SIEMENS EDA

Tessent™ SiliconInsight™ User's Manual for Tessent Shell

Software Version 2022.4
Document Revision 27

SIEMENS

About Siemens Digital Industries Software

Siemens Digital Industries Software is a leading global provider of product life cycle management (PLM) software and services with 7 million licensed seats and 71,000 customers worldwide. Headquartered in Plano, Texas, Siemens Digital Industries Software works collaboratively with companies to deliver open solutions that help them turn more ideas into successful products. For more information on Siemens Digital Industries Software products and services, visit www.siemens.com/plm.

Support Center: support.sw.siemens.com
Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

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 the 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
Tessent SiliconInsight Overview

Tessent™ SiliconInsight™ provides interactive capabilities for testing, debugging, and characterizing BIST-tested memories and logic, IEEE 1687 IJTAG instruments, and ATPG patterns. This tool also facilitates on-tester data collection and diagnosis for BIST-tested memories and logic.

Tessent SiliconInsight operates within the Tessent Shell environment. The tool uses an open ATE interface to interact with embedded instruments for pre-silicon, desktop, and ATE applications. The open interface is called the Characterization and Debug Package (CDP). The CDP contains test patterns, procedural data, and side files that enable access and control of any instrument at any hierarchy level.

Tessent SiliconInsight Architecture

Tessent SiliconInsight acts as an interface between the Tessent Shell environment and tester hardware, adaptors, and simulators.

Figure 1-1. Interactive Tessent SiliconInsight Architecture

The tool communicates with and controls the testers, adaptors, and simulators during the running of specified tests, and provides data for other applications in the Tessent Shell environment. For example, it provides data that Tessent Diagnosis can use for logic diagnosis.

When working interactively, regardless of the mode you are working in, the tool takes a Tessent Shell Database (TSDB) as input to provide design data and pattern specifications. The TSDB enables auto-generation of the test patterns.

Note

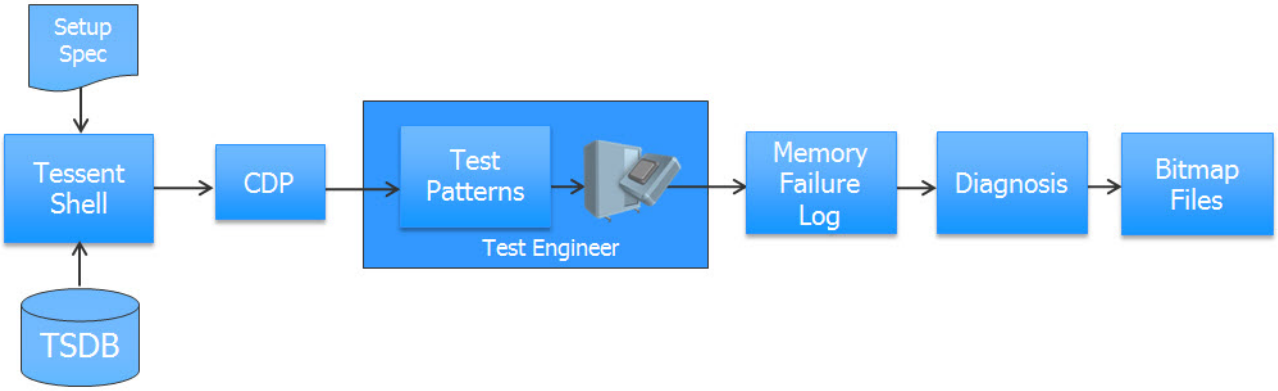
If you do not supply a TSDB, you must supply the test patterns yourself.

You supply a Tessent SiliconInsight setup specification to define the Tessent SiliconInsight mode of operation: Offline, SimDUT, Desktop, and Offline ATE.

The Tessent SiliconInsight Desktop tester process (SID tester) runs on a computer and uses the information stored within the CDP to run particular patterns for simulation, debug, or diagnostic purposes. You can also use an ATE test program to access the CDP for testing purposes and for debugging Tessent IJTAG-inserted devices.

On an ATE, you can use Tessent SiliconInsight in offline ATE diagnosis mode to diagnose memories using ESOE MemoryBIST instruments.

Figure 1-2. Offline ATE Mode Architecture



In this case, the test engineer works with the raw pattern data to convert it into JSON format that Tessent SiliconInsight can then output into a bitmap failure file.

Related Topics

Performing Diagnosis From an ATE Test Program in Offline ATE Mode

Tessent SiliconInsight Modes and Usage

Tessent SiliconInsight Modes and Usage

Tessent SiliconInsight provides various modes: Offline, Desktop, SimDUT, and Offline ATE diagnosis. Tessent SiliconInsight uses a Tessent Shell Database (TSDB) as input, and you supply a setup specification to define the mode of operation.

Table 1-1. Tessent SiliconInsight Modes of Operation

Mode	Description
Offline	Verifies test patterns without issues before using them with the Desktop and SimDUT modes.
Desktop	Performs test bring-up, debug, and characterization of devices containing Tessent Shell-inserted LogicBIST and MemoryBIST instruments in an automated interactive environment.
SimDUT	Ensures that there are no design flaws in your on-chip diagnostic hardware.
Offline ATE	Collects failure data from a previously generated CDP and saves the data to failure log files that you can diagnose. Also, debug Tessent IJTAG-inserted devices.

Offline

Use the offline mode to verify that you can run the test patterns without any issues (including syntax errors) before you use them with the Desktop and SimDUT modes.

Desktop

Use the Desktop mode for test bring-up, debug, and characterization of devices containing Tessent Shell-inserted LogicBIST and MemoryBIST instruments in an automated interactive environment.

Note

You must have a Tessent Shell Database (TSDB). Refer to the *Tessent Shell Reference Manual* for details.

If you have LogicBIST and MemoryBIST instruments inserted through the LV flow, you must use Tessent SiliconInsight for the LV flow to test and diagnose these instruments.

When you have silicon, you can use Tessent SiliconInsight in Desktop mode to perform the following:

- **Bring-up validation —** Discover under which conditions the DUT prototype fails. This is an iterative process in which you adjust and re-run your test patterns so that they exercise particular settings in the DUT.
- **Failure diagnosis —** Isolate root cause defects. During diagnosis, Tessent SiliconInsight runs test patterns and returns a list of the actual-versus-expected results for the failing cycles. You can use this data to refine and re-run the test patterns until you isolate root cause defects in your prototype that the foundry can then fix.

- **Characterization —** Isolate how the DUT performs under extreme conditions when you have a functional silicon prototype. For example, you may test for the fastest speed at which the DUT runs or the lowest voltage at which it runs. The characterization results are the performance specifications for the DUT.

You can also use Desktop mode to run and diagnose ATPG patterns (non-TSDB flow) and Tessent IJTAG-inserted instruments. If you are using ATPG, you can test the integrity of your ATPG test patterns and diagnose any failures.

SimDUT

Use the SimDUT mode to simulate the DUT and receive responses as if you are interacting with a real tester. The simulator converts the responses to failures that you can then diagnose with Tessent SiliconInsight Desktop.

You can perform pre-silicon verification to ensure that you no design flaws in your diagnostic hardware in the IP. In this case, you have a TSDB at the core or top level, and you want to verify all the diagnostic tests before tapeout.

Note

SimDUT supports chip-level simulation, not block-level simulation.

Offline ATE

Use the ATE to collect failure data from a previously generated CDP and save the data to failure log files that you can diagnose. The CDP is not integrated with the ATE. The tool supports offline diagnostic processing for Tessent MemoryBIST ESOE instruments.

For offline ATE diagnosis, an existing CDP that has been updated and fine tuned by the user through debug and characterization may be run on the ATE to perform diagnosis.

Offline ATE mode supports ATE-Connect™, which enables you to debug Tessent IJTAG-inserted devices directly on the tester via a TCP-IP connection to Tessent SiliconInsight. ATE-Connect does not require the CDP to be integrated with the ATE.

Related Topics

Tessent SiliconInsight Architecture

Chapter 2
Prepare the Tessent SiliconInsight Desktop Environment

Before using Tessent SiliconInsight, you must set up the desktop environment once for each computer. Tessent SiliconInsight Desktop runs in a Red Hat® Enterprise Linux®[1] environment on any standard Linux PC or laptop.

Note

You can find complete hardware and operating system requirements for Tessent SiliconInsight Desktop in the "Tessent SiliconInsight Configuration" section of the *Managing Tessent Software* manual.

Setting Up the Desktop Environment

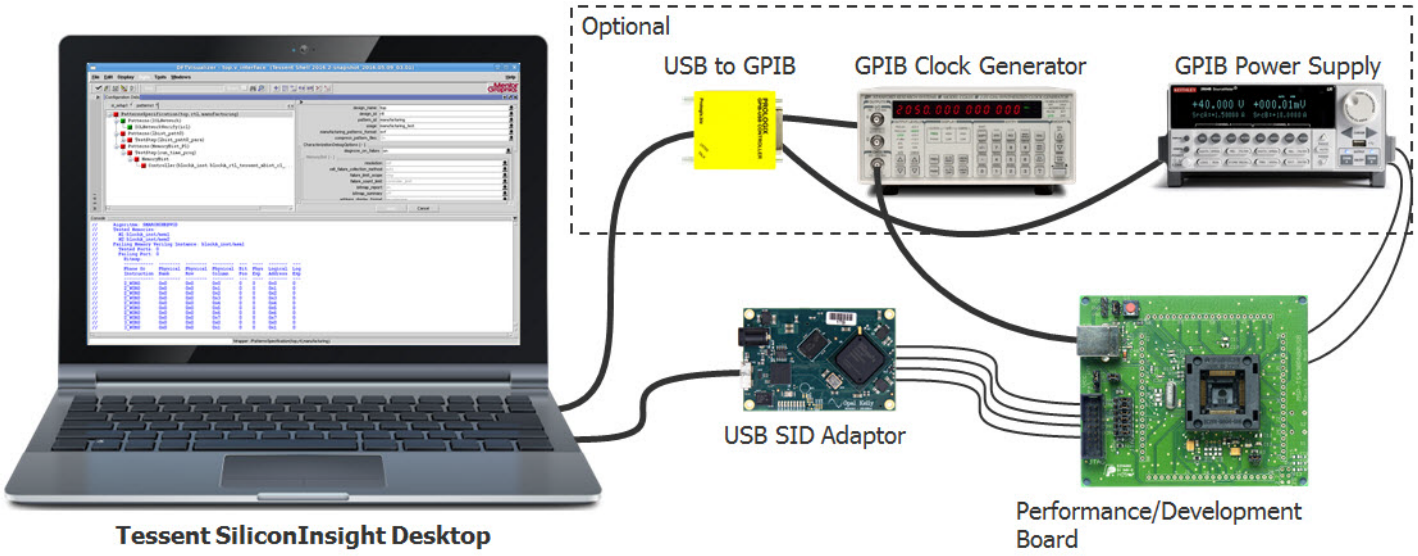
To control the DUT, attach a USB cable from the PC or laptop to an adaptor, and then connect the adaptor to standard IEEE 1149.1 pins on the performance board.

Note

Do not install additional drivers on the PC or laptop to use with the Tessent SiliconInsight Desktop adaptors. Doing so may impair the Desktop setup. Configure your Desktop environment only as described in this section.

1. Linux® is a registered trademark of Linus Torvalds in the U.S. and other countries.

Figure 2-1. Tessent SiliconInsight Desktop Hardware Configuration



For more detailed characterization activities, you can also control instruments such as power supplies and clock generators with interface buses implemented according to the General Purpose Interface Bus (GPIB) standard IEEE-488. To do this, connect a USB-to-GPIB adaptor to a GPIB instrument, which you then connect to the performance board.

When you run a test from Tessent SiliconInsight Desktop, the tool retrieves information about the test resources and design from the Tessent Shell environment. Tessent SiliconInsight Desktop then interprets the results and displays them on your computer.

Set up the desktop environment once for each machine.

Prerequisites

- A Linux PC or laptop configured with one of the following Red Hat Enterprise Linux operating systems:
  - Red Hat Enterprise Linux 7
  - Red Hat Enterprise Linux 8.2+
- USB cable(s). You need two if you are using a GPIB instrument.
- The performance board you plan to test.

Procedure

1. Configure one of the supported adaptors as described in “Supported Adaptors” on page 22.

Note

You can only use one adaptor per computer configured to run Tessent SiliconInsight Desktop. However, you may also use an additional GPIB adaptor to control other instruments.

2. Optionally, if you plan to use a GPIB instrument, configure the supported Prologix 6.0 adaptor as described in "Prologix 6.0 Adaptor" on page 70.
3. Configure your hardware as shown in Figure 2-1.
4. Based on your Red Hat Linux operating system version, configure your computer to run Tessent SiliconInsight Desktop. Perform this task while logged on as root.

Results

Completing this task enables all users in addition to *root* to access Tessent SiliconInsight Desktop. You can now proceed to "Prepare the Design Under Test" on page 75.

Related Topics

Configuring Red Hat Enterprise Linux 7 or 8 to Run Tessent SiliconInsight Desktop

Supported Adaptors

Tessent SiliconInsight Desktop supports many adaptors that you can use to connect a PC or laptop to a performance board. You can only use one adaptor per computer configured to run Tessent SiliconInsight Desktop. However, you may also use an additional GPIB adaptor to control other instruments.

Note

The TCK speed depends on the pattern format. For example, for Olimex ARM-USB-OCD-H adaptors, the observed TCK frequency is be 6 MHz for SVF patterns but 3 MHz for STIL patterns.

Supported Adaptor Specifications

Tessent SiliconInsight Desktop supports many adaptors to connect a PC or laptop to a performance board. When using adaptors in Desktop mode, the pin map specification included in the default setup specification depends on the adaptor you choose for the selected setup. In addition, each of the adaptors has an offline option to verify that the test patterns run well.

Table 2-1. Key Specifications for Supported Adaptors

Adaptor	Voltage Range	Frequency	Max Number of Pins
Opal Kelly XEM6310-LX45	1.2 V - 3.3 V	1, 5, 10, 20, 25 MHz	120
Opal Kelly XEM7310-A75	1.2 V - 3.3 V (96 pins) 3.3 V (24 pins)	10 MHz	120
Opal Kelly XEM8350-KU060	1.2 V - 1.8 V	100 MHz	330

Table 2-1. Key Specifications for Supported Adaptors (cont.)

Adaptor	Voltage Range	Frequency	Max Number of Pins
FTDI FT4232H Mini Module	3.3 V - 5.0 V	5 MHz	8
Olimex ARM-USB-OCD	2.0 V - 5.0 V	500 KHz	4
Olimex ARM-USB-OCD-H	1.65 V - 5.0 V	12 MHz	4
Tin Can Tools Flyswatter2	1.6 V - 5.0 V	12 MHz	4
Xverve Signalyzer H2, SHA40	3.3 V - 5.0 V	5 MHz	8
Xverve Signalzyer H4	3.3 V - 5.0 V	10 MHz	8
Xverve Signalyzer SP	1.2 V - 3.3 V	500 KHz	64
Xverve Signalyzer	3.3 V - 5.0 V	500 KHz	8
Amontec JTAGkey	1.4 V - 5.0 V	500 KHz	4
Amontec JTAGkey Tiny	2.4 V - 5.0 V	1.5 MHz	4

Related Topics

Default SiliconInsight Setup Specification Format

SiliconInsightSetupSpecification

Opal Kelly Adaptors

The integration of Tessent SiliconInsight and Opal Kelly adaptors form a benchtop tester for silicon validation and debug.

SiliconInsight integrates the Opal Kelly FrontPanel SDK to provide the ability to access the FPGA module from SiliconInsight. This integration enables you to program your designs into the FPGA hardware and simulate them using Tessent software, such as ATPG pattern validation, debug, and diagnosis.

The XEM6310-LX45 (or XEM6310 in this manual) is an integration module based on the Xilinx Spartan-6 FPGA. It has a USB 3.0 interface and 120 I/O pins. USB provides configuration and data transfer to the FPGA.

The XEM7310-A75 (or XEM7310 in this manual) is an integration module based on the Xilinx Artix-7 FPGA. It has a USB 3.0 interface and 120 I/O pins. The XEM7310 is compatible with the XEM6310 with minimal changes. The physical dimensions and expansion connector locations are identical. There are memory, oscillator, and some expansion connector differences.

The XEM8350-KU060 (or XEM8350 in this manual) is an integration module based on the Xilinx Kintex UltraScale FPGA. It has two independent USB 3.0 interfaces and over 330 I/O pins.

Opal Kelly XEM6310-LX45 Adaptor

Tessent SiliconInsight supports the Opal Kelly XEM6310-LX45 adaptor. This bare-board adaptor has 120 I/O pins in two banks. Using an external reference voltage, you can set the pin voltages independently in the range of 1.2V to 3.3V for each bank.

If you do not use a reference voltage, both banks are set to 3.3V. If you use a reference voltage, specify the io_standard property in the setup specification to indicate the reference voltage range.

Figure 2-2. Opal Kelly XEM6310-LX45 Adaptor

This adaptor has two high-density connectors. To connect your performance board to the XEM6310, you can either design your board with matching high-density connectors, or you can use one of the available breakout boards, BRK3010 or BRK6110. For details about this adaptor, go to https://www.opalkelly.com/products/xem6310/.

By default, the adaptor runs at 10 MHz per event vector. Assuming four events per cycle, this is equivalent to 2.5 MHz per ATPG vector. For details about the relationship between event vectors and ATPG vectors, refer to "Restrictions" in "Generating a Patterns Specification for ATPG" on page 161.

You can adjust the adaptor frequency to run slower if, for example, you have a failing test, or faster to verify the test speed limits for the tested chip. Legal event vector values are 1 MHz, 5 MHz, 10 MHz (default), 20 MHz, and 25 MHz. To control the speed at which the adaptor runs tests, specify the base_frequency property in the setup specification.

You can use the on_chip_termination property to enable a 50-ohm on-chip termination register. Enabling this register may increase signal integrity. Refer to "Default SiliconInsight Setup Specification Format" on page 83 for more information about the supported properties for this adaptor.

Note

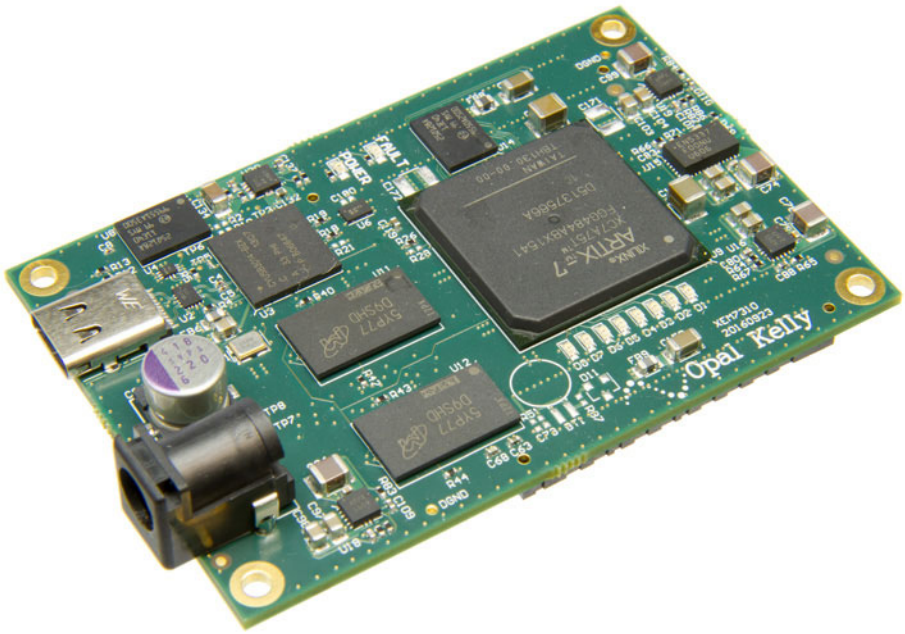
The adaptor supports transmission speeds up to 5.0 Gbps (USB 3.0); however, the actual speed depends on the USB controller that handles the connection on the PC side. Find the information about the actual established USB connection speed in the file *sid_tester.log*.

Opal Kelly XEM7310-A75 Adaptor

Tessent SiliconInsight supports the Opal Kelly XEM7310-A75 adaptor. This bare-board adaptor has 120 I/O pins in two banks. Using an external reference voltage, you can set the pin voltages independently in the range of 1.2 V to 3.3 V for each bank.

If you do not use a reference voltage, both banks are set to 3.3 V. If you use a reference voltage, specify the io_standard property in the setup specification "Sid(opalKellyXem7310)" on page 117 to indicate the reference voltage range. However, some pins are fixed at 3.3 V. See "Opal Kelly XEM7310-A75 Adaptor Pinout" on page 37 for details.

Figure 2-3. Opal Kelly XEM7310-A75 Adaptor



This adaptor has two high-density connectors. To connect your performance board to the XEM7310, you can either design your board with matching high-density connectors, or you can use one of the available breakout boards, BRK6110 or BRK7010. For details about this adaptor, go to https://www.opalkelly.com/products/xem7310/.

The adaptor runs at 10 MHz per event vector. Assuming four events per cycle, this is equivalent to 2.5 MHz per ATPG vector. For details about the relationship between event vectors and ATPG vectors, refer to "Restrictions" in "Generating a Patterns Specification for ATPG" on page 161. Unlike the XEM6310, you cannot adjust the adaptor frequency.

The XEM7310 does not have on-chip termination resistors. The on_chip_termination property defaults to off and you cannot change it. Refer to "Sid(opalKellyXem7310)" on page 117 for more information about the supported properties for this adaptor.

Note

The adaptor supports transmission speeds up to 5.0 Gbps (USB 3.0); however, the actual speed depends on the USB controller that handles the connection on the PC side. Find the information about the actual established USB connection speed in the file *sid_tester.log*.

Opal Kelly XEM8350-KU060 Adaptor

Tessent SiliconInsight supports the Opal Kelly XEM8350-KU060 adaptor. This bare-board adaptor has 330 I/O pins in two banks. You can set the pin voltages independently in the range of 1.2 V to 1.8 V for each bank.

You can set the pin voltages from 1.2 V to 1.8 V for each bank. Specify the io_standard property in the setup specification "Sid(opalKellyXem8350)" on page 121 to indicate the reference voltage range.

Figure 2-4. Opal Kelly XEM8350-KU060 Adaptor

This adaptor has two high-density I/O connectors. To connect your performance board to the XEM8350, you can either design your board with matching high-density connectors, or you can use one of the available breakout boards, BRK8350MG (recommended for access to all 330 pins) or BRK8350. For details about this adaptor, go to https://www.opalkelly.com/products/xem8350/.

The XEM8350 does not have on-chip termination resistors. The on_chip_termination property defaults to off and you cannot change it. Refer to "Sid(opalKellyXem8350)" on page 121 for more information about the supported properties for this adaptor.

Note

The adaptor supports transmission speeds up to 5.0 Gbps (USB 3.0); however, the actual speed depends on the USB controller that handles the connection on the PC side. Find the information about the actual established USB connection speed in the file *sid_tester.log*.

Guidelines for Setting XEM6310 and XEM7310 Adaptor I/O Pin Voltage

You can adjust the pin voltage in two I/O banks, with the pins in each bank having the same I/O setting. The default I/O voltage for the adaptor is 3.3 V. For any other voltage, you must manually connect the reference voltages (1.2 V - 3.3 V) from an external power supply to the board's VCCO pins.

To connect the reference voltages to the VCCO pins, you can use USB 2.0 or USB 3.0 connection; USB 3.0 is recommended because it is faster.

For voltages below 1.8 V, you must specify "io_standard 1.2 V" in the setup specification. The io_standard property specifies the pin voltage range setting for the adaptor. Specifying "1.2" implies a setting in the 1.2 V to 1.8 V range. Specifying "1.8" (the default) implies a setting in the 1.8 V to 3.3 V range.

To supply external power to a particular I/O bank through the VCCO pin, you must remove the adaptor's ferrite bead.

Caution

Because the ferrite beads can be damaged, it is generally safer to unsolder them and provide VCCO to both banks.

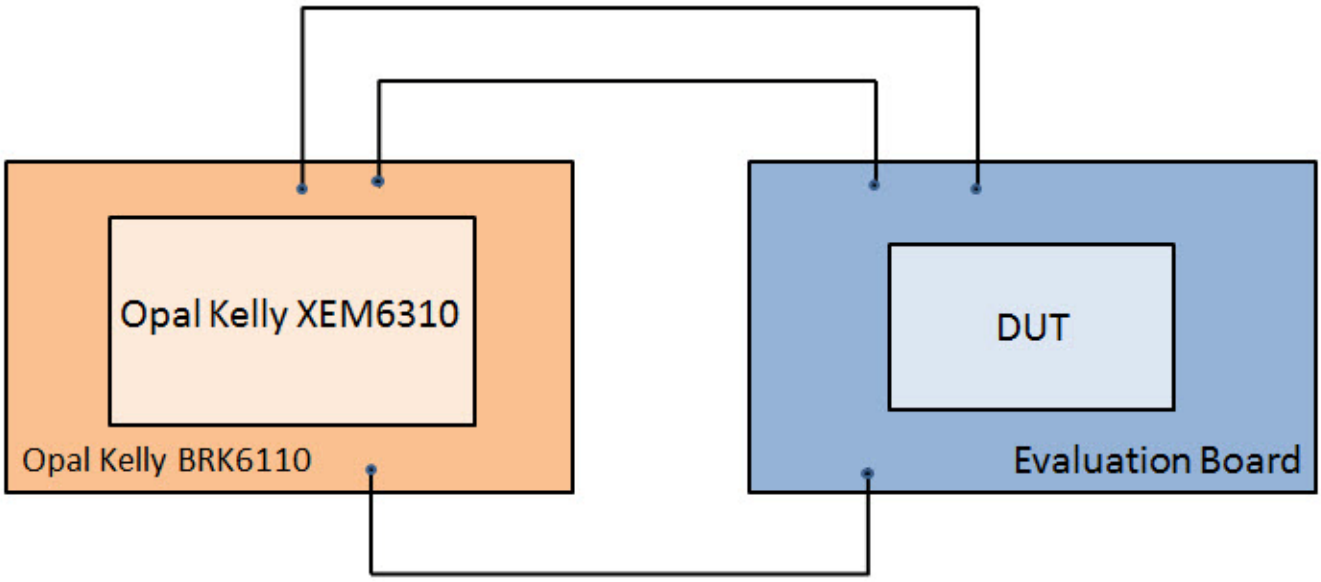
For BRK6110, some DGND pins are more stable than others. The DGND pin on the JP2A connector seems to be too close to the high voltage pins and thus provides unstable results. Preferably, use the JP1A-36, JP2A-35, JP1B-56, JP2B-55 pins, because they are not as close to the supply pins as the JP1A-1, JP1A-13, JP1A-14, JP2A-2, JP2A-14 pins.

For more information about how to configure the Opal Kelly adaptor to support voltages lower than 3.3V, refer to the appropriate Opal Kelly website XEM6310 Expansion Connectors or XEM7310 Expansion Connectors.

Signal Integrity Stability

You can use high-density connectors to connect the Opal Kelly adaptor to the BKR6110 breakout board, and then connect the breakout board to the DUT with wires. However, when you do this, the wires are ungrounded and can interfere with each other, which can cause signal integrity problems that you must stabilize.

Figure 2-5. Wiring for Opal Kelly Adaptor With BreakOut Board



The driver impedance for the Opal Kelly adaptor is set to force its impedance to be equal to the transmission line impedance of the PCB signal trace. The adaptor drives many pins at the same time and each produces an electromagnetic impulse whose strength comes from the adaptor's driver current intensity. When you have ungrounded wiring between the driver (the adaptor) and the receiver (the DUT), the impulses induct on response connections (that is, TDO) and can disturb the test. This effect intensifies when there are many pins switching simultaneously.

To decrease the driver current and thus the induction strength—thereby increasing signal integrity—enable on-chip termination with the on_chip_termination property in the Sid(opalKellyXem6310)/opalKellyXem6310 wrapper. For example:

```
...
Sid(opalKellyXem6310) {
   ...
   opalKellyXem6310 {
      on_chip_termination : on
      PinMap {
      ...
```

When you turn on on-chip termination, the tool adds an additional resistor to the Opal Kelly adaptor that is calculated to match 50 Ohm, which guarantees good communication between the adaptor and the DUT.

When you connect the Opal Kelly adaptor to the DUT directly through high-density PCB connectors on the evaluation board, signal integrity is not an issue. The PCB connections are designed to match impedance and prevent signal integrity issues. However, you can maintain the on_chip_termination property in the on state with no negative impact to testing or the equipment.

Figure 2-6. Wiring for Opal Kelly Adaptor Without Breakout Board

Related Topics

Sid(opalKellyXem6310)

Opal Kelly XEM6310-LX45 Adaptor Pinout

Tessent SiliconInsight pin names map to Opal Kelly connectors and pin numbers. Breakout boards 6110 and 3010 have different mappings. The 6110 breakout board uses a 2.00 mm pitch and the 3010 breakout board uses a 2.54 mm pitch.

Caution

When you connect the TAP pins of your DUT, do not connect them to the native JTAG pins on the board. You should treat the TAP pins as any other digital pin and wire them to one of the channels labeled P0 through P119.

Note

The notes of Table 2-2 are as follows:

- 1 — XEM6310 ground pin. (green row)
- 2 — XEM6310 power pin. (red row)
- 3 — XEM6310 pin that is not used by SiliconInsight. (grey row)
- 4 — XEM6310 pin that is not connected on BRK6110. (“n/c” in yellow cell)
- 5 — SID trigger pin for smart mode only. (“Trigger” in yellow cell)
- 6 — SID I/O pin programmable for 1.2 V or 1.8 V. (P<#> in green cell)
- 7 — XEM6310 VCCO pin. (yellow row)
- 8 — XEM6310 voltage difference from XEM7310. (+1.2VDD in yellow cell)

The following table shows the pin mappings from Tessent SiliconInsight desktop (SID) pins to the various connectors that you might use. For example, SID pin P1 connects to the HD connector pin JP2-15, which also connects to FPGA pin G16. If using a BRK6110 breakout board, SID pin P1 connects to JP1A-15.

Table 2-2. Tessent SiliconInsight Pin Map for XEM6310

SID Pin	HD Connector	FPGA Pin	Function	BRK6110 Connector	Notes
	JP2-1		DGND	JP1A-1	1
	JP2-2		+3.3VDD	JP1A-2	2
	JP2-3		VBATT	n/c	3, 4
	JP2-4		+3.3VDD	JP1A-4	2
	JP2-5	0	JTAG_TCK	JP3-6	3
	JP2-6		+3.3VDD	JP1A-6	2
	JP2-7	0	JTAG_TMS	JP3-4	3
	JP2-8	0	JTAG_TDO	JP3-8	3
	JP2-9	0	JTAG_TDI	JP3-10	3
	JP2-10	Bank 1 VREF	VREF_1	JP1A-10	3
	JP2-11	U12	L22N_2	JP1A-11	3
	JP2-12	Rfuse	Rfuse	n/c	3, 4
	JP2-13		DGND	JP1A-13	1
	JP2-14		DGND	JP1A-14	1

Table 2-2. Tessent SiliconInsight Pin Map for XEM6310 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK6110 Connector	Notes
P1	JP2-15	G16	L9P_1	JP1A-15	6
P0	JP2-16	G19	L33P_1	JP1A-16	6
P3	JP2-17	G17	L9N_1	JP1A-17	6
P2	JP2-18	F20	L33N_1	JP1A-18	6
P5	JP2-19	H19	L34P_1	JP1A-19	6
P4	JP2-20	H20	L38P_1	JP1A-20	6
P7	JP2-21	H18	L34N_1	JP1A-21	6
P6	JP2-22	J19	L38N_1	JP1A-22	6
P9	JP2-23	F16	L10P_1	JP1A-23	6
P8	JP2-24	D19	L29P_1	JP1A-24	6
P11	JP2-25	F17	L10N_1	JP1A-25	6
P10	JP2-26	D20	L29N_1	JP1A-26	6
P13	JP2-27	J17	L36P_1	JP1A-27	6
P12	JP2-28	F18	L30P_1	JP1A-28	6
P15	JP2-29	K17	L36N_1	JP1A-29	6
P14	JP2-30	F19	L30N_1	JP1A-30	6
P17	JP2-31	K16	L21P_1	JP1A-31	6
P16	JP2-32	M16	L58P_1	JP1A-32	6
P19	JP2-33	J16	L21N_1	JP1A-33	6
P18	JP2-34	L15	L58N_1	JP1A-34	6
	JP2-35		+VCCO1	JP1A-35	7
	JP2-36		DGND	JP1A-36	1
P65	JP2-37	V21	L52P_1	JP1A-37	6
P64	JP2-38	K20	L40P_GCLK11_1	JP1A-38	6
P67	JP2-39	V22	L52N_1	JP1A-39	6
P66	JP2-40	K19	L40N_GCLK10_1	JP1A-40	6
P21	JP2-41	T21	L50P_1	JP1B-41	6
P20	JP2-42	U20	L51P_1	JP1B-42	6
P23	JP2-43	T22	L50N_1	JP1B-43	6

Table 2-2. Tessent SiliconInsight Pin Map for XEM6310 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK6110 Connector	Notes
P22	JP2-44	U22	L51N_1	JP1B-44	6
P25	JP2-45	P21	L48P_1	JP1B-45	6
P24	JP2-46	R20	L49P_1	JP1B-46	6
P27	JP2-47	P22	L48N_1	JP1B-47	6
P26	JP2-48	R22	L49N_1	JP1B-48	6
P29	JP2-49	M21	L46P_1	JP1B-49	6
P28	JP2-50	N20	L47P_1	JP1B-50	6
P31	JP2-51	M22	L46N_1	JP1B-51	6
P30	JP2-52	N22	L47N_1	JP1B-52	6
P73	JP2-53	L20	L45P_1	JP1B-53	6
P72	JP2-54	M20	L42P_GCLK7_1	JP1B-54	6
	JP2-55		+VCCO1	JP1B-55	7
	JP2-56		DGND	JP1B-56	1
P75	JP2-57	L22	L45N_1	JP1B-57	6
P74	JP2-58	L19	L42N_GCLK6_1	JP1B-58	6
P77	JP2-59	H21	L41P_GCLK9_1	JP1B-59	6
P76	JP2-60	K21	L44P_1	JP1B-60	6
P79	JP2-61	H22	L41N_GCLK8_1	JP1B-61	6
P78	JP2-62	K22	L44N_1	JP1B-62	6
P53	JP2-63	F21	L37P_1	JP1B-63	6
P52	JP2-64	G20	L39P_1	JP1B-64	6
P55	JP2-65	F22	L37N_1	JP1B-65	6
P54	JP2-66	G22	L39N_1	JP1B-66	6
P57	JP2-67	D21	L31P_1	JP1B-67	6
P56	JP2-68	E20	L35P_1	JP1B-68	6
P59	JP2-69	D22	L31N_1	JP1B-69	6
P58	JP2-70	E22	L35N_1	JP1B-70	6
P61	JP2-71	B21	L19P_1	JP1B-71	6
P60	JP2-72	C20	L32P_1	JP1B-72	6

Table 2-2. Tessent SiliconInsight Pin Map for XEM6310 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK6110 Connector	Notes
P63	JP2-73	B22	L19N_1	JP1B-73	6
P62	JP2-74	C22	L32N_1	JP1B-74	6
P81	JP2-75	A21	L20N_1	JP1B-75	6
P80	JP2-76	A20	L20P_1	JP1B-76	6
P82	JP2-77	J20	L43P_GCLK5_1	JP1B-77	6
	JP2-78		DGND	JP1B-78	1
P83	JP2-79	J22	L43N_GCLK4_1	JP1B-79	6
	JP2-80		DGND	JP1B-80	1
	JP1-1		+VDC	JP2A-1	2
	JP1-2		DGND	JP2A-2	1
	JP1-3		+VDC	JP2A-3	2
	JP1-4		+1.2VDD	JP2A-4	2, 8
	JP1-5		+VDC	JP2A-5	2
	JP1-6		+1.2VDD	JP2A-6	2, 8
	JP1-7		+1.8VDD	JP2A-7	2
	JP1-8	T14	L23P_2	JP2A-8	3
	JP1-9		+3.3VDD	JP2A-9	2
	JP1-10	Y9	L43P_2	JP2A-10	3
	JP1-11		+3.3VDD	JP2A-11	2
Trigger	JP1-12	AB9	L43N_2	JP2A-12	5
	JP1-13		+3.3VDD	JP2A-13	2
	JP1-14		DGND	JP2A-14	1
P33	JP1-15	W20	L60P_1	JP2A-15	6
P32	JP1-16	T19	L74P_1	JP2A-16	6
P35	JP1-17	W22	L60N_1	JP2A-17	6
P34	JP1-18	T20	L74N_1	JP2A-18	6
P37	JP1-19	U19	L70P_1	JP2A-19	6
P36	JP1-20	P17	L72P_1	JP2A-20	6
P39	JP1-21	V20	L70N_1	JP2A-21	6

Table 2-2. Tessent SiliconInsight Pin Map for XEM6310 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK6110 Connector	Notes
P38	JP1-22	N16	L72N_1	JP2A-22	6
P41	JP1-23	C5	L2P_0	JP2A-23	6
P40	JP1-24	M17	L71P_1	JP2A-24	6
P43	JP1-25	A5	L2N_0	JP2A-25	6
P42	JP1-26	M18	L71N_1	JP2A-26	6
P45	JP1-27	D14	L49P_0	JP2A-27	6
P44	JP1-28	P18	L73P_1	JP2A-28	6
P47	JP1-29	C14	L49N_0	JP2A-29	6
P46	JP1-30	R19	L73N_1	JP2A-30	6
P49	JP1-31	E16	L66P_0	JP2A-31	6
P48	JP1-32	D9	L7P_0	JP2A-32	6
P51	JP1-33	D17	L66N_0	JP2A-33	6
P50	JP1-34	C8	L7N_0	JP2A-34	6
	JP1-35		DGND	JP2A-35	1
	JP1-36		+VCCO0	JP2A-36	7
P69	JP1-37	D7	L32P_0	JP2A-37	6
P68	JP1-38	D10	L33P_0	JP2A-38	6
P71	JP1-39	D8	L32N_0	JP2A-39	6
P70	JP1-40	C10	L33N_0	JP2A-40	6
P85	JP1-41	L17	L61P_1	JP2B-41	6
P84	JP1-42	D11	L36P_GCLK15_0	JP2B-42	6
P87	JP1-43	K18	L61N_1	JP2B-43	6
P86	JP1-44	C12	L36N_GCLK14_0	JP2B-44	6
P89	JP1-45	D6	L3P_0	JP2B-45	6
P88	JP1-46	D15	L62P_0	JP2B-46	6
P91	JP1-47	C6	L3N_0	JP2B-47	6
P90	JP1-48	C16	L62N_VREF_0	JP2B-48	6
P93	JP1-49	A3	L1P_HSWAPEN_0	JP2B-49	6
P92	JP1-50	B6	L4P_0	JP2B-50	6

Table 2-2. Tessent SiliconInsight Pin Map for XEM6310 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK6110 Connector	Notes
P95	JP1-51	A4	L1N_VREF_0	JP2B-51	6
P94	JP1-52	A6	L4N_0	JP2B-52	6
P97	JP1-53	B8	L6P_0	JP2B-53	6
P96	JP1-54	C7	L5P_0	JP2B-54	6
	JP1-55		DGND	JP2B-55	1
	JP1-56		+VCCO0	JP2B-56	7
P99	JP1-57	A8	L6N_0	JP2B-57	6
P98	JP1-58	A7	L5N_0	JP2B-58	6
P101	JP1-59	B10	L34P_GCLK19_0	JP2B-59	6
P100	JP1-60	C9	L8P_0	JP2B-60	6
P103	JP1-61	A10	L34N_GCLK18_0	JP2B-61	6
P102	JP1-62	A9	L8N_VREF_0	JP2B-62	6
P105	JP1-63	C13	L38P_0	JP2B-63	6
P104	JP1-64	B12	L37P_GCLK13_0	JP2B-64	6
P107	JP1-65	A13	L38N_VREF_0	JP2B-65	6
P106	JP1-66	A12	L37N_GCLK12_0	JP2B-66	6
P109	JP1-67	C15	L51P_0	JP2B-67	6
P108	JP1-68	B14	L50P_0	JP2B-68	6
P111	JP1-69	A15	L51N_0	JP2B-69	6
P110	JP1-70	A14	L50N_0	JP2B-70	6
P113	JP1-71	C17	L64P_0	JP2B-71	6
P112	JP1-72	B16	L63P_0	JP2B-72	6
P115	JP1-73	A17	L64N_0	JP2B-73	6
P114	JP1-74	A16	L63N_0	JP2B-74	6
P117	JP1-75	A18	L65N_0	JP2B-75	6
P116	JP1-76	B18	L65P_0	JP2B-76	6
P118	JP1-77	C11	L35P_GCLK17_0	JP2B-77	6
	JP1-78		DGND	JP2B-78	1
P119	JP1-79	A11	L35N_GCLK16_0	JP2B-79	6

Table 2-2. Tessent SiliconInsight Pin Map for XEM6310 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK6110 Connector	Notes
	JP1-80		DGND	JP2B-80	1

Opal Kelly XEM7310-A75 Adaptor Pinout

Tessent SiliconInsight pins map to Opal Kelly XEM7310 high density (HD) connectors or breakout boards, depending on your application. The breakout boards BRK6110 and BRK7010 have different mappings, but they both use the same 2.00 mm pitch headers.

Caution

When you connect the TAP pins of your DUT, do not connect them to the native JTAG pins on the board. You should treat the TAP pins as any other digital pin and wire them to one of the channels labeled P0 through P119.

Note

The notes of Table 2-3 are as follows:

- 1 — XEM7310 ground pin. (green row)
- 2 — XEM7310 power pin. (red row)
- 3 — XEM7310 pin that is not used by SiliconInsight. (grey row)
- 4 — XEM7310 pin that is not connected on BRK6110. (“n/c” in yellow cell)
- 5 — SID trigger pin. (“Trigger” in yellow cell)
- 6 — SID I/O pin programmable for 1.2 V or 1.8 V. (P<#> in green cell)
- 7 — XEM7310 VCCO pin. (yellow row)
- 8 — SID I/O pin fixed at 3.3 V. (P<#> in yellow cell)
- 9 — XEM7310 voltage difference from XEM6310. (+1.0VDD in yellow cell)

The following table shows the pin mappings from Tessent SiliconInsight desktop (SID) pins to the various connectors that you might use. For example, SID pin P1 connects to the HD connector pin MC2-15, which also connects to FPGA pin P5. If using a BRK7010 breakout board, SID pin P1 connects to J3-15. If using a BRK6110 breakout board, SID pin P1 connects to JP1A-15.

Table 2-3. Tessent SiliconInsight Pin Map for XEM7310

SID Pin	HD Connector	FPGA Pin	Function	BRK7010 Connector	BRK6110 Connector	Notes
	MC2-1		DGND	J3-1	JP1A-1	1

Table 2-3. Tessent SiliconInsight Pin Map for XEM7310 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK7010 Connector	BRK6110 Connector	Notes
	MC2-2		+3.3VDD	J3-2	JP1A-2	2
	MC2-3		+VCCBATT	J3-3	n/c	3, 4
	MC2-4		+3.3VDD	J3-4	JP1A-4	2
	MC2-5	V12	FPGA_TCK	J6-6	JP3-6	3
	MC2-6		+3.3VDD	J3-6	JP1A-6	2
	MC2-7	T13	FPGA_TMS	J6-4	JP3-4	3
	MC2-8	U13	FPGA_TDO	J6-8	JP3-8	3
	MC2-9	R13	FPGA_TDI	J6-10	JP3-10	3
	MC2-10	F4	B35_IO0	J3-10	JP1A-10	3
	MC2-11	AB12	SYS_CLK_MC2	J3-11	JP1A-11	3
Trigger	MC2-12	L6	B35_IO25	J3-12	n/c	4, 5
	MC2-13		DGND	J3-13	JP1A-13	1
	MC2-14		DGND	J3-14	JP1A-14	1
P1	MC2-15	P5	B35_L21P	J3-15	JP1A-15	6
P0	MC2-16	P6	B35_L24P	J3-16	JP1A-16	6
P3	MC2-17	P4	B35_L21N	J3-17	JP1A-17	6
P2	MC2-18	N5	B35_L24N	J3-18	JP1A-18	6
P5	MC2-19	N4	B35_L19P	J3-19	JP1A-19	6
P4	MC2-20	P2	B35_L22P	J3-20	JP1A-20	6
P7	MC2-21	N3	B35_L19N	J3-21	JP1A-21	6
P6	MC2-22	N2	B35_L22N	J3-22	JP1A-22	6
P9	MC2-23	L5	B35_L18P	J3-23	JP1A-23	6
P8	MC2-24	R1	B35_L20P	J3-24	JP1A-24	6
P11	MC2-25	L4	B35_L18N	J3-25	JP1A-25	6
P10	MC2-26	P1	B35_L20N	J3-26	JP1A-26	6
P13	MC2-27	M6	B35_L23P	J3-27	JP1A-27	6
P12	MC2-28	M3	B35_L16P	J3-28	JP1A-28	6
P15	MC2-29	M5	B35_L23N	J3-29	JP1A-29	6
P14	MC2-30	M2	B35_L16N	J3-30	JP1A-30	6

Table 2-3. Tessent SiliconInsight Pin Map for XEM7310 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK7010 Connector	BRK6110 Connector	Notes
P17	MC2-31	M1	B35_L15P	J3-31	JP1A-31	6
P16	MC2-32	K6	B35_L17P	J3-32	JP1A-32	6
P19	MC2-33	L1	B35_L15N	J3-33	JP1A-33	6
P18	MC2-34	J6	B35_L17N	J3-34	JP1A-34	6
	MC2-35		VCCO_MC2	J3-35	JP1A-35	7
	MC2-36		DGND	J3-36	JP1A-36	1
P65	MC2-37	K2	B35_L9P	J3-37	JP1A-37	6
P64	MC2-38	L3	B35_L14P	J3-38	JP1A-38	6
P67	MC2-39	J2	B35_L9N	J3-39	JP1A-39	6
P66	MC2-40	K3	B35_L14N	J3-40	JP1A-40	6
P21	MC2-41	K1	B35_L7P	J4-1	JP1B-41	6
P20	MC2-42	J5	B35_L10P	J4-2	JP1B-42	6
P23	MC2-43	J1	B35_L7N	J4-3	JP1B-43	6
P22	MC2-44	H5	B35_L10N	J4-4	JP1B-44	6
P25	MC2-45	H3	B35_L11P	J4-5	JP1B-45	6
P24	MC2-46	H2	B35_L8P	J4-6	JP1B-46	6
P27	MC2-47	G3	B35_L11N	J4-7	JP1B-47	6
P26	MC2-48	G2	B35_L8N	J4-8	JP1B-48	6
P29	MC2-49	E2	B35_L4P	J4-9	JP1B-49	6
P28	MC2-50	G1	B35_L5P	J4-10	JP1B-50	6
P31	MC2-51	D2	B35_L4N	J4-11	JP1B-51	6
P30	MC2-52	F1	B35_L5N	J4-12	JP1B-52	6
P73	MC2-53	F3	B35_L6P	J4-13	JP1B-53	6
P72	MC2-54	E1	B35_L3P	J4-14	JP1B-54	6
	MC2-55		VCCO_MC2	J4-15	JP1B-55	7
	MC2-56		DGND	J4-16	JP1B-56	1
P75	MC2-57	E3	B35_L6N	J4-17	JP1B-57	6
P74	MC2-58	D1	B35_L3N	J4-18	JP1B-58	6
P77	MC2-59	B1	B35_L1P	J4-19	JP1B-59	6

Table 2-3. Tessent SiliconInsight Pin Map for XEM7310 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK7010 Connector	BRK6110 Connector	Notes
P76	MC2-60	C2	B35_L2P	J4-20	JP1B-60	6
P79	MC2-61	A1	B35_L1N	J4-21	JP1B-61	6
P78	MC2-62	B2	B35_L2N	J4-22	JP1B-62	6
P53	MC2-63	K4	B35_L13P	J4-23	JP1B-63	6
P52	MC2-64	U15	B13_L14P	J4-24	JP1B-64	8
P55	MC2-65	J4	B35_L13N	J4-25	JP1B-65	6
P54	MC2-66	V15	B13_L14N	J4-26	JP1B-66	8
P57	MC2-67	T16	B13_L17P	J4-27	JP1B-67	8
P56	MC2-68	T14	B13_L15P	J4-28	JP1B-68	8
P59	MC2-69	U16	B13_L17N	J4-29	JP1B-69	8
P58	MC2-70	T15	B13_L15N	J4-30	JP1B-70	8
P61	MC2-71	V13	B13_L13P	J4-31	JP1B-71	8
P60	MC2-72	W14	B13_L6P	J4-32	JP1B-72	8
P63	MC2-73	V14	B13_L13N	J4-33	JP1B-73	8
P62	MC2-74	Y14	B13_L6N	J4-34	JP1B-74	8
P81	MC2-75	Y11	B13_L11P	J4-35	JP1B-75	8
P80	MC2-76	Y12	B13_L11N	J4-36	JP1B-76	8
P82	MC2-77	H4	B35_L12P_MRCC	J4-37	JP1B-77	6
	MC2-78		DGND	J4-38	JP1B-78	1
P83	MC2-79	G4	B35_L12N_MRCC	J4-39	JP1B-79	6
	MC2-80		DGND	J4-40	JP1B-80	1
	MC1-1		+VDCIN	J1-1	JP2A-1	2
	MC1-2		DGND	J1-2	JP2A-2	1
	MC1-2		+VDCIN	J1-3	JP2A-3	2
	MC1-4		+1.0VDD	J1-4	JP2A-4	2, 9
	MC1-5		+VDCIN	J1-5	JP2A-5	2
	MC1-6		+1.0VDD	J1-6	JP2A-6	2, 9
	MC1-7		+1.8VDD	J1-7	JP2A-7	2
	MC1-8	AB11	SYS_CLK_MC1	J1-8	JP2A-8	3

Table 2-3. Tessent SiliconInsight Pin Map for XEM7310 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK7010 Connector	BRK6110 Connector	Notes
	MC1-9		+3.3VDD	J1-9	JP2A-9	2
	MC1-10	M9	XADC_VN	J1-10	JP2A-10	3
	MC1-11		+3.3VDD	J1-11	JP2A-11	2
	MC1-12	L10	XADC_VP	J1-12	JP2A-12	3
	MC1-13		+3.3VDD	J1-13	JP2A-13	2
	MC1-14		DGND	J1-14	JP2A-14	1
P33	MC1-15	W9	B34_L24P	J1-15	JP2A-15	6
P32	MC1-16	V9	B34_L21P	J1-16	JP2A-16	6
P35	MC1-17	Y9	B34_L24N	J1-17	JP2A-17	6
P34	MC1-18	V8	B34_L21N	J1-18	JP2A-18	6
P37	MC1-19	R6	B34_L17P	J1-19	JP2A-19	6
P36	MC1-20	V7	B34_L19P	J1-20	JP2A-20	6
P39	MC1-21	T6	B34_L17N	J1-21	JP2A-21	6
P38	MC1-22	W7	B34_L19N	J1-22	JP2A-22	6
P41	MC1-23	U6	B34_L16P	J1-23	JP2A-23	6
P40	MC1-24	Y8	B34_L23P	J1-24	JP2A-24	6
P43	MC1-25	V5	B34_L16N	J1-25	JP2A-25	6
P42	MC1-26	Y7	B34_L23N	J1-26	JP2A-26	6
P45	MC1-27	T5	B34_L14P	J1-27	JP2A-27	6
P44	MC1-28	W6	B34_L15P	J1-28	JP2A-28	6
P47	MC1-29	U5	B34_L14N	J1-29	JP2A-29	6
P46	MC1-30	W5	B34_L15N	J1-30	JP2A-30	6
P49	MC1-31	AA5	B34_L10P	J1-31	JP2A-31	6
P48	MC1-32	R4	B34_L13P	J1-32	JP2A-32	6
P51	MC1-33	AB5	B34_L10N	J1-33	JP2A-33	6
P50	MC1-34	T4	B34_L13N	J1-34	JP2A-34	6
	MC1-35		DGND	J1-35	JP2A-35	1
	MC1-36		VCCO_MC1	J1-36	JP2A-36	7
P69	MC1-37	AB7	B34_L20P	J1-37	JP2A-37	6

Table 2-3. Tessent SiliconInsight Pin Map for XEM7310 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK7010 Connector	BRK6110 Connector	Notes
P68	MC1-38	Y4	B34_L11P	J1-38	JP2A-38	6
P71	MC1-39	AB6	B34_L20N	J1-39	JP2A-39	6
P70	MC1-40	AA4	B34_L11N	J1-40	JP2A-40	6
P85	MC1-41	R3	B34_L3P	J2-1	JP2B-41	6
P84	MC1-42	Y6	B34_L18P	J2-2	JP2B-42	6
P87	MC1-43	R2	B34_L3N	J2-3	JP2B-43	6
P86	MC1-44	AA6	B34_L18N	J2-4	JP2B-44	6
P89	MC1-45	Y3	B34_L9P	J2-5	JP2B-45	6
P88	MC1-46	AA8	B34_L22P	J2-6	JP2B-46	6
P91	MC1-47	AA3	B34_L9N	J2-7	JP2B-47	6
P90	MC1-48	AB8	B34_L22N	J2-8	JP2B-48	6
P93	MC1-49	U2	B34_L2P	J2-9	JP2B-49	6
P92	MC1-50	U3	B34_L6P	J2-10	JP2B-50	6
P95	MC1-51	V2	B34_L2N	J2-11	JP2B-51	6
P94	MC1-52	V3	B34_L6N	J2-12	JP2B-52	6
P97	MC1-53	W2	B34_L4P	J2-13	JP2B-53	6
P96	MC1-54	W1	B34_L5P	J2-14	JP2B-54	6
	MC1-55		DGND	J2-15	JP2B-55	1
	MC1-56		VCCO_MC1	J2-16	JP2B-56	7
P99	MC1-57	Y2	B34_L4N	J2-17	JP2B-57	6
P98	MC1-58	Y1	B34_L5N	J2-18	JP2B-58	6
P101	MC1-59	T1	B34_L1P	J2-19	JP2B-59	6
P100	MC1-60	AB3	B34_L8P	J2-20	JP2B-60	6
P103	MC1-61	U1	B34_L1N	J2-21	JP2B-61	6
P102	MC1-62	AB2	B34_L8N	J2-22	JP2B-62	6
P105	MC1-63	AA1	B34_L7P	J2-23	JP2B-63	6
P104	MC1-64	Y13	B13_L5P	J2-24	JP2B-64	8
P107	MC1-65	AB1	B34_L7N	J2-25	JP2B-65	6
P106	MC1-66	AA14	B13_L5N	J2-26	JP2B-66	8

Table 2-3. Tessent SiliconInsight Pin Map for XEM7310 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK7010 Connector	BRK6110 Connector	Notes
P109	MC1-67	AB16	B13_L2P	J2-27	JP2B-67	8
P108	MC1-68	AA13	B13_L3P	J2-28	JP2B-68	8
P111	MC1-69	AB17	B13_L2N	J2-29	JP2B-69	8
P110	MC1-70	AB13	B13_L3N	J2-30	JP2B-70	8
P113	MC1-71	AA15	B13_L4P	J2-31	JP2B-71	8
P112	MC1-72	W15	B13_L16P	J2-32	JP2B-72	8
P115	MC1-73	AB15	B13_L4N	J2-33	JP2B-73	8
P114	MC1-74	W16	B13_L16N	J2-34	JP2B-74	8
P117	MC1-75	Y16	B13_L1P	J2-35	JP2B-75	8
P116	MC1-76	AA16	B13_L1N	J2-36	JP2B-76	8
P118	MC1-77	V4	B34_L12P_MRCC	J2-37	JP2B-77	6
	MC1-78		DGND	J2-38	JP2B-78	1
P119	MC1-79	W4	B34_L12N_MRCC	J2-39	JP2B-79	6
	MC1-80		DGND	J2-40	JP2B-80	1

Opal Kelly XEM8350-KU060 Adaptor Pinout

Tessent SiliconInsight supports the Opal Kelly XEM8350-KU060 adaptor, which provides support for an adaptor having up to 330 pins. The Opal Kelly XEM8350 has two banks of pins, for which it automatically manages the VIO voltage levels. These pins are in the MC1 and MC2 connectors, which use a ground spine instead of ground pins.

Note

While this documentation refers to the Opal Kelly XEM8350 adaptor generally, only the XEM8350-KU060 has been tested and verified for use with Tessent SiliconInsight.

You must connect the XEM8350 to your computer using a USB 3.0 cable. The XEM8350 uses only the USB port labeled “Primary”. You can deliver power in the following ways:

- Using a J3 barrel connector.
- Using the VCC pin on the high-density MC1 connector.

Refer to the XEM8350 documentation from Opal Kelly for detailed information. You can find official information at the Opal Kelly website:

https://opalkelly.com/products/xem8350/

Caution

When you connect the TAP pins of your DUT, do not connect them to the native JTAG pins on the board. You should treat the TAP pins as any other digital pin and wire them to one of the standard channels.

Note

The adaptor supports transmission speeds up to 5.0 Gbps (USB 3.0); however, the actual speed depends on the USB controller that handles the connection on the PC side. Find the information about the actual established USB connection speed in the file *sid_tester.log*.

Note

The MC1 and MC2 connectors have a comprehensive ground spine instead of individual ground pins. Therefore, the pin map of Table 2-4 does not list ground pins.

Note

The notes of Table 2-4 are as follows:

- 1 — XEM8350 ground pin. (green row)
- 2 — XEM8350 power pin. (red row)
- 3 — XEM8350 pin that is not used by SiliconInsight. (grey row)
- 4 — SID I/O pin programmable for 1.2 V or 1.8 V. (P<#> in green cell)
- 5 — XEM7310 VIO, VREF, or VCCBATT pin. (yellow row)
- 6 — Listed pin is not connected. (“Not connected” in yellow cell)
- 7 — SID trigger pin. (“Trigger” in yellow cell)
- 8 — SID reserved pin. (“Reserved” in grey cell)
- 9 — BRK8350 connector pin difference. (rev G / rev F in yellow cell)

The following table shows the pin map for this adaptor, including for the BRK8350MG (recommended for access to all 330 pins) and BRK8350 breakout boards.

Table 2-4. Tessent SiliconInsight Pin Map for Opal Kelly XEM8350

SID Pin	HD Connector	FPGA Pin	Function	BRK8350MG Connector	BRK8350 Connector	Notes
	MC1-0		DGND			1

Table 2-4. Tessent SiliconInsight Pin Map for Opal Kelly XEM8350 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK8350MG Connector	BRK8350 Connector	Notes
	MC1-1		+VDCIN			2
	MC1-2		+VDCIN			2
	MC1-3		+VDCIN			2
	MC1-4		+VDCIN			2
	MC1-5		+VDCIN			2
	MC1-6		+VDCIN			2
	MC1-7		+VDCIN			2
	MC1-8		+VDCIN			2
	MC1-9	U11	TDI_0	J18-10		3
	MC1-10		IPMI_SDA			3
	MC1-11	T10	TDO_0	J18-8		3
	MC1-12		IPMI_SCL			3
	MC1-13	AA11	TCK_0	J18-6		3
	MC1-14	Y16	XADC_VP			3
	MC1-15	W11	TMS_0	J18-4		3
	MC1-16	AA15	XADC_VN			3
P40	MC1-17	P28	B48_L1P	J1-1	J1-31	4
P41	MC1-18	J26	B48_L14P	J1-2	J1-34 / J1-36	4, 9
P42	MC1-19	N28	B48_L1N	J1-3	J1-29	4
P43	MC1-20	H26	B48_L14N	J1-4	J1-36 / J1-34	4, 9
P44	MC1-21	C26	B48_L21P	J1-5	J1-27	4
P45	MC1-22	N24	B48_L8P	J1-6	J1-32	4
P46	MC1-23	C27	B48_L21N	J1-7	J1-25	4
P47	MC1-24	M24	B48_L8N	J1-8	J1-30	4
P48	MC1-25	T25	B48_L2P	J1-9	J1-23	4
P49	MC1-26	G26	B48_L18P	J1-10	J1-28	4
P50	MC1-27	R25	B48_L2N	J1-11	J1-21	4
P51	MC1-28	G27	B48_L18N	J1-12	J1-26	4
P52	MC1-29	F27	B48_L15P	J1-13	J1-17 / J1-19	4, 9

Table 2-4. Tessent SiliconInsight Pin Map for Opal Kelly XEM8350 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK8350MG Connector	BRK8350 Connector	Notes
P53	MC1-30	D25	B48_L22P	J1-14	J1-24	4
P54	MC1-31	E27	B48_L15N	J1-15	J1-19 / J1-17	4, 9
P55	MC1-32	D26	B48_L22N	J1-16	J1-22	4
P56	MC1-33	K26	B48_L12P	J1-17	J1-33 / J1-35	4, 9
P57	MC1-34	B26	B48_L23P	J1-18	J1-18 / J1-20	4, 9
P58	MC1-35	K27	B48_L12N	J1-19	J1-35 / J1-33	4, 9
P59	MC1-36	B27	B48_L23N	J1-20	J1-20 / J1-18	4, 9
P60	MC1-37	M27	B48_L9P	J1-21	J1-13 / J1-15	4, 9
P61	MC1-38	E25	B48_L20P	J1-22	J1-14 / J1-16	4, 9
P62	MC1-39	L27	B48_L9N	J1-23	J1-15 / J1-13	4, 9
P63	MC1-40	E26	B48_L20N	J1-24	J1-16 / J1-14	4, 9
P64	MC1-41	P26	B48_L6P	J1-25	J1-9 / J1-11	4, 9
P65	MC1-42	B25	B48_L24P	J1-26	J1-10 / J1-12	4, 9
P66	MC1-43	N26	B48_L6N	J1-27	J1-11 / J1-9	4, 9
P67	MC1-44	A25	B48_L24N	J1-28	J1-12 / J1-10	4, 9
P68	MC1-45	F28	B48_L17P	J1-29	J1-5 / J1-7	4, 9
P69	MC1-46	M25	B48_L7P	J1-30	J1-6 / J1-8	4, 9
P70	MC1-47	E28	B48_L17N	J1-31	J1-7 / J1-5	4, 9
P71	MC1-48	M26	B48_L7N	J1-32	J1-8 / J1-6	4, 9
P72	MC1-49	D28	B48_L19P	J1-33	MC1A-3	4
P73	MC1-50	G25	B48_L16P	J1-34	MC1A-5	4
P74	MC1-51	C28	B48_L19N	J1-35	MC1A-4	4
P75	MC1-52	F25	B48_L16N	J1-36	MC1A-6	4
P76	MC1-53	K28	B48_L11P	J1-37	MC1A-7	4
P77	MC1-54	P24	B48_L4P	J1-38	MC1A-9	4
P78	MC1-55	J28	B48_L11N	J1-39	MC1A-8	4
P79	MC1-56	P25	B48_L4N	J1-40	MC1A-10	4
P80	MC1-57	R26	B48_L3P	J3-1	MC1A-11	4
P81	MC1-58	L25	B48_L10P	J3-2	MC1A-13	4

Table 2-4. Tessent SiliconInsight Pin Map for Opal Kelly XEM8350 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK8350MG Connector	BRK8350 Connector	Notes
P82	MC1-59	R27	B48_L3N	J3-3	MC1A-12	4
P83	MC1-60	K25	B48_L10N	J3-4	MC1A-14	4
P84	MC1-61	A27	B48_T3U_N12	J3-5	MC1A-15	4
P85	MC1-62	J25	B48_T2U_N12	J3-6	MC1A-17	4
P86	MC1-63	L28	B48_T1U_N12	J3-7	MC1A-16	4
P87	MC1-64	N27	B48_T0U_N12	J3-8	MC1A-18	4
P88	MC1-65	H27	B48_L13P	J3-9	MC1A-19	4
P89	MC1-66	T28	B48_L5P	J3-10	MC1A-21	4
P90	MC1-67	H28	B48_L13N	J3-11	MC1A-20	4
P91	MC1-68	R28	B48_L5N	J3-12	MC1A-22	4
P92	MC1-69	G29	B47_L17P	J3-13	MC1A-23	4
P93	MC1-70	M29	B47_L5P	J3-14	MC1A-25	4
P94	MC1-71	F29	B47_L17N	J3-15	MC1A-24	4
P95	MC1-72	L29	B47_L5N	J3-16	MC1A-26	4
P96	MC1-73	J29	B47_L9P	J3-17	MC1A-27	4
P97	MC1-74	A28	B47_L23P	J3-18	MC1A-29	4
P98	MC1-75	H29	B47_L9N	J3-19	MC1A-28	4
P99	MC1-76	A29	B47_L23N	J3-20	MC1A-30	4
P100	MC1-77	P29	B47_L4P	J3-21	MC1A-31	4
P101	MC1-78	B30	B47_L19P	J3-22	MC1A-33	4
P102	MC1-79	N29	B47_L4N	J3-23	MC1A-32	4
P103	MC1-80	A30	B47_L19N	J3-24	MC1A-34	4
P104	MC1-81	K30	B47_T1U_N12	J3-25	MC1A-35	4
P105	MC1-82	M30	B47_L6P	J3-26	MC1A-37	4
P106	MC1-83	E30	B47_L18P	J3-27	MC1A-36	4
P107	MC1-84	L30	B47_L6N	J3-28	MC1A-38	4
P108	MC1-85	D30	B47_L18N	J3-29	MC1A-39	4
P109	MC1-86	B29	B47_T3U_N12	J3-30	MC1A-40	4
P110	MC1-87	C31	B47_L24P	J3-31	J2-34 / J2-36	4, 9

Table 2-4. Tessent SiliconInsight Pin Map for Opal Kelly XEM8350 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK8350MG Connector	BRK8350 Connector	Notes
P111	MC1-88	G30	B47_L15P	J3-32	J2-31	4
P112	MC1-89	B31	B47_L24N	J3-33	J2-36 / J2-34	4, 9
P113	MC1-90	F30	B47_L15N	J3-34	J2-29	4
P114	MC1-91	D29	B47_L21P	J3-35	J2-30	4
P115	MC1-92	J30	B47_L7P	J3-36	J2-27	4
P116	MC1-93	C29	B47_L21N	J3-37	J2-23	4
P117	MC1-94	J31	B47_L7N	J3-38	J2-25	4
P118	MC1-95	H32	B47_L12P	J3-39	J2-33 / J2-35	4, 9
P119	MC1-96	C32	B47_L20P	J3-40	J2-32	4
P120	MC1-97	G32	B47_L12N	J5-1	J2-35 / J2-33	4, 9
P121	MC1-98	C33	B47_L20N	J5-2	J2-28	4
P122	MC1-99	H31	B47_L11P	J5-3	J2-17 / J2-19	4, 9
P123	MC1-100	E31	B47_L16P	J5-4	J2-26	4
P124	MC1-101	G31	B47_L11N	J5-5	J2-19 / J2-17	4, 9
P125	MC1-102	D31	B47_L16N	J5-6	J2-24	4
P126	MC1-103	F33	B47_L13P	J5-7	J2-21	4
P127	MC1-104	R30	B47_L2P	J5-8	J2-13 / J2-15	4, 9
P128	MC1-105	E33	B47_L13N	J5-9	J2-22	4
P129	MC1-106	P30	B47_L2N	J5-10	J2-15 / J2-13	4, 9
P130	MC1-107	M31	B47_L1P	J5-11	J2-14 / J2-16	4, 9
P131	MC1-108	B32	B47_L22P	J5-12	J2-9 / J2-11	4, 9
P132	MC1-109	M32	B47_L1N	J5-13	J2-16 / J2-14	4, 9
P133	MC1-110	A32	B47_L22N	J5-14	J2-11 / J2-9	4, 9
P134	MC1-111	J33	B47_L8P	J5-15	J2-18 / J2-20	4, 9
P135	MC1-112	K31	B47_L10P	J5-16	J2-6 / J2-8	4, 9
P136	MC1-113	H33	B47_L8N	J5-17	J2-20 / J2-18	4, 9
P137	MC1-114	K32	B47_L10N	J5-18	J2-8 / J2-6	4, 9
P138	MC1-115	L32	B47_L3P	J5-19	J2-10 / J2-12	4, 9
P139	MC1-116	F32	B47_L14P	J5-20	J2-5 / J2-7	4, 9

Table 2-4. Tessent SiliconInsight Pin Map for Opal Kelly XEM8350 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK8350MG Connector	BRK8350 Connector	Notes
P140	MC1-117	L33	B47_L3N	J5-21	J2-12 / J2-10	4, 9
P141	MC1-118	E32	B47_L14N	J5-22	J2-7 / J2-5	4, 9
P142	MC1-119	D33	B47_T2U_N12	J5-23		4
P143	MC1-120	K33	B47_T0U_N12	J5-24		4
P0	MC1-121	E35	B46_L21P	J5-25	MC1B-1	4
P1	MC1-122	D34	B46_L23P	J5-26	MC1B-3	4
P2	MC1-123	D35	B46_L21N	J5-27	MC1B-2	4
P3	MC1-124	C34	B46_L23N	J5-28	MC1B-4	4
P4	MC1-125	B34	B46_L24P	J5-29	MC1B-5	4
P5	MC1-126	H34	B46_L9P	J5-30	MC1B-7	4
P6	MC1-127	A34	B46_L24N	J5-31	MC1B-6	4
P7	MC1-128	G34	B46_L9N	J5-32	MC1B-8	4
P8	MC1-129	B35	B46_L22P	J5-33	MC1B-9	4
P9	MC1-130	L35	B46_L3P	J5-34	MC1B-11	4
P10	MC1-131	A35	B46_L22N	J5-35	MC1B-10	4
P11	MC1-132	K35	B46_L3N	J5-36	MC1B-12	4
P12	MC1-133	E36	B46_L19P	J5-37	MC1B-13	4
P13	MC1-134	G35	B46_L7P	J5-38	MC1B-15	4
P14	MC1-135	D36	B46_L19N	J5-39	MC1B-14	4
P15	MC1-136	F35	B46_L7N	J5-40	MC1B-16	4
P16	MC1-137	F34	B46_T1U_N12	J7-1	MC1B-17	4
P17	MC1-138	A33	B46_T3U_N12	J7-2	MC1B-19	4
P18	MC1-139	K36	B46_L1P	J7-3	MC1B-18	4
P19	MC1-140	H36	B46_L11P	J7-4	MC1B-20	4
P20	MC1-141	J36	B46_L1N	J7-5	MC1B-21	4
P21	MC1-142	G36	B46_L11N	J7-6	MC1B-23	4
P22	MC1-143	C36	B46_L20P	J7-7	MC1B-22	4
P23	MC1-144	G37	B46_L12P	J7-8	MC1B-24	4
P24	MC1-145	B36	B46_L20N	J7-9	MC1B-25	4

Table 2-4. Tessent SiliconInsight Pin Map for Opal Kelly XEM8350 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK8350MG Connector	BRK8350 Connector	Notes
P25	MC1-146	F37	B46_L12N	J7-10	MC1B-27	4
P26	MC1-147	J34	B46_L5P	J7-11	MC1B-26	4
P27	MC1-148	C37	B46_L18P	J7-12	MC1B-28	4
P28	MC1-149	J35	B46_L5N	J7-13	MC1B-29	4
P29	MC1-150	B37	B46_L18N	J7-14	MC1B-31	4
P30	MC1-151	E37	B46_L14P	J7-15	MC1B-30	4
P31	MC1-152	L37	B46_L6P	J7-16	MC1B-32	4
P32	MC1-153	D38	B46_L14N	J7-17	MC1B-33	4
P33	MC1-154	K37	B46_L6N	J7-18	MC1B-35	4
P34	MC1-155	A37	B46_L17P	J7-19	MC1B-34	4
P35	MC1-156	F38	B46_L13P	J7-20	MC1B-36	4
P36	MC1-157	A38	B46_L17N	J7-21	MC1B-37	4
P37	MC1-158	E38	B46_L13N	J7-22	MC1B-39	4
P38	MC1-159	H39	B46_L8P	J7-23	MC1B-38	4
P39	MC1-160	L38	B46_L4P	J7-24	MC1B-40	4
P144	MC1-161	G39	B46_L8N	J7-25		4
P145	MC1-162	K38	B46_L4N	J7-26		4
P146	MC1-163	D39	B46_L15P	J7-27		4
P147	MC1-164	J38	B46_L2P	J7-28		4
P148	MC1-165	C39	B46_L15N	J7-29		4
P149	MC1-166	J39	B46_L2N	J7-30		4
P150	MC1-167	C38	B46_L16P	J7-31		4
P151	MC1-168	H37	B46_L10P	J7-32		4
P152	MC1-169	B39	B46_L16N	J7-33		4
P153	MC1-170	H38	B46_L10N	J7-34		4
P154	MC1-171	L39	B46_T0U_N12	J7-35		4
P155	MC1-172	F39	B46_T2U_N12	J7-36		4
	MC1-173		+3.3VDD	J7-37		2
	MC1-174		VIO1	J7-38		5

Table 2-4. Tessent SiliconInsight Pin Map for Opal Kelly XEM8350 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK8350MG Connector	BRK8350 Connector	Notes
	MC1-175		+3.3VDD	J7-37		2
	MC1-176		VIO1	J7-38		5
	MC1-177		+1.8VDD	J7-39		2
	MC1-178		VREF1	J7-40		5
	MC1-179		+1.8VDD	J7-39		2
	MC1-180		not connected			6
	MC2-0		DGND			1
	MC2-1		VREF2	J9-1		5
	MC2-2		VIO2	J9-2		5
P156	MC2-3	AV21	B44_L2P	J9-3	MC2A-1	4
	MC2-4		VIO2	J9-2		5
P157	MC2-5	AW21	B44_L2N	J9-5	MC2A-2	4
	MC2-6		VCCBATT	J9-4		5
P158	MC2-7	AN24	B44_L7P	J9-7	MC2A-3	4
P159	MC2-8	AV23	B44_L3P	J9-8	MC2A-5	4
P160	MC2-9	AP24	B44_L7N	J9-9	MC2A-4	4
P161	MC2-10	AW23	B44_L3N	J9-10	MC2A-6	4
P162	MC2-11	AN23	B44_L9P	J9-11	MC2A-7	4
P163	MC2-12	AR22	B44_L8P	J9-12	MC2A-9	4
P164	MC2-13	AP23	B44_L9N	J9-13	MC2A-8	4
P165	MC2-14	AR23	B44_L8N	J9-14	MC2A-10	4
P166	MC2-15	AU21	B44_L4P	J9-15	MC2A-11	4
P167	MC2-16	AT22	B44_L6P	J9-16	MC2A-13	4
P168	MC2-17	AV22	B44_L4N	J9-17	MC2A-12	4
P169	MC2-18	AU22	B44_L6N	J9-18	MC2A-14	4
P170	MC2-19	AP21	B44_L10P	J9-19	MC2A-15	4
P171	MC2-20	AT23	B44_L5P	J9-20	MC2A-17	4
P172	MC2-21	AR21	B44_L10N	J9-21	MC2A-16	4
P173	MC2-22	AT24	B44_L5N	J9-22	MC2A-18	4

Table 2-4. Tessent SiliconInsight Pin Map for Opal Kelly XEM8350 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK8350MG Connector	BRK8350 Connector	Notes
P174	MC2-23	AF24	B45_L22P	J9-23	J4-5	4
P175	MC2-24	AV24	B44_L1P	J9-24		4
P176	MC2-25	AG24	B45_L22N	J9-25	J4-7	4
P177	MC2-26	AW24	B44_L1N	J9-26		4
P178	MC2-27	AF25	B45_L20P	J9-27	J4-6	4
P179	MC2-28	AE27	B45_L23P	J9-28	J4-13	4
P180	MC2-29	AG25	B45_L20N	J9-29	J4-8	4
P181	MC2-30	AF27	B45_L23N	J9-30	J4-15	4
P182	MC2-31	AH24	B45_L18P	J9-31	J4-9	4
P183	MC2-32	AJ25	B45_L15P	J9-32	J4-17	4
P184	MC2-33	AJ24	B45_L18N	J9-33	J4-11	4
P185	MC2-34	AK25	B45_L15N	J9-34	J4-19	4
P186	MC2-35	AP26	B45_T1U_N12	J9-35	J4-22 / J4-10	4, 9
P187	MC2-36	AT25	B45_T0U_N12	J9-36	J4-24 / J4-14	4, 9
P188	MC2-37	AG26	B45_L24P	J9-37	J4-10 / J4-12	4, 9
P189	MC2-38	AV26	B45_L6P	J9-38	J4-14 / J4-16	4, 9
P190	MC2-39	AG27	B45_L24N	J9-39	J4-12 / J4-22	4, 9
P191	MC2-40	AV27	B45_L6N	J9-40	J4-16 / J4-18	4, 9
P192	MC2-41	AD26	B45_L21P	J11-1	J4-21	4
P193	MC2-42	AW25	B45_L2P	J11-2	J4-18 / J4-20	4, 9
P194	MC2-43	AE26	B45_L21N	J11-3	J4-28	4
P195	MC2-44	AW26	B45_L2N	J11-4	J4-20 / J4-24	4, 9
P196	MC2-45	AM26	B45_L12P	J11-5	J4-33	4
P197	MC2-46	AR26	B45_L7P	J11-6	J4-23	4
P198	MC2-47	AN26	B45_L12N	J11-7	J4-35	4
P199	MC2-48	AR27	B45_L7N	J11-8	J4-26	4
P200	MC2-49	AD25	B45_L19P	J11-9	J4-29	4
P201	MC2-50	AV28	B45_L1P	J11-10	J4-25	4
P202	MC2-51	AE25	B45_L19N	J11-11	J4-31	4

Table 2-4. Tessent SiliconInsight Pin Map for Opal Kelly XEM8350 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK8350MG Connector	BRK8350 Connector	Notes
P203	MC2-52	AW28	B45_L1N	J11-12	J4-30	4
P204	MC2-53	AP25	B45_L10P	J11-13	J4-27	4
P205	MC2-54	AH26	B45_L17P	J11-14	J4-34 / J4-32	4, 9
P206	MC2-55	AR25	B45_L10N	J11-15	J32-42	4
P207	MC2-56	AJ26	B45_L17N	J11-16	J4-36 / J4-34	4, 9
P208	MC2-57	AM24	B45_L8P	J11-17	J32-40	4
P209	MC2-58	AR28	B45_L5P	J11-18	J4-32 / J4-36	4, 9
P210	MC2-59	AM25	B45_L8N	J11-19	J32-38	4
P211	MC2-60	AT28	B45_L5N	J11-20	J32-10	4
P212	MC2-61	AL24	B45_L16P	J11-21	J32-50	4
P213	MC2-62	AN28	B45_L9P	J11-22	J32-44	4
P214	MC2-63	AL25	B45_L16N	J11-23	J32-52	4
P215	MC2-64	AP28	B45_L9N	J11-24	J32-69	4
P216	MC2-65	AH27	B45_T3U_N12	J11-25	J32-54	4
P217	MC2-66	AK26	B45_T2U_N12	J11-26	J32-68	4
P218	MC2-67	AU25	B45_L4P	J11-27	MC2A-19	4
P219	MC2-68	AL27	B45_L14P	J11-28	MC2A-21	4
P220	MC2-69	AU26	B45_L4N	J11-29	MC2A-20	4
P221	MC2-70	AL28	B45_L14N	J11-30	MC2A-22	4
P222	MC2-71	AM27	B45_L11P	J11-31	MC2A-23	4
P223	MC2-72	AK27	B45_L13P	J11-32	MC2A-25	4
P224	MC2-73	AN27	B45_L11N	J11-33	MC2A-24	4
P225	MC2-74	AK28	B45_L13N	J11-34	MC2A-26	4
P226	MC2-75	AV29	B24_L2P	J11-35	MC2A-27	4
P227	MC2-76	AT27	B45_L3P	J11-36	MC2A-29	4
P228	MC2-77	AW29	B24_L2N	J11-37	MC2A-28	4
P229	MC2-78	AU27	B45_L3N	J11-38	MC2A-30	4
P230	MC2-79	AP29	B24_L10P	J11-39	J5-6	4
P231	MC2-80	AL30	B24_L13P	J11-40	J5-9	4

Table 2-4. Tessent SiliconInsight Pin Map for Opal Kelly XEM8350 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK8350MG Connector	BRK8350 Connector	Notes
P232	MC2-81	AR30	B24_L10N	J13-1	J5-8	4
P233	MC2-82	AM30	B24_L13N	J13-2	J5-11	4
P234	MC2-83	AM31	B24_L12P	J13-3	J5-5	4
P235	MC2-84	AM32	B24_L11P	J13-4	J5-14	4
P236	MC2-85	AN31	B24_L12N	J13-5	J5-7	4
P237	MC2-86	AN32	B24_L11N	J13-6	J5-16	4
P238	MC2-87	AL29	B24_L14P	J13-7	J5-33	4
P239	MC2-88	AJ30	B24_L15P	J13-8	J5-13	4
P240	MC2-89	AM29	B24_L14N	J13-9	J5-35	4
P241	MC2-90	AK30	B24_L15N	J13-10	J5-15	4
P242	MC2-91	AU29	B24_L4P	J13-11	J5-10	4
P243	MC2-92	AN33	B24_L7P	J13-12	J5-17	4
P244	MC2-93	AU30	B24_L4N	J13-13	J5-12	4
P245	MC2-94	AP33	B24_L7N	J13-14	J5-19	4
P246	MC2-95	AT29	B24_L6P	J13-15	J5-18	4
P247	MC2-96	AW30	B24_L3P	J13-16	J5-22	4
P248	MC2-97	AT30	B24_L6N	J13-17	J5-20	4
P249	MC2-98	AW31	B24_L3N	J13-18	J5-21	4
P250	MC2-99	AP30	B24_L8P	J13-19	J5-26	4
P251	MC2-100	AJ33	B24_L18P	J13-20	J5-24	4
P252	MC2-101	AP31	B24_L8N	J13-21	J5-30	4
P253	MC2-102	AK33	B24_L18N	J13-22	J5-23	4
P254	MC2-103	AE28	B24_L24P	J13-23	J5-29	4
P255	MC2-104	AH28	B24_L22P	J13-24	J5-32	4
P256	MC2-105	AF28	B24_L24N	J13-25	J5-25	4
P257	MC2-106	AJ28	B24_L22N	J13-26	J5-28	4
P258	MC2-107	AR31	B24_L9P	J13-27	J5-27	4
P259	MC2-108	AF29	B24_L23P	J13-28	J5-34	4
P260	MC2-109	AR32	B24_L9N	J13-29	J5-31	4

Table 2-4. Tessent SiliconInsight Pin Map for Opal Kelly XEM8350 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK8350MG Connector	BRK8350 Connector	Notes
P261	MC2-110	AG29	B24_L23N	J13-30	J5-36	4
P262	MC2-111	AH29	B24_L19P	J13-31	MC2A-31	4
P263	MC2-112	AJ31	B24_L17P	J13-32	MC2A-33	4
P264	MC2-113	AJ29	B24_L19N	J13-33	MC2A-32	4
P265	MC2-114	AK31	B24_L17N	J13-34	MC2A-34	4
P266	MC2-115	AE30	B24_L21P	J13-35	MC2A-35	4
P267	MC2-116	AH31	B24_L20P	J13-36	MC2A-37	4
P268	MC2-117	AF30	B24_L21N	J13-37	MC2A-36	4
P269	MC2-118	AH32	B24_L20N	J13-38	MC2A-38	4
P270	MC2-119	AN29	B24_T1U_N12	J13-39	MC2A-39	4
P271	MC2-120	AT32	B24_T0U_N12	J13-40	MC2A-40	4
P272	MC2-121	AU31	B24_L5P	J15-1	J30-12	4
P273	MC2-122	AK32	B24_L16P	J15-2	J30-8	4
P274	MC2-123	AV31	B24_L5N	J15-3	J30-11	4
P275	MC2-124	AL32	B24_L16N	J15-4	J30-9	4
P276	MC2-125	AL33	B24_T2U_N12	J15-5	J30-31	4
P277	MC2-126	AU32	B24_L1P	J15-6	J30-27	4
P278	MC2-127	AN34	B25_L15P	J15-7	J31-28	4
P279	MC2-128	AV32	B24_L1N	J15-8	J31-31	4
P280	MC2-129	AP34	B25_L15N	J15-9	J30-28	4
P281	MC2-130	AG30	B24_T3U_N12	J15-10	J31-8	4
P282	MC2-131	AW33	B25_L1P	J15-11	J31-11	4
P283	MC2-132	AR33	B25_L5P	J15-12	J31-9	4
P284	MC2-133	AW34	B25_L1N	J15-13	J31-12	4
P285	MC2-134	AT33	B25_L5N	J15-14	J31-27	4
P286	MC2-135	AV33	B25_L2P	J15-15		4
P287	MC2-136	AT34	B25_L3P	J15-16	J8-21	4
P288	MC2-137	AV34	B25_L2N	J15-17	J8-14	4
P289	MC2-138	AU34	B25_L3N	J15-18	J8-19	4

Table 2-4. Tessent SiliconInsight Pin Map for Opal Kelly XEM8350 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK8350MG Connector	BRK8350 Connector	Notes
P290	MC2-139	AL34	B25_L21P	J15-19		4
P291	MC2-140	AT35	B25_L6P	J15-20	J8-17	4
P292	MC2-141	AL35	B25_L21N	J15-21		4
P293	MC2-142	AU35	B25_L6N	J15-22	J8-22	4
P294	MC2-143	AK35	B25_L23P	J15-23	J8-34 / J8-16	4, 9
P295	MC2-144	AM34	B25_L17P	J15-24		4
P296	MC2-145	AK36	B25_L23N	J15-25	J8-36 / J8-18	4, 9
P297	MC2-146	AM35	B25_L17N	J15-26		4
P298	MC2-147	AM36	B25_L19P	J15-27	J8-20	4
P299	MC2-148	AP36	B25_L13P	J15-28	J8-33	4
P300	MC2-149	AM37	B25_L19N	J15-29		4
P301	MC2-150	AR36	B25_L13N	J15-30	J8-35	4
P302	MC2-151	AP35	B25_T2U_N12	J15-31	J8-24	4
P303	MC2-152	AW35	B25_L4P	J15-32		4
P304	MC2-153	AU37	B25_L9P	J15-33	J8-23	4
P305	MC2-154	AW36	B25_L4N	J15-34	J8-16 / J8-34	4, 9
P306	MC2-155	AV37	B25_L9N	J15-35		4
P307	MC2-156	AR35	B25_T0U_N12	J15-36	J8-18 / J8-36	4, 9
P308	MC2-157	AV38	B25_L8P	J15-37		4
P309	MC2-158	AU36	B25_L7P	J15-38	J23-19	4
P310	MC2-159	AV39	B25_L8N	J15-39	J23-18	4
P311	MC2-160	AV36	B25_L7N	J15-40	J23-17	4
P312	MC2-161	AL37	B25_L20P	J17-1	J23-20	4
P313	MC2-162	AR37	B25_L11P	J17-2	J23-33	4
P314	MC2-163	AL38	B25_L20N	J17-3		4
P315	MC2-164	AT37	B25_L11N	J17-4	J23-35	4
P316	MC2-165	AN38	B25_L16P	J17-5		4
P317	MC2-166	AN36	B25_L14P	J17-6		4
P318	MC2-167	AP38	B25_L16N	J17-7		4

Table 2-4. Tessent SiliconInsight Pin Map for Opal Kelly XEM8350 (cont.)

SID Pin	HD Connector	FPGA Pin	Function	BRK8350MG Connector	BRK8350 Connector	Notes
P319	MC2-168	AN37	B25_L14N	J17-8		4
P320	MC2-169	AN39	B25_L18P	J17-9	J23-34	4
P321	MC2-170	AT39	B25_L10P	J17-10	J23-21	4
P322	MC2-171	AP39	B25_L18N	J17-11	J23-36	4
P323	MC2-172	AU39	B25_L10N	J17-12	J23-23	4
P324	MC2-173	AW38	B25_T1U_N12	J17-13	J23-22	4
P325	MC2-174	AL39	B25_L24P	J17-14		4
P326	MC2-175	AK37	B25_L22P	J17-15	J23-24	4
P327	MC2-176	AM39	B25_L24N	J17-16		4
P328	MC2-177	AK38	B25_L22N	J17-17		4
P329	MC2-178	AR38	B25_L12P	J17-18	J23-16	4
Trigger	MC2-179	AJ39	B25_T3U_N12	J17-19		7
Reserved	MC2-180	AT38	B25_L12N	J17-20	J23-14	8

Future Technology Devices International FT4232H Mini Module

The FT4232H Mini Module adaptor is produced by Future Technology Devices International (FTDI), Ltd. This adaptor supports up to 32 I/O pins divided into 4 channels of 8 pins.

Figure 2-7. FTDI FT4232H Mini Module Adaptor

For this adaptor's operating parameters, refer to the website at the following URL:

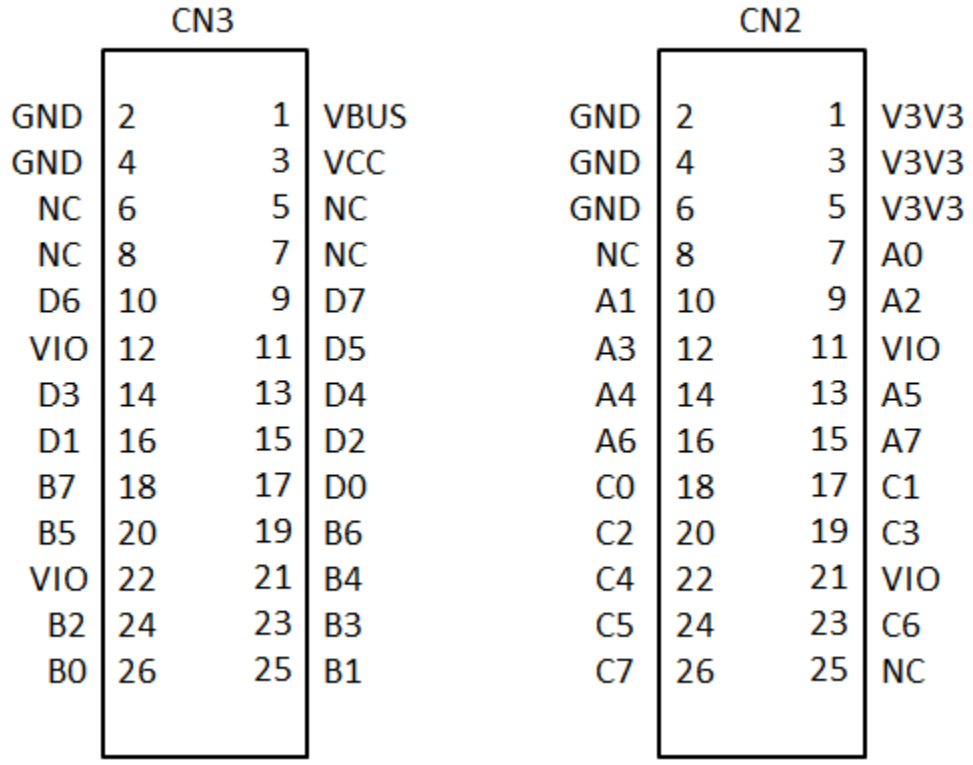
FT4232H-Mini-Module

Caution

You must connect VCC to VBUS and V3V3 to VIO before using the adaptor. Otherwise, the adaptor does not work as needed.

Tessent SiliconInsight Desktop can only simultaneously drive the pins on the same channel. In the FT4232H Mini Module adaptor, these channels are located at CN2 and CN3. The following figure shows the FT4232H Mini Module pin map as seen looking at the adaptor from the front of the columns of header pins when they are facing up.

Figure 2-8. FTDI FT4232H Mini Module Adaptor Pinout



Tessent SiliconInsight does not use the pin connections designated with “NC”. When using this adaptor ensure that the following is true:

- GND is grounded.
- VBUS connects to VCC.
- V3V3 pins are connected to VIO pins.

Olimex ARM-USB-OCD and ARM-USB-OCD-H Adaptors

Tessent SiliconInsight supports two adaptors manufactured by Olimex Ltd.—ARM-USB-OCD and ARM-USB-OCD-H.

Figure 2-9. Olimex ARM-USB-OCD and ARM-USB-OCD-H Adaptors

The only adaptor pins that you must connect to the performance board signals are listed in Table 2-5.

Table 2-5. Olimex Adaptor Connections

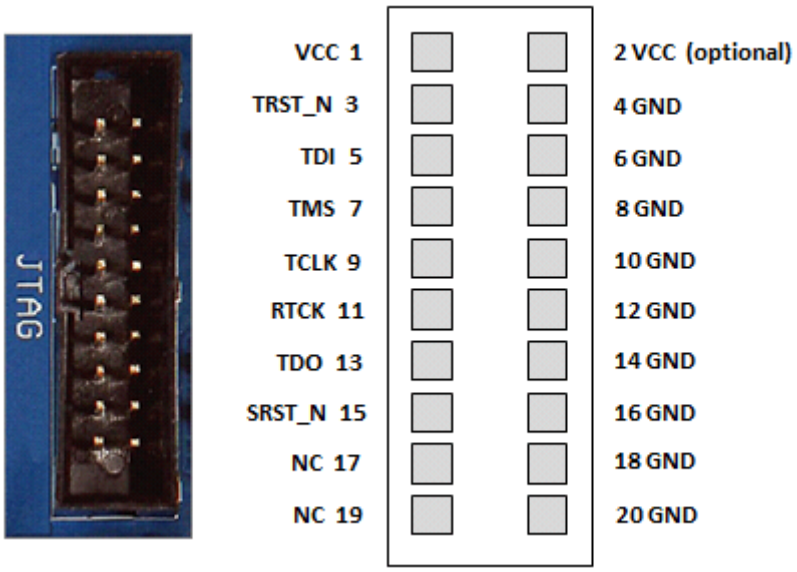
Pin Name	Pin Number	Description
TDI	5	Test Data Input (to DUT)
TMS	7	Test Mode Select (to DUT)
TCLK	9	Test Clock (to DUT)
TDO	13	Test Data Output (from DUT)
VCC	1	Voltage Reference
GND	20	Ground

Note

Ensure that all ground pins (4, 6, 8, 10, 12, 14 16, 18, 20) are grounded for good signal fidelity.

Both adaptors have the same pinout as shown in Figure 2-10.

Figure 2-10. Olimex Adaptor Pinout



Note

The Adaptor TRST_N (Test Reset) pin is not used. Tessent SiliconInsight Desktop performs a test reset using the 1149.1 standard's reset sequence. The DUT's TRST pin must be pulled up on the board to ensure that the device can operate in normal functional mode when the adaptor is not being used.

The adaptor's SRST_N pin is not used and need not be connected to the DUT.

For detailed information about these adaptors, refer to the Olimex Ltd. website at the following URL:

http://www.olimex.com

The devices are available through several distributors worldwide:

https://www.olimex.com/Distributors/

Tin Can Tools Flyswatter2 Adaptor

Tessent SiliconInsight supports the Tin Can Tools Flyswatter2 adaptor.

Figure 2-11. Tin Can Tools Flyswatter2 Adaptor

The pin connections and pinout for the Flyswatter2 adaptor are the same as that for the Olimex ARM-USB-OCD and ARM-USB-OCD-H adaptors. Refer to Table 2-5 and Figure 2-10 for details.

For detailed information about this adaptor, refer to the Tin Can Tools website at the following URL:

http://www.tincantools.com

Xverve SignalyzerSHA40 and SignalyzerH2/H4 Adaptors

The Signalyzer H2, Signalyzer H4, and Signalyzer SHA40 adaptors by Xverve Technologies, Inc., are no longer available. The following section is intended for existing Xverve Signalyzer users.

In place of the Signalyzer series adaptors, you can use the FT4232H Mini Module adaptor as described in "Future Technology Devices International FT4232H Mini Module" on page 57.

Figure 2-12. SignalyzerSHA40 and SignalyzerH2/H4 Adaptors

SignalyzerSHA40 and SignalyzerH4 support up to 32 I/O pins divided into 4 channels of 8 pins, whereas SignalyzerH2 supports up to 16 I/O pins divided into 2 channels of 8 pins. Because these adaptors are based on the USB 2.0 "high speed" protocol, they are at least five times faster than plain Signalyzer, which is based on the "full speed" version of USB 2.0.

Note

For RHEL7/8 machines, if the Signalyzer H4 or SHA40 USB exits unexpectedly after a reboot, you can log in as a superuser and run the following command: **/sbin/modprobe -r ftdi_sio**. Alternatively, you can "blacklist" the driver so that it does not load on reboot. As a superuser, add a new file called */etc/modprobe.d/blacklist.conf* that consists of one line: blacklist ftdi_sio

For the adaptors' operating parameters, refer to the Xverve Technologies website at the following URL:

`http://www.xverve.com`

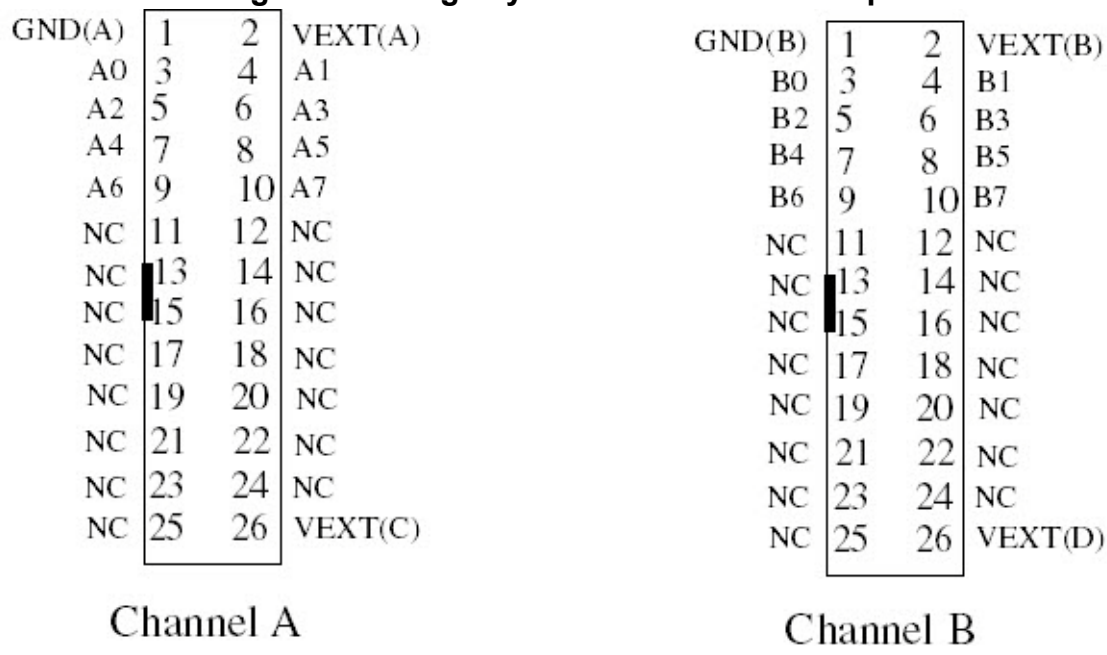
Caution

For the SignalyzerSHA40 and SignalyzerH2/H4 adaptors, Pin 2 (VEXT) and Pin 26 (VEXT) are 5.0V DC supply pins from the USB port of the computer. These pins, in contrast to older Signalyzer models, are not for VREF input and could damage the Signalyzer device if you use them as such.

SignalyzerH2 Pin Map

Tessent SiliconInsight Desktop can only simultaneously drive the pins on the same channel. In the SignalyzerH2, these channels are A and B.

Figure 2-13. Signalyzer H2 Channel Pin Map



SignalyzerSHA40 and SignalyzerH4 Pin Map

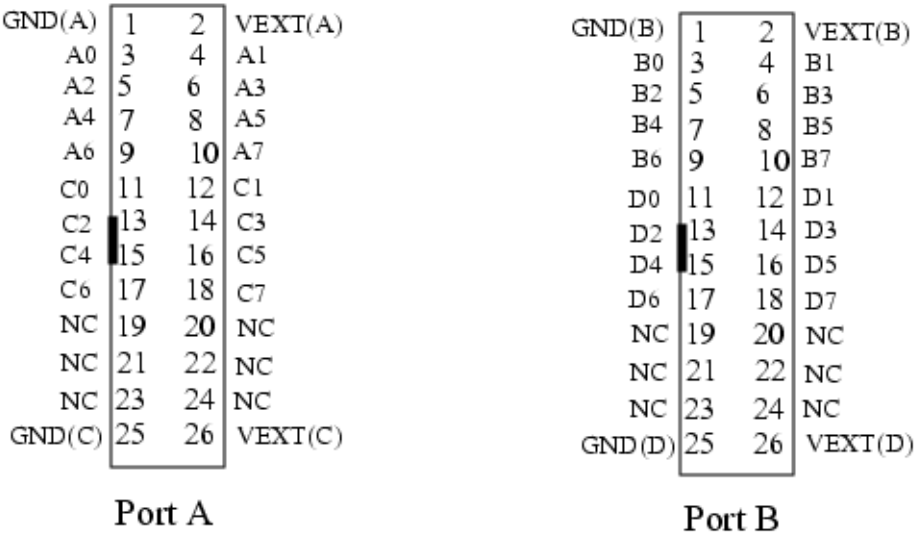
Tessent SiliconInsight Desktop can only simultaneously drive the pins on the same channel. In the Signalyzer SHA40 and SignalyzerH4 adaptors, these channels are A, B, C, and D.

Note

By default, the default setup specification uses Channel B because Channel A experiences occasional intermittent failures due to marginal DC/analog issues.

Figure 2-14 shows the Signalyzer SHA40 and SignalyzerH4 channel pin map.

Figure 2-14. Signalyzer SHA40 and SignalyzerH4 Channels Pin Map



Xverve Signalyzer Adaptor

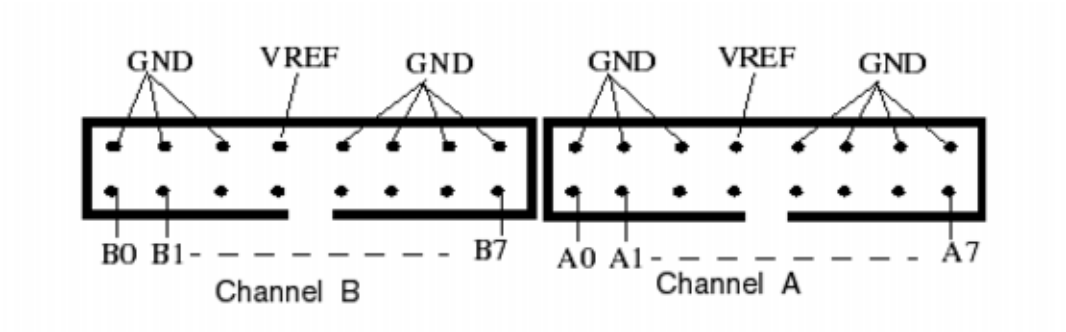
The Xverve Signalyzer adaptor produced by Xverve Technologies, Inc., is no longer available. The following section is intended for existing Xverve Signalyzer users.

In place of the Signalyzer series adaptors, you can use the FT4232H Mini Module adaptor as described in “Future Technology Devices International FT4232H Mini Module” on page 57.

Figure 2-15. Xverve Signalyzer

This Signalyzer supports up to 16 I/O pins divided into two channels, each supporting 8 I/O pins. Only one channel’s pins might be burst in a single pattern.

Figure 2-16. Xverve Signalyzer Channels Pinmap



For example, while Channel A might connect to 4 or 5 TAP pins (depending on whether TRST is contacted) and 3 or 4 other dynamic/static pins, Channel B might be used to control other user pins that are not used by the LVDB. Additionally, the Channel B pins might be used to toggle a user sequence to enable access for the Siemens EDA TAP controller, which can then be accessed through Channel A.

Note

For the best signal fidelity, ensure that all ground pins are grounded.

Xverve Signalyzer SP Adaptor

The Xverve Signalyzer SP adaptor by Xverve Technologies, Inc., is no longer available. The following section is intended for existing Xverve Signalyzer users.

In place of the Signalyzer series adaptors, you can use the FT4232H Mini Module adaptor as described in "Future Technology Devices International FT4232H Mini Module" on page 57.

The Signalyzer SP has 64 general-purpose I/O pins. All pins may be toggled at the same time. Each pin may be independently assigned to be input or output.

For Tessent SiliconInsight Desktop, the following pin-out guidelines apply:

- You can configure the Signalyzer SP hardware in two independent or one expanded GPIO port. Tessent SiliconInsight Desktop only uses the Signalyzer SP as one expanded 64-pin port. You can set voltage levels for each of the two ports independently.
- You only need to connect one of the Signalyzer SP GND pins on each Signalyzer SP port to ground (1, 11, 21, 31, 41, 49). These pins are connected together internally inside the Signalyzer SP.
- Pin 2 is a Signalyzer SP output that you can use to validate the I/O level, which is controlled by the invocation switch. Pin 2 is otherwise not used by Tessent SiliconInsight Desktop.
- Pins 12, 22, 32, and 42 are unused by Tessent SiliconInsight Desktop.

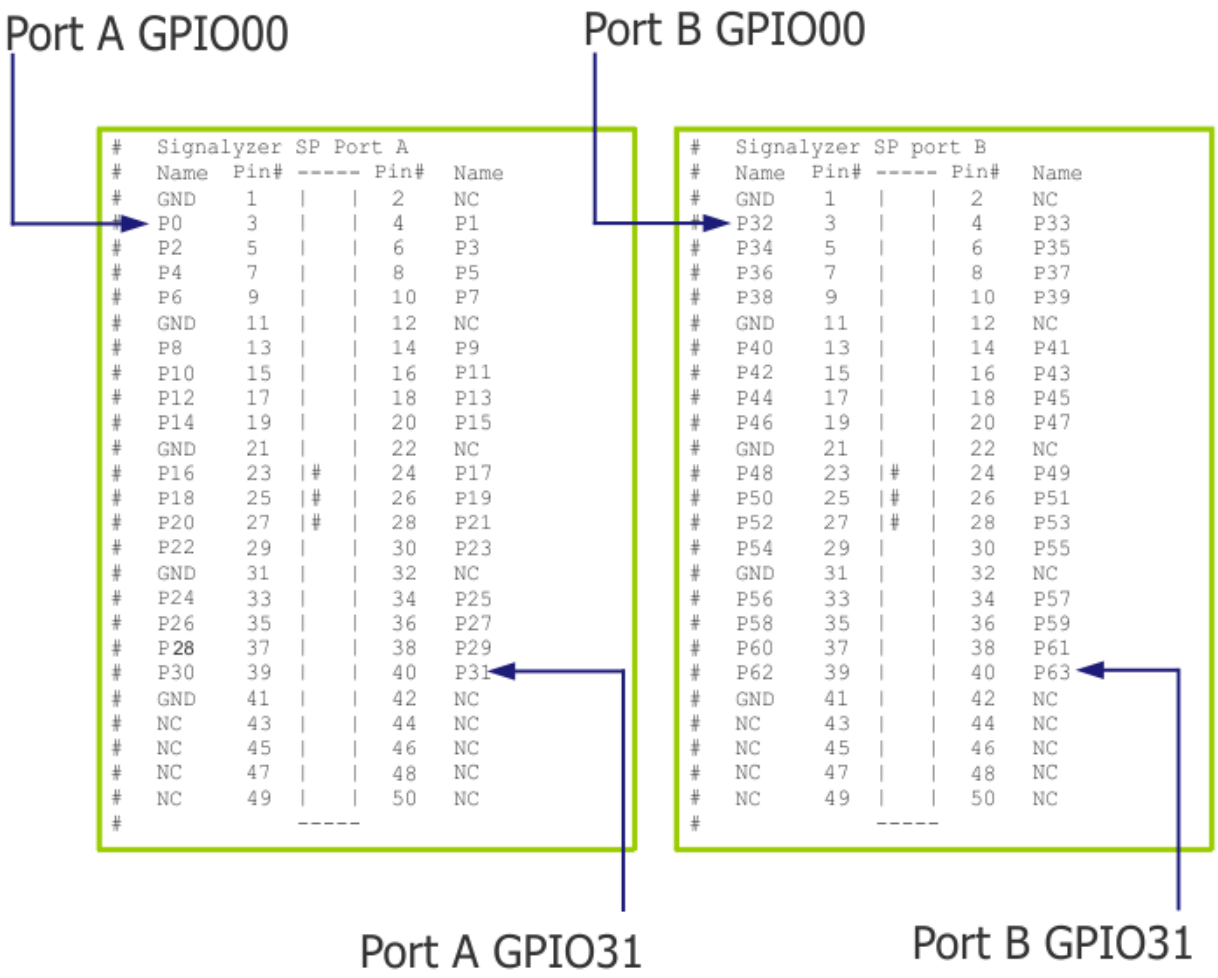
Note

For a complete pinout of the Signalyzer SP pins, see the manufacturer's website at www.xverve.com.

To support Signalyzer SP, the Tessent SiliconInsight software tree contains a hardware definition called *default.hwdef.signalyzerSP*. This file describes the names of the hardware channels that are specified in your pin map file for multi-site testing. See "Creating the Tessent SiliconInsight Pin Map File for Signalyzer SP" in the *Tessent SiliconInsight User's Manual for the LV Flow* manual for complete information.

Figure 2-17 shows the Signalyzer SP pinout for Tessent SiliconInsight Desktop as described in the *default.hwdef.signalyzerSP* file.

Figure 2-17. Hardware Definition File Signalyzer SP Pinout



This file is located in the Tessent SiliconInsight software tree at the following directory path:

Tessent_software_tree/lib/tools/etas_usb

For the adaptor's operating parameters, refer to the Xverve Technologies website at the following URL:

http://www.xverve.com

Verifying the Signalyzer SP Voltage Levels

Measure the GPIO pin voltages after you have connected the Signalyzer SP to the DUT. If the measured voltage for any GPIO pin is below the specified voltage, then the Signalyzer SP setting may not be correct.

All GPIO pins are referenced to two power pins: Port A pins are referenced to pin2 (VA) and port B pins are referenced to pin22 (VB). The power pins have reference voltages that you specify, usually between 1.2V and 3.3V.

Prerequisites

- Signalyzer SP is connected to the DUT.

Procedure

1. Generate a setup specification that contains the properties for the SignalyzerSP adaptor.

   The voltage_level_io_0_31 property specifies the reference voltage for pin2 (VA), and the voltage_level_io_32_63 property specifies the reference voltage for pin22 (VB). They can have different voltages.

   ```
   Sid(signalyzerSP) {
       cdp_directory : top.cdp;
       signalyzerSP {
         voltage_level_io_0_31 : 1.20v;
         voltage_level_io_32_63 : 1.80v;
         PinMap {
           P0 : trst;
           P1 : tdi;
           P2 : tms;
           P3 : tdo;
           P4 : tck;
           P5 : edt_bypass;
           P6 : edt_single_bypass_chain;
           P7 : edt_clock;
           P8 : refclk;
         }
       }
     }
   ```

   See “Sid(signalyzerSP)” on page 110 for details.

2. Invoke Tessent SiliconInsight and launch the SID tester process. For example:

   ```
   % tessent -shell

   SETUP>set_context patterns -silicon_insight
   SETUP>open_tsdb core1.tsdb
   SETUP>read_design top -design_id rtl -view interface
   SETUP>set_current_design top
   SETUP>read_config_data setup_spec_path
   SETUP>start_silicon_insight
   ```

Refer to "Performing Diagnosis and Interpreting the Results (TSDB Flow)" on page 133 for details.

3. Measure the voltage levels at pin2 and at pin22.

   The power pins should be functioning at the requested voltage levels as specified by the voltage_level_io_0_31 and voltage_level_io_32_63 properties, respectively. If not, disconnect the Signalyzer SP adaptor from the target device(s) and measure again. If the problem persists, contact the Signalyzer SP manufacturer.

4. Assuming that the power pin voltages are correct, verify the rest of the pins.

   Starting with the port A pins:

   a. Disconnect all the other GPIO pins, including the port B pins, from the DUT except for one port A pin.

   b. Measure the voltage at the one port A pin.

   c. If it works, then connect more port A pins to the DUT and measure the voltages.

   Repeat these steps for the port B pins.

5. For any pin that has a partial voltage, do the following:

   a. Measure the voltage between the problematic GPIO pin and other GPIO pins to verify that there are no shortages.

   b. Check whether the DUT board is configured correctly, especially if there is an interposer board placed on top of the DUT board.

   c. Check for any pull-ups or pull-downs for the associated pin on the DUT board.

   d. Check whether the design itself has a pull-up or a pull-down that connects to the problematic GPIO pin.

6. If all the GPIO pins have consistent partial voltage, reset the DUT board and the interposer board, if any.

   If the problems persists, try a new Signalyzer SP adaptor, and again, if the problem persists, contact the Signalyzer SP manufacturer.

Results

The measured voltage for all the GPIO pins should be below the preset voltage.

Amontec Adaptors

The Amontec adaptors are no longer available. This discussion is intended for existing users of the JTAGkey and JTAGkey-Tiny adaptors manufactured by Amontec. The JTAGkey adaptor has a few more features than the JTAGkey-Tiny, but these are not used by Tessent SiliconInsight Desktop. Therefore, you can use either adaptor.

The Olimex and Tin Can Tools adaptors described in "Olimex ARM-USB-OCD and ARM-USB-OCD-H Adaptors" on page 58 and "Tin Can Tools Flyswatter2 Adaptor" on page 60 offer comparable capabilities.

Figure 2-18. Amontec JTAG Adaptors

The pin connections and pinout for the Amontec adaptors are the same as those for the Olimex ARM-USB-OCD and ARM-USB-OCD-H adaptors. Refer to Table 2-5 and Figure 2-10 for details.

Prologix 6.0 Adaptor

The Tessent SiliconInsight Desktop GPIB option enables integration and usage of any GPIB-enabled instrument in a plug-and-play manner. Examples of these instruments include clock generators, power supplies, and parametric measurement units (PMUs).

For GPIB integration, Tessent SiliconInsight Desktop currently supports the Prologix 6.0 adaptor.

Figure 2-19. Prologix 6.0 Adaptor

You can purchase this adaptor from Prologix at the following URL:

www.prologix.biz

Using standard GPIB cables, you can daisy chain multiple GPIB instruments together. For example, you can connect 20 power supplies to Tessent SiliconInsight Desktop in serial.

Note

When you use multiple GPIB instruments through the GPIB bus, a “wait” statement (normally in Tcl scripts) is required so that Tessent SiliconInsight Desktop can correctly gain control of the GPIB instruments.

Red Hat Enterprise Linux Operating System Configuration

Tessent SiliconInsight Desktop supports Red Hat Enterprise Linux 7 and Red Hat Enterprise Linux 8.2+ operating systems.

You can configure the supported Red Hat Enterprise Linux operating systems to run Tessent SiliconInsight Desktop.

Configuring Red Hat Enterprise Linux 7 or 8 to Run Tessent SiliconInsight Desktop

To enable access of Tessent SiliconInsight Desktop for all users in addition to root, perform the configuration operations when logged on as root.

Prerequisites

- A Linux laptop or PC configured with the Red Hat Enterprise Linux (RHEL) 7 or 8 operating system.
- Your hardware configured as shown in Figure 2-1.

Procedure

Note

Opal Kelly adaptors require symbolic links with RHEL7 and RHEL8. Specify the link as follows:

```
ln -s /usr/lib64/libudev.so.0 /usr/lib64/libudev.so
```

1. Copy the file *<Tessent Shell Install Dir>/share/SiliconInsight/ATE/USB/Drivers/10-sid-rhel7.rules* to the */etc/udev/rules.d* directory.

   Note

   Although the filename includes “rhel7,” you can use this file for RHEL7 or RHEL8.

2. Run **udevadm control --reload-rules.**

   Note

   If you are using GPIB equipment, you do not need to perform extra setup steps.

Results

If you have completed the tasks in “Setting Up the Desktop Environment” on page 19, you can proceed to “Generating the Setup Specification for Desktop Mode (TSDB Flow)” on page 76.

Debug the Wiring Connection Between the Adaptor and the DUT

You may need to debug the wiring between the Tessent SiliconInsight Desktop adaptor and the DUT. The execute_tester_command provides options that enable you to set the logic values for adaptor pins and also to verify that you have wired the TAP pins correctly, if your DUT contains a TAP controller.

The command syntax is the following:

```
execute_tester_command [exec_vecs <adaptor_pin_list> <vector_list>]
        [find_ir_length <trst> <tms> <tdi> <tdo> <tck>]
```

Use the exec_vecs option to program any of the supported adaptors' pins to a logic value (0 or 1) to provide a baseline output voltage value. You can set any adaptor pin to any logic value. The length of each vector must be equal to the number of used adaptor pins.

Use the find_ir_length option to verify that the TAP controller is wired correctly. Specify the five adaptor pins that TAP pins are connected to in the order indicated by the command syntax: TRST, TMS, TDI, TDO, and TCK. Alternatively, you can specify find_ir_length with an uncontacted TRST pin, as follows:

```
execute_tester_command find_ir_length TMS TDI TDO TCK
```

If the command returns an error instead of the length of the instruction register, then the TAP pins are not wired correctly.

The following usage examples illustrate how to use the exec_vecs option. In the following example, the *./si_setup* file contains the setup specification with a description of the adaptor you are using. A complete pin map is not required, because the execute_tester_command command does not rely on design data.

```
set_context patterns -silicon_insight
read_config_data ./si.setup
# Can skip the next command if selected in ./si.setup
set_current_silicon_insight_setup Sid(opalKellyXem6310)
start_silicon_insight
execute_tester_command  exec_vecs  P0 1
```

The Flyswatter adaptor has four pins: TDI, TMS, TCK, and TDO. The following command sets the TCK pin to logic value 1:

```
execute_tester_command exec_vecs TCK 1
```

The following example sets the TDI and TMS pins to 0 and the TCK pin to 1.

```
execute_tester_command exec_vecs { TDI TMS TCK } 001
```

The following Tcl code segment wiggles the TMS pin 100 times:

```
for { set i 0 } { $i < 100 } { incr i } {
   execute_tester_command exec_vecs  TMS { 1 0 }
}
```

The following example for a signalyzerSHA40 adaptor sets pins C0 and C7 to 1 and the rest of the pins (C1, C2, C3, C4, C5, and C6) to 0:

```
execute_tester_command exec_vecs { C0 C1 C2 C3 C4 C5 C6 C7 }  10000001
```

The following example for an Opal Kelly adaptor sets pins P0 and P3 to logic 0 and pin P5 to logic 1:

```
execute_tester_command exec_vecs { P0 P3 P5 } 001
```

Chapter 3
Prepare the Design Under Test

After setting up the desktop environment, you must generate a SiliconInsight Setup Specification (setup specification) that represents the Tessent SiliconInsight mode you are using. You generate setup specifications once for each design.

Note

Ensure that you have not installed additional drivers on the PC or laptop to use with the Tessent SiliconInsight Desktop adaptors. Doing so may impair the Desktop setup. Configure your Desktop environment only as described in this section "Setting Up the Desktop Environment" on page 19.

The setup specification is a Tessent Shell wrapper, called SiliconInsightSetupSpecification, that contains nested wrappers and properties that describe one or more setups or modes of operation for the tool. Each setup contains the settings for all properties that control bring-up and usage of the Tessent SiliconInsight session.

The SiliconInsightSetupSpecification wrapper follows the Tessent Shell conventions for configuration-based specification syntax, as described in "Configuration-Based Specification" in the *Tessent Shell Reference Manual*.

Generating the Setup Specification for Desktop Mode (TSDB Flow)

Within the TSDB flow, the Desktop mode with automatic diagnosis capabilities applies to Tessent LogicBIST and Tessent MemoryBIST. The setup specification for Desktop functionality uses the Sid() wrapper, which includes wrappers for all of the supported adaptors and SimDUT mode.

Desktop mode also supports diagnosing and debugging Tessent IJTAG-inserted instruments, although Tessent SiliconInsight does not support automatic diagnosis for IJTAG instruments.

Prerequisites

- You have configured the Tessent SiliconInsight Desktop environment as described in "Prepare the Tessent SiliconInsight Desktop Environment" on page 19.
- For Tessent LogicBIST and Tessent MemoryBIST, you must have all TSDBs (core level and top level) for the design. For more information about TSDBs, refer to the *Tessent Shell Reference Manual*.

Procedure

1. From a shell, invoke Tessent Shell using the following syntax:

   ```
   % tessent -shell
   ```

   After invocation, the tool is in setup mode. Refer to the "tessent" command description in the *Tessent Shell Reference Manual* for additional invocation options.

2. Set the tool context to enable Tessent SiliconInsight within Tessent Shell:

   ```
   SETUP>set_context patterns -silicon_insight
   ```

3. Load the TSDBs. You must open the individual TSDBs for the cores and top level, set the top-level design, and then set the current design. For example:

   ```
   SETUP>open_tsdb core1.tsdb
   SETUP>open_tsdb core2.tsdb
   SETUP>open_tsdb top.tsdb
   SETUP>read_design top -design_id rtl -view interface
   SETUP>set_current_design top
   ```

   The tool loads the current design, the current design's top-level ICL, and the associated patterns and pattern specifications.

4. Generate the default setup specification for the Desktop mode you are using as follows:

   SETUP>create_silicon_insight_setup_specification -setup *mode_name*

   For information about the possible modes, see create_silicon_insight_setup_specification in the *Tessent Shell Reference Manual*.

   The tool generates a setup specification pre-populated with the correct selected_setup and default properties. Refer to "Default SiliconInsight Setup Specification Format" on page 83 for the format for the default setup specification for the supported modes of operation.

5. As needed, edit the default setup specification so that the specification represents your desktop environment.

   Refer to "SiliconInsightSetupSpecification" on page 92 for details about the various properties that are available for each of the adaptors.

   You can access and update the default setup specification in the following ways:

   - Open the SiliconInsightSetupSpec.cfg file in a text editor and read the modified file using the read_config_data command.
   - Use the GUI.
   - Run the set_current_silicon_insight_setup command as follows:

     set_current_silicon_insight_setup *setup_type*

     Refer to set_current_silicon_insight_setup in the *Tessent Shell Reference Manual* for a list of the setup types.

     Note

     The set_current_silicon_insight_setup command only affects the current session. If you want to save a change to the setup specification so that you can read it in using the read_config_data command in a subsequent session, you must use the write_config_data command.

   The following snippet of a setup specification indicates that you are using the Flyswatter 2 adaptor.

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(flyswatter2);
  Protocol {
    SvfTapPins {
      TRST : trst;
      TDI : tdi;
      TMS : tms;
      TDO : tdo;
      TCK : tck;
    }
  ...
```

6. As needed, customize the pin map for the specified adaptor to suit your configuration by adjusting the Sid()/PinMap wrapper.

   To use the modified pinmap, use the read_config_data command as described in "Performing Diagnosis and Interpreting the Results (TSDB Flow)" on page 133.

   For example, this is the default pin map for the flyswatter2 adaptor:

```
Sid(flyswatter2) {
    cdp_directory : top.cdp;
    flyswatter2 {
      PinMap {
        TDI : tdi;
        TMS : tms;
        TDO : tdo;
        TCK : tck;
      }
    }
  }
```

   Note

   Each adaptor supports an offline mode as indicated by the "_offline" suffix. Use this mode for testing that the patterns run well with the adaptor. For example:

```
Sid(signalyzerSHA40_offline) {
    cdp_directory : top.cdp;
    signalyzerSHA40{
      ...
```

7. If you want to launch the SID tester process on a remote machine, you can further modify the setup specification by including two optional properties within the Sid() wrapper, as shown in the following example. You may want to do this if you want to run Tessent Shell on a powerful machine (such as within a server farm) while only the SID tester process runs on the local tester that is connected to the DUT.

```
Sid(opalKellyXem6310) {
    cdp_directory : top.cdp;
    overwrite_cdp_on_startup : On;
    # Specify the name of the remote host machine to launch SID
    #  tester
    sid_tester_host_name : ststest5;
    # Specify the name of the remote shell you are using to launch
    # SID tester on the remote machine. The choices are: ssh and
    # rsh (default)
    remote_invocation_protocol : rsh
    opalKellyXem6310 {
      io_standard : 1.80;
      PinMap {
        P0 : trst;
        P1 : tdi;
        P2 : tms;
        P3 : tdo;
        P4 : tck;
        P5 : RST;
        P6 : reset;
        P7 : edt_clock;
      }
    }
  }
```

You can also use these two options for SimDUT.

Results

After you have a setup specification for a design, you can start running Tessent SiliconInsight.

For most situations, many setup specifications exist. During a session, you can verify which setup specification you are using by issuing the get_current_silicon_insight_setup command.

At any time, you can use a different setup specification by specifying the following command:

SETUP>set_silicon_insight_setup <*setup_name*>

Related Topics

PDL-Level Debugging for Tessent IJTAG-Inserted Devices

Running Non-Tessent IJTAG Instruments

Generating the Setup Specification for ATPG (Non-TSDB Flow) Desktop Mode

For ATPG, use the create_silicon_insight_setup_specification command to generate a setup specification. Generally, if you plan to run simulation, you create the setup data for SimDUT at the same time you create the setup specification for ATPG.

The SimDUT setup data includes:

- SIMDUT.v file: Wrapper that instantiates the top-level design.

- Design source dictionary: Information about the design files you have loaded.

The procedure outlined in this section generates a default setup specification for ATPG, plus the SimDUT setup data for simulating the ATPG results.

Prerequisites

- You have configured the Tessent SiliconInsight Desktop environment as described in "Prepare the Tessent SiliconInsight Desktop Environment" on page 19.

Procedure

1. From a shell, invoke Tessent Shell using the following syntax:

   ```
   % tessent -shell
   ```

   After invocation, the tool is in setup mode. Refer to the "tessent" command description in the *Tessent Shell Reference Manual* for additional invocation options.

2. Set the tool context to enable Tessent SiliconInsight within Tessent Shell:

   SETUP>set_context patterns -silicon_insight

3. (Optional) Read in the ATPG library. For example:

   SETUP>read_cell_library my_cell_library

   This is an optional but recommended step. Without the ATPG library, warnings about undefined modules for all the cells occur when you set the current design later in the process. You can ignore these warnings, but it could be difficult to separate valid warnings from ignorable warnings.

4. Read in the Verilog® netlists. For example:

   SETUP>read_verilog netlist1 netlist2 ...

   If you are only generating a setup specification, you only need to read in the top-level netlist. When you are creating the SimDUT setup files (alone or in addition to the setup specification), include all Verilog files required to run Verilog simulation. This includes Verilog files not needed for ATPG. All files imported here are included in the design source dictionary created later in the process.

5. Define the simulation library sources for SimDUT. For example:

   SETUP>set_simulation_library_sources -v my_verilog_simulation_library

6. Set the current design.

   SETUP>set_current_design

7. Generate the SimDUT.v file and accompanying SIMDUT.design_source_dictionary file.

   SETUP>run_testbench_simulation -simdut_server -generate_simdut_files_only

The -generate_simdut_files_only switch tells the tool to generate a directory that contains the SimDUT setup files. The tool returns the following message:

```
The SimDUT files were generated in directory ./simdut_outdir/
top_w_hier.simdut_gate/SIMDUT.simulation. To use this setup with
run_testbench_simulation, point to it using the flag -
simdut_output_directory.
```

8. Generate the default setup specification as follows:

   SETUP>create_silicon_insight_setup_specification

   Refer to "Default SiliconInsight Setup Specification Format" on page 83 for details about the default setup specification format.

9. As described in "Generating the Setup Specification for Desktop Mode (TSDB Flow)" on page 76, update the selected_setup property so that it represents your desktop environment and make any other adjustments to the setup specification that you need (such as to a specified adaptor's pinmap).

Examples

The following example dofile creates both the setup specification and SimDUT setup files:

```
set_context patterns -silicon
read_cell_library ../design/adk.atpg
read_verilog ../design/design_top.v
read_verilog ../design/design_modules.v
set_sim_lib_source -v ../design/adk.v
set_current_design
run_testbench_simulation -simdut_server -generate_simdut_files_only
create_silicon_insight_setup_specification
```

The following example creates only the setup specification. In this case, the specified Verilog netlist represents only the top level of the design:

```
set_context patterns -silicon
read_verilog ../design/design_top.v
set_current_design
create_silicon_insight_setup_specification
```

The following example creates only the setup files for SimDUT.

```
set_context patterns -silicon
read_cell_library ../design/adk.atpg
read_verilog ../design/design_top.v
read_verilog ../design/design_modules.v
set_sim_lib_source -v ../design/adk.v
set_current_design
run_testbench_simulation -simdut_server -generate_simdut_files_only
```

Setup Specifications for SimDUT and Offline ATE Modes

For simulation and offline ATE modes, ensure that the setup specification states “Sid(simdut)” or “No_Tester(batch_cdp)”, respectively, for the selected setup.

Refer to “Generating the Setup Specification for Desktop Mode (TSDB Flow)” on page 76 for the general process for creating the default setup specification and then editing it to reflect your desktop environment.

Simulation Mode

To run Tessent SiliconInsight in simulation mode, specify the following command:

create_silicon_insight_setup_specification -setup simdut

The tool generates a default setup specification for simulation mode that you can modify as needed, as described in “Sid(simdut)” on page 128.

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(simdut);
  Protocol {
    SvfTapPins {
      TRST : trst;
      TDI : tdi;
      TMS : tms;
      TDO : tdo;
      TCK : tck;
    }
  }
  Sid(simdut) {
    cdp_directory : top.cdp;
    SimDut {
    }
  }
```

See “Simulating Desktop, ATE, and ATPG Behavior” on page 223 for details about this mode.

Offline ATE Mode

In offline ATE mode, you are performing ATE-driven diagnosis without an integrated CDP. To run Tessent SiliconInsight in offline ATE mode, the default setup specification looks as follows.

```
SiliconInsightSetupSpecification {
  selected_setup : NoTester(batch_cdp);
  Protocol {
    SvfTapPins {
      TRST : trst;
      TDI : tdi;
      TMS : tms;
      TDO : tdo;
      TCK : tck;
    }
  }
  NoTester(batch_cdp) {
    cdp_directory : top.cdp;
  }
```

No_Tester(batch_cdp) is the default when you specify create_silicon_insight_setup_specification without the -setup option. Refer to "Performing Diagnosis From an ATE Test Program in Offline ATE Mode" on page 243 for details about this mode.

Default SiliconInsight Setup Specification Format

The format for the default setup specification for the various Tessent SiliconInsight modes of operation shows the default values for all the required properties for the supported setup modes. The value you specify for the selected_setup property tells Tessent SiliconInsight which setup you are using, and the tool automatically reads the applicable wrapper associated with the specified selected setup.

For example, if you specify "Sid(opalKellyXem6310)" for the selected setup, the tool knows to use this default setup:

```
Sid(opalKellyXem6310) {
   cdp_directory : top.cdp;
   opalKellyXem6310 {
     io_standard : 1.80;
     base_frequency : 10MHz;
     on_chip_termination : off
     PinMap {
       P0 : trst;
       P1 : tdi;
       P2 : tms;
       P3 : tdo;
       P4 : tck;
       P5 : edt_bypass;
       P6 : edt_single_bypass_chain;
       P7 : edt_clock;
       P8 : refclk;
     }
   }
}
```

Refer to the reference pages starting with "SiliconInsightSetupSpecification" on page 92 for details about the customizable fields, including optional properties that you can use.

```
SiliconInsightSetupSpecification {
  selected_setup : NoTester(batch_cdp);
  Protocol {
    SvfTapPins {
      TRST : trst;
      TDI : tdi;
      TMS : tms;
      TDO : tdo;
      TCK : tck;
    }
  }
  NoTester(batch_cdp) {
    cdp_directory : top.cdp;
  }
  Sid(jtagkey) {
    cdp_directory : top.cdp;
    jtagkey {
      PinMap {
        TDI : tdi;
        TMS : tms;
        TDO : tdo;
        TCK : tck;
      }
    }
  }
  Sid(jtagkey_offline) {
    cdp_directory : top.cdp;
    jtagkey {
      PinMap {
        TDI : tdi;
        TMS : tms;
        TDO : tdo;
        TCK : tck;
      }
      operation_mode : offline;
    }
  }
  Sid(flyswatter2) {
    cdp_directory : top.cdp;
    flyswatter2 {
      PinMap {
        TDI : tdi;
        TMS : tms;
        TDO : tdo;
        TCK : tck;
      }
    }
  }
  Sid(flyswatter2_offline) {
    cdp_directory : top.cdp;
    flyswatter2 {
      PinMap {
        TDI : tdi;
        TMS : tms;
        TDO : tdo;
        TCK : tck;
      }
      operation_mode : offline;
```

```
  }
}
Sid(olimex_arm_usb_ocd) {
  cdp_directory : top.cdp;
  olimex_arm_usb_ocd {
    PinMap {
      TDI : tdi;
      TMS : tms;
      TDO : tdo;
      TCK : tck;
    }
  }
}
Sid(olimex_arm_usb_ocd_offline) {
  cdp_directory : top.cdp;
  olimex_arm_usb_ocd {
    PinMap {
      TDI : tdi;
      TMS : tms;
      TDO : tdo;
      TCK : tck;
    }
    operation_mode : offline;
  }
}
Sid(olimex_arm_usb_ocd_h) {
  cdp_directory : top.cdp;
  olimex_arm_usb_ocd_h {
    PinMap {
      TDI : tdi;
      TMS : tms;
      TDO : tdo;
      TCK : tck;
    }
  }
}
Sid(olimex_arm_usb_ocd_h_offline) {
  cdp_directory : top.cdp;
  olimex_arm_usb_ocd_h {
    PinMap {
      TDI : tdi;
      TMS : tms;
      TDO : tdo;
      TCK : tck;
    }
    operation_mode : offline;
  }
}
Sid(signalyzer) {
  cdp_directory : top.cdp;
  signalyzer {
    PinMap {
      A0 : trst;
      A1 : tdi;
      A2 : tms;
      A3 : tdo;
      A4 : tck;
      A5 : edt_bypass;
```

```
      A6 : edt_single_bypass_chain;
      A7 : edt_clock;
    }
  }
}
Sid(signalyzer_offline) {
  cdp_directory : top.cdp;
  signalyzer {
    PinMap {
      A0 : trst;
      A1 : tdi;
      A2 : tms;
      A3 : tdo;
      A4 : tck;
      A5 : edt_bypass;
      A6 : edt_single_bypass_chain;
      A7 : edt_clock;
    }
    operation_mode : offline;
  }
}
Sid(signalyzerSHA40) {
  cdp_directory : top.cdp;
  signalyzerSHA40 {
    PinMap {
      B0 : trst;
      B1 : tdi;
      B2 : tms;
      B3 : tdo;
      B4 : tck;
      B5 : edt_bypass;
      B6 : edt_single_bypass_chain;
      B7 : edt_clock;
    }
  }
}
Sid(signalyzerSHA40_offline) {
  cdp_directory : top.cdp;
  signalyzerSHA40 {
    PinMap {
      B0 : trst;
      B1 : tdi;
      B2 : tms;
      B3 : tdo;
      B4 : tck;
      B5 : edt_bypass;
      B6 : edt_single_bypass_chain;
      B7 : edt_clock;
    }
    operation_mode : offline;
  }
}
Sid(signalyzerH2) {
  cdp_directory : top.cdp;
  signalyzerH2 {
    PinMap {
      B0 : trst;
      B1 : tdi;
```

```
        B2 : tms;
        B3 : tdo;
        B4 : tck;
        B5 : edt_bypass;
        B6 : edt_single_bypass_chain;
        B7 : edt_clock;
      }
    }
  }
  Sid(signalyzerH2_offline) {
    cdp_directory : top.cdp;
    signalyzerH2 {
      PinMap {
        B0 : trst;
        B1 : tdi;
        B2 : tms;
        B3 : tdo;
        B4 : tck;
        B5 : edt_bypass;
        B6 : edt_single_bypass_chain;
        B7 : edt_clock;
      }
      operation_mode : offline;
    }
  }
  Sid(signalyzerH4) {
    cdp_directory : top.cdp;
    signalyzerH4 {
      PinMap {
        B0 : trst;
        B1 : tdi;
        B2 : tms;
        B3 : tdo;
        B4 : tck;
        B5 : edt_bypass;
        B6 : edt_single_bypass_chain;
        B7 : edt_clock;
      }
    }
  }
  Sid(signalyzerH4_offline) {
    cdp_directory : top.cdp;
    signalyzerH4 {
      PinMap {
        B0 : trst;
        B1 : tdi;
        B2 : tms;
        B3 : tdo;
        B4 : tck;
        B5 : edt_bypass;
        B6 : edt_single_bypass_chain;
        B7 : edt_clock;
      }
      operation_mode : offline;
    }
  }
  Sid(signalyzerSP) {
    cdp_directory : top.cdp;
```

```
  signalyzerSP {
    // See Sid(signalyzerSP) for supported voltage levels
    voltage_level_io_0_31 : voltage;
    voltage_level_io_32_63 : voltage;
    PinMap {
      P0 : trst;
      P1 : tdi;
      P2 : tms;
      P3 : tdo;
      P4 : tck;
      P5 : edt_bypass;
      P6 : edt_single_bypass_chain;
      P7 : edt_clock;
      P8 : refclk;
    }
  }
}
Sid(signalyzerSP_offline) {
  cdp_directory : top.cdp;
  signalyzerSP {
    voltage_level_io_0_31 : 1.20v;
    voltage_level_io_32_63 : 1.80v;
    PinMap {
      P0 : trst;
      P1 : tdi;
      P2 : tms;
      P3 : tdo;
      P4 : tck;
      P5 : edt_bypass;
      P6 : edt_single_bypass_chain;
      P7 : edt_clock;
      P8 : refclk;
    }
    operation_mode : offline;
  }
}
Sid(opalKellyXem6310) {
  cdp_directory : top.cdp;
  opalKellyXem6310 {
    io_standard : 1.80;
    base_frequency : 10MHz;
    on_chip_termination : off
    PinMap {
      P0 : trst;
      P1 : tdi;
      P2 : tms;
      P3 : tdo;
      P4 : tck;
      P5 : edt_bypass;
      P6 : edt_single_bypass_chain;
      P7 : edt_clock;
      P8 : refclk;
    }
  }
}
Sid(opalKellyXem6310_offline) {
  cdp_directory : top.cdp;
  opalKellyXem6310 {
```

```
        io_standard : 1.80;
        base_frequency : 10MHz;
        on_chip_termination : off
        PinMap {
          P0 : trst;
          P1 : tdi;
          P2 : tms;
          P3 : tdo;
          P4 : tck;
          P5 : edt_bypass;
          P6 : edt_single_bypass_chain;
          P7 : edt_clock;
          P8 : refclk
        }
        operation_mode : offline;
      }
    }

  Sid(ftdi4232H_mini_module) {
      cdp_directory : top.cdp;
      ftdi4232H_mini_module {
        PinMap {
          B0 : trst;
          B1 : tdi;
          B2 : tms;
          B3 : tdo;
          B4 : tck;
          B5 : edt_bypass;
          B6 : edt_single_bypass_chain;
          B7 : edt_clock;
        }
      }
    }
    Sid(ftdi4232H_mini_module_offline) {
      cdp_directory : top.cdp;
      ftdi4232H_mini_module {
        PinMap {
          B0 : trst;
          B1 : tdi;
          B2 : tms;
          B3 : tdo;
          B4 : tck;
          B5 : edt_bypass;
          B6 : edt_single_bypass_chain;
          B7 : edt_clock;
        }
        operation_mode : offline;
```

```
    Sid(simdut) {
      cdp_directory : top.cdp;
      SimDut {
      }
    }
    Ate(ate1) {
      host_name : machine_1;
      port_number : 3211;
    }
     }
}
```

SiliconInsightSetupSpecification

Specifies the mode of operation for the current Tessent SiliconInsight session.

Usage

```
SiliconInsightSetupSpecification {
  selected_setup : setup_mode;              // Setup modes listed below
  Protocol {
    SvfTapPins {
      TRST : pin_name;                      // Default: tdi
      TDI  : pin_name;                      // Default: tdi
      TMS  : pin_name;                      // Default: tms
      TDO  : pin_name;                      // Default: tdo
      TCK  : pin_name;                      // Default: tck
    }
  }
// Choice of the following wrappers for selected_setup mode
  NoTester(batch_cdp) {                     // Default selected setup
  }
  Sid(jtagkey) {
  }
  Sid(flyswatter2) {
  }
  Sid(olimex_arm_usb_ocd) {
  }
  Sid(olimex_arm_usb_ocd_h) {               // See Sid(olimex_arm_usb_ocd)
  }
  Sid(signalyzer) {
  }
  Sid(signalyzerSHA40) {
  }
  Sid(signalyzerH2) {                       // See Sid(signalyzerSHA40)
  }
  Sid(signalyzerH4) {                       // See Sid(signalyzerSHA40)
  }
  Sid(signalyzerSP) {
  }
  Sid(opalKellyXem6310) {
  }
  Sid(opalKellyXem7310) {
  }
  Sid(opalKellyXem8350) {
  }
  Sid(simdut) {
  }
  Ate(tester_name) {
  }
}
```

Description

The SiliconInsight Setup Specification specifies the mode of operation for the current session and the top-level pin connections for the TAP controller.

Specifying the selected_setup property with a supported setup mode causes the tool to automatically use the default wrapper specific to that mode. You can customize the wrapper as needed.

By default, the tool names the setup specification "SiliconInsightSetupSpec.cfg," stores it in the current working directory, and populates the selected_setup field with "NoTester(batch_cdp)".

To save a change to the setup specification so that you can read it in using the read_config_data command in a subsequent session, you must use the write_config_data command.

When using adaptors in Desktop mode, the pin map specification included in the default setup specification depends on the adaptor you choose for the selected setup. In addition, each of the adaptors has an offline option, which you can use to verify that the test patterns run well with the specified adaptor.

Arguments

- selected_setup : *setup_mode*;

  A required property that specifies the mode of operation for the current session. The choices are:

  - NoTester(batch_cdp) — This is the default mode. In this mode, you are not connected to a tester and the tool only generates a CDP.
  - Sid(*desktop_setup*) — In Desktop mode, you can perform analyses on Tessent Shell-inserted LogicBIST and MemoryBIST instruments, ATPG patterns, and Tessent IJTAG-inserted instruments. This mode of operation also includes simulation (SimDUT).

    Deskop setups for supported adaptors include an offline mode.
  - Ate(*tester_name*) — In ATE mode, you can use ATE-Connect to debug Tessent IJTAG-inserted devices directly on an ATE.

- Protocol/SvfTapPins { *signals* ... }

  A required wrapper that specifies the pin connections/signals for the TAP controller.

NoTester(batch_cdp)

Generate a CDP that you can use for diagnostic purposes.

Usage

```
SiliconInsightSetupSpecification {
  selected_setup : NoTester(batch_cdp);
  Protocol {
    ...
    }
  }
  NoTester(batch_cdp) {
    cdp_directory : name;                    // Default: top.cdp
  }
}
```

Description

This is the default setup specification mode for operating Tessent SiliconInsight, in which the tool generates a CDP only.

You can use this mode to perform ATE-driven diagnosis on a CDP that is not integrated into the ATE software. Refer to "Performing Diagnosis From an ATE Test Program in Offline ATE Mode" for details.

Arguments

- cdp_directory : *name*;

  The name of the CDP directory. The default is top.cdp.

Examples

The following example shows the default setup specification when you specify the create_silicon_insight_setup_specification command.

```
SiliconInsightSetupSpecification {
  selected_setup : NoTester(batch_cdp);
  Protocol {
    SvfTapPins {
      TRST : trst;
      TDI : tdi;
      TMS : tms;
      TDO : tdo;
      TCK : tck;
    }
  }
  NoTester(batch_cdp) {
    cdp_directory : top.cdp;
  }
```

Sid(jtagkey)

Additional specification: Sid(ijtag_offline)

Usage

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(ijtag);
  Protocol {
    ...
    }
  }
  Sid(jtagkey) {
    cdp_directory : name;                    // Default: top.cdp
    overwrite_cdp_on_startup : on | off;     // Optional
    load_patterns_on_demand : on | off;      // Optional
    sid_tester_host_name : host_name;        // Optional
    remote_invocation_protocol : ssh | rsh; // Optional
    jtagkey {
      PinMap {
        TDI : pin_name;                      // Default: tdi
        TMS : pin_name;                      // Default: tms
        TDO : pin_name;                      // Default: tdo
        TCK : pin_name;                      // Default: tck
      }
      operation_mode : online | offline | simulation;    // Optional
      timeout        : integer;              // Default: auto
      startup_dofile : dofile_name;
    }
  }
}
```

Description

This is the selected setup and associated wrapper when you are performing analyses on MemoryBIST and LogicTest instruments and ATPG patterns using an Amontec JTAGkey or JTAGkey-Tiny adaptor as your Desktop connection to the performance board. Refer to “Setting Up the Desktop Environment” for details.

Arguments

- cdp_directory : *name*;

  The name of the CDP directory. The default is top.cdp.

- overwrite_cdp_on_startup : on | off;

  An optional property that you can add to the default setup specification if you want the tool to automatically overwrite the CDP. The default is off.

- load_patterns_on_demand : on | off;

  An optional property that you can add to the default setup specification if you want to postpone loading the patterns until right before running them. By default, the tool loads the patterns at the same time as it loads the CDP. Setting this property to on enables you to load only the patterns that you want to run, rather than the whole pattern set.

- sid_tester_host_name : *host_name*;

  An optional property that you can add to the default setup specification if you want to launch the SID tester process on a remote machine. The default host name is localhost. When using this property, you must specify an absolute pathname for the cdp_directory property. In addition, the paths must be to the same respective locations on the remote and local machines.

  When using a remote machine, you must set the TESSENT_TMP_LOCATION environment variable to a directory path that is also visible to both the remote and host machines. For more information about this variable, refer to get_scratch_directory in the *Tessent Shell Reference Manual*.

- remote_invocation_protocol: ssh | rsh;

  An optional property that you can add to the default setup specification when you are launching the SID tester process on a remote machine with the sid_tester_host_name property. The default is rsh.

- jtagkey/PinMap { *signals* ... }

  A required wrapper that specifies the pin connections/signals for this adaptor. Refer to "Amontec Adaptors" for details.

- jtagkey/operation_mode : online | offline | simulation;

  An optional property that you can set to "offline" when you are using Sid(ijtag_offline) as the selected setup, which you can use to verify that the patterns run correctly. You can also set it to "simulation" when you want to validate that the patterns pass when you run them using the adaptor.

- jtagkey/timeout *integer*

  An optional property that you use when you have set the operation mode to simulation. This property specifies the time in seconds for which Tessent SiliconInsight should wait for design elaboration in SimDUT mode. This property is useful for bigger designs when elaboration can take hours. You can set the timeout from 1 second to 65535 seconds. The default is 600 seconds.

- jtagkey/startup_dofile *dofile_name*

  An optional property that you use when you have set the operation mode to simulation. This property specifies the path to the startup_dofile that you use for SimDUT mode. This dofile contains the commands to set up the SimDUT simulation. Refer to "Sid(simdut)" on page 128 for more information.

Examples

The following example shows the default setup specification when you are using Sid(jtagkey_offline).

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(ijtag_offline);
  Protocol {
    SvfTapPins {
     TRST : trst;
     TDI  : tdi;
     TMS  : tms;
     TDO  : tdo;
     TCK  : tck;
    }
  }
  Sid(jtagkey_offline) {
    cdp_directory : top.cdp;
    jtagkey {
      PinMap {
        TDI : tdi;
        TMS : tms;
        TDO : tdo;
        TCK : tck;
      }
      operation_mode : offline;
    }
  }
}
```

Sid(flyswatter2)

Additional specification: Sid(flyswatter2_offline)

Usage

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(flyswatter2);
  Protocol {
    ...
    }
  }
  Sid(flyswatter2) {
    cdp_directory : name;                    // Default: top.cdp
    overwrite_cdp_on_startup : on | off;     // Optional
    load_patterns_on_demand : on | off;      // Optional
    sid_tester_host_name : host_name;        // Optional
    remote_invocation_protocol : ssh | rsh; // Optional
    flyswatter2 {
      PinMap {
        TDI : pin_name;                      // Default: tdi
        TMS : pin_name;                      // Default: tms
        TDO : pin_name;                      // Default: tdo
        TCK : pin_name;                      // Default: tck
      }
      operation_mode : online | offline | simulation;    // Optional
      timeout        : integer;              // Default: auto
      startup_dofile : dofile_name;
    }
  }
}
```

Description

This is the selected setup and associated wrapper when you are performing analyses on MemoryBIST and LogicTest instruments and ATPG patterns using a Tin Can Tools Flyswatter2 adaptor as your Desktop connection to the performance board. Refer to "Setting Up the Desktop Environment" for details.

Arguments

- cdp_directory : *name*;

  The name of the CDP directory. The default is top.cdp.

- overwrite_cdp_on_startup : on | off;

  An optional property that you can add to the default setup specification if you want the tool to automatically overwrite the CDP. The default is off.

- load_patterns_on_demand : on | off;

  An optional property that you can add to the default setup specification if you want to postpone loading the patterns until right before running them. By default, the tool loads the patterns at the same time as it loads the CDP. Setting this property to on enables you to load only the patterns that you want to run, rather than the whole pattern set.

- sid_tester_host_name : *host_name*;

  An optional property that you can add to the default setup specification if you want to launch the SID tester process on a remote machine. The default host name is localhost. When using this property, you must specify an absolute pathname for the cdp_directory property. In addition, the paths must be to the same respective locations on the remote and local machines.

  When using a remote machine, you must set the TESSENT_TMP_LOCATION environment variable to a directory path that is also visible to both the remote and host machines. For more information about this variable, refer to get_scratch_directory in the *Tessent Shell Reference Manual*.

- remote_invocation_protocol: ssh | rsh;

  An optional property that you can add to the default setup specification when you are launching the SID tester process on a remote machine with the sid_tester_host_name property. The default is rsh.

- flyswatter2/PinMap { *signals* ... }

  A required wrapper that specifies the pin connections/signals for this adaptor. Refer to "Tin Can Tools Flyswatter2 Adaptor" for details.

- flyswatter2/operation_mode : online | offline | simulation;

  An optional property that you set to "offline" when you are using Sid(flyswatter_offline) as the selected setup, which you can use to verify that the patterns run correctly. You can also set it to "simulation" when you want to validate that the patterns pass when you run them using the adaptor.

- flyswatter2/timeout *integer*;

  An optional property that you use when you have set the operation mode to simulation. This property specifies the time in seconds for which Tessent SiliconInsight should wait for design elaboration in SimDUT mode. This property is useful for bigger designs when elaboration can take hours. You can set the timeout from 1 second to 65535 seconds. The default is 600 seconds.

- flyswatter2/startup_dofile *dofile_name*;

  An optional property that you use when you have set the operation mode to simulation. This property specifies the path to the startup_dofile that you use for SimDUT mode. This dofile contains the commands to set up the SimDUT simulation. Refer to "Sid(simdut)" on page 128 for more information.

Examples

The following example shows the default setup specification when you are using Sid(flyswatter2_offline).

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(flyswatter2_offline);
  Protocol {
    SvfTapPins {
     TRST : trst;
     TDI  : tdi;
     TMS  : tms;
     TDO  : tdo;
     TCK  : tck;
    }
  }
  Sid(flyswatter2_offline) {
    cdp_directory : top.cdp;
    flyswatter2 {
      PinMap {
        TDI : tdi;
        TMS : tms;
        TDO : tdo;
        TCK : tck;
      }
      operation_mode : offline;
    }
  }
}
```

Sid(olimex_arm_usb_ocd)

Additional specifications: Sid(olimex_arm_usb_ocd_offline), Sid(olimex_arm_usb_ocd_h), Sid(olimex_arm_usb_ocd_h_offline)

Usage

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(olimex_arm_usb_ocd);
  Protocol {
    ...
    }
  }
  Sid(olimex_arm_usb_ocd) {
    cdp_directory : name;                    // Default: top.cdp
    overwrite_cdp_on_startup : on | off;     // Optional
    load_patterns_on_demand : on | off;      // Optional
    sid_tester_host_name : host_name;        // Optional
    remote_invocation_protocol : ssh | rsh; // Optional
    olimex_arm_usb_ocd {
      PinMap {
        TDI : pin_name;                      // Default: tdi
        TMS : pin_name;                      // Default: tms
        TDO : pin_name;                      // Default: tdo
        TCK : pin_name;                      // Default: tck
      }
      operation_mode : online | offline | simulation;    // Optional
      timeout        : integer;              // Default: auto
      startup_dofile : dofile_name;
    }
  }
}
```

Description

This is the selected setup and associated wrapper when you are performing analyses on MemoryBIST and LogicTest instruments and ATPG patterns using an Olimex ARM-USB-OCD (or Olimex ARM-USB-OCD-H) adaptor as your Desktop connection to the performance board. Refer to “Setting Up the Desktop Environment” for details.

The Olimex ARM-USB-OCD-H adaptor uses the same default pin map.

Arguments

- cdp_directory : *name*;

  The name of the CDP directory. The default is top.cdp.

- overwrite_cdp_on_startup : on | off;

  An optional property that you can add to the default setup specification if you want the tool to automatically overwrite the CDP. The default is off.

- load_patterns_on_demand : on | off;

  An optional property that you can add to the default setup specification if you want to postpone loading the patterns until right before running them. By default, the tool loads the patterns at the same time as it loads the CDP. Setting this property to on enables you to load only the patterns that you want to run, rather than the whole pattern set.

- sid_tester_host_name : *host_name*;

  An optional property that you can add to the default setup specification if you want to launch the SID tester process on a remote machine. The default host name is localhost. When using this property, you must specify an absolute pathname for the cdp_directory property. In addition, the paths must be to the same respective locations on the remote and local machines.

  When using a remote machine, you must set the TESSENT_TMP_LOCATION environment variable to a directory path that is also visible to both the remote and host machines. For more information about this variable, refer to get_scratch_directory in the *Tessent Shell Reference Manual*.

- remote_invocation_protocol: ssh | rsh;

  An optional property that you can add to the default setup specification when you are launching the SID tester process on a remote machine with the sid_tester_host_name property. The default is rsh.

- olimex_arm_usb_ocd/PinMap { *signals* ... }

  A required wrapper that specifies the pin connections/signals for this adaptor. Refer to "Olimex ARM-USB-OCD and ARM-USB-OCD-H Adaptors" for details.

- olimex_arm_usb_ocd/operation_mode : online | offline | simulation;

  An optional property that you set to "offline" when you are using Sid(olimex_arm_usb_ocd_offline) or Sid(olimex_arm_usb_ocd_h_offline) as the selected setup, which you can use to verify that the patterns run correctly. You can also set it to "simulation" when you want to validate that the patterns pass when you run them using the adaptor.

- olimex_arm_usb_ocd/timeout *integer*;

  An optional property that you use when you have set the operation mode to simulation. This property specifies the time in seconds for which Tessent SiliconInsight should wait for design elaboration in SimDUT mode. This property is useful for bigger designs when elaboration can take hours. You can set the timeout from 1 second to 65535 seconds. The default is 600 seconds.

- olimex_arm_usb_ocd/startup_dofile *dofile_name;*

  An optional property that you use when you have set the operation mode to simulation. This property specifies the path to the startup_dofile that you use for SimDUT mode. This dofile contains the commands to set up the SimDUT simulation. Refer to "Sid(simdut)" on page 128 for more information.

Examples

The following example shows the default setup specification when you are using Sid(olimex_arm_usb_ocd_offline).

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(olimex_arm_usb_ocd_offline);
  Protocol {
    SvfTapPins {
     TRST : trst;
     TDI  : tdi;
     TMS  : tms;
     TDO  : tdo;
     TCK  : tck;
    }
  }
  Sid(olimex_arm_usb_ocd_offline) {
    cdp_directory : top.cdp;
    olimex_arm_usb_ocd
      PinMap {
        TDI : tdi;
        TMS : tms;
        TDO : tdo;
        TCK : tck;
      }
      operation_mode : offline;
    }
  }
}
```

Sid(signalyzer)

Additional specification: Sid(signalyzer_offline)

Usage

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(signalyzer);
  Protocol {
    ...
    }
  }
  Sid(signalyzer) {
    cdp_directory : name;                    // Default: top.cdp
    overwrite_cdp_on_startup : on | off;     // Optional
    load_patterns_on_demand : on | off;    // Optional
    sid_tester_host_name : host_name;        // Optional
    remote_invocation_protocol : ssh | rsh; // Optional
    signalyzer {
      PinMap {
        A0 : pin_name;                       // Default: trst
        A1 : pin_name;                       // Default: tdi
        A2 : pin_name;                       // Default: tms
        A3 : pin_name;                       // Default: tdo
        A4 : pin_name;                       // Default: tck
        A5 : pin_name;                       // Default: edt_bypass
        A6 : pin_name;                 // Default: edt_single_bypass_chain
        A7 : pin_name;                       // Default: edt_clock
      }
      operation_mode : online | offline | simulation;    // Optional
      timeout        : integer;              // Default: auto
      startup_dofile : dofile_name;
    }
  }
}
```

Description

This is the selected setup and associated wrapper when you are performing analyses on MemoryBIST and LogicTest instruments and ATPG patterns using an Xverve Signalyzer adaptor as your Desktop connection to the performance board. Refer to “Setting Up the Desktop Environment” for details.

Arguments

- cdp_directory : *name*;

  The name of the CDP directory. The default is top.cdp.

- overwrite_cdp_on_startup : on | off;

  An optional property that you can add to the default setup specification if you want the tool to automatically overwrite the CDP. The default is off.

- load_patterns_on_demand : on | off;

  An optional property that you can add to the default setup specification if you want to postpone loading the patterns until right before running them. By default, the tool loads the patterns at the same time as it loads the CDP. Setting this property to on enables you to load only the patterns that you want to run, rather than the whole pattern set.

- sid_tester_host_name : *host_name*;

  An optional property that you can add to the default setup specification if you want to launch the SID tester process on a remote machine. The default host name is localhost. When using this property, you must specify an absolute pathname for the cdp_directory property. In addition, the paths must be to the same respective locations on the remote and local machines.

  When using a remote machine, you must set the TESSENT_TMP_LOCATION environment variable to a directory path that is also visible to both the remote and host machines. For more information about this variable, refer to get_scratch_directory in the *Tessent Shell Reference Manual*.

- remote_invocation_protocol: ssh | rsh;

  An optional property that you can add to the default setup specification when you are launching the SID tester process on a remote machine with the sid_tester_host_name property. The default is rsh.

- signalyzer/PinMap { *signals* ... }

  A required wrapper that specifies the pin connections/signals for this adaptor. Refer to "Xverve Signalyzer Adaptor" for details.

- signalyzer/operation_mode : online | offline | simulation;

  An optional property that you set to "offline" when you are using Sid(signalyzer_offline) as the selected setup, which you can use to verify that the patterns run correctly. You can also set it to "simulation" when you want to validate that the patterns pass when you run them using the adaptor.

- signalyzer/timeout *integer*;

  An optional property that you use when you have set the operation mode to simulation. This property specifies the time in seconds for which Tessent SiliconInsight should wait for design elaboration in SimDUT mode. This property is useful for bigger designs when elaboration can take hours. You can set the timeout from 1 second to 65535 seconds. The default is 600 seconds.

- signalyzer/startup_dofile *dofile_name;*

  An optional property that you use when you have set the operation mode to simulation. This property specifies the path to the startup_dofile that you use for SimDUT mode. This dofile contains the commands to set up the SimDUT simulation. Refer to "Sid(simdut)" on page 128 for more information.

Examples

The following example shows the default setup specification when you are using Sid(signalyzer_offline).

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(signalyzer_offline);
  Protocol {
    SvfTapPins {
     TRST : trst;
     TDI  : tdi;
     TMS  : tms;
     TDO  : tdo;
     TCK  : tck;
    }
  }
  Sid(signalyzer_offline) {
    cdp_directory : top.cdp;
    signalyzer {
      PinMap {
        A0 : trst;
        A1 : tdi;
        A2 : tms;
        A3 : tdo;
        A4 : tck;
        A5 : edt_bypass;
        A6 : edt_single_bypass_chain;
        A7 : edt_clock;
       }
      operation_mode : offline;
    }
  }
}
```

Sid(signalyzerSHA40)

Additional specifications: Sid(signalyzerSHA40_offline), Sid(signalyzerH2), Sid(signalyzerH2_offline), Sid(signalyzerH4), and Sid(signalyzerH4_offline)

Usage

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(signalyzerSHA40);
  Protocol {
    ...
    }
  }
  Sid(signalyzerSHA40) {
    cdp_directory : name;                    // Default: top.cdp
    overwrite_cdp_on_startup : on | off;     // Optional
    load_patterns_on_demand : on | off;      // Optional
    sid_tester_host_name : host_name;        // Optional
    remote_invocation_protocol : ssh | rsh; // Optional
    signalyzerSHA40 {
      PinMap {
        B0 : pin_name;                       // Default: trst
        B1 : pin_name;                       // Default: tdi;
        B2 : pin_name;                       // Default: tms;
        B3 : pin_name;                       // Default: tdo;
        B4 : pin_name;                       // Default: tck;
        B5 : pin_name;                       // Default: edt_bypass;
        B6 : pin_name;                 // Default: edt_single_bypass_chain;
        B7 : pin_name;                       // Default: edt_clock;
      }
      operation_mode : online | offline | simulation;    // Optional
      timeout        : integer;              // Default: auto
      startup_dofile : dofile_name;
    }
  }
}
```

Description

This is the selected setup and associated wrapper when you are performing analyses on MemoryBIST and LogicTest instruments and ATPG patterns using an Xverve SignalyzerSHA40 (or SignalyzerH2 or SignalyzerH4) adaptor as your Desktop connection to the performance board. Refer to “Setting Up the Desktop Environment” for details.

The Xverve SignalyzerH2 and SignalyzerH4 adaptors use the same default pin map.

Arguments

- cdp_directory : *name*;

  The name of the CDP directory. The default is top.cdp.

- overwrite_cdp_on_startup : on | off;

  An optional property that you can add to the default setup specification if you want the tool to automatically overwrite the CDP. The default is off.

- load_patterns_on_demand : on | off;

  An optional property that you can add to the default setup specification if you want to postpone loading the patterns until right before running them. By default, the tool loads the patterns at the same time as it loads the CDP. Setting this property to on enables you to load only the patterns that you want to run, rather than the whole pattern set.

- sid_tester_host_name : *host_name*;

  An optional property that you can add to the default setup specification if you want to launch the SID tester process on a remote machine. The default host name is localhost. When using this property, you must specify an absolute pathname for the cdp_directory property. In addition, the paths must be to the same respective locations on the remote and local machines.

  When using a remote machine, you must set the TESSENT_TMP_LOCATION environment variable to a directory path that is also visible to both the remote and host machines. For more information about this variable, refer to get_scratch_directory in the *Tessent Shell Reference Manual*.

- remote_invocation_protocol: ssh | rsh;

  An optional property that you can add to the default setup specification when you are launching the SID tester process on a remote machine with the sid_tester_host_name property. The default is rsh.

- signalyzerSHA40/PinMap { *signals* ... }

  A required wrapper that specifies the pin connections/signals for this adaptor. Refer to "Xverve SignalyzerSHA40 and SignalyzerH2/H4 Adaptors" for details.

- signalyzerSHA40/operation_mode : online | offline | simulation;

  An optional property that you set to "offline" when you are using Sid(signalyzerSHA40_offline), Sid(signalyzerH2_offline), or Sid(signalyzerH4_offline) as the selected setup, which you can use to verify that the patterns run correctly. You can also set it to "simulation" when you want to validate that the patterns pass when you run them using the adaptor.

- signalyzerSHA40/timeout *integer;*

  An optional property that you use when you have set the operation mode to simulation. This property specifies the time in seconds for which Tessent SiliconInsight should wait for design elaboration in SimDUT mode. This property is useful for bigger designs when elaboration can take hours. You can set the timeout from 1 second to 65535 seconds. The default is 600 seconds.

- signalyzerSHA40/startup_dofile *dofile_name;*

  An optional property that you use when you have set the operation mode to simulation. This property specifies the path to the startup_dofile that you use for SimDUT mode. This dofile contains the commands to set up the SimDUT simulation. Refer to "Sid(simdut)" on page 128 for more information.

Examples

The following example shows the default setup specification when you are using Sid(signalyzerSHA40_offline).

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(signalyzerSHA40_offline);
  Protocol {
    SvfTapPins {
     TRST : trst;
     TDI  : tdi;
     TMS  : tms;
     TDO  : tdo;
     TCK  : tck;
    }
  }
  Sid(signalyzerSHA40_offline) {
    cdp_directory : top.cdp;
    signalyzerSHA40 {
      PinMap {
        B0 : trst;
        B1 : tdi;
        B2 : tms;
        B3 : tdo;
        B4 : tck;
        B5 : edt_bypass;
        B6 : edt_single_bypass_chain;
        B7 : edt_clock;
       }
       operation_mode : offline;
    }
  }
}
```

Sid(signalyzerSP)

Additional specifications: Sid(signalyzerSP_offline)

Usage

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(signalyzerSP);
  Protocol {
    ...
    }
  }
  Sid(signalyzerSP) {
    cdp_directory : name;                    // Default: top.cdp
    overwrite_cdp_on_startup : on | off;     // Optional
    load_patterns_on_demand : on | off;      // Optional
    sid_tester_host_name : host_name;        // Optional
    remote_invocation_protocol : ssh | rsh; // Optional
    signalyzerSP {
      voltage_level_io_0_31 : voltage;
      voltage_level_io_32_63 : voltage;
      PinMap {
        P0 : pin_name;                       // Default: trst;
        P1 : pin_name;                       // Default: tdi;
        P2 : pin_name;                       // Default: tms;
        P3 : pin_name;                       // Default: tdo;
        P4 : pin_name;                       // Default: tck;
        P5 : pin_name;                       // Default: edt_bypass;
        P6 : pin_name;                 // Default: edt_single_bypass_chain;
        P7 : pin_name;                       // Default: edt_clock;
        P8 : pin_name;                       // Default: refclk;
      }
      operation_mode : online | offline | simulation;    // Optional
      timeout        : integer;              // Default: auto
      startup_dofile : dofile_name;
    }
  }
}
```

Description

This is the selected setup and associated wrapper when you are performing analyses on MemoryBIST and LogicTest instruments and ATPG patterns using an Xverve SignalyzerSP adaptor as your Desktop connection to the performance board. Refer to “Setting Up the Desktop Environment” for details.

Refer to “Xverve Signalyzer SP Adaptor” for pinout guidelines. This adaptor enables you to define the reference voltages at two power pins with two different voltages.

Arguments

- cdp_directory : *name*;

  The name of the CDP directory. The default is top.cdp.

- overwrite_cdp_on_startup : on | off;

  An optional property that you can add to the default setup specification if you want the tool to automatically overwrite the CDP. The default is off.

- load_patterns_on_demand : on | off;

  An optional property that you can add to the default setup specification if you want to postpone loading the patterns until right before running them. By default, the tool loads the patterns at the same time as it loads the CDP. Setting this property to on enables you to load only the patterns that you want to run, rather than the whole pattern set.

- sid_tester_host_name : *host_name*;

  An optional property that you can add to the default setup specification if you want to launch the SID tester process on a remote machine. The default host name is localhost. When using this property, you must specify an absolute pathname for the cdp_directory property. In addition, the paths must be to the same respective locations on the remote and local machines.

  When using a remote machine, you must set the TESSENT_TMP_LOCATION environment variable to a directory path that is also visible to both the remote and host machines. For more information about this variable, refer to get_scratch_directory in the *Tessent Shell Reference Manual*.

- remote_invocation_protocol: ssh | rsh;

  An optional property that you can add to the default setup specification when you are launching the SID tester process on a remote machine with the sid_tester_host_name property. The default is rsh.

- signalyzerSP/voltage_level_io_0_31 : *voltage;*

  A required property that sets the voltage for the first port of 32 pins. Supported values are: 1.20v, 1.25v, 1.30v, 1.35v, 1.40v, 1.45v, 1.50v, 1.60v, 1.65v, 1.70v, 1.80v, 1.90v, 2.50v, 2.80v, 3.00v, and 3.30v.

- signalyzerSP/voltage_level_io_32_63 : *voltage;*

  A required property that sets the voltage for the second port of 32 pins. Supported values are: 1.20v, 1.25v, 1.30v, 1.35v, 1.40v, 1.45v, 1.50v, 1.60v, 1.65v, 1.70v, 1.80v, 1.90v, 2.50v, 2.80v, 3.00v, and 3.30v.

- signalyzerSP/PinMap { *signals* ... }

  A required wrapper that specifies the pin connections/signals for this adaptor. Refer to "Xverve Signalyzer SP Adaptor" for details.

- signalyzerSP/operation_mode : online | offline | simulation;

  An optional property that you set to "offline" when you are using Sid(signalyzerSP_offline) as the selected setup, which you can use to verify that the patterns run correctly. You can also set it to "simulation" when you want to validate that the patterns pass when you run them using the adaptor.

- signalyzerSP/timeout *integer*;

  An optional property that you use when you have set the operation mode to simulation. This property specifies the time in seconds for which Tessent SiliconInsight should wait for design elaboration in SimDUT mode. This property is useful for bigger designs when elaboration can take hours. You can set the timeout from 1 second to 65535 seconds. The default is 600 seconds.

- signalyzerSP/startup_dofile *dofile_name;*

  An optional property that you use when you have set the operation mode to simulation. This property specifies the path to the startup_dofile that you use for SimDUT mode. This dofile contains the commands to set up the SimDUT simulation. Refer to "Sid(simdut)" on page 128 for more information.

Examples

The following example shows the default setup specification when you are using Sid(signalyzerSP_offline).

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(signalyzerSP_offline);
  Protocol {
    SvfTapPins {
     TRST : trst;
     TDI  : tdi;
     TMS  : tms;
     TDO  : tdo
     TCK  : tck;
    }
  }
  Sid(signalyzerSP_offline) {
    cdp_directory : top.cdp;
    signalyzerSP {
      voltage_level_io_0_31 : 1.20v;
      voltage_level_io_32_63 : 1.80v;
      PinMap {
        P0 : trst;
        P1 : tdi;
        P2 : tms;
        P3 : tdo;
        P4 : tck;
        P5 : edt_bypass;
        P6 : edt_single_bypass_chain;
        P7 : edt_clock;
        P8 : refclk;
      }
      operation_mode : offline;
    }
  }
}
```

Sid(opalKellyXem6310)

Additional specification: Sid(opalKellyXem6310_offline)

Usage

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(opalKellyXem6310);
  Protocol {
    ...
    }
  }
  Sid(opalKellyXem6310) {
    cdp_directory : name;                    // Default: top.cdp
    overwrite_cdp_on_startup : on | off;     // Optional
    load_patterns_on_demand : on | off;      // Optional
    sid_tester_host_name : host_name;        // Optional
    remote_invocation_protocol : ssh | rsh; // Optional
    opalKellyXem6310 {
      io_standard : 1.20 | 1.80;             // Optional
      base_frequency : frequency;            // Optional, default: 10MHz
      on_chip_termination : on | off;        // Optional
      use_onboard_ram : on | off | smart;    // Optional
      true_HiZ : on | off;                   // Optional
      serial_number : string ;               // Optional
      PinMap {
        P0 : pin_name;                       // Default: trst;
        P1 : pin_name;                       // Default: tdi;
        P2 : pin_name;                       // Default: tms;
        P3 : pin_name;                       // Default: tdo;
        P4 : pin_name;                       // Default: tck;
        P5 : pin_name;                       // Default: edt_bypass;
        P6 : pin_name;                 // Default: edt_single_bypass_chain;
        P7 : pin_name;                       // Default: edt_clock;
        P8 : pin_name;                       // Default: refclk;
      }
      operation_mode : online | offline | simulation;    // Optional
      timeout        : integer;              // Default: auto
      startup_dofile : dofile_name;
    }
  }
}
```

Description

This is the selected setup and associated wrapper when you are performing analyses on MemoryBIST and LogicTest instruments and ATPG patterns using an Opal Kelly XEM6310-LX45 adaptor as your Desktop connection to the performance board. Refer to "Setting Up the Desktop Environment" for details.

Refer to "Opal Kelly Adaptors" for guidelines for setting the I/O pin voltage. This adaptor enables you to adjust the pin voltage in two I/O banks.

Arguments

- cdp_directory : *name*;

  The name of the CDP directory. The default is *top.cdp*.

- overwrite_cdp_on_startup : on | off;

  An optional property that you can add to the default setup specification if you want the tool to automatically overwrite the CDP. The default is off.

- load_patterns_on_demand : on | off;

  An optional property that you can add to the default setup specification if you want to postpone loading the patterns until right before running them. By default, the tool loads the patterns at the same time as it loads the CDP. Setting this property to on enables you to load only the patterns that you want to run, rather than the whole pattern set.

- sid_tester_host_name : *host_name*;

  An optional property that you can add to the default setup specification if you want to launch the SID tester process on a remote machine. The default host name is localhost. When using this property, you must specify an absolute pathname for the cdp_directory property. In addition, the paths must be to the same respective locations on the remote and local machines.

  When using a remote machine, you must set the TESSENT_TMP_LOCATION environment variable to a directory path that is also visible to both the remote and host machines. For more information about this variable, refer to get_scratch_directory in the *Tessent Shell Reference Manual*.

- remote_invocation_protocol: ssh | rsh;

  An optional property that you can add to the default setup specification when you are launching the SID tester process on a remote machine with the sid_tester_host_name property. The default is rsh.

- opalKellyXem6310/io_standard : 1.20 | 1.80;

  An optional property that specifies the pin voltage range setting. 1.20 implies a setting in the 1.2 V to 1.8 V range. 1.80 (default) implies a setting in the 1.8 V to 3.3 V range.

- opalKellyXem6310/base_frequency : *frequency*;

  An optional property that controls the speed at which the adaptor runs tests. Supported values are: 1MHz, 5MHz, 10MHz (default), 20MHz, and 25MHz.

- opalKellyXem6310/on_chip_termination : on | off;

  An optional property that controls a 50-ohm on-chip termination resistor. Specifying “on” enables the resistor, which may increase signal integrity.

- opalKellyXem6310/use_onboard_ram : on | off | smart;

  An optional property that specifies that you are using an onboard RAM chip (rather than distributed RAM on FPGA) with the Opal Kelly adaptor for the purposes of improving processing speeds for large designs. The “smart” setting enables smart mode, which enables

faster loading and execution time for large pattern files. Smart mode is most effective for ATPG STIL patterns in large designs. It requires the mtsihpdesktop_c license.

Note

When you set this property to "on" or "smart", the Opal Kelly adaptor automatically sets the base_frequency value to 10 MHz. This value overrides any manual settings of this value.

- opalKellyXem6310/true_HiZ : on | off;

  An optional property that enables you to choose true high-Z behavior for the Opal Kelly adaptor pins when not driven. True high-Z behavior means that the I/O is effectively disconnected from the net (same effect as if the adaptor is physically removed) and has no impact on the voltage for the net. By default (off), adaptor pins behave in a low-Z state.

- opalKellyXem6310/serial_number : *string* ;

  An optional property that specifies the serial number of the adaptor that you want to connect to. If you do not specify this property, the tool uses the first board it detects. If you specify this property but the SiliconInsight tool does not find a matching serial number when starting, it prompts you with a list of connected devices and their serial numbers.

  Tip

  You can supply a random serial number, such as "ABCD," if you want the tool to prompt you during startup with this information.

- opalKellyXem6310/PinMap { *signals* ... }

  A required wrapper that specifies the pin connections/signals for this adaptor. Refer to "Opal Kelly Adaptors" for details.

- opalKellyXem6310/operation_mode : online | offline | simulation;

  An optional property that you set to "offline" when you are using Sid(opalKellyXem6310_offline) as the selected setup, which you can use to verify that the patterns run correctly. You can also set it to "simulation" when you want to validate that the patterns pass when you run them using the adaptor.

- opalKellyXem6310/timeout *integer*;

  An optional property that you use when you have set the operation mode to simulation. This property specifies the time in seconds for which Tessent SiliconInsight should wait for design elaboration in SimDUT mode. This property is useful for bigger designs when elaboration can take hours. You can set the timeout from 1 second to 65535 seconds. The default is 600 seconds.

- opalKellyXem6310/startup_dofile *dofile_name*;

  An optional property that you use when you have set the operation mode to simulation. This property specifies the path to the startup_dofile that you use for SimDUT mode. This dofile contains the commands to set up the SimDUT simulation. Refer to "Sid(simdut)" on page 128 for more information.

Examples

The following example shows the default setup specification when you are using Sid(opalKellyXem6310_offline).

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(opalKellyXem6310_offline);
  Protocol {
    SvfTapPins {
     TRST : trst;
     TDI  : tdi;
     TMS  : tms;
     TDO  : tdo
     TCK  : tck;
    }
  }
  Sid(opalKellyXem6310_offline) {
    cdp_directory : top.cdp;
    opalKellyXem6310 {
      io_standard : 1.80;
      base_frequency : 10MHz;
      on_chip_termination : off
      PinMap {
        P0 : trst;
        P1 : tdi;
        P2 : tms;
        P3 : tdo;
        P4 : tck;
        P5 : edt_bypass;
        P6 : edt_single_bypass_chain;
        P7 : edt_clock;
        P8 : refclk;
      }
      operation_mode : offline;
    }
  }
}
```

Sid(opalKellyXem7310)

Additional specification: Sid(opalKellyXem7310_offline).

Usage

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(opalKellyXem7310) ;
  Protocol {
    ...
    }
  }
  Sid(opalKellyXem7310) {
    cdp_directory : name ;                    // Default: top.cdp
    overwrite_cdp_on_startup : on | off ;     // Optional
    load_patterns_on_demand : on | off ;      // Optional
    sid_tester_host_name : host_name ;        // Optional
    remote_invocation_protocol : ssh | rsh ; // Optional
    opalKellyXem7310 {
      io_standard : 1.20 | 1.80 ;             // Optional
      base_frequency : 10MHz ;                // Optional
      on_chip_termination : off ;             // Optional
      use_onboard_ram : on | off | smart ;    // Optional
      true_HiZ : on ;                         // Optional
      serial_number : string ;                // Optional
      PinMap {
        P0 : pin_name ;                       // Default: trst
        P1 : pin_name ;                       // Default: tdi
        P2 : pin_name ;                       // Default: tms
        P3 : pin_name ;                       // Default: tdo
        P4 : pin_name ;                       // Default: tck
        P5 : pin_name ;                       // Default: edt_bypass
        P6 : pin_name ;              // Default: edt_single_bypass_chain
        P7 : pin_name ;                       // Default: edt_clock
        P8 : pin_name ;                       // Default: refclk
      }
      operation_mode  : online | offline | simulation ; // Optional
      timeout         : integer ;                       // Default: auto
      startup_dofile  : dofile_name ;                   // Optional
    }
  }
}
```

Description

This is the selected setup and associated wrapper when you are performing analyses on MemoryBIST and LogicTest instruments and ATPG patterns using an Opal Kelly XEM7310-A75 adaptor as your Desktop connection to the performance board. Refer to “Setting Up the Desktop Environment” for details.

Arguments

- cdp_directory : *name* ;

  The name of the CDP directory. The default is *top.cdp*.

- overwrite_cdp_on_startup : on | off ;

  An optional property that you can add to the default setup specification if you want the tool to automatically overwrite the CDP. The default is off.

- load_patterns_on_demand : on | off ;

  An optional property that you can add to the default setup specification if you want to postpone loading the patterns until right before running them. By default, the tool loads the patterns at the same time as it loads the CDP. Setting this property to on enables you to load only the patterns that you want to run, rather than the whole pattern set.

- sid_tester_host_name : *host_name* ;

  An optional property that you can add to the default setup specification if you want to launch the SID tester process on a remote machine. The default host name is localhost. When using this property, you must specify an absolute pathname for the cdp_directory property. In addition, the paths must be to the same respective locations on the remote and local machines.

  When using a remote machine, you must set the TESSENT_TMP_LOCATION environment variable to a directory path that is also visible to both the remote and host machines. For more information about this variable, refer to get_scratch_directory in the *Tessent Shell Reference Manual*.

- remote_invocation_protocol : ssh | rsh ;

  An optional property that you can add to the default setup specification when you are launching the SID tester process on a remote machine with the sid_tester_host_name property. The default is rsh.

- opalKellyXem7310/io_standard : 1.20 | 1.80 ;

  An optional property that specifies the pin voltage setting, either 1.20 or 1.80. The default is 1.80.

- opalKellyXem7310/base_frequency : 10MHz ;

  An optional property that indicates the speed at which the adaptor runs tests. The XEM7310 has only one base frequency. You cannot change increase or decrease the value. The default is 10MHz, which is the only supported value.

- opalKellyXem7310/on_chip_termination : off ;

  An optional property but the XEM7310 does not have on-chip termination resistors to turn on. The default is off, which is the only supported value.

- opalKellyXem7310/use_onboard_ram : on | off | smart ;

  An optional property that specifies that you are using an onboard RAM chip (rather than distributed RAM inside the FPGA) on the XEM7310 for the purposes of improving processing speeds for large designs. The “smart” setting enables smart mode, which enables faster loading and execution time for large pattern files. Smart mode is most effective for ATPG STIL patterns in large designs. It requires the mtsihpdesktop_c license.

- opalKellyXem7310/true_HiZ : on ;

  An optional property that specifies true high-Z behavior for the Opal Kelly adaptor pins when not driven. True high-Z behavior means that the I/O is effectively disconnected from the net (same effect as if the adaptor is physically removed) and has no impact on the voltage for the net. The default is on, which is the only supported value.

- opalKellyXem7310/serial_number : *string* ;

  An optional property that specifies the serial number of the adaptor that you want to connect to. If you do not specify this property, the tool uses the first board it detects. If you specify this property but the SiliconInsight tool does not find a matching serial number when starting, it prompts you with a list of connected devices and their serial numbers.

  Tip

  You can supply a random serial number, such as "ABCD," if you want the tool to prompt you during startup with this information.

- opalKellyXem7310/PinMap : { *signals ...* } ;

  A required wrapper that specifies the pin connections/signals for this adaptor. Refer to Opal Kelly XEM6310-LX45 Adaptor Pinout for details.

- opalKellyXem7310/operation_mode : online | offline | simulation ;

  An optional property that you set to "offline" when you are using Sid(opalKellyXem7310_offline) as the selected setup, which you can use to verify that the patterns run correctly. You can also set it to "simulation" when you want to validate that the patterns pass when you run them using the adaptor in simulation. The default is online.

- opalKellyXem7310/timeout : *integer* ;

  An optional property that you use when you have set the operation mode to simulation. This property specifies the time in seconds for which Tessent SiliconInsight should wait for design elaboration in SimDUT mode. This property is useful for bigger designs when elaboration can take hours. You can set the timeout from 1 second to 65535 seconds. The default is 600 seconds.

- opalKellyXem7310/startup_dofile : *dofile_name* ;

  An optional property that you use when you have set the operation mode to simulation. This property specifies the path to the startup_dofile that you use for SimDUT mode. This dofile contains the commands to set up the SimDUT simulation. Refer to "Sid(simdut)" on page 128 for more information.

Examples

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(opalKellyXem7310);
  Sid(opalKellyXem7310) {
    cdp_directory : top.cdp;
    opalKellyXem7310 {
      operation_mode : online;

      io_standard: 1.80;
      on_chip_termination : off;
      true_HiZ : on;
      use_onboard_ram : smart;
      base_frequency: 10MHz;

      PinMap {
        P0: pta[5];
        P1: pta[0];
        P2: pta[1];
        …
        P119: pta[56];
      }
    }
  }
}
```

Sid(opalKellyXem8350)

Additional specification: Sid(opalKellyXem8350_offline).

Usage

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(opalKellyXem8350) ;
  Protocol {
    ...
    }
  }
  Sid(opalKellyXem8350) {
    cdp_directory : name ;                    // Default: top.cdp
    overwrite_cdp_on_startup : on | off ;     // Optional
    load_patterns_on_demand : on | off ;      // Optional
    sid_tester_host_name : host_name ;        // Optional
    remote_invocation_protocol : ssh | rsh ; // Optional
    opalKellyXem8350 {
      io_standard : 1.20 | 1.80 ;             // Optional
      base_frequency : 100MHz ;               // Optional
      on_chip_termination : off ;             // Optional
      use_onboard_ram : smart ;               // Optional
      true_HiZ : on ;                         // Optional
      serial_number : string ;                // Optional
      PinMap {
        P0 : pin_name ;                       // Default: trst
        P1 : pin_name ;                       // Default: tdi
        P2 : pin_name ;                       // Default: tms
        P3 : pin_name ;                       // Default: tdo
        P4 : pin_name ;                       // Default: tck
        P5 : pin_name ;                       // Default: edt_bypass
        P6 : pin_name ;                 // Default: edt_single_bypass_chain
        P7 : pin_name ;                       // Default: edt_clock
        P8 : pin_name ;                       // Default: refclk
      }
      operation_mode  : online | offline | simulation ; // Optional
      timeout         : integer ;                       // Default: auto
      startup_dofile  : dofile_name ;                   // Optional
    }
  }
}
```

Description

This is the selected setup and associated wrapper when you are performing analyses on MemoryBIST and LogicTest instruments and ATPG patterns using an Opal Kelly XEM8350-KU060 adaptor as your Desktop connection to the performance board. Refer to "Setting Up the Desktop Environment" for details.

Arguments

- cdp_directory : *name* ;

  The name of the CDP directory. The default is *top.cdp*.

- overwrite_cdp_on_startup : on | off ;

  An optional property that you can add to the default setup specification if you want the tool to automatically overwrite the CDP. The default is off.

- load_patterns_on_demand : on | off ;

  An optional property that you can add to the default setup specification if you want to postpone loading the patterns until right before running them. By default, the tool loads the patterns at the same time as it loads the CDP. Setting this property to on enables you to load only the patterns that you want to run, rather than the whole pattern set.

- sid_tester_host_name : *host_name* ;

  An optional property that you can add to the default setup specification if you want to launch the SID tester process on a remote machine. The default host name is localhost. When using this property, you must specify an absolute pathname for the cdp_directory property. In addition, the paths must be to the same respective locations on the remote and local machines.

  When using a remote machine, you must set the TESSENT_TMP_LOCATION environment variable to a directory path that is also visible to both the remote and host machines. For more information about this variable, refer to get_scratch_directory in the *Tessent Shell Reference Manual*.

- remote_invocation_protocol : ssh | rsh ;

  An optional property that you can add to the default setup specification when you are launching the SID tester process on a remote machine with the sid_tester_host_name property. The default is rsh.

- opalKellyXem8350/io_standard : 1.20 | 1.80 ;

  An optional property that specifies the pin voltage setting, either 1.20 or 1.80. The default is 1.80.

- opalKellyXem8350/base_frequency : 100MHz ;

  An optional property that indicates the speed at which the adaptor runs tests. Only a speed of 100MHz is supported.

- opalKellyXem8350/on_chip_termination : off ;

  An optional property that turns off the 50-ohm on-chip termination resistor. This resistor cannot be turned on.

- opalKellyXem8350/use_onboard_ram : smart ;

  An optional property that specifies that you are using an onboard RAM chip (rather than distributed RAM on FPGA) with the Opal Kelly adaptor for the purposes of improving processing speeds for large designs. The only available setting, “smart”, turns on smart mode, which enables faster loading and execution time for large pattern files.

- opalKellyXem8350/true_HiZ : on ;

  An optional property that specifies true high-Z behavior for the Opal Kelly adaptor pins when not driven. True high-Z behavior means that the I/O is effectively disconnected from the net (same effect as if the adaptor is physically removed) and has no impact on the voltage for the net. This is the only available behavior.

- opalKellyXem8350/serial_number : *string* ;

  An optional property that specifies the serial number of the adaptor that you want to connect to. If you do not specify this property, the tool uses the first board it detects. If you specify this property but the SiliconInsight tool does not find a matching serial number when starting, it prompts you with a list of connected devices and their serial numbers.

  Tip

  You can supply a random serial number, such as "ABCD," if you want the tool to prompt you during startup with this information.

- opalKellyXem8350/PinMap : { *signals ...* } ;

  A required wrapper that specifies the pin connections/signals for this adaptor. Refer to "Opal Kelly XEM8350-KU060 Adaptor Pinout" on page 43 for details.

- opalKellyXem8350/operation_mode : online | offline | simulation ;

  An optional property that you set to "offline" when you are using Sid(opalKellyXem8350_offline) as the selected setup, which you can use to verify that the patterns run correctly. You can also set it to "simulation" when you want to validate that the patterns pass when you run them using the adaptor in simulation.

- opalKellyXem8350/timeout : *integer* ;

  An optional property that you use when you have set the operation mode to simulation. This property specifies the time in seconds for which Tessent SiliconInsight should wait for design elaboration in SimDUT mode. This property is useful for bigger designs when elaboration can take hours. You can set the timeout from 1 second to 65535 seconds. The default is 600 seconds.

- opalKellyXem8350/startup_dofile : *dofile_name* ;

  An optional property that you use when you have set the operation mode to simulation. This property specifies the path to the startup_dofile that you use for SimDUT mode. This dofile contains the commands to set up the SimDUT simulation. Refer to "Sid(simdut)" on page 128 for more information.

Examples

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(opalKellyXem8350);
  Sid(opalKellyXem8350) {
    cdp_directory : top.cdp;
    opalKellyXem8350 {
      operation_mode : online;

      io_standard: 1.80;
      on_chip_termination : off;
      true_HiZ : on;
      use_onboard_ram : smart;
      base_frequency: 100MHz;
      PinMap {
        P1: pta[0];
        P118: pta[1];
        P295: pta[2];
        P189: ptc[1];
        …
      }
    }
  }
}
```

Sid(ftdi4232H_mini_module)

Additional specification: Sid(ftdi4232H_mini_module_offline)

Usage

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(ftdi4232H_mini_module);
  Protocol {
    ...
    }
  }
  Sid(ftdi4232H_mini_module) {
    cdp_directory : name;                    // Default: top.cdp
    overwrite_cdp_on_startup : on | off;     // Optional
    load_patterns_on_demand : on | off;      // Optional
    sid_tester_host_name : host_name;        // Optional
    remote_invocation_protocol : ssh | rsh; // Optional
    ftdi4232H_mini_module {
      PinMap {
        B0 : pin_name;                      // Default: trst
        B1 : pin_name;                      // Default: tdi
        B2 : pin_name;                      // Default: tms
        B3 : pin_name;                      // Default: tdo
        B4 : pin_name;                      // Default: tck
        B5 : pin_name;                      // Default: edt_bypass
        B6 : pin_name;                 // Default: edt_single_bypass_chain
        B7 : pin_name;                      // Default: edt_clock
      }
      operation_mode : online | offline | simulation;    // Optional
      timeout        : integer;             // Default: auto
      startup_dofile : dofile_name;
    }
  }
}
```

Description

This is the selected setup and associated wrapper when you are performing analyses on MemoryBIST and LogicTest instruments and ATPG patterns using a Future Technology Devices International (FTDI) FT4232H Mini Module adaptor as your Desktop connection to the performance board. Refer to “Setting Up the Desktop Environment” for details.

Arguments

- cdp_directory : *name*;

  The name of the CDP directory. The default is top.cdp.

- overwrite_cdp_on_startup : on | off;

  An optional property that you can add to the default setup specification if you want the tool to automatically overwrite the CDP. The default is off.

- load_patterns_on_demand : on | off;

  An optional property that you can add to the default setup specification if you want to postpone loading the patterns until right before running them. By default, the tool loads the patterns at the same time as it loads the CDP. Setting this property to on enables you to load only the patterns that you want to run, rather than the whole pattern set.

- sid_tester_host_name : *host_name*;

  An optional property that you can add to the default setup specification if you want to launch the SID tester process on a remote machine. The default host name is localhost. When using this property, you must specify an absolute pathname for the cdp_directory property. In addition, the paths must be to the same respective locations on the remote and local machines.

  When using a remote machine, you must set the TESSENT_TMP_LOCATION environment variable to a directory path that is also visible to both the remote and host machines. For more information about this variable, refer to get_scratch_directory in the *Tessent Shell Reference Manual*.

- remote_invocation_protocol: ssh | rsh;

  An optional property that you can add to the default setup specification when you are launching the SID tester process on a remote machine with the sid_tester_host_name property. The default is rsh.

- ftdi4232H_mini_module/PinMap { *signals* ... }

  A required wrapper that specifies the pin connections/signals for this adaptor. Refer to "Future Technology Devices International FT4232H Mini Module" for details.

- ftdi4232H_mini_module/operation_mode : online | offline | simulation;

  An optional property that you set to "offline" when you are using Sid(ftdi4232H_mini_module_offline) as the selected setup, which you can use to verify that the patterns run correctly. You can also set it to "simulation" when you want to validate that the patterns pass when you run them using the adaptor.

- ftdi4232H_mini_module/timeout *integer*;

  An optional property that you use when you have set the operation mode to simulation. This property specifies the time in seconds for which Tessent SiliconInsight should wait for design elaboration in SimDUT mode. This property is useful for bigger designs when elaboration can take hours. You can set the timeout from 1 second to 65535 seconds. The default is 600 seconds.

- ftdi4232H_mini_module/startup_dofile *dofile_name*;

  An optional property that you use when you have set the operation mode to simulation. This property specifies the path to the startup_dofile that you use for SimDUT mode. This dofile contains the commands to set up the SimDUT simulation. Refer to "Sid(simdut)" on page 128 for more information.

Examples

The following example shows the default setup specification when you are using Sid(ftdi4232H_mini_module_offline).

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(ftdi4232H_mini_module_offline);
  Protocol {
    SvfTapPins {
     TRST : trst;
     TDI  : tdi;
     TMS  : tms;
     TDO  : tdo;
     TCK  : tck;
    }
  }
  Sid(ftdi4232H_mini_module_offline) {
    cdp_directory : top.cdp;
    ftdi4232H_mini_module {
      PinMap {
        B0 : trst;
        B1 : tdi;
        B2 : tms;
        B3 : tdo;
        B4 : tck:
        B5 : edt_bypass;
        B6 : edt_single_bypass_chain;
        B7 : edt_clock
       }
       operation_mode : offline;
    }
  }
}
```

Sid(simdut)

Use Desktop mode to run simulation.

Usage

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(simdut);
  Protocol {
    ...
    }
  }
  Sid(simdut) {
    cdp_directory : name;                    // Default: top.cdp
    overwrite_cdp_on_startup : on | off;     // Optional
    load_patterns_on_demand : on | off;      // Optional
    sid_tester_host_name : host_name;        // Optional
    remote_invocation_protocol : ssh | rsh; // Optional
    simdut {
      startup_dofile: dofile_name;           // Optional
      }
    }
  }
}
```

Description

This is the selected setup and associated wrapper when you are simulating the behavior of LogicBIST, MemoryBIST, and ATPG instruments in Desktop mode, as well as MemoryBIST in offline ATE mode.

For more information about running simulation, refer to "Simulating Desktop, ATE, and ATPG Behavior."

Arguments

- cdp_directory : *name*;

  The name of the CDP directory. The default is top.cdp.

- overwrite_cdp_on_startup : on | off;

  An optional property that you can add to the default setup specification if you want the tool to automatically overwrite the CDP. The default is off.

- load_patterns_on_demand : on | off;

  An optional property that you can add to the default setup specification if you want to postpone loading the patterns until right before running them. By default, the tool loads the patterns at the same time as it loads the CDP. Setting this property to on enables you to load only the patterns that you want to run, rather than the whole pattern set.

- sid_tester_host_name : *host_name*;

  An optional property that you can add to the default setup specification if you want to launch the SID tester process on a remote machine. The default host name is localhost.

When using this property, you must specify an absolute pathname for the cdp_directory property. In addition, the paths must be to the same respective locations on the remote and local machines.

When using a remote machine, you must set the TESSENT_TMP_LOCATION environment variable to a directory path that is also visible to both the remote and host machines. For more information about this variable, refer to get_scratch_directory in the *Tessent Shell Reference Manual*.

- remote_invocation_protocol: ssh | rsh;

  An optional property that you can add to the default setup specification when you are launching the SID tester process on a remote machine with the sid_tester_host_name property. The default is rsh.

- simdut/startup_dofile : *dofile_name*;

  An optional property that you can add to the default setup specification that specifies a startup dofile that provides an alternative way to start SimDUT for designs with or without TSDBs. The following sample SimDUT startup dofile retrieves the SimDUT setup information from the setup specification and invokes the simulator:

```
set_sim_lib_sources -v ../verilog/adk.v   \
-v ../data/mem/*.v -v ../data/picdram.v run_testbench_simulations -
simulator_options  \
"+nospecify" -simdut_server
```

Examples

The following example shows the default setup specification for SimDUT mode.

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(simdut);
  Protocol {
    SvfTapPins {
     TRST : trst;
     TDI  : tdi;
     TMS  : tms;
     TDO  : tdo
     TCK  : tck;
    }
  }
  Sid(simdut) {
    cdp_directory : top.cdp;
    SimDut {
    }
  }
}
```

Ate(tester_name)

Perform IJTAG debug of Tessent IJTAG-inserted devices on a specified tester.

Usage

```
SiliconInsightSetupSpecification {
  selected_setup : Ate(tester_name);
  Protocol {
    ...
    }
  }
  Ate(tester_name) {
    cdp_integration_type : Partial;
    cdp_directory : name;
    host_name : host_name;                          // Default: localhost
    port_number : port_number;
    load_patterns_on_demand : on | off;             // Optional
    pattern_transmission_method : nfs_file | stream_file;
                                                    // Default: nfs_file
    initial_control_string: "string";               // Optional
    PinMap {
      P0 : pin_name;
      P1 : pin_name;
      P2 : pin_name;
      P3 : pin_name;
  }
}
```

Description

This is the selected setup and associated wrapper that enables ATE-Connect. ATE-Connect enables you to debug Tessent IJTAG-inserted devices on an ATE.

When using ATE-Connect, you must prepare your test program based on the ATE vendor documentation and specify the SiliconInsightSpecification as specified by the ATE vendor. See "ATE-Connect for IJTAG Debugging on Testers" on page 293 for details.

Arguments

- cdp_integration_name : Partial;

  A property that indicates that the CDP libraries are not fully integrated into the ATE software environment. Instead, the integration occurs through a TCP-IP communication protocol between the Tessent Shell environment and the ATE. This setting is required for ATE-Connect mode.

- cdp_directory : *name*;

  The name of the CDP directory. The default is <*design_name*>*.cdp*.

- host_name : *host_name*;

  A property that specifies the name of the tester controller machine to use to establish the TCP-IP connection to the ATE software running on that machine.

- port_number : *port_number*;

  A required property that specifies the port number on the tester controller that the TCP-IP connection should use for communication between Tessent SiliconInsight and the ATE. This property may be defined by the ATE vendor based on its security protocols.

- load_patterns_on_demand : on | off;

  An optional property that you can add to the default setup specification if you want to postpone loading the patterns until right before running them. By default, the tool loads the patterns at the same time as it loads the CDP. Setting this property to on enables you to load only the patterns that you want to run, rather than the whole pattern set.

- pattern_transmission_method : nfs_file | stream_file; (*Defined by the ATE vendor.*)

  A property that specifies how Tessent SiliconInsight transmits pattern data it generates to the tester. The following choices are available:

  - nfs_file — The pattern file generated by Tessent SiliconInsight is saved to the CDP, and the path to the pattern file is sent to the tester. This method requires the CDP directory to be accessible by the machines running Tessent Shell and the test program. This is the default.
  - stream_file — The pattern file generated by Tessent SiliconInsight is saved to the CDP and the data is streamed to the tester. This method is slower than using nfs_file, but it does not require the CDP to be accessible by the machine running the test program.

- initial_control_string : "*string*"; (*Defined by the ATE vendor.*)

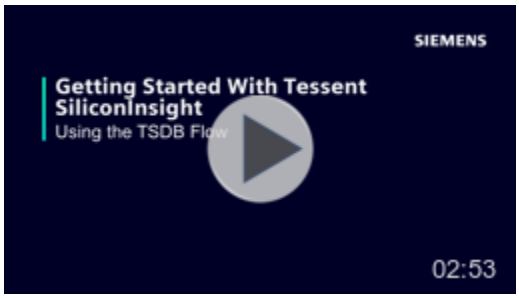
  An optional property that directs Tessent SiliconInsight to initiate a new session by sending the specified string value to the ATE as the first data sent over the TCP-IP socket, which enables the tester to reset or initialize resources as needed. If the ATE vendor specifies this string, you must include it in the setup specification.

- PinMap { *signals* ... }

  A required wrapper that specifies the pin connections/signals for ATE-Connect. Specify the ATE pins followed by the design pin names. Unspecified ATE pins are assumed to be uncontacted.

Accessing SiliconInsight Getting Started Videos

Getting Started With Tessent SiliconInsight Using the TSDB Flow is a video that shows some basic steps in starting and configuring Tessent SiliconInsight.



This video shows how to load an existing Tessent Shell DataBase (TSDB) and set it up for use with SiliconInsight. It also discusses the SiliconInsight setup specification that you use to configure SiliconInsight.

Chapter 4
Diagnosis and Debug on the Desktop

Tessent SiliconInsight Desktop provides an interactive diagnosis and debug environment that can help you validate and characterize your silicon prototype. The tool supports diagnosing and debugging Tessent LogicBIST and Tessent MemoryBIST instruments. In addition, it supports PDL-level and top-level ICL debugging for third-party IEEE 1687 IJTAG devices.

Note

Desktop mode also supports diagnosis and debug for ATPG patterns. However, for ATPG patterns, you use a non-TSDB flow as described in "Support for ATPG (Non-TSDB Flow)."

Performing Diagnosis and Interpreting the Results (TSDB Flow)

You can automatically diagnose any failure by selecting a failing Patterns wrapper and clicking the **Diagnose** button. The tool displays the diagnosis results in the Transcript area.

Prerequisites

- You must have a signed-off TSDB. For more information, refer to Chapter 9 in the *Tessent Shell Reference Manual*.

Note

By default, when you create the TSDB with the write_tsdb_data command, Tessent Shell saves the scan chain data for only the first 1024 patterns.

If you have a failing pattern greater than 1023, before running diagnosis, you can re-run fault simulation and generate a new TSDB that includes more patterns. To do this, specify write_tsdb_data with the -max_scan_load_unload_size option. For details, refer to "Performing LogicBIST Fault Simulation and Pattern Creation" in the *Hybrid TK/LBIST User's Manual*.

- You have generated and customized a setup specification as described in "Prepare the Design Under Test" on page 75.

Procedure

1. From a shell, invoke Tessent Shell using the following syntax:

   ```
   % tessent -shell
   ```

   After invocation, the tool is in setup mode. Refer to the "tessent" command description in the *Tessent Shell Reference Manual* for additional invocation options.

2. Set the tool context to enable Tessent SiliconInsight within Tessent Shell:

   ```
   SETUP> set_context patterns -silicon_insight
   ```

   In this context, you can use the set_silicon_insight_option command to configure Tessent SiliconInsight options. For example, if you have compressed STIL or SVF pattern files, enable using compressed patterns with the -allow_compressed_patterns option.

3. Load the TSDBs. You must open the individual TSDBs for the cores and top level, set the top-level design, and then set the current design. For example:

   ```
   SETUP> open_tsdb core1.tsdb
   SETUP> open_tsdb core2.tsdb
   SETUP> open_tsdb top.tsdb
   SETUP> read_design top -design_id rtl -view interface
   SETUP> set_current_design top
   ```

   The tool loads the current design, the current design's top-level ICL, and the associated patterns specification.

   You can override the patterns specification by using the read_config_data command to load another one. You can also save the default patterns specification to a file using the write_config_data command and load it in subsequent sessions using the read_config_data command.

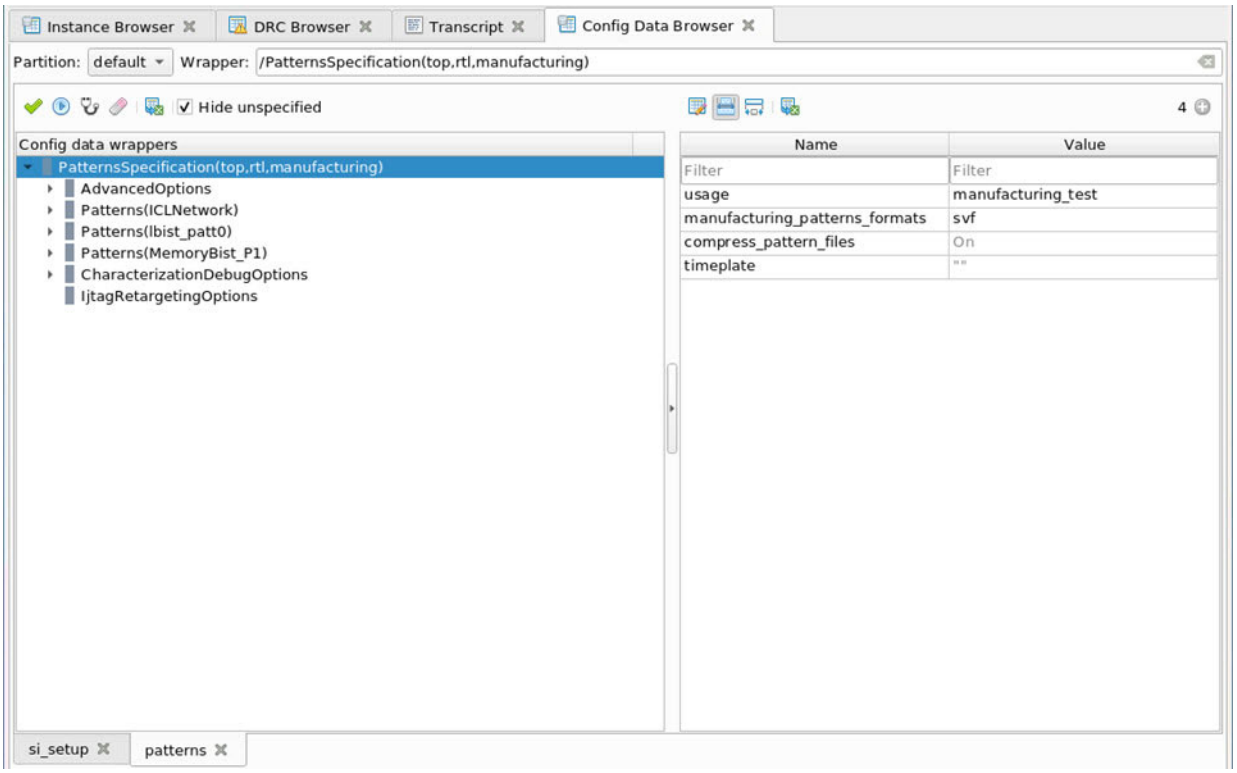
4. Load the setup specification as follows:

   SETUP> read_config_data *setup_spec_path*

5. Start Tessent SiliconInsight in interactive mode:

   SETUP> start_silicon_insight -gui

   The GUI displays.



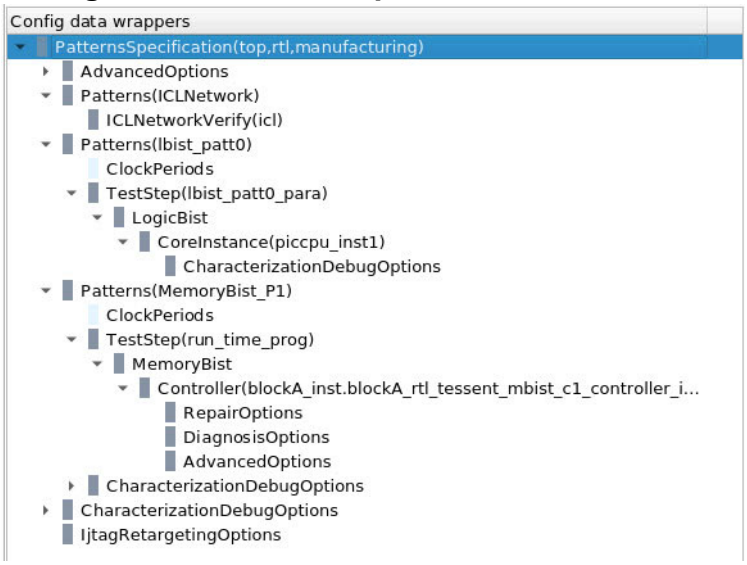
When you click the right arrow ( ) next to the loaded PatternsSpecification, the GUI provides a cascading display of the embedded wrappers and instruments within the loaded patterns specification. The PatternsSpecification wrapper contains one or more Patterns wrappers, each of which describes patterns to be tested or simulated.

Figure 4-1 shows the PatternsSpecification wrapper cascade in the Config data wrappers pane that contains various patterns and test steps. Status indicator icons on the right-hand side of the pane for each test indicate the results of the test: a green checkmark indicates a passing test and a red exclamation point indicates a failing test.

Note

For details about the PatternsSpecification wrapper and its contents, refer to the *Tessent Shell Reference Manual*. You may see a wrapper named "SimulationOptions," which controls simulations when generating testbenches. For purposes of diagnosis and debug using the Tessent SiliconInsight GUI, you can ignore this wrapper.

Figure 4-1. PatternsSpecification Cascade

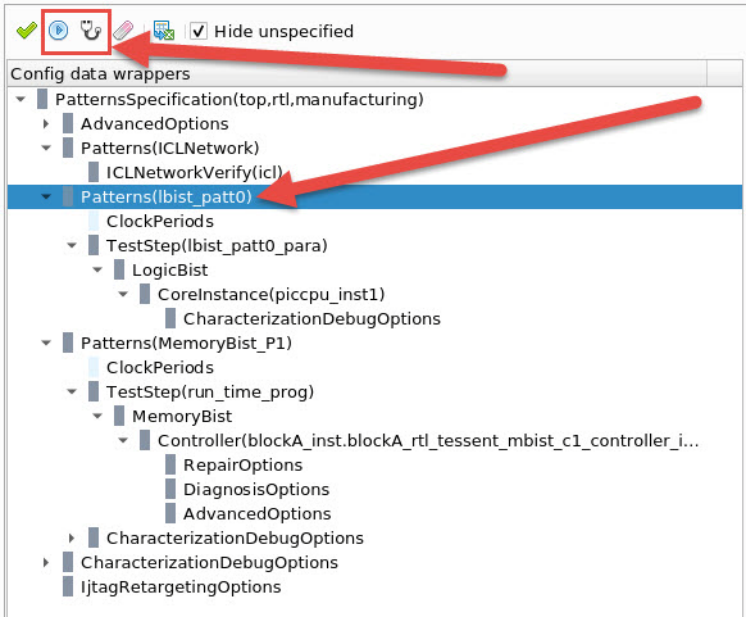


6. Under Config data wrappers, click the Patterns wrapper you want to test.

   Figure 4-2 shows a PatternsSpecification wrapper that contains three Patterns wrappers, including lbist_patt0 for a LogicBIST instrument. By default, diagnose_on_failure is off.

   For descriptions of the other wrappers, refer to "PatternsSpecification" in the *Tessent Shell Reference Manual*.

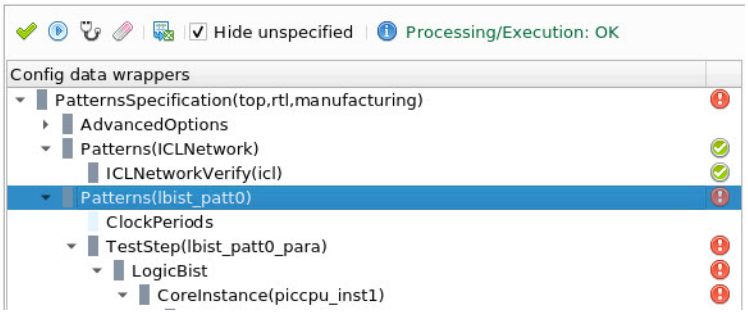
Figure 4-2. Patterns Wrapper for Test lbist_patt0



7. Click **Process/Execute** ( ) to run the test.

The red icons in Figure 4-3 show that the test for lbist_patt0 failed. When a test passes, the icons turn green and change to checkmarks.

Figure 4-3. Test lbist_patt0 Failure



The **Transcript** tab reports the failure as follows:

```
//  Command: execute_cdp_test lbist_patt0  -collect_data_type_list { variable instrument diagnosis }
//  Test 'lbist_patt0' failed.
//  Variable failures and unmapped failures of pattern 'lbist_patt0.svf' :
//
//  PatternSet        Variable                                                          Pin                  Expected
//                                                                                                           Actual
//  ----------------  ----------------------------------------------------------------  ---  -------------------------
//  lbist_patt0_para  piccpu_inst1.piccpu_gate_tessent_edt_lbist_i.misr_lbist_patt0_para  tdo  b00100011111111011000010
//                                                                                              b01001110110000000001000010
//  ----------------  ----------------------------------------------------------------  ---  -------------------------
//  Diagnostic Result: CDPResult/TestExecutionResult/DiagnosticResult/PatternSets(0)/PatternSetData(0)/InstrumentResult(0)
//  PatternsSpec Path: Patterns(lbist_patt0)/TestStep(lbist_patt0_para)/LogicBist/piccpu_inst1
//
//  Failing instrument icl name     = piccpu_inst1
//  Failing instrument Verilog name = piccpu_inst1
//   Patterns(lbist_patt0) : fail
//         TestStep(lbist_patt0_para) : fail
//                  LogicBist : fail
//                           CoreInstance(piccpu_inst1) : fail
```

The tool reports “Unknown” for Verilog instance names when the ICL has not been extracted using Tessent Shell.

You can introspect the test results using Tessent Shell commands such as report_config_data, get_config_element, and get_config_value.

8. Click **Diagnose** () to diagnose failures.

Note

When you specify diagnose_on_failure on, you only need to click **Process/Execute**. The tool runs the test and automatically supplies the diagnosis information for any failures.

9. Debug the DUT.
10. Exit the Tessent SiliconInsight session:

```
SETUP> stop_silicon_insight
```

Results

Refer to “LogicBIST Diagnosis Results” on page 203.

Related Topics

CDP Result Structure

Characterization Debug Options Display in the GUI

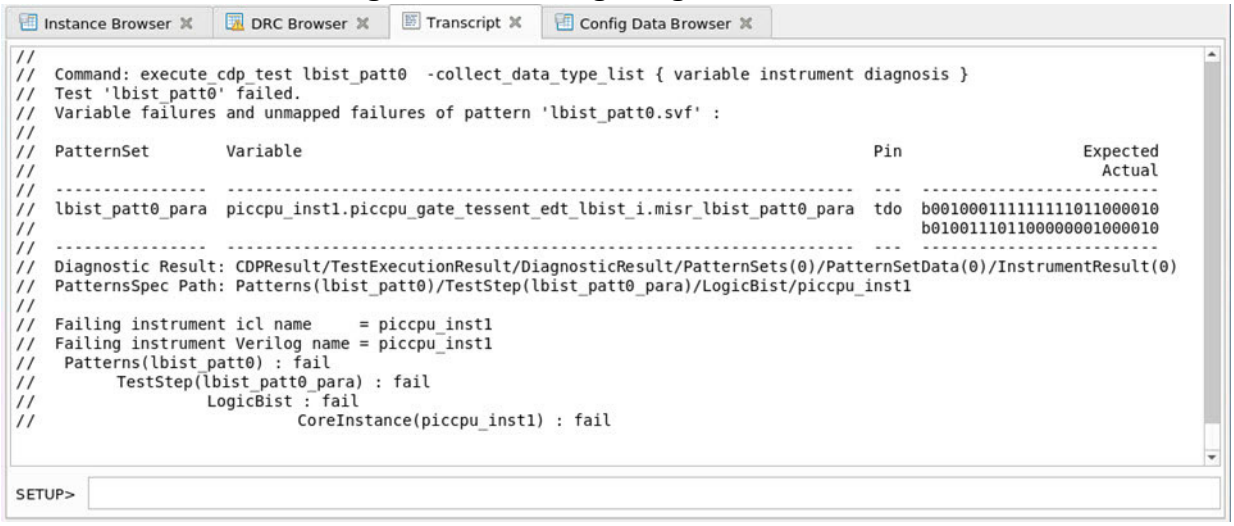
Performing Manual Debug

Performing Manual Debug

After diagnosis, you can examine the results and adjust the specific instrument or CharacterizationDebugOptions and run the diagnosis again. By fine tuning and rerunning the test and diagnosis in an iterative process you can pinpoint root causes of the silicon failures.

Suppose you have the following results after running and diagnosing test pattern lbist_patt0. This discussion continues from section "Performing Diagnosis and Interpreting the Results (TSDB Flow)" on page 133.

Figure 4-4. Failing Diagnosis Results



Prerequisites

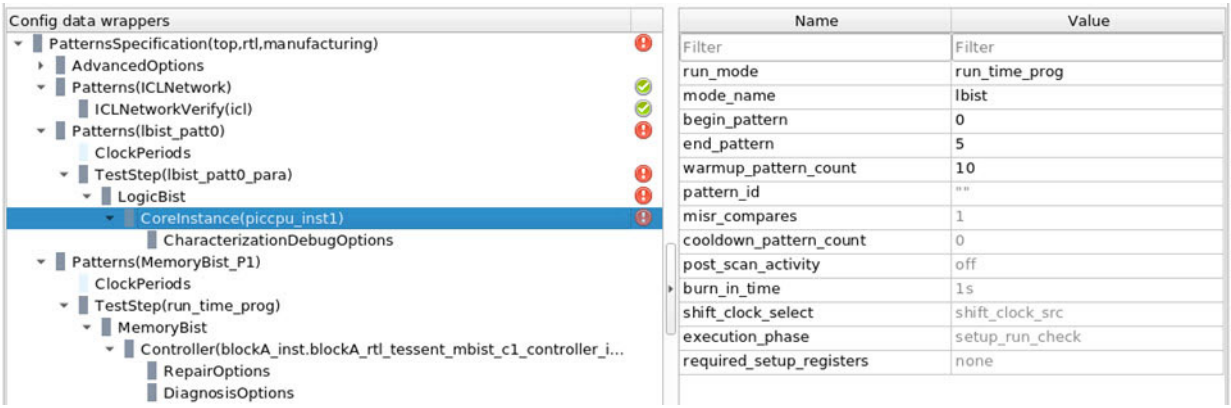
- You have successfully run a test in Tessent SiliconInsight Desktop and have diagnosis results. See "Performing Diagnosis and Interpreting the Results (TSDB Flow)" on page 133 for details.

Procedure

1. In the **Config Data Wrappers** tab and in the Config data wrappers pane, select a failing instance to debug.

   In Figure 4-5, the CoreInstance(piccpu_inst1) instance is selected for debug. The options associated with the instance are in the property table pane on the right. The dimmed lettering indicates options that were specified higher in the wrapper hierarchy.

Figure 4-5. Selected Failing Instance and Associated Configuration Options



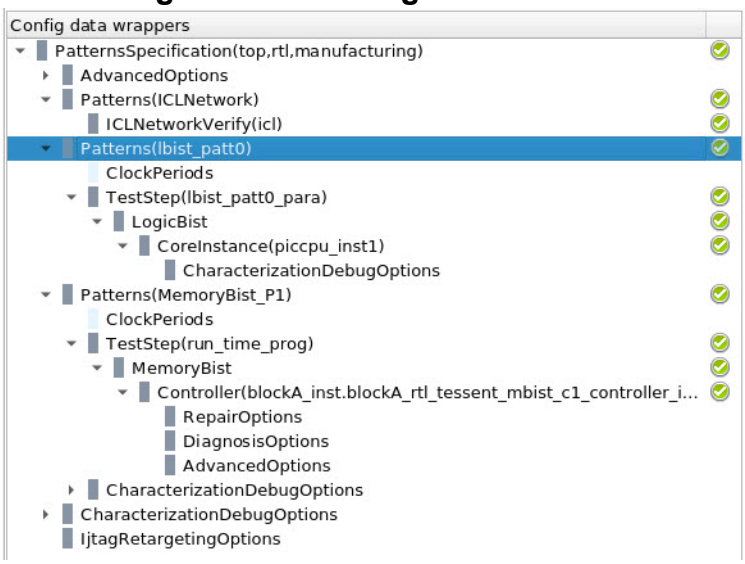
2. In the property table pane, double-click the begin_pattern field, change 0 to 3, and then click **Apply**.

   You know from the diagnosis transcript that the failure occurred at pattern 2.

3. Select the pattern wrapper Patterns(lbist_patt0), and then click **Process/Execute** ( ).

   Now the test passes. The passing results propagate up the hierarchy. If MemoryBist_P1 was failing, you would debug that failure.

Figure 4-6. Passing Test Results



If you click **Process/Execute** while at the CoreInstance level, you receive the following error:

```
Error: Execution/diagnosis of 'CoreInstance' level isn't supported.
   Can only execute/diagnose at the PatternsSpecification,
   PatternsGroup, Patterns, CustomPatterns, or ScanPatterns
   wrapper level.
```

Results

The debug process helps you pinpoint root cause defects in the silicon.

Diagnosing and Debugging IJTAG Networks

The first test run for a DUT designed using the 1687 IJTAG Network standard is the ICLNetwork test. This test verifies that the IJTAG network functions as designed. If the test fails, subsequent testing and diagnosis of instruments in the network may not be reliable. As design sizes grow, so do the IJTAG networks, resulting in a higher probability of defects in the network.

Assumptions and limitations:

- Multi-chain interfaces and OneHotScanGroup are not supported.
- Only defects in the IJTAG scan network are supported.
- Suspect analysis using the analyze_icl_suspect command assumes a single fault model.
- ATE support for IJTAG network diagnosis is not supported.
- The GUI does not currently display the defective and suspect scan elements.

Overview of Tessent IJTAG Conventions

IJTAG networks are composed of sets of reconfigurable scan chains. Testing a network is complicated because in addition to testing the scan chains for defects, you must also test the additional network elements that enable network reconfiguration.

This discussion uses the following terminology. For details about Tessent IJTAG, refer to the *Tessent IJTAG User's Manual*.

- TDR — Test data register. A sequence of flops used to interface with instruments.
- SIB — Segment insertion bit. Used to build hierarchical IJTAG networks and consists of one or more bit shift-update registers and a mux. A SIB has one client and one or more host interfaces, where the number of hosts is dependent on the length of the shift-update register.

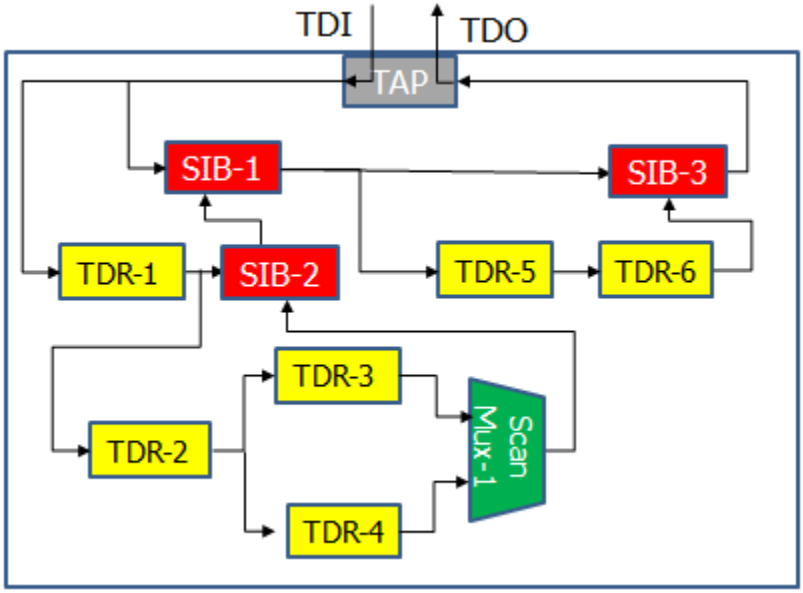
  If the SIB is asserted, it is in host mode. The input to the SIB passes through the host subnet to scan out (TDO). If the SIB is de-asserted, it is in client mode and shifts from scan in (TDI) to scan out (TDO), bypassing the host sub-network. By default, SIBs are set to client mode when the network is reset (set_icl_network -reset_current_scan_path) or at the beginning of a debug session.

- Scan mux — A multi-input mux with a shift-update register that enables choosing one of <*N*> input TDRs to be sent out. The number of input TDRs is dependent on the length of shift-update register.
- Scan elements — The TDRs, SIBs, and scan muxes in an IJTAG network. By default, scan elements are initialized to pick their 0 inputs.
- Scan path — A sequence of scan elements that start from the primary scan input pin (TDI), traverses a path through the network, and ends with the scan primary output pin (TDO). The default setup path is the shortest scan path from TDI to TDO.

Any scan element may be defective. For example, one or more flops in a TDR may be stuck at 0 or 1. A SIB may be stuck at 0 or 1. A shift-update register in a scan mux may be stuck at 0 or 1, resulting in the selection of the same input no matter what it is programmed to do. Additionally, employing bit patterns as flush patterns may detect other defects such as slow-to-rise, slow-to-fall, and fast-to-rise.

The following figure shows a simple IJTAG network with six TDRs, three SIBs, and one scan mux.

Figure 4-7. IJTAG Network Example



The following convention denotes the state of a given SIB or scan mux in a scan path:

{<*sib_name*> | <*scanmux_name*>}@{0 | 1}

SIB-1@0 indicates that SIB-1 is programmed to pick the client input path, while SIB-1@1 indicates that SIB-1 is programmed to pick the host path. ScanMux-1@0 indicates that the scan mux is programmed to pick the first path (that is, the top path).

The following examples illustrate valid scan paths:

```
SIB-1@0, SIB-3@0

SIB-1@0, TDR-5, TDR-6, SIB-3@1

TDR-1, TDR-2, TDR-4, ScanMux-1@1, SIB-2@1, SIB-1@1, SIB-3@0
```

Diagnosing and Debugging an IJTAG Network

Running and diagnosing IJTAG network tests in the Tessent SiliconInsight GUI follows a similar process as running and diagnosing Tessent LogicBIST and Tessent MemoryBIST instrument tests. For debugging, you use the command line.

Prerequisites

- You must have a signed-off TSDB or a top-level ICL file.
- You have generated and customized a setup specification as described in "Prepare the Design Under Test" on page 75.
- For simulation with SimDUT, you also need all the design files and libraries required by the simulator.

Procedure

1. Invoke Tessent SiliconInsight with Tessent Shell, load the TSDBs and setup specification, and start the Tessent SiliconInsight GUI.

   Ensure that you set the context as follows:

   ```
   set_context patterns -silicon_insight -ijtag
   ```

2. In the **Config Data Browser** tab, expand the loaded patterns specification in the Config data wrappers pane. Select the Patterns(ICLNetwork) wrapper.

3. Click **Process/Execute** () to run the test.

4. Click **Diagnose** () to diagnose the failures.

   The default name of the test is ICLNetwork. The results include a list of possible suspects for defects along the network.

   You can also use the following command to run and diagnose the IJTAG network:

   ```
   execute_cdp_test ICLNetwork -force_diagnose_on_failure \
     -collect_data_type_list { variable diagnosis }
   ```

5. Analyze each suspect listed in the diagnosis results by specifying the analyze_icl_suspect command.

   The following example shows a result that indicates that the behavior of the specified register suspect indicates that it has stuck-at-0 defect.

analyze_icl_suspect block1_I1.tdr1.R[7:0] –report

```
block1_I1.tdr1.R[7:0]    register_stuck@0
```

The following example shows a result that indicates the tool could not determine whether the specified SIB suspect was stuck at 1.

analyze_icl_suspect block1_I1.sib –report

```
block1_I1.sib1@1         ?
```

6. (*Optional*) For indeterminate results for suspects, examine the IJTAG network in more detail using the manual debugging commands as described in "Manually Debugging an IJTAG Network" on page 144.

Results

The results of the ICLNetwork test provide a list of suspect scan elements. The following example shows that there are two suspects to analyze. The first one is when block1_I1.sib is in host mode, and the other is the TDR register block1_I1.tdr1.R[7:0].

```
Good: block1_I1.sib1@0 block1_I1.sib2@0 block1_I1.sib2@1
 block1_I1.tdr2.R[7:0] block1_I2.sib1@0 block1_I2.sib1@1
 block1_I2.sib2@0 block1_I2.sib2@1 block1_I2.tdr1.R[7:0]
 block1_I2.tdr2.R[7:0] block2_I1.tdr.R[10:0] block3_I1.Selector[1:0]
 block3_I1.M1@0 block3_I1.M1@1 block3_I1.M1@2 block3_I1.M1@3
 block3_I1.tdr1.R[7:0]
...

Suspect: block1_I1.sib1@1 block1_I1.tdr1.R[7:0]
```

Manually Debugging an IJTAG Network

When you receive indeterminate results for suspects, as denoted by question marks (?), you can investigate the IJTAG network manually to verify whether the suspect is actually a defect.

Commands such as report_icl_network, assert_icl_sib, deassert_icl_sib, and set_icl_scan_mux enable you to textually "visualize" the IJTAG network. You can traverse the network opening and closing various hierarchies for better understanding of how the network is constructed. You can also introspect each node to understand its properties by using the report_icl_network command.

Prerequisites

- You have successfully run an ICLNetwork test in Tessent SiliconInsight Desktop and have diagnosis results that include a list of suspects.

Procedure

1. Explore the network.

The following commands are available for examining the network:

- get_icl_network
- get_icl_scan_mux
- report_icl_network

2. Configure a valid scan path through the network from TDI to TDO.

   The following commands are available:

   - assert_icl_sib
   - deassert_icl_sib
   - isolate_icl_node
   - set_icl_network
   - set_icl_scan_mux

   Specify the analyze_icl_suspects -verify_scan_path command to verify the scan path you have configured. Based on the pass or failure result, repeat this step as many times as required with different scan paths.

3. Generate your own algorithmic sequence to perform the analysis.

   At any point, you may need to analyze the scan paths for suspects in greater depth than performed by the analyze_icl_suspects -verify_scan_path command. The following utility commands are available:

   - create_icl_setup_pattern — Generates a setup pattern that configures the IJTAG network to use the scan path you defined.
   - create_icl_flush_pattern — Generates a flush pattern to test the IJTAG network along the scan path you defined.

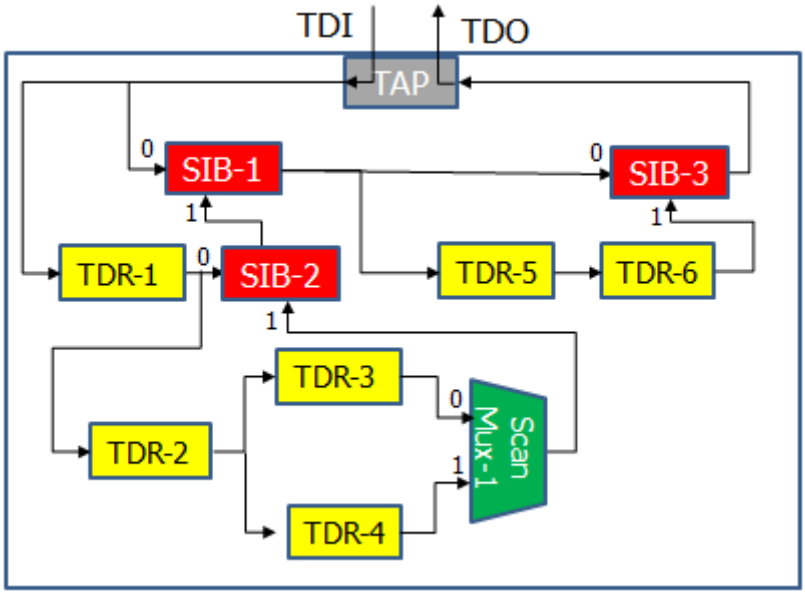
   The analyze_icl_suspects –verify_scan_path command calls these two utility commands.

Examples

This example illustrates how to configure a valid scan path. In the following design, the default setup scan path is:

```
SIB-1@0, SIB-3@0
```

Figure 4-8. Scan Path Example for Manual IJTAG Network Debugging



The default setup path is the shortest path from TDI to TDO with the scan elements in 0 states. Suppose you want to examine the scan path:

```
TDR-1, TDR-2, TDR-4, ScanMux-1@1, SIB-2@1, SIB-1@1, SIB-3@0
```

Specify the following commands:

```
assert_icl_sib SIB-1
set_icl_scan_mux -selected_input 1
assert_icl_sib SIB-2
```

Debugging Usage Examples

The following examples illustrate debugging IJTAG networks.

Example 1: Specifying a Scan Path

In the following design, the default setup scan path is:

```
SIB-1@0, SIB-3@0
```

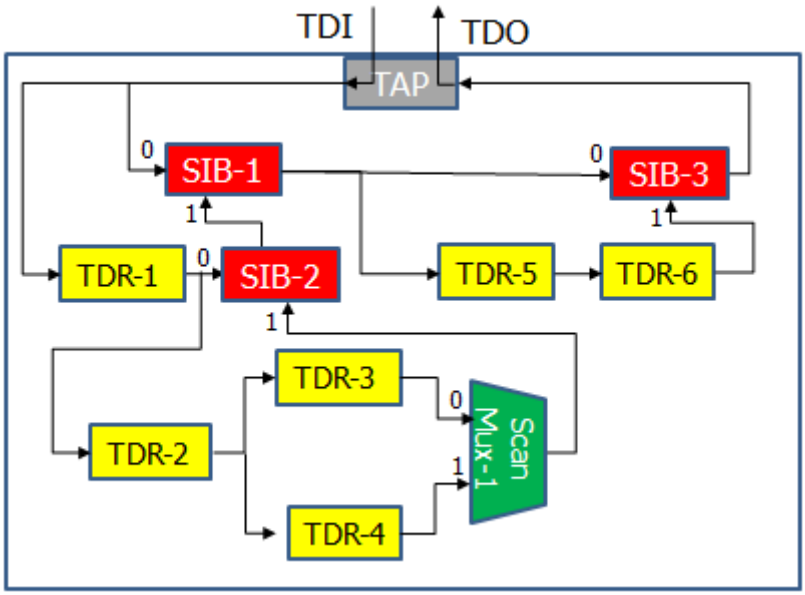
The default setup path is the shortest path from TDI to TDO with the scan elements in 0 states. Suppose you want to examine the scan path:

```
TDR-1, TDR-2, TDR-4, ScanMux-1@1, SIB-2@1, SIB-1@1, SIB-3@0
```

Specify the following commands:

```
assert_icl_sib SIB-1
set_icl_scan_mux -selected_input 1
assert_icl_sib SIB-2
```

Figure 4-9. Scan Path Example for Specifying a Scan Path



Example 2: Overshift Technique With SIB or Scan Mux Suspect

You can use the overshift technique when debugging potential SIB and scan mux suspects to determine what state they are stuck at. After exploring the network with the report_icl_network and get_icl_network commands, check if a given SIB or scan mux is stuck at a given state. To do this, configure the IJTAG network for a minimum length scan path with the suspect SIB or scan mux in a stuck state, whether 0 or 1. Next, generate a flush pattern with a longer length than if the SIB or scan mux were stuck in the opposite state. Run the flush pattern and check the output at TDO. If the flush pattern is observed at TDO, then you have confirmed that the SIB or scan mux is stuck at that state.

In the following example, suppose you would like to diagnose whether SIB-3 is stuck at 0 or stuck at 1. The stuck-at-0 path is:

```
SIB-1@0, SIB-3@0
```

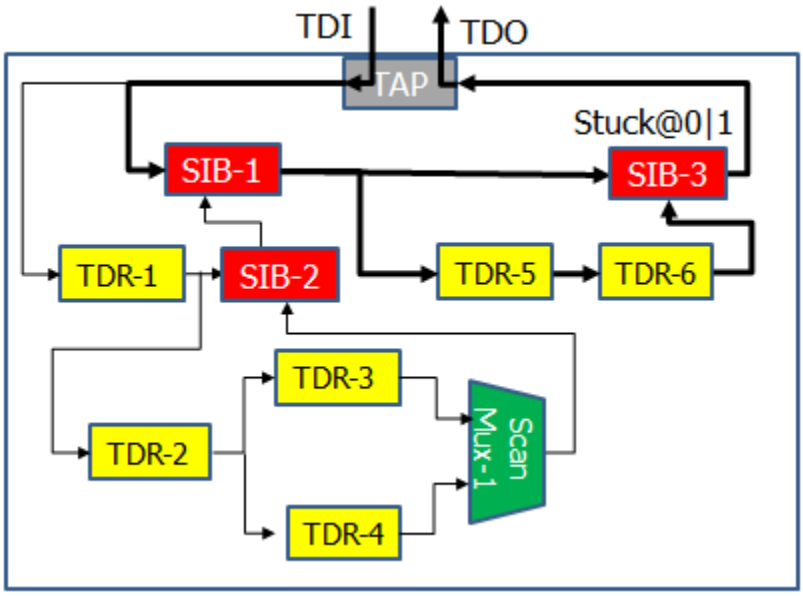
The stuck-at-1 path is:

```
SIB-1@0, TDR-5, TDR-6, SIB-3@1
```

If SIB-3 is stuck at 0, it fails when you try to assert it because the flush pattern 0011001100 arrives at TDO too soon. If SIB-3 is stuck at 1, it fails when you try to de-assert it because the

flush pattern 0011001100 arrives at TDO too late. You can detect either of these conditions by flushing an appropriately long bit pattern 001100110011 and checking when it appears at TDO.

Figure 4-10. Debug SIB Suspect



Example 3: Overshift Technique With TDR Suspect

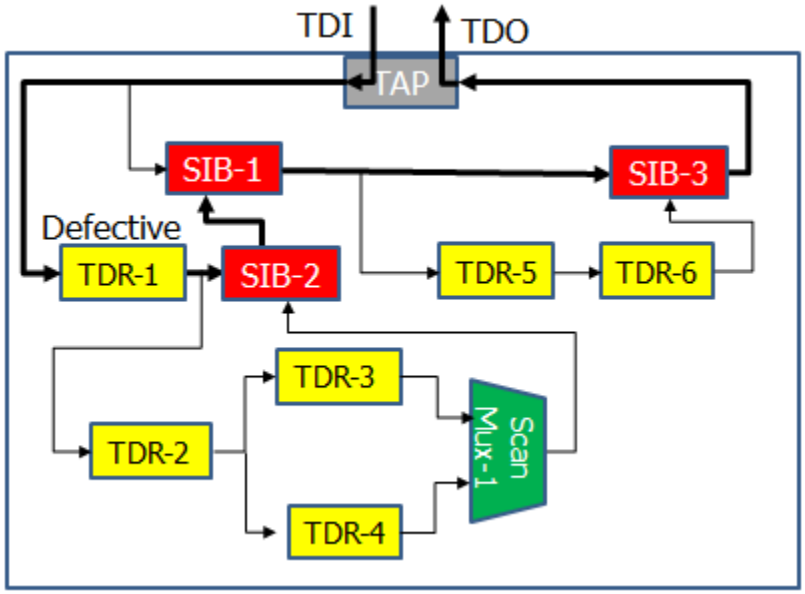
You can also use the overshift technique to determine if there is a discrepancy between the TDR length specified in ICL and the actual length of that TDR in the design.

In the following example, suppose you would like to check if suspect TDR-1 is bad due to a length discrepancy between ICL and the design. The minimum scan path is:

```
TDR-1, SIB-2@0, SIB-1@1, SIB-3@0
```

If the TDR-1 length in the design is smaller than the ICL length, the TDR fails because the flush pattern 0011001100 arrives at TDO too soon. If the length of TDR-3 in the design is longer than the length specified in ICL, it fails because the flush pattern 0011001100 arrives at TDO too late. You can detect either of these conditions by flushing an appropriately long bit pattern such as 001100110011 and checking when it appears at TDO. You can also calculate the actual length of the TDR (in the design) by calculating when the pattern arrives.

Figure 4-11. Debug TDR Suspect



PDL-Level Debugging for Tessent IJTAG-Inserted Devices

Tessent SiliconInsight enables you to test and diagnose third-party IJTAG devices inserted with Tessent IJTAG, as well as Tessent MemoryBIST and LogicBIST instruments, at the PDL level, which in turn enables you to debug your devices faster than manually generating patterns and running them on the tester.

In Desktop mode, you can use interactive IJTAG for bring-up, validation and failure diagnosis to isolate root cause defects. In addition, in SimDUT mode, interactive IJTAG is useful for pre-silicon verification.

Prerequisites

- You have a signed-off TSDB.
- You have a setup specification as described in "Generating the Setup Specification for Desktop Mode (TSDB Flow)" on page 76.

Procedure

1. To enable PDL-level diagnosis and debugging, run the following command after you have set the context, loaded the TSDBs and setup specification, and started Tessent SiliconInsight as described in "Performing Diagnosis and Interpreting the Results (TSDB Flow)" on page 133:

   SETUP>set_silicon_insight_option -interactive_ijtag on

2. Run PDL commands directly in the command console or run a dofile with a sequence of PDL commands. After specifying the iApply command, the set of PDL commands leading up to the iApply command are converted to chip-level patterns and run on the DUT. For example:

   ```
   ANALYSIS> iWrite register 0b1000

   ANALYSIS> iRead register 0b0001

   ANALYSIS> iApply
   ```

   For details about the PDL commands that Tessent Shell, and thus Tessent SiliconInsight, support, refer to the *Tessent Shell Reference Manual*.

Results

In this example, the tool runs the chip-level patterns corresponding to the iWrite and iRead commands and validates the chip-level response corresponding to the iRead command. The tool returns any mismatches between the chip-level response and the actual data corresponding to the register level "0b0001." If there is no mismatch, the tool shows no data.

PDL-Level Debugging for Non-Tessent IJTAG-Inserted Devices

You can generate patterns, run tests, diagnose, and debug any IJTAG network if you have the top-level ICL. You do not need a TSDB. Instead, you read in the top-level ICL.

Prerequisites

- You have a top-level ICL that describes the IJTAG network.
- You have a setup specification as described in "Generating the Setup Specification for Desktop Mode (TSDB Flow)" on page 76.

Procedure

1. After invoking Tessent Shell, set the tool context to enable Tessent SiliconInsight within Tessent Shell:

   ```
   SETUP>set_context patterns -silicon_insight
   ```

2. Read in the top-level ICL and then set the current design to the ICL module you plan to work with. For example:

   ```
   SETUP>read_icl <top_level_icl_file>
   SETUP>set_current_design <design_module_in_icl>
   ```

3. Load the setup specification as follows:

   ```
   SETUP>read_config_data <setup_spec_path>
   ```

4. To debug any issues with the IJTAG network, enable interactive IJTAG.

```
SETUP>set_silicon_insight_option -interactive_ijtag on
```

5. Run PDL commands directly in the command console or run a dofile with a sequence of PDL commands.

   Refer to "PDL-Level Debugging for Tessent IJTAG-Inserted Devices" on page 149 for more information.

PDL Commands Supported in Interactive IJTAG Mode

Interactive IJTAG mode supports level 0 and level 1 PDL commands as described by the IEEE 1687 standard.

Interactive IJTAG mode supports the following level 0 PDL commands.

Table 4-1. Level 0 PDL Commands

Command	Parameters	Purpose
iPDLLevel	('0' | '1') '-version STD_1687_2014'	Identify PDL flavor.
iPrefix	instance_path	Specify hierarchical prefix.
iReset	'-sync'?	Reset the network.
iWrite	(register | port | alias) value	Queue data to be written.
iRead	(register | port | alias) value	Queue data to be read.
iScan	scanInterface_name '-ir'? length '-si' siData '-so' soData	Queue data to be scanned.
iOverrideScanInterface	<scanInterfaceList> -captureEn (on) | off -updateEn (on) | off -broadcast on | (off)	Indicate the capture, update, and broadcast behavior to be imposed on a list of scan interfaces.
iApply	[-together]	Run queued operations.
iClock	ClockPort	Specify a system clock, which is required to be running.
iClockOverride	ToClockPort '-source' clockPort -freqMultiplier mult -freqDivider div -period int [unit]	Override definition of system clock when it is generated on-chip.
iRunLoop	(cyclecount ('-tck' | '-sck' port)? | '-time' value	Pulse a number of clocks.

Table 4-1. Level 0 PDL Commands (cont.)

Command	Parameters	Purpose
iProc	procName '{' arguments* '}' '{'commands+'}'	Wrapper for a PDL procedure.
iProcsForModule	[namespace::]moduleName - iProcNameSpace nameSpace_name	Identify the module in the ICL with which subsequent iProcs are associated.

Interactive IJTAG mode supports the following level 1PDL commands.

Table 4-2. Level 1 PDL Commands

Command	Parameters	Purpose
iGetReadData	register | port | alias | ScanInterface [-bin | -hex | -dec]	Return the value from the most recently applied iRead operation on a register or output port (or an alias consisting of either or both). May contain x-values.
iGetMiscompares	register | port | alias | ScanInterface [-bin | -hex | -dec]	Return the XOR of the value from the most recently applied iRead operation on a register or output port (or an alias consisting of either or both) and the value expected for that iRead operation. May contain x-values.
iGetStatus	[-clear]	Return the decimal number of iApply miscompares that have occurred since the last iGetStatus -clear command. Clear the count afterwards if directed.
iSetFail	message [-quit]	Return the message string to the controlling program to indicate an unexpected condition, with optional directive to abort execution.

Related Topics

Interactive IJTAG Tutorial

Running Non-Tessent IJTAG Instruments

You can use Tessent SiliconInsight to generate patterns for any third-party or standalone instrument as long as you have the PDL for the instrument. The process for running an IJTAG test pattern uses a test pattern that you have saved to SVF, stores it in the CDP, creates a test for the test pattern, and runs the test on the DUT. This process does not use a TSDB.

Prerequisites

- You have configured your computer as described in "Prepare the Tessent SiliconInsight Desktop Environment" on page 19.
- You have generated and customized a setup specification as described in "Generating the Setup Specification for Desktop Mode (TSDB Flow)" on page 76, excluding loading TSDBs, which are not required for the following procedure.
- You have PDL for the instrument you want to test.

Procedure

1. From a shell, invoke Tessent Shell using the following syntax:

   ```
   % tessent -shell
   ```

2. Set the tool context to enable Tessent SiliconInsight within Tessent Shell:

   SETUP>set_context patterns -ijtag -silicon_insight

   You do not need the -ijtag switch if you are not generating patterns in the current session.

3. Read in the setup specification. For example:

   SETUP>read_config_data my_config_file.cfg

4. Start Tessent SiliconInsight:

   SETUP>start_silicon_insight

5. Generate your IJTAG test pattern and save it as an annotated SVF test pattern with the write_patterns command.

   Refer to "Writing PDL, Pattern, and Testbench Files" in the *Tessent IJTAG User's Manual*.

6. Add the SVF pattern file to the CDP and create a test for that pattern file. For example:

   SETUP>add_cdp_test test1 -pattern my_pattern_path.svf

   The test manipulates and runs test pattern files. Currently, you can only specify one test pattern file per test.

   To view your tests, specify the get_cdp_test_list command. To delete a test, specify the delete_cdp_test command.

7. Optionally, use a GPIB instrument to change settings such as the power supply. You must use instrument-specific commands along with the execute_gpib_command.
8. Run the test. For example:

   SETUP>execute_cdp_test test1 -collect_data_type_list variable

   This command runs the test named “test1” on the Desktop. It collects the failures during test pattern execution and maps the failures into variable cycles as defined in the annotated SVF pattern file.

   If you do not specify any switch, the tool runs the test and returns the pass/fail status for the specified test.

   The execute_cdp_test command also stores the data as a config data object that you can introspect using the report_config_data command.
9. Shut down Tessent SiliconInsight.

   SETUP>stop_silicon_insight

Results

After running a given test, Tessent SiliconInsight Desktop generates a tabulated report. The following example shows the results for the test pattern associated with the test named “test1.”

```
// command: execute_cdp_test test1 -collect_data_type_list variable
// Test 'test1' failed.
// Variable failures and unmapped failures of pattern
// 'DLV2_1S0_ETMemory_Full.svf' :
//
// PatternSet Variable Pin Expected
// Actual
// ---------- ---------------------------------- --- --------
// pattern_1 MyBlock1.MyBlock1Instrument.go(1) tdo b1
// b0
// ---------- ---------------------------------- --- --------
// pattern_1 MyBlock1.MyBlock1Instrument.done(1) tdo b1
// b0
// ---------- ---------------------------------- --- --------
//
// Command ID Bit Offsets
// ---------- -----------
// 4 0
// ---------- -----------
```

Related Topics

Examples of Controlling Instruments With GPIB

Examples of Controlling Instruments With GPIB

You can control various settings by using GPIB instruments, such as power supplies and clock generators, with Tessent SiliconInsight Desktop.

Refer to "execute_gpib_command" in the *Tessent Shell Reference Manual* for details.

Example 1

In the following example, the GPIB power supply sets the pin voltage levels to 3.9V. The power supply was configured to listen at address 19.

```
SETUP>execute_gpib_command “SetVoltage 3.9V” -address 19
```

Example 2

The following example sets the voltage for an Agilent Power Supply (connected to GPIB address 19) to 1.03V.

```
SETUP>execute_gpib_command "SOURCE:VOLTAGE:LEVEL:IMMEDIATE:AMPLITUDE 1.03"
-address 19
```

Example 3

The following example measures DC voltage on an Agilent 34410A/11A digital multimetre and assigns the voltage to a variable.

```
execute_gpib_command \ "MEASURE:VOLTAGE:DC? AUTO" -address 22 -read_result
```

Alternatively, you can also create a tcl variable to catch the output value, as follows:

```
set voltage_value [ regsub {// Note :} [ catch_output {execute_gpib_command \
"MEASURE:VOLTAGE:DC? AUTO" -address 22 -read_result} ] "" }
```

Example 4

The following example sets the low voltage level for the clock pulse (0) for a Stanford ResearchSystem (SRS) CG635 clock generator (connected to GPIB address 6) to 1.5V. It also sets the high voltage for the clock pulse (1) to 2.7V. Thus, the clock pulse swings between 1.5V to 2.7V.

```
SETUP>execute_gpib_command "QOUT 0,1" -address 19
SETUP>execute_gpib_command "QOUT 1,2.7" -address 19
```

Example 5

The following example sets the frequency for the same clock generator (SRS CG635 at GPIB address 6) to 25 MHZ.

```
SETUP>execute_gpib_command "FREQ 25000000" -address 19
```

Chapter 5
Support for ATPG (Non-TSDB Flow)

In Desktop mode, Tessent SiliconInsight supports testing and diagnosing flat and hierarchical ATPG patterns. The tool automatically diagnoses test failures down to failing cells or flops. It can also diagnose chain failures.

Note

ATPG pattern testing and diagnosis in Tessent SiliconInsight is not based on the TSDB. This means you have to use dedicated commands to specify design and pattern data.

ATPG Test Step Wrappers

Tessent Shell provides two ATPG test step wrappers that you specify when generating a PatternsSpecification wrapper for ATPG: UserDefinedATPG and UserDefinedRetargetedATPG. In addition, for hierarchical designs, you can use the CoreModule wrapper to define the cores.

- **UserDefinedATPG —** For flat designs, adds a flat ATPG pattern to the PatternsSpecification
- **UserDefinedRetargetedATPG —** For hierarchical designs, adds a retargeted ATPG pattern to the PatternsSpecification
- **CoreModule —** Defines each core module within the retargeted ATPG test; located beneath UserDefinedRetargetedATPG

Table 5-1. Wrapper Properties for ATPG

Wrapper	Required Property	Optional Property
UserDefinedATPG:	scan_pattern_file: Top-level ATPG pattern file.	layout_database: Top-level LDB, which enables the tool to perform layout-aware diagnosis with Tessent Diagnosis.
	flat_model_file: Flat model for the top-level design.	support_pattern_files: The pattern file or files, compressed or uncompressed, that are included in the scan_pattern file using the STIL 2005 Include statement. **Note:** The pattern names indicated in the value for this property must reflect exactly how those patterns are included in the main STIL pattern file. For example, if the name is included (with an 'Include' statement) as *my.pat.gz*, this property must specify it as *my.pat.gz*; similarly, if it is *my.pat* in the STIL file, it must also be *my.pat* in this property.

Table 5-1. Wrapper Properties for ATPG (cont.)

Wrapper	Required Property	Optional Property
UserDefinedRetargeted ATPG:	scan_pattern_file: Top-level ATPG pattern file.	layout_database: One LDB per CoreModule, which enables the tool to perform layout-aware diagnosis with Tessent Diagnosis.
	tcd_file: Chip-level TCD file for the retargeted ATPG test patterns. For each core in the TCD file, the tool automatically creates the CoreModule wrappers.	support_pattern_files: The pattern file or files, compressed or uncompressed, that are included in the scan_pattern file using the STIL 2005 Include statement. **Note:** The pattern names indicated in the value for this property must reflect exactly how those patterns are included in the main STIL pattern file. For example, if the name is included (with an 'Include' statement) as *my.pat.gz*, this property must specify it as *my.pat.gz*; similarly, if it is *my.pat* in the STIL file, it must also be *my.pat* in this property.
UserDefinedRetargeted ATPG/CoreModule:	pattern_file: ATPG pattern file for the specified core.	layout_database: Top-level LDB, which enables the tool to perform layout-aware diagnosis with Tessent Diagnosis.
	flat_model_file: Flat model for the specified core.	

PatternsSpecification Configuration Data Syntax

For the ATPG usage scenario within Tessent SiliconInsight, the PatternsSpecification/Patterns/TestStep configuration data syntax is:

```
TestStep(<name>) {
  ...
  CharacterizationDebugOptions { ...
  }
  UserDefinedATPG {
    scan_pattern_file     : <scan_pattern_file> ;
    flat_model_file       : <flat_model_file>   ;
    layout_database       : <layout_database>   ;
    support_pattern_files : <support_pattern_file> ... ;


    CharacterizationDebugOptions { ...
    }
  }
  UserDefinedRetargetedATPG {
    scan_pattern_file     : <scan_pattern_file> ;
    tcd_file              : <file_path> ;
    support_pattern_files : <support_pattern_file> ... ;
    CharacterizationDebugOptions { ...
    }
    CoreModule(module_name) {
      flat_model_file     : <file_path>       ;
      pattern_file        : <file_path>       ;
      layout_database     : <layout_database> ;
    }
  }
  .
  .
  .
  }
```

For details about the properties you can adjust within the CharacterizationDebugOptions wrappers, refer to "Characterization Debug Options" on page 171.

Restrictions

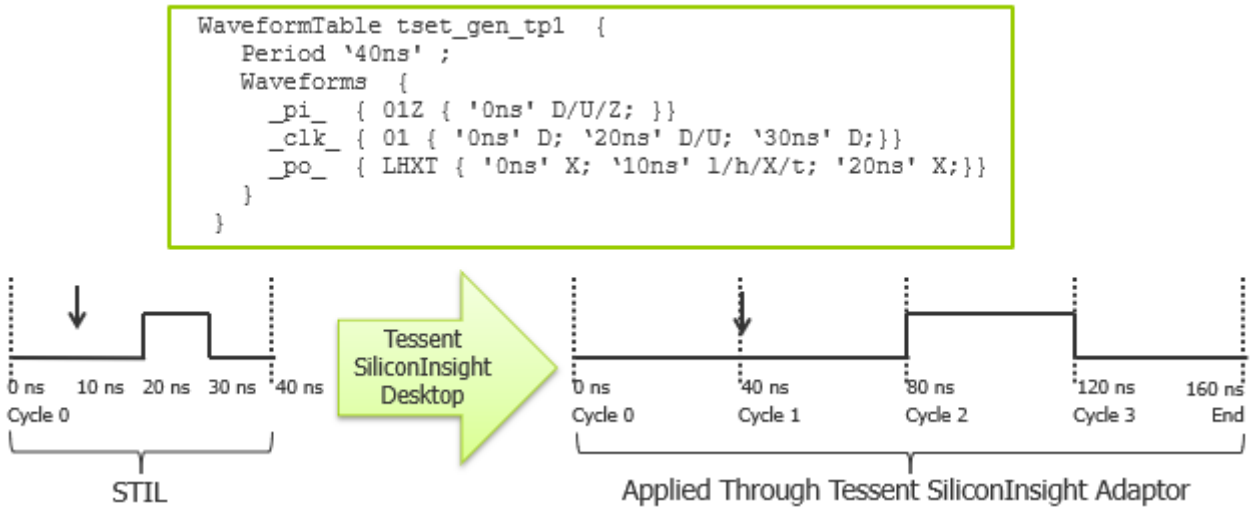
The following restrictions apply to ATPG wrappers in a given PatternsSpecification:

- You can only specify one ATPG wrapper per TestStep wrapper. Choose either UserDefinedATPG or UserDefinedRetargetedATPG. For more information, refer to "TestStep" in the *Tessent Shell Reference Manual*.
- When a TestStep contains an ATPG wrapper, no other TestStep is permitted in the Patterns wrapper.
- If there is an ATPG wrapper anywhere in a PatternsSpecification, that PatternsSpecification may not contain any other instruments (MemoryBIST,

LogicBIST, and so on) in any of its wrappers. That is, the PatternsSpecification wrapper is used exclusively for ATPG testing.

STIL test pattern files contain WaveformTable sections that specify timing in waveforms.

Figure 5-1. STIL Test Pattern File Waveform Table



Tessent SiliconInsight Desktop honors the order of events within a waveform but ignores the exact timing. The actual timing depends on the following:

- Adaptor type.
- Base frequency set in the setup specification.
- Number of events in the WaveformTable,

The adaptor applies one event per cycle. The total period of a waveform and the frequency in which patterns are applied depends on the number of events or edges in the waveform. The effective frequency slows down the more edges you have in your waveforms.

Generating a Patterns Specification for ATPG

Tessent Shell provides ATPG test step wrappers that you specify when generating a PatternsSpecification wrapper for ATPG. Using Tessent Shell commands, you can generate a PatternsSpecification that contains all the data needed for the scan tests.

Prerequisites

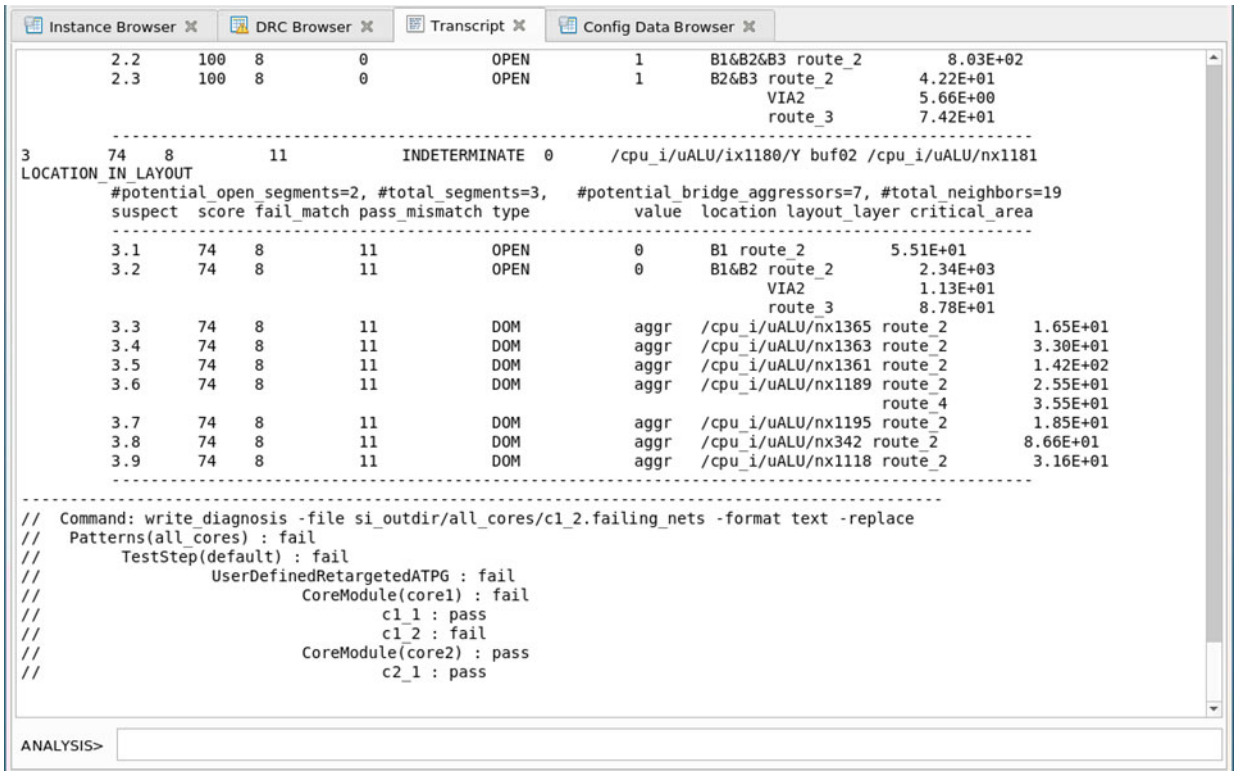
- You have prepared a setup specification as described in "Generating the Setup Specification for ATPG (Non-TSDB Flow) Desktop Mode" on page 79.

Procedure

1. In the **Transcript** tab, specify the add_config_element to build the test step wrapper.
2. In the **Transcript** tab, specify the set_config_value command to load the design data associated with that test step.
3. In the **Config Data Browser** tab, select a CoreModule node in the "Config data wrappers" pane on the left. Click the right arrows (▸) as needed to reveal the CoreModules if they are not visible.

Results

The following figure shows an example of what displays in the property table pane on the right when you select a CoreModule node.



Examples

The following example dofile generates a new PatternsSpecification for a hierarchical design with retargeted patterns and populates it with the required data. It then loads the setup specification and starts Tessent SiliconInsight in GUI mode.

```
set_context patterns -silicon_insight

# Create a brand new PatternsSpecification and set the usage to
# manufacturing_test.
set ps [add_config_element \
   /PatternsSpecification(top,gate,manufacturing)]
set_config_value -in_wrapper $ps usage manufacturing_test

# Add a flat ATPG test. This is the top-level test.
add_config_element -in_wrapper $ps Patterns(glue_logic)
add_config_element -in_wrapper \
   $ps Patterns(glue_logic)/TestStep(default)
set atpg [add_config_element -in_wrapper \
   $ps Patterns(glue_logic)/TestStep(default)/UserDefinedATPG]
set_config_value -in_wrapper $atpg scan_pattern_file \
   ./results/pat_glue_top.stil
set_config_value -in_wrapper $atpg flat_model_file
   ./results/chip_top_glue.flat

# Providing a layout database is optional.
set_config_value -in_wrapper $atpg layout_database ./results/top_glue.ldb

# The following sections only apply if you have retargeted patterns.
# If you only have chip-level patterns, skip to the write_config_data command.
# Add a hierarchical ATPG test.
add_config_element -in_wrapper $ps Patterns(all_cores)
add_config_element -in_wrapper $ps Patterns(all_cores)/TestStep(default)
```

```
set atpg [add_config_element -in_wrapper $ps \
   Patterns(all_cores)/TestStep(default)/UserDefinedRetargetedATPG]
set_config_value -in_wrapper $atpg scan_pattern_file \
   ./results/pat_core_top.stil
set_config_value -in_wrapper $atpg tcd_file ./results/chip.tcd

# Add CoreModule specific data for core module core1.
set_config_value -in_wrapper $atpg CoreModule(core1)/pattern_file \
   ./results/pat_edt_core1.ascii
set_config_value -in_wrapper $atpg CoreModule(core1)/flat_model_file \
   ./results/design_core1.flat
set_config_value -in_wrapper $atpg CoreModule(core1)/layout_database \
   ./results/core1.ldb

# Add CoreModule specific data for core module core2.
set_config_value -in_wrapper $atpg CoreModule(core2)/pattern_file \
   ./results/pat_edt_core2.ascii
set_config_value -in_wrapper $atpg CoreModule(core2)/flat_model_file \
   ./results/design_core2.flat
set_config_value -in_wrapper $atpg CoreModule(core2)/layout_database \
   ./results/core2.ldb.

# Store the PatternsSpecification.
write_config_data ./atpg_patterns_spec_bypass -wrapper $ps -replace

# PatternsSpecification in now complete, proceed to run the tool.
# Load the setup specification.
read_config_data <si setup spec>

# Start Tessent SiliconInsight.
start_silicon_insight -gui
```

Refer to the *Tessent Shell Reference Manual* for full details about PatternsSpecification wrappers.

Running and Diagnosing ATPG Patterns Usage Example

This usage example shows how to run logic tests and diagnose ATPG patterns. You can use the command line to limit the number of failing cycles collected during pattern execution.

For purposes of this usage example assume you have used the dofile as shown in "Generating a Patterns Specification for ATPG" on page 161.

Prerequisites

- You must have generated an ATPG PatternsSpecification. You can also read in a previously generated PatternsSpecification using the read_config_data command.

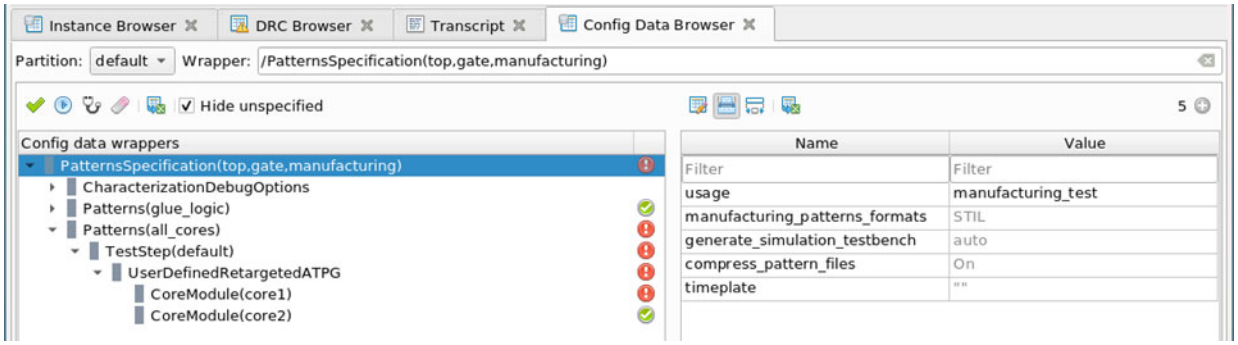
- If you want to run and diagnose a subset of ATPG patterns, write out the subset with the write_patterns command. For example:

```
set_context pattern -scan
read_flat_model design.v.flat
read_patterns pat.stil -stil
write_patterns subset_pat.stil -stil -begin N -end M -external -rep
```

Procedure

1. Run logic tests in the **Config Data Browser** tab.
   a. Select PatternsSpecification in the "Config data wrappers" pane on the left.
   b. Click the **Process/Execute** button ( ). There is a failing test.
   c. Click the right arrow ( ) beside PatternsSpecification to show its tests. Each test is a Patterns wrapper. Patterns(glue_logic) passed but Patterns(all_cores) failed.
   d. Click the right arrow beside Patterns(all_cores). Then click the right arrows beside TestStep(default) and UserDefinedRetargetedATPG.

   Tessent SiliconInsight runs the top-level patterns for all Patterns wrappers and returns the test results. The "Config data wrappers" pane shows that CoreModule(core1) failed while CoreModule(core2) passed. In the **Transcript** tab, the tool reports additional details of the c1_1 core instance pass and the c1_2 core instance failure for CoreModule(core1).



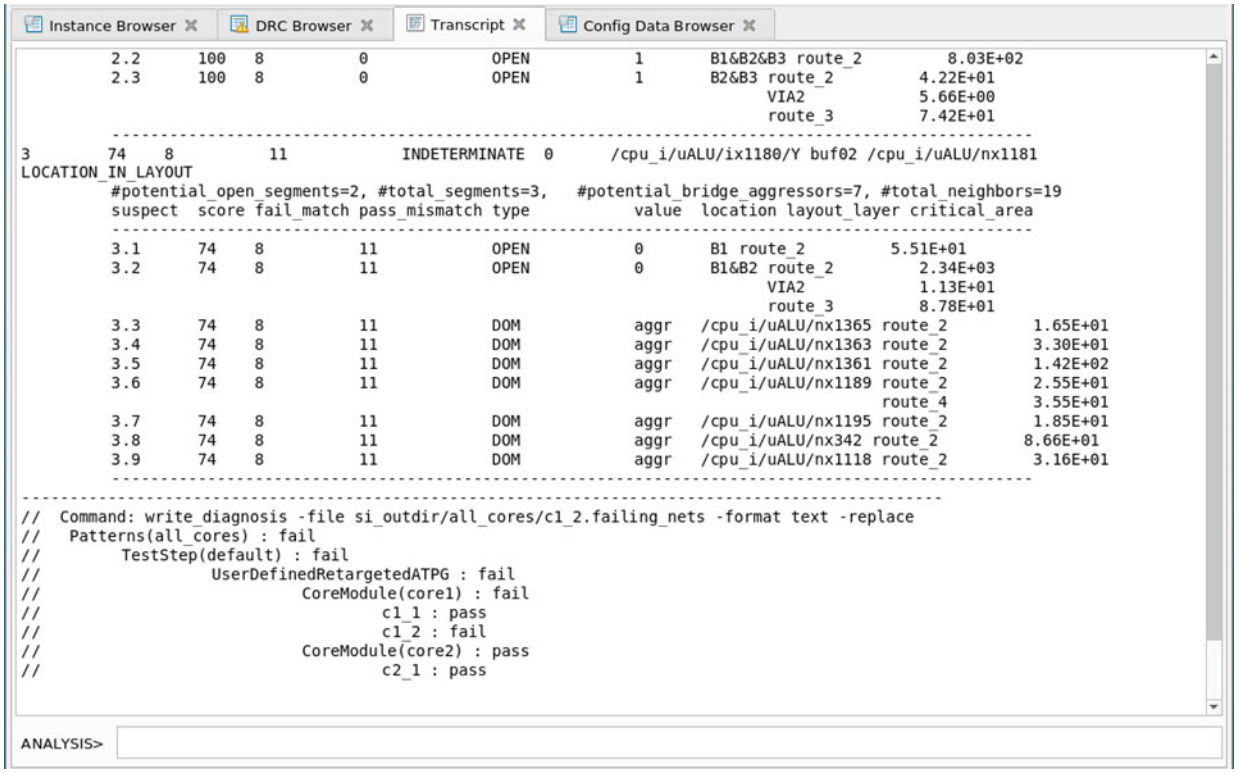
2. In the **Config Data Browser** tab, select Patterns(all_cores) and then click the **Diagnose** button ( ) to diagnose the failure.
3. (*optional*) To limit the number of raw failing cycles collected during pattern execution, use the command field in the **Transcript** tab. Run the following command in the command field beside the mode indicator label (SETUP>).

   set_silicon_insight_option -maximum_failing_cycles <number>

   This command is useful, for example, when there is a scan chain defect, which results in many cycle failures that do not need to be collected.

The next time you click the **Process/Execute** button in the **Config Data Browser** tab, the tool runs the full pattern but only collects the failing cycles for the specified subset.

4. Investigate the failing nets more closely.
   a. In the **Config Data Browser** tab, if the "Hide unspecified" option is active (☑ Hide unspecified), click it to deactivate.
   b. In the "Config data wrappers" pane, click the right arrow beside Patterns(all_cores) to reveal its CharacterizationDebugOptions.
   c. Click the right arrow beside CharacterizationDebugOptions, and then select ATPG.
   d. Double-click the diagnosis_result property in the property table pane on the right.
   e. A small Set Property Value window opens. Choose "failing_nets" and click **Apply**.
   f. Click the "Hide unspecified" box to activate the hiding of unspecified wrappers.
   g. Select Patterns(all_cores).
   h. Click the **Diagnose** button.
   i. View the diagnosis results in the **Transcript** tab.



5. (*Optional*) Run the diagnosis in the background while you continue working.

a. In the **Config Data Browser** tab, select the ATPG wrapper under CharacterizationDebugOptions of the Patterns(all_cores) test.

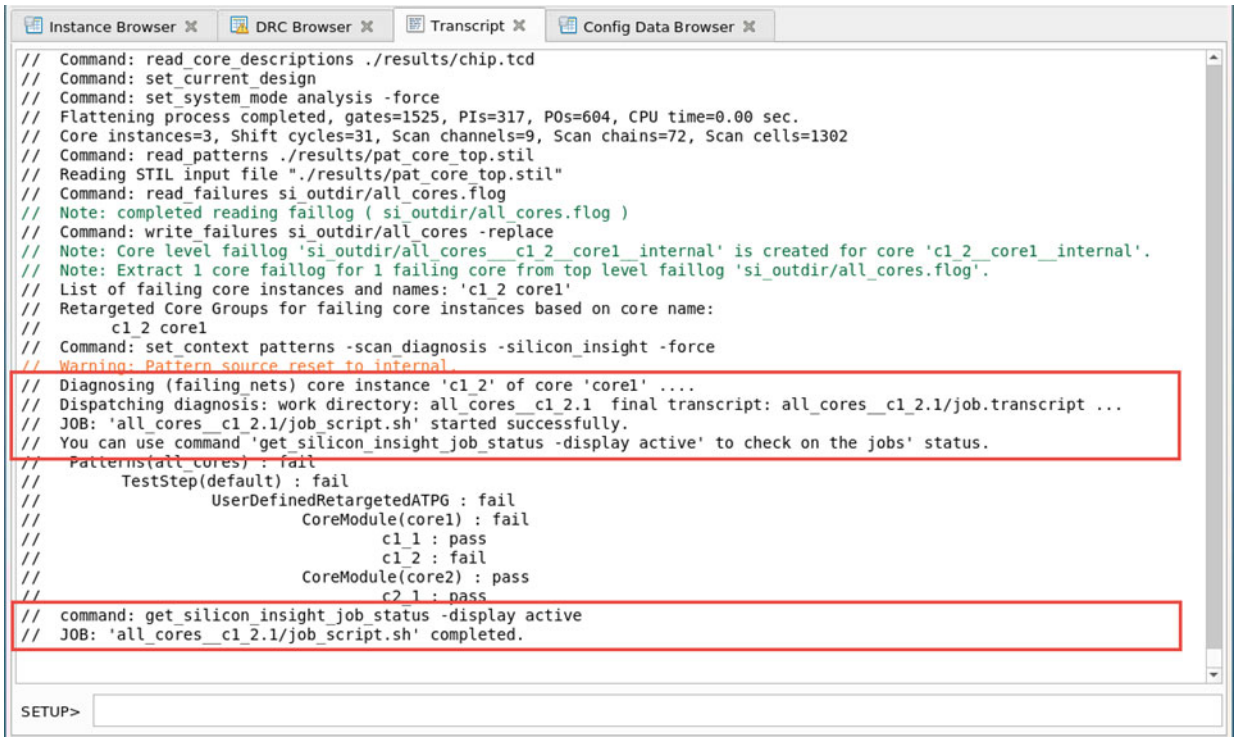
b. Double-click the wait_for_diagnosis_result property in the property table pane on the right.

c. A small Set Property Value window opens. Choose “off” and click **Apply**.

d. Select Patterns(all_cores).

e. Click the **Diagnose** button.

The tool returns a message in the transcript that it is dispatching diagnosis.



Refer to “user_log_tag” on page 199 for information about customizing the job directory name as listed after the JOB: entry in the transcript.

6. (*Optional*) To check on the diagnosis job status, use the “get_silicon_insight_job_status -display_active” command in the **Transcript** tab.

Related Topics

Characterization Debug Options

Dofile Example: Integrating Test and Pattern Generation

You can combine and automate test pattern generation, test execution, and result evaluation by incrementing (or decrementing) a test parameter of interest, generating new test patterns, running the patterns, and capturing the results in successive loops.

The following example generates ATPG patterns with an increasing shift-toggle rate for a design with low-power EDT controller until the patterns start to fail. The tool captures and reports the results.

```
# Set up SiliconInsight with pre-existing setup and pattern specification
set_context patterns -silicon_insight -scan
read_config_data ./SiliconInsightSetupSpec.cfg
read_config_data ./atpg_patterns_spec

start_silicon_insight -gui

# Run pre-existing patterns -- should pass
process_patterns_specification  -config_objects /
PatternsSpecification(top,gate,manufacturing)
execute_cdp_test stuck_at

# Read flat model and create new node for experiment
read_flat_model ./design.flat.gz
add_config_element /PatternsSpecification(top,gate,manufacturing)/
   Patterns(shift_power_experiment)
add_config_element /PatternsSpecification(top,gate,manufacturing)/
   Patterns(shift_power_experiment)/TestStep(default)
add_config_element/PatternsSpecification(top,gate,manufacturing)/
   Patterns(shift_power_experiment)/TestStep(default)/UserDefinedATPG

# Set  basic parameters for experiment
set toggle_rate 12
set toggle_reject [ expr $toggle_rate + 5 ]
set test_fail 0
set result_table {}
set incr_step 3

# START LOOP
# Generate patterns and test until patterns fail
#
while { $test_fail == 0 } {

  # Generate patterns with new toggle rate
  reset_state
  set_power_control shift on -sw $toggle_rate -rej $toggle_reject
  puts "// Generating patterns with shift toggle rate $toggle_rate"
  create_patterns
  report_power_metrics -shift
  write_patterns ./pat_experiment_${toggle_rate}_${toggle_reject}.stil -stil1999 -replace

  # Run newly generated test patterns
  set_config_value /PatternsSpecification(top,gate,manufacturing)/
     Patterns(shift_power_experiment)/TestStep(default)/UserDefinedATPG/
     scan_pattern_file ./pat_experiment_${toggle_rate}_${toggle_reject}.stil
  process_patterns_specification  -config_objects /
     PatternsSpecification(top,gate,manufacturing)/Patterns(shift_power_experiment)
  set test_fail [ execute_cdp_test shift_power_experiment ]

  # Capture results and settings for reporting
  report_power_metrics -shift > ./shift_power.rep
  set f [ open ./shift_power.rep ]
  set rep_table [ split [ regexp -all -inline {\S+} [ read $f ] ] ]
  close $f
  set avg_load [ lindex $rep_table 13 ]
  set avg_unload [ lindex $rep_table 19 ]
  if { $test_fail == 0 } { set flag "   PASS" } else { set flag "** FAIL **"}
  set result_line "   $toggle_rate   $toggle_reject   $avg_load   $avg_unload   $flag "
```

```
  lappend result_table $result_line

  # Increment toggle rate for next experiment
  set toggle_rate [ expr $toggle_rate + $incr_step ]
  set toggle_reject [ expr $toggle_reject + $incr_step ]
}

# Report results summary
puts ""
puts "-------------------------------------------------------------------------------"
puts "Results"
puts "-------------------------------------------------------------------------------"
puts "   ATPG setting shift              Actual average shift                Test"
puts "toggle_rate  reject_rate    load transitions   resp. transitions     result"
foreach line $result_table { puts $line }
puts "-------------------------------------------------------------------------------"
```

The output is formatted as follows:

```
-------------------------------------------------------------------------------
Results
-------------------------------------------------------------------------------
toggle_rate  reject_rate    load transitions   resp. transitions      result
    12            17           12.82%              13.69%              PASS
    15            20           15.04%              16.30%              PASS
    18            23           16.49%              18.15%              PASS
    21            26           18.17%              19.49%              PASS
    24            29           18.65%              20.33%              PASS
    27            32           19.18%              20.68%              PASS
    30            35           18.88%              20.58%              PASS
    33            38           19.17%              20.60%              PASS
    36            41           19.17%              20.52%           ** FAIL **
-------------------------------------------------------------------------------
```

Chapter 6
Characterization Debug Options

Tessent SiliconInsight uses the CharacterizationDebugOptions wrapper to define the diagnosis algorithms that Tessent SiliconInsight automatically runs when an instrument detects failures. The algorithms collect data to help you analyze root cause failures in the silicon.

These options apply to the LogicBIST, MemoryBIST, and ATPG instruments in Desktop mode.

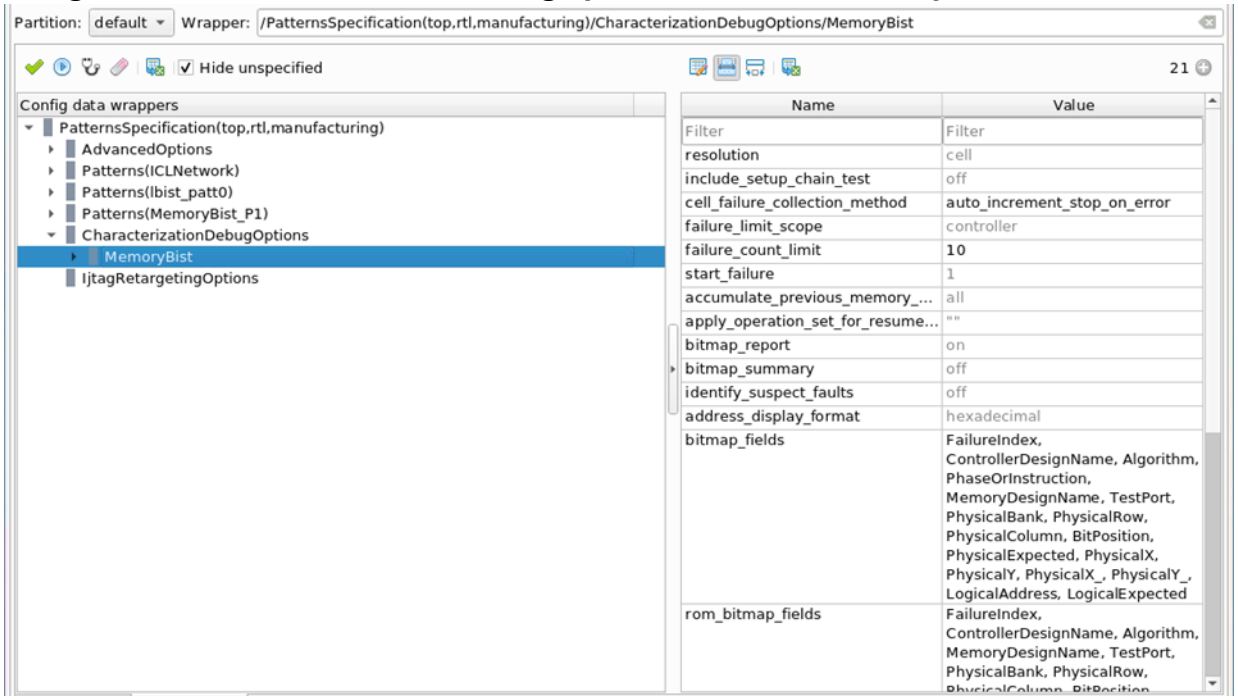
Characterization Debug Options Display in the GUI

The “Config data wrappers” pane in the GUI displays the characterization debug options.

You can configure the CharacterizationDebugOptions properties for different instrument types at different levels of the PatternsSpecification hierarchy. The children inherit the property settings downward in the hierarchy unless you specifically customize a setting at a particular level. Inherited and default settings display in gray, and customized settings display in black.

Figure 6-1 shows the CharacterizationDebugOptions wrapper for a MemoryBIST instrument at the PatternsSpecification level. To view the MemoryBIST settings, use the **Config Data Browser** tab and the “Config data wrappers” pane on the left. Click the right arrows (▸) beside the PatternsSpecification wrapper and then the CharacterizationDebugOptions wrapper. Select MemoryBIST to show its properties and values in the property table pane on the right.

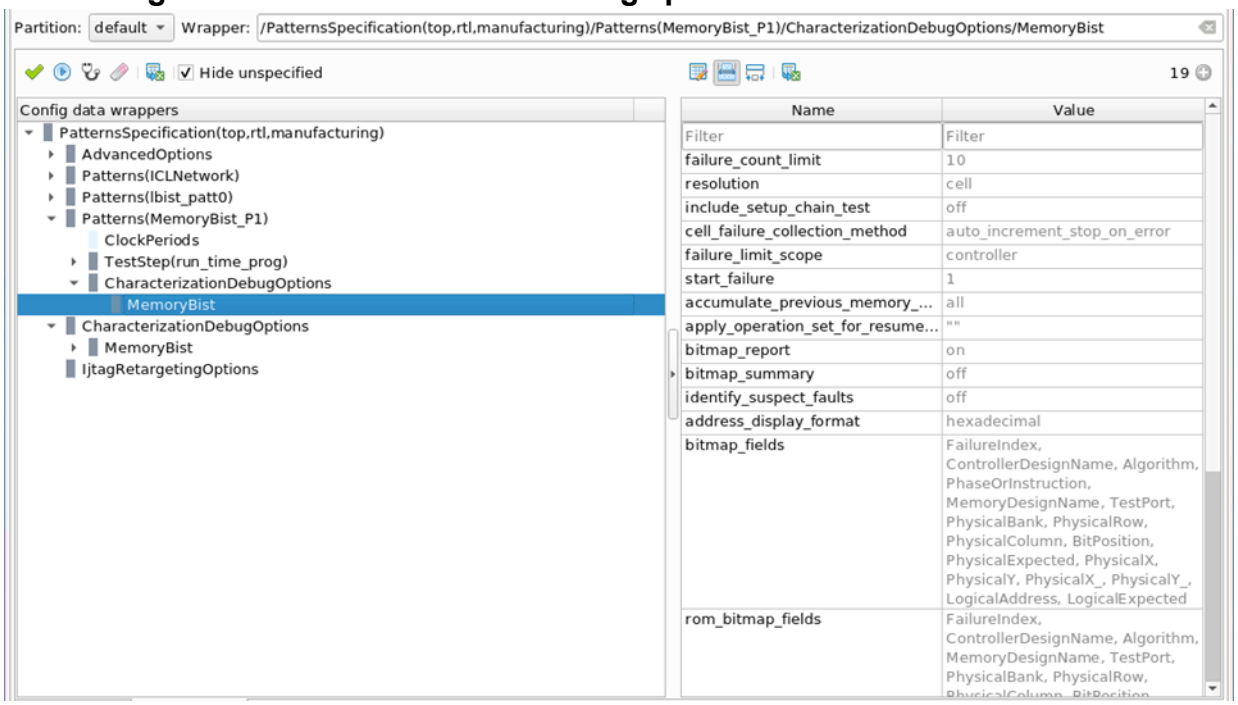
Figure 6-1. CharacterizationDebugOptions at the PatternsSpecification Level



This example shows that the diagnostic resolution is “cell”. In addition, the cell failure collection method is auto_increment_stop_on_error (ESOE), using a controller (from failure_limit_scope) with failure count limit of ten. Because diagnosis_on_failure is enabled (not shown but visible if you select the CharacterizationDebugOptions wrapper), when the test for this controller fails on the tester and the controller detects the failure, the tool uses the ESOE algorithm to collect up to ten failures per controller and bitmap all of the failures to failing cells.

Figure 6-2 shows the CharacterizationDebugOptions at the MemoryBIST instrument level. The settings display in gray because they were all inherited from the CharacterizationDebugOptions at the PatternsSpecification wrapper level.

Figure 6-2. CharacterizationDebugOptions at the Instrument Level



Characterization Debug Options Hierarchy in the Patterns Specification

In the GUI, you can configure the CharacterizationDebugOptions properties for different instrument types at different levels of the PatternsSpecification hierarchy. The property settings are inherited downward by the children in the hierarchy unless you specifically customize a setting at a particular level.

Within the PatternsSpecification configuration data syntax, the CharacterizationDebugOptions wrapper occurs within the following wrappers:

- PatternsSpecification
- Patterns
- TestStep
- TestStep/UserDefinedATPG and TestStep/UserDefinedRetargetedATPG
- TestStep/LogicBist/CoreInstance
- TestStep/LVMemoryBist/Controller
- TestStep/MemoryBist/Controller
- PatternsGroup

The following high-level view of the PatternsSpecification syntax shows the CharacterizationDebugOptions wrapper hierarchy for the instruments that Tessent SiliconInsight supports. The characterization debug options available for each instrument are described in "Characterization Debug Options Descriptions" on page 178.

```
PatternsSpecification(<design_name>,<design_id>,<pattern_id>) {
  usage                          : <usage>       ;
  manufacturing_patterns_formats : <format>, ... ;
  compress_pattern_files         : <boolean>     ;
  CharacterizationDebugOptions {
    diagnose_on_failure : <diagnose_on_failure> ; // default: Off
    ATPG {
    }
    IclNetwork {
    }
    LogicBist {
    }
    LVMemoryBist {
    }
    MemoryBist {
    }
  }
  SimulationOptions {
  }
  AdvancedOptions {
  }

  Patterns(<patterns_name>) {
    CharacterizationDebugOptions {
      diagnose_on_failure : <diagnose_on_failure> ; // default: Off
      ATPG {
      }
      IclNetwork {
      }
      LogicBist {
      }
      LVMemoryBist {
      }
      MemoryBist {
    }
    ClockPeriods {
    }
    .
    .
```

```
  TestStep(<name>) {
    ...
    CharacterizationDebugOptions {
      // Same as /Patterns/CharacterizationDebugOptions
    }
    UserDefinedATPG {
      CharacterizationDebugOptions {
        // Same as PatternsSpecification/CharacterizationDebugOptions
        // ATPG
      }
    }
    UserDefinedATPG {
      CharacterizationDebugOptions {
        // Same as PatternsSpecification/CharacterizationDebugOptions
        // ATPG
      }
    }
    LogicBist {
      CoreInstance {
        CharacterizationDebugOptions {
          // Same as PatternsSpecification/CharacterizationDebugOptions
          // LogicBist
        }
      }
    }
    LVMemoryBist {
      Controller {
        CharacterizationDebugOptions {
          // Same as PatternsSpecification/CharacterizationDebugOptions
          // LVMemoryBist
        }
      }
    }
    MemoryBist {
      Controller {
        CharacterizationDebugOptions {
          // Same as PatternsSpecification/CharacterizationDebugOptions
          // MemoryBist
        }
      }
    }
    .
    .
}
```

```
  PatternsGroup(<patterns_group_name>) {
    CharacterizationDebugOptions {
      diagnose_on_failure : <diagnose_on_failure> ; // default: Off
      LogicBist {
      }
      MemoryBist {
    }
    .
    .

  }
  .
  .
}
```

For details about the PatternsSpecification configuration data syntax, refer to "PatternsSpecification" in the *Tessent Shell Reference Manual*.

Characterization Debug Options Descriptions

Characterization debug options are specified within their applicable instrument wrappers: LogicBIST, MemoryBIST, and ATPG. In addition, the LVMemoryBist wrapper enables you to control bitmap collection and output formatting when merging an LV-flow MemoryBIST CDP into a Tessent Shell MemoryBIST CDP. The IclNetwork wrapper applies to IJTAG network diagnosis and debug.

The following table lists the characterization debug options and the CharacterizationDebugOptions wrappers associated with each option.

Note

Some properties are only controllable at certain levels of the PatternsSpecification CDO hierarchy. For example, layout_database and VendorXYMapDirectories for MemoryBIST are controlled only at the top level, within the PatternsSpecification wrapper.

Table 6-1. Characterization Debug Option Overview

Characterization Debug Option	LogicBist	Memory Bist	ATPG	LV Memory Bist	Icl Network
accumulate_previous_memory_steps		x			
address_display_format		x		x	
PRTControl/ate_callback_file		x			
bitmap_fields		x		x	
bitmap_report		x		x	
bitmap_summary		x		x	
cell_failure_collection_method		x			
diagnose_on_failure	x	x	x	x	
diagnosis_result			x		x
diagnosis_setup_dofile			x		
diagnostic_patterns_symbols		x			
failing_flops_summary			x		
failing_pattern_identification_algorithm	x				
failure_count_limit		x		x	
failure_limit_scope		x			
generate_marker_file		x			
include_setup_chain_test		x			
layout_database		x			

Table 6-1. Characterization Debug Option Overview (cont.)

Characterization Debug Option	LogicBist	Memory Bist	ATPG	LV Memory Bist	Icl Network
ExecutionOptions/ max_failing_flops_per_pattern	x				
ExecutionOptions/max_failing_patterns	x				
maximum_1hot_patterns_per_file			x		
maximum_diagnosis_patterns			x		
maximum_patterns_to_expand			x		
maximum_parallel_diagnosis_servers			x		
PRTControl/ pre_first_pause_ate_callback		x			
PRTControl/ pre_pause_to_end_ate_callback		x			
PRTControl/ pre_pause_to_pause_ate_callback		x			
PRTControl/ pre_second_pause_ate_callback		x			
PRTControl/ pre_start_to_pause_ate_callback		x			
remote_protocol			x		
remote_server_hosts			x		
resolution	x	x			
start_failure		x			
start_failure_count				x	
startup_cache_file			x		
store_additional_memory_data		x			
user_log_tag			x		
VendorXYMapDirectories { }		x			
wait_for_diagnosis_result			x		

accumulate_previous_memory_steps

Wrapper: MemoryBist

Syntax

```
accumulate_previous_memory_steps : { integer | all };
```

Description

When performing diagnosis, controls the number of previous memory test steps to include in the generated INIT diagnosis pattern for the current memory test step. By default, the tool automatically keeps the gonogo testing for all the MemoryBIST controllers included before the current memory test step.

This property takes a positive integer or zero, or the value “all” to limit the number of previous memory test steps to perform along with the current test step being diagnosed. The default is all. Minimizing the number of previous memory test steps lessens pattern generation time and the pattern size when you have large designs.

Setting this property to zero excludes all of the memory test steps prior to the memory test step being performed. Setting this property to “all” retains the full context from when the failure occurred during gonogo pattern execution.

address_display_format

Wrapper: MemoryBist and LVMemoryBist

Syntax

```
address_display_format : { hexadecimal | decimal };
```

Description

Specifies the address display format for the various MemoryBIST Diagnosis bitmap results. The possible values are hexadecimal or decimal. The default value is hexadecimal.

ate_callback_file

Wrapper: MemoryBist

Syntax

```
ate_callback_file : file_path;
```

Description

Specifies a user-supplied file containing Tcl callback procedures to invoke at the beginning of the various phases of the parallel retention test. Use these user-defined Tcl procedures to vary such factors as voltage and frequency. This file supports the following phases:

- **StartToPause —** Refer to the pre_start_to_pause_ate_callback option for more details.
- **FirstRetentionPause —** Refer to the pre_first_pause_ate_callback option for more details.
- **PauseToPause —** Refer to the pre_pause_to_pause_ate_callback option for more details.
- **SecondRetentionPause —** Refer to the pre_second_pause_ate_callback option for more details.
- **PauseToEnd —** Refer to the pre_pause_to_end_ate_callback option for more details.

This option is located within the PRTControl wrapper.

The tool ignores this option when not dealing with a parallel retention test TestStep (that is, when the parallel_retention_time option for a given TestStep is set to “off”).

The default value is an empty string (“”), which means that the parallel retention test does not require any callbacks.

bitmap_fields

Wrapper: MemoryBist and LVMemoryBist

Syntax

```
bitmap_fields : identifier1, identifier2, ... ;
```

Description

Provides the list of parameters that display in the various MemoryBIST diagnosis bitmap results. For details, refer to Diagnostic Bitmap File.

You can specify any combination of the following identifiers.

Identifiers	
AddressPort	Algorithm [1]
BitPosition [1]	ControllerDesignName [1]
ControllerName	CounterA

Identifiers	
FailureIndex [1]	LogicalAddress [1]
LogicalBank	LogicalColumn
LogicalExpected [1]	LogicalRow
Memory	MemoryDesignName [1]
MemoryInstance	MemoryMacroModule
MemoryModule	PhaseOrInstruction [1]
PhysicalAddress	PhysicalBank [1]
PhysicalColumn [1]	PhysicalExpected [1]
PhysicalRow [1]	PhysicalX [1]
PhysicalX_ [1]	PhysicalY [1]
PhysicalY_ [1]	PortDataLine
RepeatLoopA	RepeatLoopB
RunId	TestPort [1]

1. Default value

bitmap_report

Wrapper: MemoryBist and LVMemoryBist

Syntax

```
bitmap_report : { on | off | on_with_parsable_data };
```

Values

- on

  Returns the bitmap data in tabular format. This is the default.

- off

  Turns off bitmap reporting.

- on_with_parsable_data

  Returns the bitmap data in both parsable format and tabular format.

Description

Generates a bitmap report in human readable hierarchical format. Refer to "MemoryBIST Diagnosis Results" for an example.

bitmap_summary

Wrapper: MemoryBist and LVMemoryBist

Syntax

```
bitmap_summary : { on | off };
```

Description

When enabled, the tool generates a bitmap summary in tabular format. Refer to "MemoryBIST Diagnosis Results" for an example.

cell_failure_collection_method

Wrapper: MemoryBist

Syntax

```
cell_failure_collection_method : { auto | auto_increment_stop_on_error };
```

Description

Defines how the tool should collect cell failures when you have specified the resolution cell option.

Currently, auto is equivalent to auto_increment_stop_on_error, in which the tool collects cell failures using the auto-increment stop-on-nth-error (ESOE) method and generates diagnostic patterns based on this method.

Requirements for this property depend on what you specify for the failure_limit_scope option.

The ESOE method works in the following way:

1. Apply the initialization pattern once (Init): Set the initial failure limit counter value (typically 0).
2. Apply the execute pattern in a loop (Exec): Load the failure limit in the counter, perform the test, and scan out any failure data.
3. Apply the increment pattern in a loop (Inc): Increment the failure limit counter.

The Exec and Inc patterns iterate until the maximum number of ESOE failures is reached or until the Exec pattern stops failing.

diagnose_on_failure

Wrapper: LogicBist, MemoryBist, ATPG, LVMemoryBist

Syntax

```
diagnose_on_failure : { on | off };
```

Description

When enabled, specifies to automatically diagnose the failures.

For LogicBIST and MemoryBIST, the tool generates diagnostic pattern wrappers as described by the resolution property.

For pass/fail results only, disable diagnose_on_failure for the applicable LogicBIST, MemoryBIST, or LVMemoryBIST controller within the TestStep wrapper.

diagnosis_setup_dofile

Wrapper: ATPG

Syntax

```
diagnosis_setup_dofile : dofile_name ;
```

Description

Specifies a user-defined dofile that contains options for Tessent Diagnosis. Tessent Diagnosis sources the dofile before diagnosis. An example diagnosis setup dofile may contain commands such as set_diagnosis_options -time_limit 1000.

This option applies to failing_nets and failing_nets_iterative_diagnosis diagnosis results. By default, no dofile is used.

Refer to the *Tessent Diagnosis User's Manual* for details.

diagnosis_result

Wrapper: ATPG and IclNetwork

Syntax

```
// FOR ATPG
diagnosis_result : { failing_pseudo_flops | failing_flops | failing_nets |
failing_nets_iterative_diagnosis };

// FOR ICLNETWORK
diagnosis_result : { suspects_only | diagnosed_suspects };
```

Values

- failing_pseudo_flops

  Diagnoses failures down to failing pseudo flops for compressed designs and failing flops for non-compressed ATPG by using Tessent SiliconInsight. This is the default.

- failing_flops

  Diagnoses failures to the failing flops by using Tessent SiliconInsight. If the design is compressed, that tool automatically generates and runs 1-hot patterns for the failing patterns and interprets the results. For non-compressed designs, this setting and the failing_pseudo_flops produce the same result.

- failing_nets

  Diagnoses the failing design down to failing nets/cells by using Tessent Diagnosis. If an LDB is available, the tool automatically performs layout-aware diagnosis. If chain test failures are observed, chain diagnosis is automatically performed.

- failing_nets_iterative_diagnosis

  Diagnoses the failing design down to failing nets/cells by using Tessent Diagnosis and also generates new patterns focusing on the failures from the previous run. It reruns and runs the new patterns to improve the results by passing the results of both runs to Tessent Diagnosis.

- suspects_only

  For IJTAG/ICL networks, diagnoses the network quickly to report only suspects, as with the following example:

  ```
  Suspect: block2_I1.tdr.R[11:0] sib_I2.Mux@1
  ```

- diagnosed_suspects

For IJTAG/ICL networks, diagnoses the network to find failing suspects. For each suspect, it then further analyzes the suspect to determine whether it is good or bad, and, if bad, the reason. The following is an example of this output:

```
//  Analyzing suspects: 'block2_I1.tdr.R[11:0] sib_I2.Mux@1' ...
//  Warning: Can't definitely determine if this Sib is stuck at
//  client(@0).

// ICL Analysis Result:
// ====================
// ---------------------  ---------------------------------
// signature              status
// ---------------------  ---------------------------------
// block2_I1.tdr.R[11:0]  register_inconsistent_length@12@11
// sib_I2.Mux@1           ?
//
//  command: analyze_icl_suspect block2_I1.tdr.R[11:0] -report

Analysis Result: register_inconsistent_length@12@11
//  command: analyze_icl_suspect sib_I2.Mux@1 -report
//  Warning: Can't definitely determine if this Sib is stuck at
//  client(@0).

Analysis Result: ?
//  command: analyze_icl_suspect sib_I2.Mux@0 -report

Analysis Result: good
```

Description

For ATPG, specifies what kind of diagnosis result Tessent SiliconInsight generates if ATPG fails.

For IJTAG/ICL networks, specifies whether to provide a quick report of suspects or to examine the network and analyze potential suspects.

diagnostic_patterns_symbols

Wrapper: MemoryBist

Syntax

```
diagnostic_patterns_symbols : { identifier1 identifier2 identifier3 };
```

Description

Adds an optional additional identifier to the names of the ESOE diagnosis test pattern files in the characterization and debug package (CDP).

By default, the filenames are of the format "<*test_name*>_<*index*>.{stil|svf}": for example, *MemoryBist_P1_1.stil* for the test "MemoryBist_P1". If you specify this option, the identifier is added between the test name and the index: "<*test_name*>_<*identifier*>_<*index*>.{stil|svf}".

Examples

Suppose you set the following value:

```
diagnostic_patterns_symbols : { INIT EXEC INCR };
```

for the test "MemoryBist_P1". This produces the following STIL files:

- *MemoryBist_P1_INIT_1.stil*
- *MemoryBist_P1_EXEC_2.stil*
- *MemoryBist_P1_INCR_3.stil*

failing_flops_summary

Wrapper: ATPG

Syntax

```
failing_flops_summary : { none | scan_cells | patterns | full };
```

Values

- none

  Disables generation of summary reports. This is the default.

- scan_cells

  Returns a list of failing cells and pattern counts for the failing cells.

- patterns

  Returns a list of all failing patterns and their failing flop/cell count per failing pattern.

- full

  Returns a list of all failing patterns, their failure counts, and a description of each failing cell in the pattern.

Description

Specifies to automatically generate a summary report of failing flops and failing pseudo flops after a diagnosis run.

Refer to "report_silicon_insight_result" for details.

failing_pattern_identification_algorithm

Wrapper: LogicBist

Syntax

```
failing_pattern_identification_algorithm : { binary_search | linear_search
| hybrid_search };
```

Values

- binary_search

  Run binary search-based diagnostics to find failing patterns.

- linear_search

  Runs linear search-based diagnostics to find failing patterns.

- hybrid_search

  Runs a combination of binary search and linear search-based diagnostics in which the tool first runs a linear search and then after a predefined number of consecutive passes switches to a binary search. This is the default.

Description

Specifies the type of algorithm to use to collect the failing patterns.

failure_count_limit

Wrapper: MemoryBist and LVMemoryBist

Syntax

```
failure_count_limit : { integer | controller_limit };
```

Description

The failure_count_limit property defines the maximum number of iterations that the Execute and Increments test steps cycle though during auto_increment_stop_on_error failure collection. You can specify a positive integer or “controller_limit”. For LVMemoryBist, this property can also define the maximum number of iterations that a patchable SOE pattern merged from a LV-flow MemoryBIST CDP cycles through during classic stop_on_error failure collection.

For MemoryBist, the default is 100.

For LVMemoryBist, the default is controller_limit. However, the default is never used because the value is initialized from the MaxFailsPerCtrl parameter, which comes from the merged LV-flow MemoryBIST CDP.

failure_limit_scope

Wrapper: MemoryBist

Syntax

```
failure_limit_scope : controller;
```

Description

Defines the scope in which the tool should apply the failure_count_limit property during cell failure collection. The tool applies the failure_count_limit property per controller.

generate_marker_file

Wrapper: MemoryBist

Syntax

```
generate_marker_file : { on | off };
```

Description

For X-Y coordinate results for failing RAM memory cells, optionally enables the generation of a marker file with the ".lay" extension so that you can view X-Y coordinates results in the Calibre DRC viewer.

Refer to "Adding Physical X-Y Coordinate Data to Bitmap Reports" on page 268 for more information.

Using this option requires you to have specified an LDB and a VendorXYMapDirectories path. For information about the marker file, refer to "Marker File Semantics" in the *Tessent Diagnosis User's Manual*.

include_setup_chain_test

Wrapper: MemoryBist

Syntax

```
include_setup_chain_test : { on | off };
```

Description

Specifies to enable setup chain test diagnosis and incorporate the test into the CDP. The setup chain test is an optional test that the tool performs in addition to other MemoryBIST diagnosis

methods (such as ESOE). When going into diagnosis, the tool performs the setup chain test first, and then proceeds to the next diagnosis test. If the setup chain test fails, no further diagnosis is performed and failure results are reported.

Setup chain test patterns exercise all chains through the MemoryBIST controller and interfaces to verify the serial load/unload operation. The patterns shift the registers in all configuration/ status chains as well as the BIRA registers, if they are present.

Setup chain test diagnosis is only supported when diagnose_on_failure property is on.

layout_database

Wrapper: MemoryBist

Syntax

```
layout_database : directory_path [, directory_path, …];
```

Description

Specifies the pathnames of layout databases (LDB), which can be previously generated with Tessent Diagnosis or generated from within Tessent SiliconInsight if you have the LEF/DEF files.

Specifying LDBs enables the X-Y coordinate feature as described in “Adding Physical X-Y Coordinate Data to Bitmap Reports” on page 268.

In the patterns specification, you can also set the LDB path through the command prompt as follows:

```
# design name is top
set_config_value PatternsSpecification(top,rtl,manufacturing)/   \
    CharacterizationDebugOptions/MemoryBist/layout_database \
    { ../data/AnotherLayoutDB MyLayoutDB }
```

Note

Braces ({ }) are only required for multiple LDB paths. You can omit them if you specify only a single LDB path.

max_failing_flops_per_pattern

Wrapper: LogicBist

Syntax

```
ExecutionOptions {
  max_failing_flops_per_pattern : integer;
}
```

Description

This option specifies the maximum number of failing flops per pattern. It takes an integer that ranges from 0 to the number of flops in all scan chains.

This option is located within the ExecutionOptions wrapper.

The default is unlimited.

max_failing_patterns

Wrapper: LogicBist

Syntax

```
ExecutionOptions {
  max_failing_patterns : integer;
}
```

Description

This option specifies the maximum number of failing patterns. It takes an integer that ranges from 0 to the number of failing patterns.

This option is located within the ExecutionOptions wrapper.

The default is unlimited.

maximum_1hot_patterns_per_file

Wrapper: ATPG

Syntax

```
maximum_1hot_patterns_per_file : integer;
```

Description

Specifies the maximum number of 1hot patterns that can be saved into a pattern file.

Examples

If you are pregenerating the 1hot patterns for 0:99 (that is, 100 patterns) and specify a value of 20, each pattern file contains at most 20 patterns, so the tool generates five pattern files. These files contain patterns 0:19, 20:39, 40:59, 60:79, and 80:99.

maximum_diagnosis_patterns

Wrapper: ATPG

Syntax

```
maximum_diagnosis_patterns : integer;
```

Description

Limits the number of additional patterns that may be generated in the second run of Tessent Diagnosis when you have asked for failing_nets_iterative_diagnosis.

The default is unlimited.

maximum_parallel_diagnosis_servers

Wrapper: ATPG

Syntax

```
maximum_parallel_diagnosis_servers : integer;
```

Description

Enables using parallel servers for generation of 1hot patterns in a diagnosis session and specifies the maximum number of parallel servers that can be active for the duration.

The tool uses the servers specified by the remote_server_hosts property and the protocol specified by the remote_protocol property. If remote_server_hosts is “localhost” (the default value), all parallel processes occur on the local machine. If you have specified other hosts, the tool distributes generation to a number of servers equal to the value this property specifies. It is not necessary for “localhost” to be one of the hosts.

A value of zero (default) disables using parallel servers, causing the tool to ignore remote_server_hosts and remote_protocol. Use a value of one and a different remote_server_host to distribute the generation to a different host than the current host. (A value of one without specifying a different host is functionally identical to the default value of zero.)

maximum_patterns_to_expand

Wrapper: ATPG

Syntax

```
maximum_patterns_to_expand : integer;
```

Description

Limits the maximum number of 1-hot patterns that the tool generates when you have a compressed design and have asked for failing_flops.

The default is 10.

pre_first_pause_ate_callback

Wrapper: MemoryBist

Syntax

```
pre_first_pause_ate_callback : tcl_proc;
```

Description

Specifies a user-defined Tcl procedure to invoke at the beginning of the first retention pause of the parallel retention test. Use this procedure to vary such factors as voltage and frequency. It must exist in the callback file specified by the ate_callback_file option.

This option is located within the PRTControl wrapper.

The tool ignores this option when not dealing with a parallel retention test TestStep (that is, when the parallel_retention_time option for a given TestStep is set to “off”).

The default value is an empty string (“”), which means the first retention pause requires no callback at the beginning.

pre_pause_to_end_ate_callback

Wrapper: MemoryBist

Syntax

```
pre_pause_to_end_ate_callback : tcl_proc;
```

Description

Specifies a user-defined Tcl procedure to invoke at the beginning of the “pause to end” phase of the parallel retention test. Use this procedure to vary such factors as voltage and frequency. It must exist in the callback file specified by the ate_callback_file option.

This option is located within the PRTControl wrapper.

The tool ignores this option when not dealing with a parallel retention test TestStep (that is, when the parallel_retention_time option for a given TestStep is set to “off”).

The default value is an empty string (“”), which means the “pause to end” phase requires no callback at the beginning.

pre_pause_to_pause_ate_callback

Wrapper: MemoryBist

Syntax

```
pre_pause_to_pause_ate_callback  : tcl_proc;
```

Description

Specifies a user-defined Tcl procedure to invoke at the beginning of the “pause to pause” phase of the parallel retention test. Use this procedure to vary such factors as voltage and frequency. It must exist in the callback file specified by the ate_callback_file option.

This option is located within the PRTControl wrapper.

The tool ignores this option when not dealing with a parallel retention test TestStep (that is, when the parallel_retention_time option for a given TestStep is set to “off”).

The default value is an empty string (“”), which means the “pause to pause” phase requires no callback at the beginning.

pre_second_pause_ate_callback

Wrapper: MemoryBist

Syntax

```
pre_second_pause_ate_callback  : tcl_proc;
```

Description

Specifies a user-defined Tcl procedure to invoke at the beginning of the second retention pause of the parallel retention test. Use this procedure to vary such factors as voltage and frequency. It must exist in the callback file specified by the ate_callback_file option.

This option is located within the PRTControl wrapper.

The tool ignores this option when not dealing with a parallel retention test TestStep (that is, when the parallel_retention_time option for a given TestStep is set to “off”).

The default value is an empty string (“”), which means the second retention pause requires no callback at the beginning.

pre_start_to_pause_ate_callback

Wrapper: MemoryBist

Syntax

```
pre_start_to_pause_ate_callback : tcl_proc;
```

Description

Specifies a user-defined Tcl procedure to invoke at the beginning of the “start to pause” phase of the parallel retention test. Use this procedure to vary such factors as voltage and frequency. It must exist in the callback file specified by the ate_callback_file option.

This option is located within the PRTControl wrapper.

The tool ignores this option when not dealing with a parallel retention test TestStep (that is, when the parallel_retention_time option for a given TestStep is set to “off”).

The default value is an empty string (“”), which means the “start to pause” phase requires no callback at the beginning.

pregenerate_1hot_patterns

Wrapper: ATPG

Pregenerates 1hot patterns for the specified list of patterns, enabling you to use the 1hot pattern interactively or during an offline interpretation flow.

Syntax

```
pregenerate_1hot_patterns : pattern_list ;
```

Description

The pregenerate_1hot_patterns property contains a list of patterns that you want to pregenerate 1hot patterns for when you build the CDP. Pregenerating 1hot patterns enables you, in particular, to use the patterns in an offline interpretation flow by collecting failures on the ATE into a standard ATPG *.flog* file. Then you can use the process_atpg_failures command to interpret the failure log files. Refer to "ATPG Offline Interpretation Flow" on page 288 for more information about this flow.

The <*pattern_list*> consists of pattern numbers and ranges separated by spaces. A pattern range consists of a start pattern number and an end pattern number joined by a colon. For example:

```
1 5 76:85
```

remote_protocol

Wrapper: ATPG

Syntax

```
remote_protocol : ssh | qrsh ;
```

Description

Specifies the protocol to use when submitting parallel processes to hosts specified by remote_server_hosts. By default, the tool uses ssh.

remote_server_hosts

Wrapper: ATPG

Syntax

```
remote_server_hosts : hostname[:maximum_servers] ... ;
```

Description

Specifies one or more network hosts on which to run parallel servers for 1hot pattern pregeneration.

- If you do not specify this property and do not specify the maximum_parallel_diagnosis_servers property (or specify a value of zero for that property), diagnosis does not use parallel servers.
- If you do not specify this property but do specify a value for maximum_parallel_diagnosis_servers, this value defaults to localhost and diagnosis uses localhost for as many parallel servers as you specified.

- If you specify this property but do not specify the optional maximum_servers value for some or all hosts, diagnosis uses the maximum_parallel_diagnosis_servers value to determine how many processes to run the remote hosts that you did not specify a value for.

Except as described previously, the tool does not assume localhost as a parallel server host. If you want to run parallel servers locally alongside other hosts, you must explicitly declare it.

Examples

Suppose you have set the following property values:

```
maximum_parallel_diagnosis_servers : 5;
remote_server_hosts: ststest1:6 ststest2:2 ststest3;
```

Then diagnosis can run on six servers on the host ststest1, two servers on the host ststest2, and five servers (specified by the maximum_parallel_diagnosis_servers property) on the host ststest3.

resolution

Wrapper: MemoryBist and LogicBist

Syntax

```
// FOR MEMORYBIST
resolution : { cell | memory };

// FOR LOGICBIST
resolution : { flop | pattern | misr };
```

MemoryBIST Values

- cell

  This is applicable to RAM memories. Generates diagnostic patterns that provide failure data down to the memory cell level, suitable for bitmaps. This is the default.

- memory

  This is applicable to RAM and ROM memories. Adapts the GoNoGo pattern to provide failure data down to the memory level, suitable for identifying failing memories.

LogicBIST Values

- flop

  Generates diagnostic patterns to permit running failed pattern search algorithms and flop unload patterns. This permits down-to failed flop diagnostics. This is the default.

- pattern

Generates diagnostic patterns to permit running failed pattern search algorithms and returns a list of failing patterns.

- misr

  Does not process additional diagnostics but returns a list of failing misrs.

Description

Generates diagnostic pattern wrappers when you have enabled the diagnose_on_failure property. You must specify this property for all controller instances of the applicable instrument found within the TestStep wrapper.

start_failure

Wrapper: MemoryBist

Syntax

```
start_failure : integer;
```

Description

Controls the ESOE diagnostic starting point for auto_increment_stop_on_error failure collection. This property enables you to debug a specific failure or range of failures. Specify an integer corresponding to the SOE number where you want to begin Execute and Increments test step iterations during failure collection.

The default is 1. The number of iterations is limited by the setting of failure_count_limit, which specifies the upper end of iterations.

start_failure_count

Wrapper: LVMemoryBist

Syntax

```
start_failure_count : integer;
```

Description

The start_failure_count property defines the SOE number that begins the iterations the patchable SOE pattern merged from a LV-flow MemoryBIST CDP cycles though during classic stop_on_error failure collection. The default is 1. However, the default is never used because the value is initialized from the StartFailureNumber parameter, which comes from the merged LV-flow MemoryBIST CDP. The number of iterations is limited by the setting of failure_count_limit, which specifies the upper end of iterations.

startup_cache_file

Wrapper: ATPG

Syntax

```
startup_cache_file : filename;
```

Description

Enables the user to specify a startup cache file to be used by Tessent Diagnosis to reduce the diagnosis runtime by re-using the pattern verification information without rerunning the verification, assuming you have made no changes to the patterns.

This option applies to failing_nets and failing_nets_iterative_diagnosis diagnosis results. By default, no cache file is used.

store_additional_memory_data

Wrapper: MemoryBist

Syntax

```
store_additional_memory_data : { on | off };
```

Description

Specifies whether to add vendor physical X-Y map files for tested memories to the CDP.

The tool adds the vendor files of tested memories to the *<cdp_path>/Design/mentor/memory_bist/* directory under the following scenario:

- The store_additional_memory_data property is set to “on”.
- The tool discovers the vendor physical X-Y map files in a directory path specified in the VendorXYMapDirectories wrapper of CharacterizationDebugOptions/MemoryBist.
- The tool discovers the tested memory in one of the layout databases (LDB) specified by the layout_database property in CharacterizationDebugOptions/MemoryBist.

user_log_tag

Wrapper: ATPG

Syntax

```
user_log_tag : tag;
```

Description

Enables you to customize the default job directory name. The job directory is listed in the transcript during processing when you have set the wait_for_diagnosis_result option to off. For example:

```
...
Diagnosing (failing_nets) core instance 'c1_2' of 'core1' ....
Dispatching diagnosis: working directory: all_cores__c1_2.3 final
transcript: all_cores__c1_2.3/job.transcript ...
JOB: 'all_cores__c1_2.3/job_script.sh' started successfully.
```

Suppose you set the user_log_tag option to "xyz." The listed JOB would be all_cores__c1_2_xyz.3, where 3 indicates the number of jobs initiated for the current test.

VendorXYMapDirectories

Wrapper: MemoryBist

Syntax

```
VendorXYMapDirectories {
  directory_path1;
  directory_path2;
  .
  .
  }
```

Description

Specifies one or more directory paths to Vendor X-Y Map files. This is required for generating physical X-Y coordinate data for failing RAM memory cells as described in "Adding Physical X-Y Coordinate Data to Bitmap Reports" on page 268.

The names are case sensitive. The tool ignores all data within the bitmap files except the logical address (#address) data.

You can add a directory path through the command prompt as follows, for example:

```
# design name is top
set dirWrap [add_config_element -in_wrapper   \
PatternsSpecification(top,rtl,manufacturing)/ \
CharacterizationDebugOptions/MemoryBist VendorXYMapDirectories]

# repeat add_config_element for each directory
add_config_element -in $dirWrap ../data/14nmlpp_sp_rams \
-type property
```

wait_for_diagnosis_result

Wrapper: ATPG

Syntax

```
wait_for_diagnosis_result : { on | off };
```

Description

Defines when to return the control of the tool back to the user.

By default, this option is on, and the tool waits for diagnosis to complete before giving the control back to the user.

Chapter 7
Diagnosis Results

The LogicBIST and MemoryBIST diagnosis results display in the transcript area of the GUI. In addition, the tool generates a results file. You can introspect the results that are stored in memory in the /CDPResult partition.

LogicBIST Diagnosis Results

The diagnosis failure results display show the failing patterns, the failing chains per pattern, all failing scan cells per chain, and the expected and actual values for each failing scan cell.

The following example shows logicBIST diagnosis results. The transcript shows that the test failed at patterns 1 and 2.

```
//Failing instrument icl name     = piccpu_inst1
//Failing instrument Verilog name = piccpu_inst1

//Saved Tessent Diagnosis execution dofile.
//  Diagnose failure by running following command:
//  >>dofile lbist_patt0_lbist_patt0_para__piccpu_inst1_td_dofile.do

//Failing pattern No. = 1
//  Failing chain = chain8
//    Failing flops:
//      ScanCell = piccpu_inst1/inst_reg_7 Expected = 1 Actual = 0
//Failing pattern No. = 2
//  Failing chain = chain8
//    Failing flops:
//      ScanCell = piccpu_inst1/inst_reg_7 Expected = 1 Actual = 0
```

In addition, as shown in the example transcript for lbist_patt0, the log contains:

```
//Saved Tessent Diagnosis execution dofile.
//  Diagnose failure by running following command:
//  >>dofile lbist_patt0_lbist_patt0_para__piccpu_inst1_td_dofile.do
```

This tells you that a Tessent Diagnosis dofile was automatically saved by the tool in the file lbist_patt0_lbist_patt0_para__piccpu_inst1_td_dofile.do.

Within the saved dofile, you can find the name of the .flog failure log and access the log to further investigate into the root causes for the failures to the failing scan cells. The result of further analysis may be similar to:

```
symptom=1 #suspects=3 #explained_patterns=2
1     2
suspect  score fail_match pass_mismatch type    value  pin_pathname cell_name net_pathname
-------------------------------------------------------------------------------------------
1        100   2          0             STUCK   1      /inst_reg_7/QB dff      /n150
2        100   2          0             STUCK   1      /U920/A0       oai221   /n150
3        100   2          0             EQ2     1      /U920/A1       oai221   /n767
-------------------------------------------------------------------------------------------
```

MemoryBIST Diagnosis Results

When you set bitmap_report to on, the GUI transcript area displays the bitmap results in a hierarchical format.

For example:

```
// Failing Controller Verilog Instance: core_inst1/
top_rtl_tessent_mbist_c1_controller_inst
//    Warning: Some memories for this controller may still be untested because the
failure_count_limit (10) has been reached during StopOnError diagnosis.
// If this is the case, please increase that limit if you want to have them tested.
//    Tested Controller Steps: 0-2
//    Step: 0
//      Algorithm: SMARCHCHKBVCD
//      Tested Memories:
//        M1:core_inst1/blockA_clka_i1/mem1
//        M2:core_inst1/blockA_clka_i1/mem2
//        M3:core_inst1/blockA_clka_i1/mem3
//        M7:core_inst1/blockB_clka_i1/mem1
//        M4:core_inst1/blockA_clka_i2/mem1
//        M5:core_inst1/blockA_clka_i2/mem2
//        M6:core_inst1/blockA_clka_i2/mem3
//      Failing Memory Verilog Instance: core_inst1/blockA_clka_i1/mem1
//        Tested Ports: 0
//        Failing Port: 0
//          Bitmap:
//          ----------  ---------  -------- --------- ---- ----- ----       ----
//          Phase Or    Physical   Physical  Physical  Bit  Phys  Logical    Log
//          Instruction  Bank       Row       Column   Pos  Exp   Address    Exp
//         -----------  --------   --------  --------- ----   -----         ----
//          2_W1R1      0x0        0x0       0x0        4    1     0x0        1
//          2_W1R1      0x0        0x0       0x1        4    1     0x1        1
//          2_W1R1      0x0        0x0       0x2        4    1     0x2        1
//          2_W1R1      0x0        0x0       0x3        4    1     0x3        1
//          2_W1R1      0x0        0x0       0x4        4    1     0x4        1
//          2_W1R1      0x0        0x0       0x5        4    1     0x5        1
//          2_W1R1      0x0        0x0       0x6        4    1     0x6        1
//          2_W1R1      0x0        0x0       0x7        4    1     0x7        1
//          3_W1R1      0x0        0x0       0x0        4    1     0x0        1
//          3_W1R1      0x0        0x0       0x1        4    1     0x1        1
```

When you set bitmap_summary to on, the GUI transcript area displays a bitmap summary for each memory controller. For example:

```
// Bitmap Summary:
// ----------- -------- -------- --------- ----  ------  ------   ----
// Phase Or     Physical Physical  Physical   Bit  Phys Logical   Log
// Instruction  Bank     Row       Column     Pos  Exp  Address   Exp
// ----------- -------- -------- --------- ----  ------  ------   ----
// 2_W1R1,3_W1R1 0x0     0x0       0x0-0x7    4    1    0x0-0x7   1
```

ATPG Diagnosis Results

The ATPG diagnosis results displays the number of symptoms and suspects, and details about the suspects.

For more information about the diagnosis results refer to “Layout-Aware Diagnosis Report” in the *Tessent Diagnosis User’s Manual*.

```
//  Verifying 100 external patterns.
//  incorrect_patterns=0, simulated_patterns=100, simulation_time=0.43sec.
Tessent Shell 2016.2-snapshot_2016.04.04_03.01   Mon Apr 04 03:06:34 GMT 2016

tracking_info_begin
core_instance "c1_2"
core_name core1
core_mode internal
core_id c1_2__core1__internal
tracking_info_end

#diagnosis_modes=1
#logic_symptoms=1
#total_symptoms=1 #total_suspects=3 total_CPU_time=1.03sec

diagnosis_mode=logic
#symptoms=1 #suspects=3 CPU_time=1.03sec fail_log=si_outdir/all_cores___c1_2__core1__internal
#failing_patterns=12, #passing_patterns=88
#unexplained_failing_patterns=0

symptom=1 #suspects=3 #explained_patterns=12
8     9     21    23    43    53    77    79    82    92
93    97
suspect  score fail_match pass_mismatch type          value  pin_pathname cell_name net_pathname
layout_status
-----------------------------------------------------------------------------------------
1        100   12         0             STUCK          0      /cpu_i/uALU/u10/ix216/Y ao22 /cpu_i/
uALU   /u10/nx10  LOCATION_IN_LAYOUT
        #potential_open_segments=1, #total_segments=3,   #potential_bridge_aggressors=0,
#total_neighbors=11
        suspect  score fail_match pass_mismatch type           value  location layout_layer
critical_area
        ---------------------------------------------------------------------------------
        1.1      100   12         0             CELL           0      /cpu_i/uALU/u10/ix216
       1.2      100   12         0            OPEN          0       B1&B2 route_2        1.12E+02
                                                                      VIA2           5.66E+00
                                                                      route_3        3.83E+02
        ---------------------------------------------------------------------------------
2        100   12         0             OPEN/DOM       both   /cpu_i/uALU/ix1136/Y buf02 /cpu_i/
uALU/n   x1137  LOCATION_IN_LAYOUT
        #potential_open_segments=1, #total_segments=5,   #potential_bridge_aggressors=0,
#total_neighbors=na
        suspect  score fail_match pass_mismatch type           value  location layout_layer
critical_area
        ---------------------------------------------------------------------------------
        2.1      100   12         0             OPEN           both    B1&B2&B3 route_2
1.25E+02
                                                                     VIA2          1.13E+01
                                                                     route_3       1.84E+02
        ---------------------------------------------------------------------------------
3        100   12         0             BRIDGE_2WAY    0      /cpu_i/uALU/ix1136/Y buf02 /cpu_i/
uALU/n   x1137  LOCATION_IN_LAYOUT
        suspect  score fail_match pass_mismatch type           value  pin_pathname cell_name
net_pathname layout_layer critical_area
        ---------------------------------------------------------------------------------
        3.1      100   12         0             BRIDGE_2WAY    0      /cpu_i/uALU/u10/ix21/Y xor2
/cpu_i/uALU/ALU_SUM_6_ route_2         1.45E+01
        ---------------------------------------------------------------------------------
```

CDP Result Structure

Tessent SiliconInsight stores the results of the last test run in memory in the Tessent Shell CDPResult partition. This data structure is useful for purposes of introspecting the test results.

The top-level CDPResult structure looks as follows:

```
CDPResult {
   Command : performed_command;        // Performed command
   ExecutionStatus : <Success | Failure>;
                                       // Status of the command execution
   ResultType : None | StringList | ErrorMessage | TestExecutionResult;
                                       // Result type
   Version : version_number;           // Format version of the structure
}
```

CDP Result Structure for the None Result Type

The following example shows the structure of the results returned for the None result type:

```
CDPResult {
   Command : delete_test -name glue_logic
   ExecutionStatus : Success;
   None : "";
   ResultType : None;
   Version : 1;
}
```

CDP Result Structure for the StringList Result Type

The following example shows the structure of the results returned for the StringList result type:

```
CDPResult {
   Command : get_test_list;
   ExecutionStatus : Success;
   ResultType : StringList;
   StringList : all_cores, glue_logic;
   Version : 1;
}
```

CDP Result Structure for the ErrorMessage Result Type

The following example shows the structure of the results returned for the ErrorMessage result type:

```
CDPResult {
   Command : execute_cdp_test -test bogus;
   ErrorMessage : Test 'bogus' doesn't exist.;
   ExecutionStatus : Failure;
   ResultType : ErrorMessage;
   Version : 1;
}
```

CDP Result Structure for the TestExecutionResult Result Type

The TestExecutionResult result type returns various results depending on the test data you collect, as specified by the execute_cdp_test -collect_data_type_list command.

Refer to "execute_cdp_test" in the *Tessent Shell Reference Manual* for details.

Go-NoGo tests and all passing data collection results

For go-nogo tests (whether failed or passed) and all passing data collection tests, CDPResult lists the TestName and TestStatus: Passed or Failed. For example:

```
CDPResult {
   Command : execute_cdp_test -test MemoryBist_P1;
   ExecutionStatus : Success;
   ResultType : TestExecutionResult;
   TestExecutionResult {
      TestName : MemoryBist_P1;
      TestStatus : Passed;
}
Version : 1;
```

Failing variable results

Specifying the -collect_data_type_list option expands the CDPResult structure when you have failures. The structure always includes the following properties:

- TestExecutionResultTypes property, which lists the result types that are available for introspection. The possibilities are: VariableFailures, InstrumentsResult, and DiagnosisResult.
- VariableFailures property: Details about the failures.

For example, for variable failures:

```
CDPResult {
   Command : execute_cdp_test -collect_data_type_list variable \
-test MemoryBist_P1;
   ExecutionStatus : Success;
   ResultType : TestExecutionResult;
   TestExecutionResult {
      TestExecutionResultTypes : VariableFailures;
      TestName : MemoryBist_P1;
      TestStatus : Failed;
      VariableFailures {
        ExecStatus : FAIL;
        Pattern : MemoryBist_P1.svf;
        PatternSets(0) {
          PatternSetName : run_time_prog;
          Variables(0) {
            Actual : b0;
            Expected : b1;
            Pin : tdo;
            VarName : \
blockA_inst.blockA_rtl_tessent_mbist_c1_controller_\
inst.MBISTPG_GO(1);
          }
          Variables(1) {
            Actual : b00000001;
            Expected : b00000000;
            Pin : tdo;
            VarName : blockA_inst.mem1_interface_inst.GO_ID_REG;
          }
        }
    }
  }
  Version : 1;
}
```

Failing instrument results

For failure results, the CDPResult structure further expands when you specify -collect_data_type_list with the instrument option. For example, for instrument failures:

```
CDPResult {
   Command : execute_cdp_test -collect_data_type_list instrument -test
   MemoryBist_P1;
   ExecutionStatus : Success;
   ResultType : TestExecutionResult;
   TestExecutionResult {
     InstrumentsResult {
       PatternSets(0) {
          FailingInstruments : \
blockA_inst.blockA_rtl_tessent_mbist_c1_controller_inst;
          ParticipatingInstruments : \
blockA_inst.blockA_rtl_tessent_mbist_c1_controller_inst;
          PatternSetName : run_time_prog;
       }
     }
     TestExecutionResultTypes : VariableFailures, InstrumentsResult;
     TestName : MemoryBist_P1;
     TestStatus : Failed;
     VariableFailures {
       ExecStatus : FAIL;
       Pattern : MemoryBist_P1.svf;
       PatternSets(0) {
       PatternSetName : run_time_prog;
       Variables(0) {
         Actual : b0;
         Expected : b1;
         Pin : tdo;
         VarName : \
blockA_inst.blockA_rtl_tessent_mbist_c1_controller_inst.MBISTPG_GO(1);
       }
     Variables(1) {
       Actual : b00000001;
       Expected : b00000000;
       Pin : tdo;
       VarName : blockA_inst.mem1_interface_inst.GO_ID_REG;
...
```

Failing Diagnosis Results (MemoryBIST)

Failing diagnosis test results include a DiagnosticResult property. For example:

```
CDPResult {
   Command : execute_cdp_test -collect_data_type_list { variable \
instrument diagnosis } -force_diagnose_on_failure -test MemoryBist_P1;
   ExecutionStatus : Success;
   ResultType : TestExecutionResult;
   TestExecutionResult {
     DiagnosticResult {
       PatternSets(0) {
         FailingInstruments : \
blockA_inst.blockA_rtl_tessent_mbist_c1_controller_inst;
         ParticipatingInstruments : \
blockA_inst.blockA_rtl_tessent_mbist_c1_controller_inst;
         PatternSetData(0) {
           InstrumentGroupType : MemoryBist;
           FailingInstruments : \
blockA_inst.blockA_rtl_tessent_mbist_c1_controller_inst;
           ParticipatingInstruments : \
blockA_inst.blockA_rtl_tessent_mbist_c1_controller_inst;

           InstrumentResult(0) {
             Algorithm : ESOE_inline;
              FailsInfo : 8 Info: Possible serial datapath short and/or
write enable short if the algorithm is only failing on Phase 3, 9 Info:
Possible serial datapath short and/or write enable short if the algorithm
is only failing on Phase 3;
              Failures : 0 blockA_inst/
blockA_rtl_tessent_mbist_c1_controller_inst SMARCHCHKBVCD \
2_W0R0
blockA_inst/mem1 0 0x0 0x0 0x0 0 0 0x0 0, 1 blockA_inst/
blockA_rtl_tessent_mbist_c1_controller_inst SMARCHCHKBVCD \
2_W0R0
blockA_inst/mem1 0 0x0 0x0 0x1 0 0 0x1 0, 2 blockA_inst/
blockA_rtl_tessent_mbist_c1_controller_inst SMARCHCHKBVCD \
2_W0R0
blockA_inst/mem1 0 0x0 0x0 0x2 0 0 0x2 0, 3
...
              Header : FailureIndex ControllerDesignName Algorithm
PhaseOrInstruction MemoryDesignName TestPort PhysicalBank PhysicalRow
PhysicalColumn BitPosition PhysicalExpected LogicalAddress
LogicalExpected;
              InstrumentName : None;
             }
           }
           PatternSetName : run_time_prog;
         }
       }
       TestExecutionResultTypes : VariableFailures, DiagnosticResult;
       TestName : MemoryBist_P1;
       TestStatus : Failed;
       VariableFailures {
         ExecStatus : FAIL;
         Pattern : MemoryBist_P1.svf;
         PatternSets(0) {
```

```
        PatternSetName : run_time_prog;
        Variables(0) {
...
```

Failing diagnosis results (LogicBIST)

```
CDPResult {
   Command : execute_cdp_test -collect_data_type_list { variable \
instrument diagnosis } -force_diagnose_on_failure -test lbist_patt0;
   ExecutionStatus : Success;
   ResultType : TestExecutionResult;
   TestExecutionResult {
     DiagnosticResult {
       PatternSets(0) {
         FailingInstruments : piccpu_inst1;
         ParticipatingInstruments : piccpu_inst1;
         PatternSetData(0) {
           InstrumentGroupType : LogicBist;
           FailingInstruments : piccpu_inst1;
           ParticipatingInstruments : piccpu_inst1;
           InstrumentResult(0) {
             Algorithm : hybrid_search;
             InstrumentName : piccpu_inst1;
             LogicBistFailure(0) {
               actual : 0;
               cell : 0;
               chain : chain8;
               expected : 1;
               pattern : 1;
             }
             LogicBistFailure(1) {
               actual : 0;
               cell : 0;
               chain : chain8;
               expected : 1;
               pattern : 2;
             }
           }
       }
       PatternSetName : lbist_patt0_para;
     }
   }
   TestExecutionResultTypes : VariableFailures, DiagnosticResult;
   TestName : lbist_patt0;
   TestStatus : Failed;
   VariableFailures {
     ExecStatus : FAIL;
     Pattern : lbist_patt0.svf;
     PatternSets(0) {
...
```

Failing diagnosis results (ATPG)

```
CDPResult {
   Command : execute_cdp_test -test glue_logic -collect_data_type_list \
variable;
   ExecutionStatus : Success;
   ResultType : TestExecutionResult;
   TestExecutionResult {
     TestExecutionResultTypes : VariableFailures, DiagnosticResult;
     TestName : glue_logic;
     TestStatus : Failed;
     VariableFailures {
       ExecStatus : FAIL;
       Pattern : pat_glue_top.stil;
       PatternSets : "";
       UnmappedFailures(0) {
         Cycle : 739;
         Expected : b1;
         Pin : SFRWE_c2_2;
       }
       UnmappedFailures(1) {
         Cycle : 742;
         Expected : b1;
         Pin : edt_channels_out3_c2_2;
       }
...
       UnmappedFailuresFormat : Expanded;
       }
       DiagnosticResult {
         PatternSets(0) {
           PatternSetName : default;
           PatternSetData(0) {
             InstrumentGroupType : UserDefinedATPG;

...}
```

The CDPResult structure for the UserDefinedRetargetedATPG instrument group type includes core module data. For example:

```
...
PatternSetName : default;
PatternSetData(0) {
   CoreModules(0) {
     CoreModuleName : core1;
     ParticipatingCoreInstances : c1_1, c1_2;
     FailingCoreInstances : c1_2;
   }
   CoreModules(1) {
     CoreModuleName : core2;
     ParticipatingCoreInstances : c2_1;
     FailingCoreInstances : "";
   }
   InstrumentGroupType : UserDefinedRetargetedATPG;
   ParticipatingInstruments : core1, core2;
   FailingInstruments : core1;
  }
  ParticipatingInstruments : core1, core2;
  FailingInstruments : core1;
 }
...
```

Related Topics

CDP Result Introspection

CDP Result Introspection

You can introspect the test results stored in the in-memory CDPResult partition by using Tessent Shell commands such as report_config_data, get_config_element, and get_config_value.

Note

You can only introspect the last performed test because the CDPResult structure exists in memory for only the last performed test.

Test Pass/Fail Status Example

You can return the pass/fail status for any type of CDP test. The following Tcl example returns Passed or Failed.

```
get_config_value /CDPResult/TestExecutionResult/TestStatus
```

Failing Variable Example

The following TCL example returns all failing variables for a failing test that was just performed regardless of the type of instrument exercised in it. Suppose you had run:

execute_cdp_test my_test –collect_data_type_list variable

```
if { ! [get_config_value -exists /CDPResult/TestExecutionResult/
VariableFailures] } {
   return -code error "There are no Variable Failures."
}
set pat_sets_coll [get_config_elements /CDPResult/TestExecutionResult/
VariableFailures/PatternSets(*)]

foreach_in_collection pat_set $pat_sets_coll {
   set pat_set_name [get_config_value -in_wrapper $pat_set PatternSetName]
   set vars_coll [get_config_elements -in_wrapper $pat_set Variables(*)]
   puts "Failing variables for PatternSet '$pat_set_name':"
   foreach_in_collection var $vars_coll {
      puts "\tVarName = '[get_config_value -in_wrapper $var VarName]'"
      puts "\tPin = '[get_config_value -in_wrapper $var Pin]'"
      puts "\tExpected = '[get_config_value -in_wrapper $var Expected]'"
      puts "\tActual = '[get_config_value -in_wrapper $var Actual]'"
      puts ""
   }
}
```

Diagnostic Data Examples for Different Instrument Group Types

Diagnostic data is uniform through /CDPResult/TestExecutionResult/DiagnosticResult/PatternsSets(*)/PatternSetData(*). The structure and the data under PatternSetData(*) differ depending on the value of the InstrumentGroupType property.

The following example for the MemoryBist instrument group type returns the names of the failing memories. Suppose you had run:

execute_cdp_test MemoryBist_P1 –force_diagnose_on_failure -collect_data_type_list diagnosis

```
# Initialize the failing memory names list
set failing_mem_names [list]
if { ! [get_config_value -exists /CDPResult/TestExecutionResult/DiagnosticResult] } {
   return -code error "There are no DiagnosticResult."
}
set pat_sets_coll [get_config_elements /CDPResult/TestExecutionResult/DiagnosticResult/
PatternSets(*)]
foreach_in_collection pat_set $pat_sets_coll {
  set pat_set_data_coll [get_config_elements -in_wrapper $pat_set PatternSetData(*)]
  foreach_in_collection pat_set_data $pat_set_data_coll {
    set inst_result_coll [get_config_elements -in_wrapper $pat_set_data InstrumentResult(*)]
    foreach_in_collection inst_res $inst_result_coll {
      set fails_list [get_config_value -in_wrapper $inst_res Failures]
      set header [get_config_value -in_wrapper $inst_res Header]
      set mem_name_idx [lsearch $header MemoryDesignName]
      if { $mem_name_idx < 0 } {
        return -code error "Bitmap does not contain MemoryDesignName."
      }
      foreach fail $fails_list {
        set mem_name [lindex $fail $mem_name_idx]
        # Add the name to the list if not already there
        if { [lsearch $failing_mem_names $mem_name] < 0 } {
          lappend failing_mem_names $mem_name
         }
       }
     }
   }
 }
puts "List of failing memories = '$failing_mem_names'"
```

The following example for the LogicBist instrument group type returns the pattern number, chain name, and flop/cell index for each failure.

```
if { ! [get_config_value -exists /CDPResult/TestExecutionResult/DiagnosticResult] } {
   return -code error "There are no DiagnosticResult."
}
set pat_sets_coll [get_config_elements /CDPResult/TestExecutionResult/DiagnosticResult/
PatternSets(*)]
foreach_in_collection pat_set $pat_sets_coll {
  set pat_set_data_coll [get_config_elements -in_wrapper $pat_set PatternSetData(*)]
  foreach_in_collection pat_set_data $pat_set_data_coll {
    set inst_result_coll [get_config_elements -in_wrapper $pat_set_data InstrumentResult(*)]
    foreach_in_collection inst_res $inst_result_coll {
      set inst_name [get_config_value -in_wrapper $inst_res InstrumentName]
      set fails_coll [get_config_elements -in_wrapper $inst_res LogicBistFailure(*)]
      puts "Failures for instrument '$inst_name':"
      puts "\tpattern\tchain\tcell"
      foreach_in_collection fail $fails_coll {
        set pat_num [get_config_value -in_wrapper $fail pattern]
        set chain [get_config_value -in_wrapper $fail chain]
        set cell_idx [get_config_value -in_wrapper $fail cell]
        puts "\t$pat_num\t$chain\t$cell_idx"
      }
    }
  }
}
```

For ATPG tests, the tool saves the diagnostic results in the log files. The only ATPG diagnostic data saved in the CDPResult wrapper is for UserDefinedRetargetedATPG, and it consists of the list of passing and failing core instances per core module.

The following UserDefinedRetargetedATPG example extracts the failing/passing core instances for each core module:

```
if { ! [get_config_value -exists /CDPResult/TestExecutionResult/DiagnosticResult] } {
   return -code error "There are no DiagnosticResult."
}
set pat_sets_coll [get_config_elements /CDPResult/TestExecutionResult/DiagnosticResult/
PatternSets(*)]
foreach_in_collection pat_set $pat_sets_coll {
  set pat_set_data_coll [get_config_elements -in_wrapper $pat_set PatternSetData(*)]
  foreach_in_collection pat_set_data $pat_set_data_coll {
    set ip_grp_type [get_config_value -in_wrapper $pat_set_data InstrumentGroupType]
    if { $ip_grp_type ne "UserDefinedRetargetedATPG" } {
      return -code error "Expected result for UserDefinedRetargetedATPG; got '$ip_grp_type'"
    }
    set core_module_coll [get_config_elements -in_wrapper $pat_set_data CoreModules(*)]
    foreach_in_collection coreModuleObj $core_module_coll {
      set module_name [get_config_value -in_wrapper $coreModuleObj CoreModuleName]
      set core_insts [get_config_value -in_wrapper $coreModuleObj ParticipatingCoreInstances]
      set failing_core_insts [get_config_value -in_wrapper $coreModuleObj FailingCoreInstances]
      puts "core module name: '$module_name'"
      puts "\tparticipating core instances: '$core_insts'"
      puts "\tfailing core instances: '$failing_core_insts'"
    }
  }
}
```

Failing Flop Diagnosis for ATPG Patterns Examples

The following example shows a Tcl proc for creating a Tcl list of failing scan cells. At the end of the example, you see how you can use this proc to add cell constraints.

```
proc list_fail_chain_id {} {

  set failing_flop_data_collection [get_config_elements -in_wrapper /CDPResult/
TestExecutionResult/DiagnosticResult/PatternSets(0)/PatternSetData(0)/
FailingFlops(0) FlopData(*)]
  set raw_chain_id_list {}

  foreach_in_collection FlopData $failing_flop_data_collection {
    lappend raw_chain_id_list [ list  [get_config_value -in_wrapper
       $FlopData Chain] [get_config_value -in_wrapper $FlopData CellId]  ]
  }

  return [ join [ lsort -unique $raw_chain_id_list ] ]
}

ANALYSIS> set failing_scan_cells [ list_fail_chain_id ]
chain11 2 chain15 3 chain16 4 chain17 1
ANALYSIS> foreach { fail_chain fail_cell } [ list_fail_chain_id ] {
add_cell_constrain $fail_chain $fail_cell ox }
//  sub-command: add_cell_constraints chain11 2 ox
//  sub-command: add_cell_constraints chain15 3 ox
//  sub-command: add_cell_constraints chain16 4 ox
//  sub-command: add_cell_constraints chain17 1 ox
ANALYSIS> report_cell_constraints
//  constraint   chain     cell    constraint       pattern scope
//     value      name   position  properties
//  ----------  -------  --------  ----------  ----------------------
//       OX      chain11      2      (Dynamic)   Chain + Scan Patterns
//       OX      chain15      3      (Dynamic)   Chain + Scan Patterns
//       OX      chain16      4      (Dynamic)   Chain + Scan Patterns
//       OX      chain17      1      (Dynamic)   Chain + Scan Patterns
ANALYSIS>
```

The following example shows a Tcl proc for creating a Tcl list of failing scan cells, patterns, or chain/chainIDs. The proc checks whether results exists, whether hierarchical results exist, and then reports the results based on your input.

```
proc sid_list { { list_type cell } } {

  # Ensure proc is called with valid argument
  if { [ expr [ lsearch { chain_id pattern cell } $list_type ] == -1 ] } {
    puts -nonewline "//  Error: Invalid argument "
    puts $list_type
    puts "//         Valid arguments: chain_id pattern cell"
    return -1
  }

  # Determine if pseudo flop or failing flop results exist. Never
  # occur simultaneously.
  set pseudo_result [ get_name_list [ get_config_elements -in /CDPResult/
TestExecutionResult/DiagnosticResult/PatternSets(0)/PatternSetData(0)
FailingPseudoFlops(0) -silent ]]
  set flop_result   [ get_name_list [ get_config_elements -in /CDPResult/
TestExecutionResult/DiagnosticResult/PatternSets(0)/PatternSetData(0)
FailingFlops(0) -silent ]]
  if { $pseudo_result != "" } {
      puts "//  Note: Returning failing pseudo flop results"
      set failing_flopdata_collection [ get_config_elements -in /
CDPResult/TestExecutionResult/DiagnosticResult/PatternSets(0)/
PatternSetData(0) FailingPseudoFlops(*)]
      set failing_flop_collection [ get_config_elements -in /CDPResult/
TestExecutionResult/DiagnosticResult/PatternSets(0)/PatternSetData(0)/
FailingPseudoFlops(0) FlopData(*)]
  } elseif {  $flop_result != "" } {
      puts "//  Note: Returning failing flop results"
      set failing_flopdata_collection [ get_config_elements -in /
CDPResult/TestExecutionResult/DiagnosticResult/PatternSets(0)/
PatternSetData(0) FailingFlops(*)]
      set failing_flop_collection [ get_config_elements -in /CDPResult/
TestExecutionResult/DiagnosticResult/PatternSets(0)/PatternSetData(0)/
FailingFlops(0) FlopData(*)]
  } else {
    puts "//  Error: No failing flop or pseudo flop data found in /
CDPResult"
    return -1
  }

  # Determine if more than one set of results exist. This could happen
  # if multiple cores fail.
  # If that is the case, only report first

  if [string match "*retarget*" [get_config_value /CDPResult/
TestExecutionResult/TestName] ] {
    set failing_flopset_coreinstance_list {}

    foreach_in_collection failing_flop_set $failing_flopdata_collection {
      lappend failing_flopset_coreinstance_list [get_name_list
$failing_flop_set]
      lappend failing_flopset_coreinstance_list [get_config_value -in
$failing_flop_set CoreInstance]
    }

    # Report core name (if applicable). Warn if multiple failing flop sets
    # (multiple cores) exist
    set m 0
```

```
    foreach { flopset core } $failing_flopset_coreinstance_list {
      incr m
      if { $m == 1 && $core !=""}  {
        puts -nonewline "//  Note: Returning data for core instance: "
        puts $core
      } elseif { $m > 1 } {
        puts -nonewline "//  Warning: Ignoring data in: "
        puts $flopset
        if { $core != "" } {
           puts -nonewline "//           Representing core instance: "
           puts $core
        }
      }
    }
  }
  puts "//"

  # Extract three separate lists from the failing flop data. Different
  # results for pseudo versus flop
  set raw_chain_id_list {}
  set raw_pattern_list {}
  set raw_cell_list {}


  if { $pseudo_result != "" } {
    foreach_in_collection FlopData $failing_flop_collection {
      lappend raw_chain_id_list [ list [get_config_value -in_wrapper
$FlopData ChannelId] [get_config_value -in_wrapper $FlopData
ChannelCycleId] ]
      lappend raw_pattern_list [get_config_value -in_wrapper $FlopData
PatternNumber]
    }
    switch $list_type {
      chain_id { return [ join [ lsort -unique $raw_chain_id_list ]] }
      pattern  { return [ lsort -integer -unique $raw_pattern_list  ] }
      cell     {
        puts "//  Error: Can't report failing cell names for pseudo-flop
results"
        return -1
      }
      default  {
        puts "//  Error: Unexpected"
        return -1
      }
    }
  } else {
    foreach_in_collection FlopData $failing_flop_collection {
      lappend raw_chain_id_list [ list [get_config_value -in_wrapper
$FlopData Chain] [get_config_value -in_wrapper $FlopData CellId] ]
      lappend raw_pattern_list [get_config_value -in_wrapper $FlopData
PatternNumber]
      lappend raw_cell_list [get_config_value -in_wrapper $FlopData
CellName]
    }
    switch $list_type {
      chain_id { return [ join [ lsort -unique $raw_chain_id_list ]] }
      pattern  { return [ lsort -integer -unique $raw_pattern_list  ] }
      cell     { return [ lsort -unique $raw_cell_list  ] }
```

```
      default  {
        puts "//  Error: Unexpected"
        return -1
      }
    }
  }
}

ANALYSIS> sid_list cell
//  Note: Returning failing flop results
//
cpu_i/uPMI/ix496 cpu_i/uPMI/ix506 cpu_i/uPMI/ix736 cpu_i/uPMI/ix926
ANALYSIS> sid_list chain_id
//  Note: Returning failing flop results
//
chain11 2 chain15 3 chain16 4 chain17 1
ANALYSIS> sid_list pattern
//  Note: Returning failing flop results
//
0 2 8 10 16 18 24 26 32 34 44 48 56 58 64 66 72 80 88 90 96 98 104 106 112
114 120 122 128 130 136 138 144 146 152 154 160 168 170 176 178 184 186 192
194 200 202 208 210 216 218 224 227 229 231 232 240 242 248 250 256 258 264
265 268 269 271 272 274 280 282 292 296 299 301 303 304 306 312 314 320 328
330 336 338 344 346 352 354 364 368 370 376 378 384 386 392 394 400 402 408
410 416 418 424 426 432 434 440 442 452 456 464 466 472 474 480 482 488 490
ANALYSIS>
```

Parallel Retention Test Diagnosis Considerations

In general, the MemoryBist parallel retention test specifically targets testing of cell retention time and should be viewed as a complement to the regular MemoryBist GO pattern testing. You should not expect bitmap failures to correspond between regular tests and parallel retention tests, aside from solid failures that occur in memory test algorithm phases that are common to both the parallel retention test and regular test. This is because the parallel retention tests runs only a subset of the memory test algorithms that the regular test runs.

Furthermore, when you deal with failures of a marginal or intermittent nature, you must consider how the test conditions differ between parallel retention tests and regular tests. These differences can cause intermittent failures to fluctuate in a different way or appear in only one of the two tests:

- Testing sequences are much shorter in the parallel retention test because of the small subset of the algorithm under test.
- The parallel retention test resynchronizes all controllers at the same point of the algorithm three times during the test, even with the retention time set to zero, whereas the regular test synchronizes all controllers only once, at the beginning of the test.

Chapter 8
Simulating Desktop, ATE, and ATPG Behavior

You can use SimDUT mode to simulate the behavior of LogicBIST, MemoryBIST, and ATPG instruments in Desktop mode, as well as MemoryBIST in offline ATE mode. In addition, you can simulate pattern execution as if using a specific adaptor as the execution device.

Note

SimDUT supports chip-level simulation, not block-level simulation. In addition, it supports Verilog and VHDL constructs; using other languages, such as System Verilog, may cause issues with simulation.

Tessent SiliconInsight supports the following simulators:

- Siemens EDA ModelSim™/Questa™ simulator
- Cadence® Incisive® Enterprise Simulator commonly referred to as NCSim
- Synopsys® VCS® (Verilog compiler simulator)

Simulating Diagnosis of LogicBIST and MemoryBIST Instruments (TSDB Flow)

Simulation enables you to validate devices by simulating the application of test patterns on a Tessent SiliconInsight Desktop tester.

Prerequisites

- You have generated a setup specification for SimDUT as described in "Prepare the Design Under Test."
- You have a signed-off chip-level TSDB. For more information about TSDBs, refer to Chapter 9 in the *Tessent Shell Reference Manual*.
- You have Verilog design files and pattern specification files.
- You have a licensed installation of a supported simulator.
- You can load the design's Verilog netlist into the simulator. As a recommendation, simulate your Verilog netlist with a testbench file to ensure that the netlist is a functional design netlist.

Procedure

1. From a shell, invoke Tessent Shell using the following syntax:

   ```
   % tessent -shell
   ```

   After invocation, the tool is in setup mode. Refer to the "tessent" command description in the *Tessent Shell Reference Manual* for additional invocation options.

2. Set the tool context to enable Tessent SiliconInsight within Tessent Shell:

   SETUP>set_context patterns -silicon_insight

3. Load the TSDBs. You must load the individual TSDBs for the cores and top level, set the top-level design, and then set the current design. For example:

   SETUP>open_tsdb core1.tsdb
   SETUP>open_tsdb core2.tsdb
   SETUP>open_tsdb top.tsdb
   SETUP>read_design top -design_id rtl -view interface
   SETUP>set_current_design top

   The tool loads the current design, the current design's top-level ICL, and associated patterns specification.

4. Load the setup specification that specifies to use SimDUT mode:

   SETUP>read_config_data *setup_spec_path*

   Optionally, your setup specification can include a SimDUT startup dofile to automatically start the simulator and load the design files.

5. Specify any additional library sources necessary for simulation:

   SETUP>set_simulation_library_sources -v *file_path* **-v** *file_path* **...**

   The set_simulation_library_sources command defines the source directories and files in which to search for the simulation Verilog library cells that the run_testbench_simulations command uses. In particular, specify the top design name file and the top module ports files.

6. Run the simulation using the simulation commands you previously used for TSDB sign-off.

Note

SimDUT simulates the patterns as if they are running on Desktop, which means that timing restrictions may be more relaxed than pattern execution on a tester.

```
#run the following command for the ModelSim/Questa simulator
SETUP>run_testbench_simulations -simulator_options "+nospecify" -simdut_server

#run the following command for the NC-Sim simulator
SETUP>run_testbench_simulations -simulator_options "nc-sim_options" \
        -simulator incisive -simdut_server

#run the following command for the VCS simulator
SETUP>run_testbench_simulations -simulator vcs -simdut_server
```

The run_testbench_simulations command compiles the source files and invokes the simulator to run the testbench. When you have specified the top design name and top module ports, the tool generates a *SIMDUT.v* testbench file by default.

To generate the *SIMDUT.v* testbench file without starting a simulation, run run_testbench_simulation with the -generate_scripts_only switch. This enables you to manually modify the *SIMDUT.v* file and then use it in the simulation by specifying the path to the *SIMDUT.v* file with the run_testbench_simulations -simdut_output_directory switch.

Refer to the *Tessent Shell Reference Manual* for the complete syntax for these commands. Refer to "Adding Asynchronous Clocks to Simulations" for information about adding free-running clocks to the simulation.

7. Start Tessent SiliconInsight:

```
SETUP>start_silicon_insight -gui
```

8. Run and diagnose tests as described in "Performing Diagnosis and Interpreting the Results (TSDB Flow)" on page 133.

9. (*Optional*) Inject faults.

For example, the following command injects a stuck-at 0 fault at the SIMDUT_TB.DUT_inst.memory00.q_a signal.

```
SETUP>add_simdut_fault -signal SIMDUT_TB.DUT_inst.memory00.q_a -fault 0
```

Note

The simulator injects faults on nets, not ports.

a. If a net consists of escaped identifiers, you can inject a fault as shown in the following command:

```
SETUP>add_simdut_fault -signal { /SIMDUT_TB.DUT_inst/m8051_i/u4/\
LDATAA[3]  } -fault 1
```

The braces ({ }) prevent the tool from flattening the pathname thus making it possible to use fault injection. You can specify the add_simdut_fault command multiple times. The tool supports stuck-at 0 (SA0) and stuck-at 1 (SA1) fault injection.

b. To delete injected faults, specify the delete_simdut_fault command.

c. To list the simdut faults that are currently active in the simulator, specify the report_simdut_faults command.

10. Run and diagnose tests as described in "Performing Diagnosis and Interpreting the Results (TSDB Flow)" on page 133.

11. Exit the Tessent SiliconInsight session:

```
SETUP>stop_silicon_insight
```

Results

After running a given test, the tool generates a tabulated report that contains the results from the simulated test execution. Whereas the results are based on simulation, the format of the results is identical to those returned by Tessent SiliconInsight when you test the actual silicon.

The example shows the variable failure results for test1.

```
//  sub-command: execute_cdp_test test1 -collect_data_type_list variable
//  Test 'test1' failed.
//  Variable failures and unmapped failures of pattern 'test1.svf' :
//
//  PatternSet    Variable            Pin                    Expected
//                                                             Actual
//  ------------  ------------------  ---  -------------------------
//  lbist_normal  top_edt_i.misr(51)  tdo  b110011111100010101101101
//                                         b010101101100101001101101
//  ------------  ------------------  ---  -------------------------
//  lbist_normal  top_edt_i.misr(77)  tdo  b000110101100111110010010
//                                         b000110111000111110000110
//  ------------  ------------------  ---  -------------------------
//  lbist_normal  top_edt_i.misr(92)  tdo  b101101110100001110001011
//                                         b111010101110101001010101
//  ------------  ------------------  ---  -------------------------
```

Examples

The following dofile example shows the SimDUT simulation process using NC-Sim.

```
set_context pattern -silicon_insight
open_tsdb design1/top.tsdb
file mkdir design1_outdir
set_tsdb_output_directory design1_outdir
create_silicon_insight_setup_spec -select_current_setup Sid(simdut)
read_config_data design1/my_patspec.cfg
set_sim_lib_sources -v techlib_adk.tnt/4.1/verilog/adk.v -v data/mem/*.v
   -v data/picdram.v

run_testbench_simulations -simdut_server -simulator_options
   {-timescale "1ns/1ps"} -simulator incisive
start_silicon_insight
execute_cdp_test LogicBist -collect_data_type_list variable
stop_silicon_insight
```

Related Topics

SimDUT Signal Naming Convention

SimDUT Fault Injection for Memory Models

Simulating Diagnosis of ATPG Patterns (Non-TSDB Flow)

When you are simulating test patterns for ATPG and other non-TSDB designs, you must generate the SIMDUT.v file prior to running the simulation.

Prerequisites

- You have generated a setup specification for SimDUT and created the SIMDUT.v and design source dictionary as described in "Generating the Setup Specification for ATPG (Non-TSDB Flow) Desktop Mode" on page 79.
- You have Verilog design files and pattern specification files.
- You have a licensed installation of a supported simulator.
- You can load the design's Verilog netlist into the simulator. As a recommendation, simulate your Verilog netlist with a testbench file to ensure that the netlist is a functional design netlist.

Procedure

1. From a shell, invoke Tessent Shell using the following syntax:

   ```
   % tessent -shell
   ```

   After invocation, the tool is in setup mode. Refer to the "tessent" command description in the *Tessent Shell Reference Manual* for additional invocation options.

2. Set the tool context to enable Tessent SiliconInsight within Tessent Shell:

   SETUP>set_context patterns -silicon_insight

3. Load the setup specification that specifies to use SimDUT mode:

   SETUP>read_config_data *setup_spec_path*

   Optionally, your setup specification can include a SimDUT startup dofile to automatically start the simulator and load the design files.

4. Run the simulation by running one of the following commands, depending on the simulator you are using:

   Note

   SimDUT simulates the patterns as if they are running on Desktop, which means that timing restrictions may be more relaxed than pattern execution on a tester.

   ```
   #run the following command for the ModelSim/Questa simulator
   SETUP>run_testbench_simulations -simulator_options "+nospecify" \
           -simdut_server -simdut_output_directory \
           ./simdut_outdir/top_w_hier.simdut_gate/SIMDUT.simulation

   #run the following command for the NC-Sim simulator
   SETUP>run_testbench_simulations -simulator_options "nc-sim_options" \
           -simulator incisive -simdut_server -simdut_output_directory \
           ./simdut_outdir/top_w_hier.simdut_gate/SIMDUT.simulation

   #run the following command for the VCS simulator
   SETUP>run_testbench_simulations -simulator vcs -simdut_server \
           -simdut_output_directory \
           ./simdut_outdir/top_w_hier.simdut_gate/SIMDUT.simulation
   ```

   The run_testbench_simulations command compiles the source files and invokes the simulator to run the testbench.

   The -simdut_output_directory references the directory that contains the SimDUT setup files, which you generated at the same time you generated the setup specification for ATPG. See "Generating the Setup Specification for ATPG (Non-TSDB Flow) Desktop Mode" for details.

5. Start Tessent SiliconInsight:

   SETUP>start_silicon_insight -gui

6. Run and diagnose tests as described in "Performing Diagnosis and Interpreting the Results (TSDB Flow)" on page 133.

7. (*Optional*) Inject faults.

   For example, the following command injects a stuck-at 0 fault at the SIMDUT_TB.DUT_inst.memory00.q_a signal.

   SETUP>add_simdut_fault -signal SIMDUT_TB.DUT_inst.memory00.q_a -fault 0

   Note

   The simulator injects faults on nets, not ports.

a. If a net consists of escaped identifiers, you can inject a fault as shown in the following command:

```
SETUP>add_simdut_fault -signal { /SIMDUT_TB.DUT_inst/m8051_i/u4/\
LDATAA[3]   } \
        -fault 1
```

The braces ({ }) prevent the tool from flattening the pathname thus making it possible to use fault injection. You can specify the add_simdut_fault command multiple times. The tool supports stuck-at 0 (SA0) and stuck-at 1 (SA1) fault injection.

b. To delete injected faults, specify the delete_simdut_fault command.

c. To list the simdut faults that are currently active in the simulator, specify the report_simdut_faults command.

8. Run and diagnose tests as described in "Performing Diagnosis and Interpreting the Results (TSDB Flow)" on page 133.

9. Exit the Tessent SiliconInsight session:

```
SETUP>stop_silicon_insight
```

Related Topics

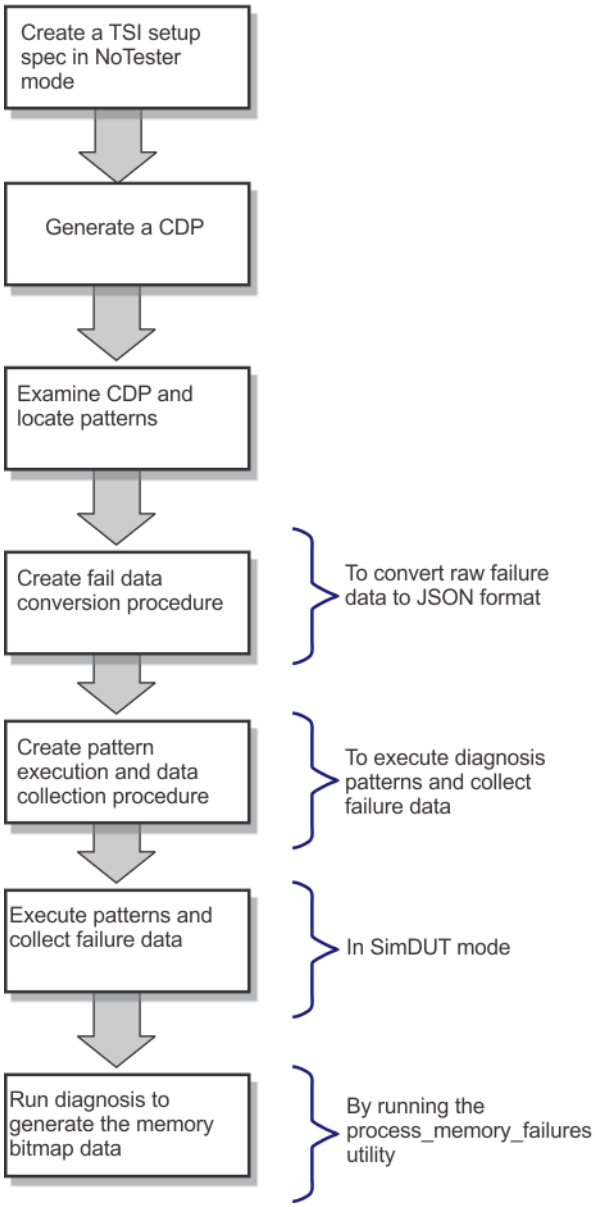
SimDUT Signal Naming Convention

SimDUT Fault Injection for Memory Models

Simulating Diagnosis Patterns in Offline ATE Mode

For pre-silicon validation, you can simulate the application of ESOE MemoryBIST patterns on a Tessent SiliconInsight Desktop tester. The simulator converts the responses to failures that you can then diagnose with Tessent SiliconInsight Desktop.

Figure 8-1. High-Level Flow for Diagnosing ESOE MemoryBIST Instruments by Simulating an ATE



Prerequisites

- You have generated a setup specification for SimDUT as described in "Prepare the Design Under Test" on page 75.
- You have a signed-off chip-level TSDB. For more information about TSDBs, refer to Chapter 9 in the *Tessent Shell Reference Manual*.
- You have Verilog design files and pattern specification files.
- You have a licensed installation of a supported simulator.

- You can load the design's Verilog netlist into the simulator. As a recommendation, simulate your Verilog netlist with a testbench file to ensure that the netlist is a functional design netlist.

Procedure

1. From a shell, invoke Tessent Shell using the following syntax:

   ```
   % tessent -shell
   ```

   After invocation, the tool is in setup mode. Refer to the "tessent" command description in the *Tessent Shell Reference Manual* for additional invocation options.

2. Set the tool context to enable Tessent SiliconInsight within Tessent Shell:

   ```
   SETUP>set_context patterns -silicon_insight
   ```

3. Load the TSDBs. You must load the individual TSDBs for the cores and top level, set the top-level design, and then set the current design. For example:

   ```
   SETUP>open_tsdb core1.tsdb
   SETUP>open_tsdb core2.tsdb
   SETUP>open_tsdb top.tsdb
   SETUP>read_design top -design_id rtl -view interface
   SETUP>set_current_design top
   ```

   The tool loads the current design, the current design's top-level ICL, and associated patterns specification.

4. Load the previously created pattern specification. For example:

   ```
   SETUP>read_config_data my_patspec.cfg
   ```

5. Specify to generate the patterns in STIL format:

   ```
   SETUP>set_config_value PatternsSpecification(top,rtl,manufacturing)/
       manufacturing_patterns_formats stil
   ```

6. Start Tessent SiliconInsight:

   ```
   SETUP>start_silicon_insight -gui
   ```

7. Enable diagnosis for MemoryBIST as follows:

   a. Select a MemoryBIST controller in the Config data wrappers pane on the left side of the **Config Data Browser** tab.

   b. To display the CharacterizationDebugOptions wrapper, click the right arrow (▸) buttons in the Config data wrappers pane. Select CharacterizationDebugOptions and double-click diagnosis_on_failure in the property pane to enable it.

   c. Click **Apply**.

Note

Regardless of the setting of the compare_go and compare_go_id properties in the DiagnosisOptions for a given controller in the main pattern, the ESOE diagnosis patterns is automatically generated for this controller if the diagnose_on_failure property is set to On.

8. Generate the CDP by selecting **PatternsSpecification** under the Config data wrappers pane and then clicking **Process/Execute** (). You can also generate the CDP by using the following command:

   SETUP>process_patterns_specification

9. Exit the session.

   SETUP>stop_silicon_insight

10. After examining the CDP and locating the patterns, source the following Tcl procedure. This Tcl procedure saves the raw failure data that you collect in JSON format.

```
proc save_json_failure_log_version_2 { flog_fname cdp test_name diag_pat_name pin
raw_fail_list} {
  set json_str "\{ \n\
    \"Version\": 2, \n\
    \"Type\": \"ESOE\", \n\
    \"TrackingInfo\": \{\"Design\": \"$cdp\"\}, \n\
    \"FailDataType\": \"RawSTIL\", \n\
    \"RawSTIL\": \[\{ \n\
        \"UserTestName\": \"$test_name\", \n\
        \"Failures\": \[\{ \n\
            \"TesterDiagPatternName\": \"$diag_pat_name\", \n\
            \"Pin\": \"$pin\", \n\
            \"StartingSOEID\": 1, \n\
            \"FailureFormat\": \"FailingCycles\", \n\
            \"FailData\": \["
  set fail_cnt 0
  foreach fail $raw_fail_list {
    if { $fail_cnt > 0 } {
      append json_str ","
    }
    append json_str "\n\t\t\t\t\"$fail\""
    incr fail_cnt
  }
  append json_str "\n\t\t\t\]\n\t\t\}\]\n\t\}\]\n\}"
  set chan [open $flog_fname w]
  puts $chan $json_str
  close $chan

  return -code ok
```

11. After initiating Tessent SiliconInsight in SimDUT mode, add your tests using the add_cdp_test command. Refer to "Setup Specifications for SimDUT and Offline ATE Modes" on page 82 for information about initiating and preparing SimDUT.

Note

SimDUT simulates the patterns as if they are running on Desktop, which means that timing restrictions maybe be more relaxed than pattern execution on a tester.

12. Generate a data collection loop as described in "Preparing the Test Patterns and the Test Program" on page 248 and collect the raw failure data. After you have collected the data, run the previously sourced Tcl procedure to save the data in JSON format.
13. Exit SimDUT mode and run the process_memory_failures utility.

Related Topics

MemoryBIST Offline ATE Diagnosis and Validation Tutorial

Performing Simulation Using Adaptors

Tessent SiliconInsight supports using the SimDUT simulation engine with your adaptors. This adaptor simulation enables you to validate diagnosis patterns as they would actually perform given a specific adaptor's characteristics, such as pin count and application frequency.

Restrictions and Limitations

- Adaptor simulation mode has the same restrictions and limitations as SimDUT mode. Refer to the Notes under "Simulating Desktop, ATE, and ATPG Behavior" on page 223 and "Simulating Diagnosis of LogicBIST and MemoryBIST Instruments (TSDB Flow)" on page 223.

Prerequisites

- You have generated a setup specification for an adaptor in simulation mode that specifies a startup dofile that automatically starts simulation. Refer to "Example 1: Setup Specification and SimDUT Startup Dofile" on page 234.
- You have a signed-off chip-level TSDB. For more information about TSDBs, refer to Chapter 9 in the *Tessent Shell Reference Manual*.
- You have Verilog design files and pattern specification files.
- You have a licensed installation of a supported simulator.
- You can load the design's Verilog netlist into the simulator. As a recommendation, simulate your Verilog netlist with a testbench file to ensure that the netlist is a functional design netlist.

Note

The line numbers used in this procedure refer to the command flow dofile shown in "Example 2: Dofile Example for Performing Adaptor Simulation" on page 235.

Procedure

1. In Tessent Shell, set the context and load the design files, TSDB, setup specification, and patterns specification. (See lines 1- 17.)

   See "Example 1: Setup Specification and SimDUT Startup Dofile" on page 234 for details about creating the setup specification when you are performing adaptor simulation.

2. (*Optional*) Set the option to ignore unconnected pins between the design (elaborated in SimDUT) and pins specified in the pinmap wrapper within the setup specification. (See lines 19-21.)

   Set this option when the design has more pins than the adaptor can handle, which is an important consideration for adaptors with four pins such as Olimex, IJTAGKey, and Flyswatter. The tool reports an error when a pattern has more pins than specified in the pinmap wrapper for the selected adaptor.

3. In Tessent SiliconInsight, create the CDP and run the tests. (See lines 23-53.) The following step is important if you have set the option to ignore unconnected pins.

   - (*Optional*) For each unconnected pin, specify the add_simdut_fault command (lines 32-37) to ensure that the unconnected pins are in a stable and known state. Otherwise, the test may fail.

Examples

Example 1: Setup Specification and SimDUT Startup Dofile

Specify the setup specification as follows when performing adaptor simulation. The selected setup must specify an adaptor, not SimDUT mode. The following example configures the Opal Kelly adaptor to perform simulation with a timeout of 1200 seconds and using a SimDUT startup dofile.

```
SiliconInsightSetupSpecification {
  selected_setup : Sid(opalKellyXem6310);
  Protocol {
    SvfTapPins {
     TRST : trst;
     TDI  : tdi;
     TMS  : tms;
     TDO  : tdo
     TCK  : tck;
    }
  }
  Sid(opalKellyXem6310) {
    cdp_directory : top.cdp;
    opalKellyXem6310 {
      io_standard : 1.80;
      base_frequency : 10MHz;
      on_chip_termination : off
      PinMap {
        P0 : trst;
        P1 : tdi;
        P2 : tms;
        P3 : tdo;
        P4 : tck;
        P5 : edt_bypass;
        P6 : edt_single_bypass_chain;
        P7 : edt_clock;
        P8 : refclk;
      }
      operation_mode : simulation;
      timeout : 1200;
      startup_dofile : simdut_startup.dofile;
    }
  }
}
```

The following example shows a simdut_startup.dofile that automatically loads additional library sources for simulation and starts the simulator (in this case, ModelSim/Questa):

```
set_simulation_library_sources -v ../verilog/adk.v \-v ../data/mem/*.v -v
../data/picdram.v
run_testbench_simulations -simulator_options "+nospecify" -simdut_server
```

Example 2: Dofile Example for Performing Adaptor Simulation

The following dofile example shows a typical command flow as detailed in the procedure listed above. The highlighted command lines illustrate additional considerations when performing simulation with adaptors.

```
1  # Set context to silicon_insight
2
3  set_context pattern -silicon_insight
4
5  # Read in TSDB and designs
6
7  open_tsdb top.tsdb
8  read_design top -design_id rtl -view interface
```

```
set_current_design top

# Load the setup specification

read_config_data si.setup

# Read in the patterns specification of interest

read_config_data my_patspec.cfg

# Set option to ignore uncontacted pin data

set_si_options -ignore_uncontacted_pin_data on

# Start Tessent SiliconInsight

start_silicon_insight

# Generate the CDP from the pattern specification

process_patterns_specification -unprocessed_only -config_objects \
/PatternsSpecification(top,rtl,manufacturing)

# Set uncontacted pins to constant values

add_simdut_fault -signal SIMDUT_TB.DUT_inst.RST -fault 0
add_simdut_fault -signal SIMDUT_TB.DUT_inst.reset -fault 0
add_simdut_fault -signal SIMDUT_TB.DUT_inst.trst -fault 1
add_simdut_fault -signal SIMDUT_TB.DUT_inst.edt_clock -fault 0

# Run the patterns

execute_cdp_test lbist_patt0 -collect_data_type_list \
{ variable instrument diagnosis }

# Inject faults and re-run the test
add_simdut_fault -signal SIMDUT_TB.DUT_inst.piccpu_inst1.inst_reg_7.QB \
-fault 1
add_simdut_fault -signal SIMDUT_TB.DUT_inst.blockA_inst.mem1.Q[0] \
-fault 1
execute_cdp_test lbist_patt0 -collect_data_type_list \
{ variable instrument diagnosis }

stop_silicon_insight
exit
```

Related Topics

Generating the Setup Specification for Desktop Mode (TSDB Flow)

Simulating Diagnosis of LogicBIST and MemoryBIST Instruments (TSDB Flow)

Save Simulation Waveforms as VCD Files

When you have simulation mismatches, you may want to save the waveform results in a VCD file so that you can view them in a schematic viewer.

Save the waveform results by specifying the -store_simulation_waveforms option. For example:

```
SETUP>run_testbench_simulations -simulator_options "+nospecify" -simdut_server
      -store_simulation_waveforms on
```

The VCS and Incisive (irun) tools store the waveform results in VCD files. ModelSim/Questa provides a *vsim.wlf* file that you convert to with the following command:

```
wlf2vcd vsim.wlf <vcd_filename>.vcd
```

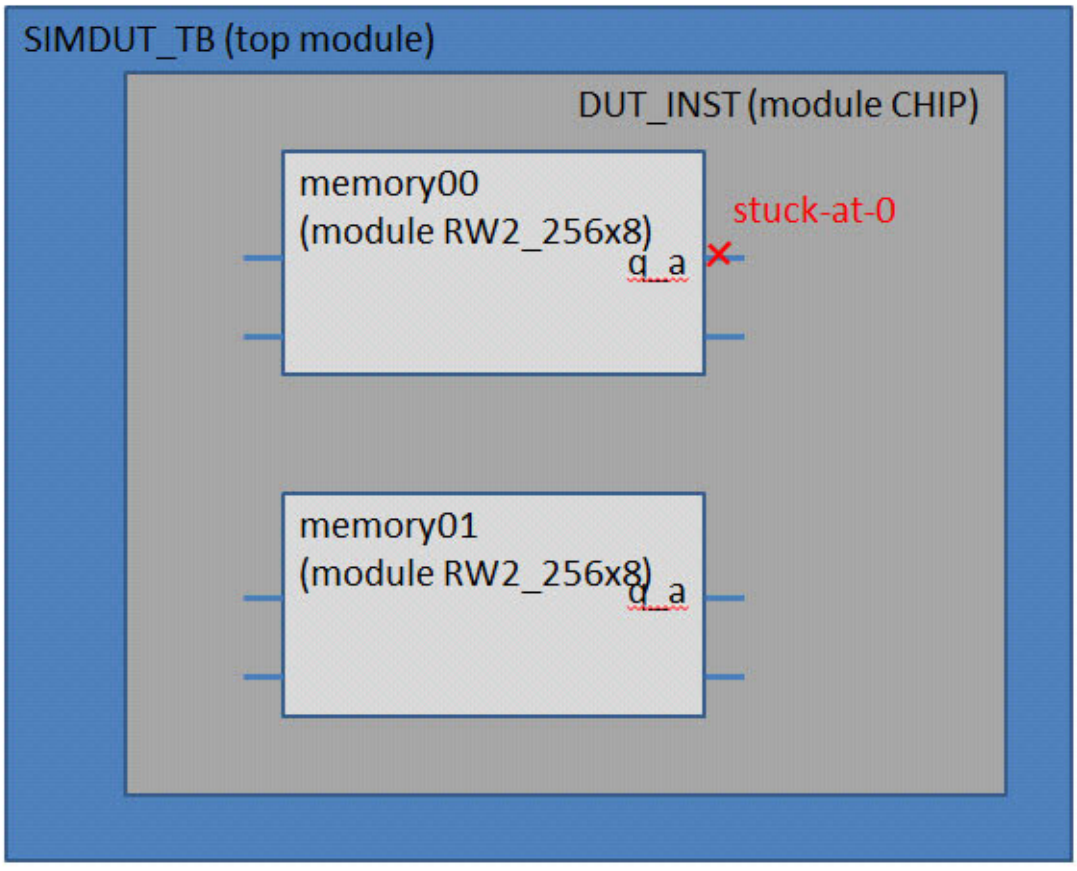
SimDUT Fault Injection and Signal Naming Convention

When you inject a stuck-at-0 fault or a stuck-at-1 fault into the specified signal, SimDUT uses the Verilog simulation capabilities to force the signal to 0 or 1, respectively. When you specify a pin pathname for the fault you want to inject, Verilog forces the associated signal to the specified value.

For example, given the following command, SimDUT injects a stuck-at fault on a specified signal, as shown in the figure.

```
add_simdut_fault -signal SIMDUT_TB.DUT_inst.memory00.q_a -fault 0
```

Figure 8-2. SimDUT Stuck-At Fault Insertion



When you inject a fault at a specific pin pathname, SimDUT actually injects the fault at the whole net, not only on the specific pin. The fault acts like a fault injected on the upstream driver pin of the associated net. If you specify the input port of a cell and this port is attached to a net with fanout, then the fault acts like a fault injected on the driver port of that net.

SimDUT Signal Naming Convention

For the add_simdut_fault command, derive the name for the signal using the Verilog source code. You can use "." or "/" to separate the names of the modules and pins.

The following example shows a sample Verilog netlist that has two modules corresponding to Figure 8-2.

```
module SIMDUT1;
  CHIP SIMDUT_TB.DUT_inst(…);

endmodule

module CHIP (…);
……
    RW2_256x8 memory00 (…);
    RW2_256x8 memory01 (…);

……
endmodule

module RW2_256x8 (......);
......
        output  q_a;
......
endmodule
```

The signal name starts with the top-module name followed by the lower-level module names based on the hierarchy of the Verilog netlist. By following the hierarchy shown in the sample Verilog netlist, you derive the signal name "SIMDUT_TB.DUT_inst.memory00.q_a."

SimDUT Fault Injection for Memory Models

SimDUT fault injection relies on the Verilog simulator to force values on specified signals. This means that the actual resolution of injected faults depends on the memory model you use.

- **High-level RTL memory models —** Injecting faults might be limited to the external interface of the memory, that is, the address or data ports.
- **Low-level gate memory models —** To inject faults at the cell level, the internal structure of the memory module must be represented in a model you are using.

Examples

The first example shows the Verilog source code for a high-level RTL memory model in which fault injection at the cell level is not possible. The second example shows the Verilog source code for a low-level memory model for injecting faults at the cell level.

High-Level RTL Memory Model

With this model, you can only inject faults on the memory interface, as follows:

add_simdut_fault -signal { /mem_inst/q[3] } -fault 0

```
module SYNC_1RW_64x8 (
   Q,
   CLK,
   CEN,
   WEN,
   A,
   D,
   OEN
);
   reg [BITS-1:0]          MEM [word_depth-1:0];
   reg [BITS-1:0]          Q_REG;

assign Q = (~OEN) ? Q_REG : wordz ;
integer i;
initial begin
    for (i=0;i<word_depth; i=i+1)
        begin
            MEM[i]={BITS{1'b0}};
        end
    Q_REG={BITS{1'b0}};
end

always @ (posedge CLK) begin
   if (~CEN) begin
     if (~WEN) begin
         MEM[A] <= D;
         Q_REG  <= D;
      end else begin
         Q_REG  <= MEM[A];
      end
   end
end
endmodule
```

Low-Level Gate Memory Model

In this example, all memory cells are represented by a distinct entity in the memory model. Perform fault injection at the cell level as follows:

add_simdut_fault -signal { /mem_instance/mem_cell[63][4]/q } -fault 0

You can generate low-level memory models by synthesizing high-level RTL models.

```
module SYNC_1RW_64x8 ( Q, CLK, CEN, WEN, A, D, OEN );
  output [7:0] Q;
  input [5:0] A;
  input [7:0] D;
  input CLK, CEN, WEN, OEN;
  wire \MEM[63][7];
  wire \MEM[63][6];
  wire \MEM[63][5];
  ...
  wire \MEM[0][0];

  DFF \MEM_reg[63][7]   ( .D(n682), .CLK(CLK), .Q(\MEM[63][7] ) );
  DFF \MEM_reg[63][6]   ( .D(n681), .CLK(CLK), .Q(\MEM[63][6] ) );
  DFF \MEM_reg[63][5]   ( .D(n680), .CLK(CLK), .Q(\MEM[63][5] ) );
  ...
endmodule
```

Adding Asynchronous Clocks to Simulations

In the TSDB flow, you can add free-running clocks to simulations. To do this, you must update the patterns specification.

Prerequisites

- You have Verilog design files and pattern specification files.

Procedure

1. To add asynchronous clocks to a simulation:

If you want to...	Do the following:

Define additional asynchronous clocks	Within the patterns wrapper, modify (or add) a ClockPeriods data wrapper. For example:
	`Patterns(LogicBist) {`
	`  ClockPeriods {`
	`  clk : 10ns;`
	`  **edt_clock : 12ns;**`
	`  }`
	`  TestStep(serial_load_0) {`
	`  …`
	`  }`
	`}`

Add clocks for ATPG tests	Use the following command: `add_clocks`

2. Regenerate the patterns after adding asynchronous clocks.

Chapter 9
Performing Diagnosis From an ATE Test Program in Offline ATE Mode

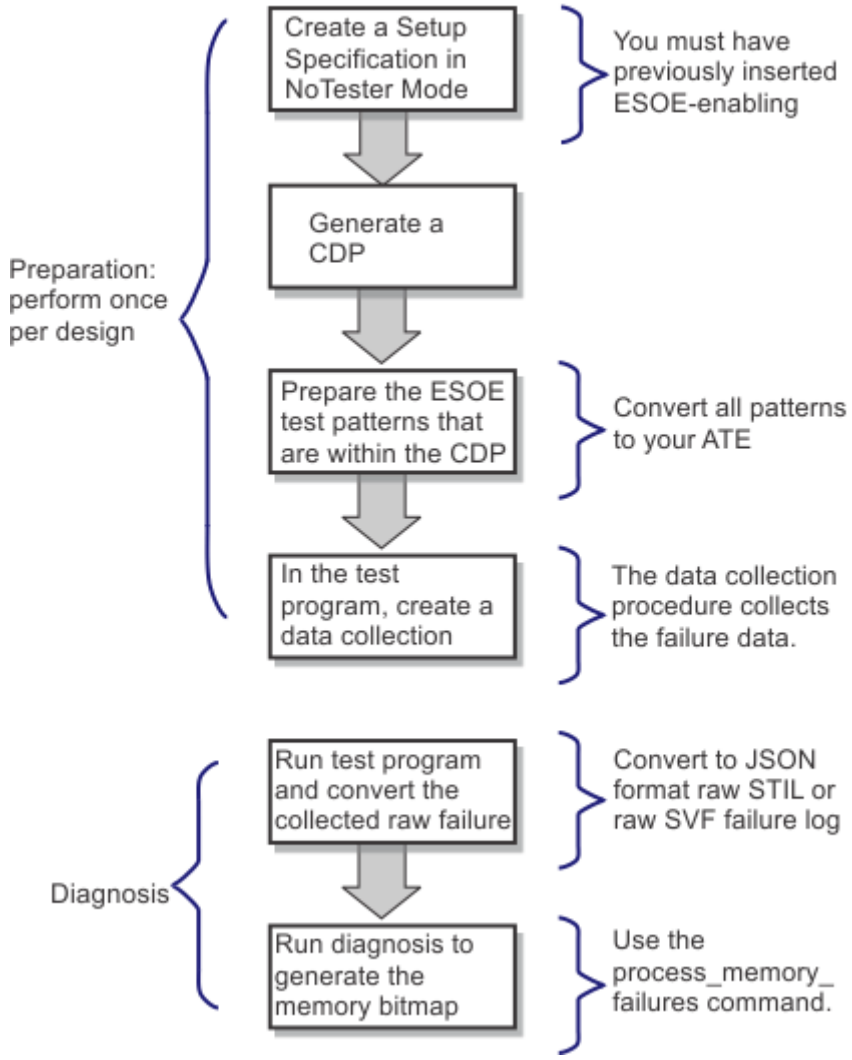
Tessent SiliconInsight supports using ATE test programs in offline ATE diagnosis mode both for MemoryBIST instruments that are ESOE-enabled and for ATPG flop failure diagnosis. For MemoryBIST instruments, you can diagnose the instruments, generate failing memory reports for any MemoryBIST instrument, and generate setup chain test failure reports. For ATPG flop failure diagnosis, you can pregenerate 1hot patterns and run tests to create failure log files that you can analyze offline. In these usage models, the CDP software is not integrated with the ATE.

ESOE Diagnosis Flow

Performing ESOE diagnosis from an ATE test program requires some preparation steps.

Figure 9-1 shows the high-level flow for diagnosing memory with ESOE test patterns.

Figure 9-1. High-Level Flow for Diagnosing ESOE MemoryBIST Instruments From an ATE



The following topics describe the steps in the ESOE diagnosis flow:

Flow Differences Between the LV Flow and Tessent Shell Flow

The Tessent Shell flow differs from the LV flow at a number of crucial points.

The following table summarizes the differences between the Tessent SiliconInsight for the LV flow and the Tessent SiliconInsight for Tessent Shell flow for offline ESOE MemoryBIST.

Table 9-1. Flow Differences Between the LV Flow and Tessent Shell Flow

Step	LV Flow	Tessent Shell
Enable ESOE diagnosis.	In the etmanufacturing file, set the CompareGoID property to On.	In the patterns specification, set the diagnose_on_failure property in the CharacterizationDebugOptions wrapper to On.
Generate the CDP with ESOE diagnosis patterns.	Invoke the ETVerify (etv) tool.	Use the process_patterns_specification command after starting Tessent SiliconInsight, opening the TSDB, reading the design, reading the config data (patterns specification) and creating/loading the SiliconInsight setup file.
Prepare the test patterns.	Locate the ESOE test patterns within the CDP directory structure and use your pattern translator to convert them to the format supported by your ATE.	Locate the unique ESOE test patterns for each test within the CDP directory structure by using the summary.json file and your pattern translator to convert them to the format supported by your ATE.

Table 9-1. Flow Differences Between the LV Flow and Tessent Shell Flow

Step	LV Flow	Tessent Shell
STIL raw failure log files JSON format.	"Version" : 1 "FailingCycles" identifier provides the lists of failing cycles.	"Version" : 2 "Failures" list after "UserTestName" enables multiple sets of ESOE test patterns for the same user test and for each item in the list: • "TesterDiagPatternName" identifier. • "Pin" identifier to identify the scanout pin. • "FailureFormat" : "FailingCycles" identifier. • "FailData" identifier to provide the lists of failing cycles.
SVF raw failure log files JSON format.	"Version" : 1 "CommandID:BitOffsets" identifier provides the lists of failing SVF commands and bit offsets.	"Version" : 2 "Failures" list after "UserTestName" enables multiple sets of ESOE test patterns for the same user test and for each item in the list: • "TesterDiagPatternName" identifier. • "Pin" identifier to identify the scanout pin. • "FailureFormat" : "CommandID:BitOffsets" identifier. • "FailData" identifier to provide the lists of failing SVF command and bit offsets.
Bitmap files in JSON format.	--	FailuresInfo block to provide informational messages for certain failures when applicable.More supported parameters: ControllerDesignName, Algorithm, MemoryDesignName, Addressport.

Generating a CDP

Before performing ATE-driven diagnosis, you must generate a CDP. The CDP contains test patterns and mapping information that is used to map raw failure data to memory bitmaps.

Prerequisites

- When generating and inserting the MemoryBIST IP using a DFT specification, you have inserted the required ESOE-enabling hardware into the MemoryBIST controller. Refer to "Enhanced Stop-On-Nth-Error Approach" in the *Tessent MemoryBIST User's Manual* for details.
- When generating the MemoryBIST test patterns with ESOE diagnosis enabled and a controller is testing both RAM and ROM memories, you must limit diagnosis to the RAM controller steps by creating a pattern for each of the RAM controller steps and specifying the freeze_step property in the PatternsSpecification/Patterns/TestStep/Controller/AdvancedOptions wrapper.
- As a recommendation, use the freeze_step property as described above if a small (< 4096) hardware failure limit for the controller was previously specified or when one of the memories has a catastrophic failure. In the latter case, you may also want to use the AdvancedOptions/enable_memory_list property to select a limited number of memories within the frozen controller step for testing and diagnosis.
- You have a signed-off chip-level TSDB.
- If you have a legacy non-Tessent Shell CDP, refer to merge_cdp in the *Tessent Shell Reference Manual* for information about merging its contents into the current Tessent Shell-based CDP. You can also merge a pre-existing Tessent Shell-based CDP into the new CDP using this command.

Procedure

1. Start Tessent SiliconInsight:

   ```
   SETUP>start_silicon_insight
   ```

2. Open the TSDB:

   ```
   SETUP>open_tsdb top.tsdb
   SETUP>read_design top -design_id rtl -view interface
   SETUP>set_current_design top
   ```

3. Generate a setup specification in NoTester(batch_cdp) mode as described in "Generating the Setup Specification for Desktop Mode (TSDB Flow)" on page 76.

   NoTester(batch_cdp) is the default mode, so you do not need to customize the setup specification. The create_silicon_insight_setup_spec command automatically loads the setup specification.

4. Load the patterns specification of interest:

   SETUP>read_config_data *pattern_specification_name*

Note

Regardless of the setting of the compare_go and compare_go_id properties in the DiagnosisOptions for a given controller in the main pattern, the ESOE diagnosis patterns are automatically generated for this controller if the diagnose_on_failure property in the CharacterizationDebugOptions is set to On.

5. (*optional*) Add the necessary properties to the patterns specifications to generate physical X-Y coordinate data for failing RAM memory cells and add the data to the bitmap file results.

   The X-Y coordinate feature is an optional add-on to the ESOE diagnosis flow. Refer to "Adding Physical X-Y Coordinate Data to Bitmap Reports" on page 268" for details about how to implement this feature.

6. Generate the CDP:

   SETUP>process_patterns_spec

7. Stop Tessent SiliconInsight:

   SETUP>stop_silicon_insight

Results

Tessent SiliconInsight generates a CDP that contains a test for each patterns wrapper in the PatternsSpecification. By default, the CDP is named *design_name*.cdp and is located in the current working directory.

Preparing the Test Patterns and the Test Program

After generating a CDP that contains your test patterns, prepare the test patterns and the test program.

Prerequisites

- You have generated a CDP that contains test patterns as described in "Generating a CDP" on page 247.

Procedure

1. Locate the unique test patterns for each test within the directory structure using the summary.json file located in the test directory:

   cdp_name/Tests/<*user_test_name*>

   The unique pattern names are indicated in bold.

```
{"<pattern_set_name>": {"CDPMemoryDiagnosticAvailability": [{
   "TestMethod": "auto_increment_stop_on_error",
   "PatternDistribution": [{
     "InitPatternName": "<unique_pattern_name >_1.<svf|stil>",
     "ExecutePatternName": "<unique_pattern_name >_2.<svf|stil>",
     "IncrementPatternName": "<unique_pattern_name >_3.<svf|stil>",
         "TestedControllers": [{….
```

For a test performing the Parallel Retention Test, the summary.json file shows more patterns. In the following example, the “PreExecuteP1PatternName” and “PreExecuteP2PatternName” are optional and included only if you define the pre_first_pause_ate_callback and pre_second_pause_ate_callback options, respectively, in the PRTControl wrapper.

```
{
  "<pattern_set_name>": {"CDPMemoryDiagnosticAvailability": [{
   "TestMethod": "auto_increment_stop_on_error",
   "PatternDistribution": [ {
     "InitPatternName": "<unique_pattern_name>_6.<svf|stil>",
     "ExecutePatternName": "<unique_pattern_name>_7.<svf|stil>",
     "IncrementPatternName": "<unique_pattern_name>_8.<svf|stil>",
     "TestedControllers": [ { …
     } ]
   },{
     "InitPatternName": "<unique_pattern_name>_9.<svf|stil>",
     "ExecutePatternName": "<unique_pattern_name>_12.<svf|stil>",
     "IncrementPatternName": "<unique_pattern_name>_13.<svf|stil>",
     "PreExecuteP1PatternName": "<unique_pattern_name>_11.<svf|stil>",
     "PreExecuteSPPatternName": "<unique_pattern_name>_10.<svf|stil>",
     "TestedControllers": [ { …
     } ]
   },{
     "InitPatternName": "<unique_pattern_name>_14.<svf|stil>",
     "ExecutePatternName": "<unique_pattern_name>_19.<svf|stil>",
     "IncrementPatternName": "<unique_pattern_name>_20.<svf|stil>",
     "PreExecuteP1PatternName": "<unique_pattern_name>_16.<svf|stil>",
     "PreExecuteSPPatternName": "<unique_pattern_name>_15.<svf|stil>",
     "PreExecuteP2PatternName": "<unique_pattern_name>_18.<svf|stil>",
     "PreExecutePPPatternName": "<unique_pattern_name>_17.<svf|stil>",
     "TestedControllers": [ { …
```

2. Use your pattern translator to convert the patterns to the format supported by your ATE.

   Ensure that there is a one-to-one mapping between the STIL pattern cycle counts and the vector cycles of the translated ATE format. This mapping simplifies reporting of failing cycles in terms of the STIL pattern file.

3. In the test program, for each MemoryBist wrapper configuration, implement a data collection procedure that collects the failing cycle data.

   For a MemoryBist regular test, each data collection procedure consists of the following three patterns:

   - **Initialize (Init) —** Set the initial failure limit counter value (typically 0) and run once.

- **Execute (Exec) —** Run the pattern. Load the failure limit in the counter, perform the test, and scan out any failure data.
- **Increment (Inc) —** Increment the failure limit counter.

The Execute and Increment patterns iterate until the tool reaches the maximum number of failures or until the Execute pattern stops failing.

For a MemoryBist parallel retention test, a data collection procedure is required for each of the retention test phases (StartToPause, PauseToPause, and PauseToEnd), for which data collection is necessary.

For the StartToPause phase, data collection consists of the following three patterns:

- **Initialize (Init)** — Set the initial failure limit counter value (typically 0) and run once.
- **Execute (Exec)** — Run the pattern. Load the failure limit in the counter, perform the StartToPause phase test, and scan out any failure data.
- **Increment (Inc)** — Increment the failure limit counter.

The Execute and Increment patterns iterate until the tool reaches the maximum number of failures or until the Execute pattern stops failing.

For the PauseToPause phase, data collection consists of the following four or five patterns with optional instrument manipulations:

- **Initialize (Init)** — Set the initial failure limit counter value (typically 0) and run once.
- **Perform pre-StartToPause instrument manipulation (optional)** — Before the StartToPause phase, optionally vary the voltage, frequency, and other such factors.
- **Execute StartToPause (PreExecSP)** — Perform the StartToPause phase memory initialization. Ignore any failure data.
- **Perform pre-FirstPause instrument manipulation** (optional) — Before the first retention pause, optionally vary voltage, frequency, and other such factors. This is only possible if the following pattern (PreExecP1) exists.
- **Execute FirstPause (PreExecP1)** (optional) — This pattern only exists if the pre_first_pause_ate_callback option is defined in the PRTControl wrapper. Perform the first retention pause.
- **Perform pre-PauseToPause instrument manipulation** (optional) — Before the PauseToPause phase, optionally vary the voltage, frequency, and other such factors.
- **Execute (Exec)** — Run the pattern. Load the failure limit in the counter, perform the PauseToPause phase test, and scan out any failure data.
- **Increment (Inc)** — Increment the failure limit counter.

The StartToPause, FirstPause (optional), Execute, and Increment patterns, and the optional intermediate instrument manipulations, iterate until the tool reaches the maximum number of failures or until the Execute pattern stops failing.

For the PauseToEnd phase, data collection consists of the following five to seven patterns with optional instrument manipulations:

- **Initialize (Init)** — Set the initial failure limit counter value (typically 0) and run once.
- **Perform pre-StartToPause instrument manipulation** (optional) — Before the StartToPause phase, optionally vary the voltage, frequency, and other such factors.
- **Execute StartToPause (PreExecSP)** — Perform the StartToPause phase memory initialization. Ignore any failure data.
- **Perform pre-FirstPause instrument manipulation** (optional) — Before first retention pause, optionally vary the voltage, frequency, and other such factors. This is only possible if the following pattern (PreExecP1) exists.
- **Execute FirstPause (PreExecP1)** (optional) — This pattern only exists if the pre_first_pause_ate_callback option is defined in the PRTControl wrapper. Perform the first retention pause.
- **Perform pre-PauseToPause instrument manipulation** (optional) — Before the PauseToPause phase, optionally vary the voltage, frequency, and other such factors.
- **Execute PauseToPause (PreExecPP)** — Perform the PauseToPause phase memory initialization. Ignore any failure data.
- **Perform pre-SecondPause instrument manipulation** (optional) — Before the second retention pause, optionally the vary voltage, frequency, and other such factors. This is only possible if the following pattern (PreExecP2) exists.
- **Execute SecondPause (PreExecP2)** (optional) — This pattern only exists if the pre_second_pause_ate_callback option is defined in the PRTControl wrapper. Perform the second retention pause.
- **Perform pre-PauseToEnd instrument manipulation** (optional) — Before the PauseToEnd phase, optionally vary the voltage, frequency, and other such factors.
- **Execute (Exec)** — Run the pattern. Load the failure limit in the counter, perform the PauseToEnd phase test, and scan out any failure data.
- **Increment (Inc)** — Increment the failure limit counter.

The StartToPause, FirstPause (optional), PauseToPause, SecondPause (optional), Execute, and Increment patterns, and the optional intermediate instrument manipulations, iterate until the tool reaches the maximum number of failures or until the Execute pattern stops failing.

Each wrapper may contain multiple controllers, as shown in the following Examples section.

Results

The tool generates the following test patterns for the MemoryBist_P1 test:

- MemoryBist_P1.stil: GoNoGo pattern
- MemoryBist_P1_1.stil: ESOE initialize diagnostic pattern
- MemoryBist_P1_2.stil: ESOE execute diagnostic pattern
- MemoryBist_P1_3.stil: ESOE increment diagnostic pattern

For a MemoryBist parallel retention test, the number of test patterns the tool generates varies.

For the simplest setup, where the PRTControl wrapper is unspecified (that is, no ATE callback is specified), the tool generates the following patterns:

- MemoryBist_ParallelRetentionTest_P1.stil: PRT GoNoGo pattern
- MemoryBist_ParallelRetentionTest_P1_1.stil: ESOE initialize diagnostic pattern for StartToPause phase
- MemoryBist_ParallelRetentionTest_P1_2.stil: ESOE execute diagnostic pattern for StartToPause phase
- MemoryBist_ParallelRetentionTest_P1_3.stil: ESOE increment diagnostic pattern for StartToPause phase
- MemoryBist_ParallelRetentionTest_P1_4.stil: ESOE initialize diagnostic pattern for PauseToPause phase
- MemoryBist_ParallelRetentionTest_P1_5.stil: ESOE StartToPause memory initialization diagnostic pattern for PauseToPause phase
- MemoryBist_ParallelRetentionTest_P1_6.stil: ESOE execute diagnostic pattern for PauseToPause phase
- MemoryBist_ParallelRetentionTest_P1_7.stil: ESOE increment diagnostic pattern for PauseToPause phase
- MemoryBist_ParallelRetentionTest_P1_8.stil: ESOE initialize diagnostic pattern for PauseToEnd phase
- MemoryBist_ParallelRetentionTest_P1_9.stil: ESOE StartToPause memory initialization diagnostic pattern for PauseToEnd phase
- MemoryBist_ParallelRetentionTest_P1_10.stil: ESOE PauseToPause memory initialization diagnostic pattern for PauseToEnd phase
- MemoryBist_ParallelRetentionTest_P1_11.stil: ESOE execute diagnostic pattern for PauseToEnd phase

- MemoryBist_ParallelRetentionTest_P1_12.stil: ESOE increment diagnostic pattern for PauseToEnd phase

For the most complex setup, where the PRTControl wrapper specifies ATE callbacks for all phases of the Parallel Retention Test, the tool generates the following patterns:

- MemoryBist_ParallelRetentionTest_P1_1.stil: PRT GoNoGo StartToPause phase pattern
- MemoryBist_ParallelRetentionTest_P1_2.stil: PRT GoNoGo first retention time pause pattern
- MemoryBist_ParallelRetentionTest_P1_3.stil: PRT GoNoGo PauseToPause phase pattern
- MemoryBist_ParallelRetentionTest_P1_4.stil: PRT GoNoGo second retention time pause pattern
- MemoryBist_ParallelRetentionTest_P1_5.stil: PRT GoNoGo PauseToEnd phase pattern
- MemoryBist_ParallelRetentionTest_P1_6.stil: ESOE initialize diagnostic pattern for StartToPause phase
- MemoryBist_ParallelRetentionTest_P1_7.stil: ESOE execute diagnostic pattern for StartToPause phase
- MemoryBist_ParallelRetentionTest_P1_8.stil: ESOE increment diagnostic pattern for StartToPause phase
- MemoryBist_ParallelRetentionTest_P1_9.stil: ESOE initialize diagnostic pattern for PauseToPause phase
- MemoryBist_ParallelRetentionTest_P1_10.stil: ESOE StartToPause memory initialization diagnostic pattern for PauseToPause phase
- MemoryBist_ParallelRetentionTest_P1_11.stil: ESOE first retention pause diagnostic pattern for PauseToPause phase
- MemoryBist_ParallelRetentionTest_P1_12.stil: ESOE execute diagnostic pattern for PauseToPause phase
- MemoryBist_ParallelRetentionTest_P1_13.stil: ESOE increment diagnostic pattern for PauseToPause phase
- MemoryBist_ParallelRetentionTest_P1_14.stil: ESOE initialize diagnostic pattern for PauseToEnd phase
- MemoryBist_ParallelRetentionTest_P1_15.stil: ESOE StartToPause memory initialization diagnostic pattern for PauseToEnd phase
- MemoryBist_ParallelRetentionTest_P1_16.stil: ESOE first retention pause diagnostic pattern for PauseToEnd phase

- MemoryBist_ParallelRetentionTest_P1_17.stil: ESOE PauseToPause memory initialization diagnostic pattern for PauseToEnd phase
- MemoryBist_ParallelRetentionTest_P1_18.stil: ESOE second retention pause diagnostic pattern for PauseToEnd phase
- MemoryBist_ParallelRetentionTest_P1_19.stil: ESOE execute diagnostic pattern for PauseToEnd phase
- MemoryBist_ParallelRetentionTest_P1_20.stil: ESOE increment diagnostic pattern for PauseToEnd phase

Examples

Each wrapper may contain multiple controllers. For example, suppose you have two wrappers, MemoryBist_P1 and MemoryBist_P2. MemoryBist_P1 contains two controllers, and MemoryBist_P2 contains one controller. You still implement one data collection procedure for MemoryBist_P1 and one for MemoryBist_P2, for a total of six patterns: two each for Init, Exec, and Inc.

The following example shows a pseudo data collection procedure for MemoryBist_P1.

```
// Initialize pattern
execute_pattern( MemoryBist_P1_1.stil )

// Exit the loop if the next test pattern passes
loop = 0
status = fail
while (status == fail and loop < MaxLoop) {

// Execute pattern
     status = execute_and_get_failing_cycles( MemoryBist_P1_2.stil,
     fail_cycle_container )

// Increment pattern
   execute_pattern( MemoryBist_P1_3.stil )

// Save the failure log
     store_data( loop, fail_cycle_data )

// Tester iterates the Execute and Increment patterns

     loop++
}
```

The following example shows a pseudo data collection procedure for the PauseToEnd phase with voltage changes for all phases and retention pauses for MemoryBist_ParallelRetentionTest_P1:

```
// Initialize pattern
execute_pattern( MemoryBist_ParallelRetentionTest_P1_14.stil )

loop = 0
status = fail
while (status == fail and loop < MaxLoop) {

// StartToPause voltage change
     change_voltage( V1 )

// StartToPause memory initialisation pattern
     execute_pattern( MemoryBist_ParallelRetentionTest_P1_15.stil )

// First retention pause voltage change
     change_voltage( V2 )

// First retention pause pattern
     execute_pattern( MemoryBist_ParallelRetentionTest_P1_16.stil )

// PauseToPause voltage change
     change_voltage( V3 )

// PauseToPause memory initialisation pattern
     execute_pattern( MemoryBist_ParallelRetentionTest_P1_17.stil )

// Second retention pause voltage change
     change_voltage( V4 )

// Second retention pause pattern
     execute_pattern( MemoryBist_ParallelRetentionTest_P1_18.stil )

// PauseToEnd voltage change
     change_voltage( V5 )

// Exit the loop if the next test pattern passes
// Execute pattern
     status = execute_and_get_failing_cycles
        ( MemoryBist_ParallelRetentionTest_P1_19.stil,
        fail_cycle_container )

// Increment pattern
   execute_pattern( MemoryBist_ParallelRetentionTest_P1_20.stil )

// Save the failure log
     store_data( loop, fail_cycle_data )

// Tester iterates the StartToPause, FirstPause, PauseToPause,
//  SecondPause, Execute and Increment patterns and voltage changes

     loop++
}
```

Collecting and Converting Failure Data

Load the test pattern and test program, and run the test program on your ATE.

Restrictions and Limitations

- The state of the device on the tester remains the same between execution of the diagnostic patterns in the data collection procedure.
- Phase lock loops (PLLs), if any, remain locked during data collection. Because the PLL initial sequence with a user-defined sequence is applied only to the Initialize pattern, “keep-alive” clocks are mandatory between one pattern execution and another.

Prerequisites

- You have prepared the test patterns and the test program as described in “Preparing the Test Patterns and the Test Program” on page 248.

Procedure

1. Load the test program and all the test patterns into the ATE.
2. Run the test program.
3. To be compatible with process_memory_failures, convert the raw STIL or raw SVF failure log files to JavaScript® Object Notation (JSON) format.
   - For STIL patterns, convert to CDP pattern cycles in JSON format as required by the process_memory_failures command.
   - For SVF patterns, convert to the failing command ID and bit offset in JSON format as required by the process_memory_failures command.

Examples

Raw STIL ESOE Failure Log File: JSON Format

```
{
"Version": 2,   // Failure log file version (Tessent Shell supports 2)
 "Type":"ESOE", // Other types include Compstat, LBIST, Flop, SOE,
                   MEMORY, and SETUPCHAIN
 "TrackingInfo": { // Optional and free-form value pairs to identify
                   // failures and enable backmapping to a specific
                   // design, lot, or set of coordinates
                "Design":"<device_name>",
                "DeviceID":"<deviceId>",
                "XCoordinate":"<x_coord>",
                "YCoordinate":"<y_coord>"
 },
 "FailDataType": "RawSTIL",
 "RawSTIL": [
    {
     "TestConditions": { // Optional and free-form value pairs to specify
                         // test conditions for the failure collection
       "Voltage":"<voltage>",
       "Temperature":"<temperature>",
     },
     "UserTestName":"MemoryBist_P1", // Corresponds to the Patterns
                                     // wrapper name in the patterns spec
                                     // used to generate the CDP
     "Failures":[
      {
      "TesterDiagPatternName":"MemoryBist_P1_2",//  Name of the CDP ESOE
                                                 // EXEC pattern used
                                                 // for failure extraction
       "Pin":"tdo",                 // Name of the scanout pin on which
                                    // the failing cycles were sampled
       "StartingSOEID": 1,          // First SOE run number (default is 1)
       "FailureFormat" : "FailingCycles",
       "FailData" : ["14,42,69,109,136,7","15,16,55"]
         // Actual failing cycles, where each string in the
         // array contains the list of failing cycles for one
         // SOE run
     },
     {
     "TesterDiagPatternName":"MemoryBist_P1_5",
     "Pin":"tdo",
     "StartingSOEID": 1, // First SOE run number (default is 1)
     "FailureFormat" : "FailingCycles",
     "FailData" : [….]
     }
  ]
},
{
     "TestConditions": { // Optional and free-form value pairs to specify
                         // test conditions for the failure collection

       "Voltage":"<voltage>",
       "Temperature":"<temperature>",
     },
 "UserTestName":"MemoryBist2_P1",
```

```
 "Failures": [
   {
    "TesterDiagPatternName":"MemoryBist2_P1_2",
    "Pin":"tdo",
    "StartingSOEID": 1,
    "FailureFormat" : "FailingCycles",
    "FailData" : [….]
   }
  ]
 }
]
}
```

Raw SVF ESOE Failure Log File: JSON Format

```
{
"Version": 2,       // Failure log file version (Tessent Shell supports 2)
 "Type":"ESOE",     // Other types include Compstat, LBIST, Flop, SOE,
                       MEMORY, and SETUPCHAIN
 "TrackingInfo": { // Optional and free-form value pairs to identify
                   // failures and enable backmapping to a specific
                   // design, lot, or set of coordinates
                "Design":"<device_name>",
                "DeviceID":"<deviceId>",
                "XCoordinate":"<x_coord>",
                "YCoordinate":"<y_coord>"
 },
 "FailDataType": "RawSVF",
 "RawSVF": [
      {
     "TestConditions": { // Optional and free-form value pairs to specify
                         // test conditions for the failure collection
       "Voltage":"<voltage>",
       "Temperature":"<temperature>",
     },
       "UserTestName":"MemoryBist_P1",// Corresponds to the Patterns
                                      // wrapper name in the patterns spec
                                      // used to generate the CDP
       "Failures": [
         {
         "TesterDiagPatternName":"MemoryBist_P1_2",// Name of the CDP ESOE
                                                   // EXEC pattern used
                                                   // for failure extraction
          "Pin":"tdo",              // Name of the scanout pin on which
                                    // the failing cycles were sampled
          "StartingSOEID": 1,       // First SOE run number (default is 1)
          // SVF failures (one string per SOE run) where each string in
          // the array contains the list of failing SVF commands and a
          // comma separated list of failing bit offsets

          // In a given SOE Run, failing commands and their bits are
          // separated using ";"

          // The following failures are for 2 SOE runs (1 and 2). We know
          // this due to the setting "StartingSOEID": 1

          // The first SOE run ("25:3,4;29:6") contains failures for 2 SVF
          // commands (25 and 29), where the failing bits for SVF command
          // 25 are 3 and 4, while the second SVF command (29) has only
          // one failing bit, 6.
          "FailureFormat" : "CommandID:BitOffsets",
          "FailData": ["25:3,4;29:6","25:3,4;29:8,66"]
          },
          {
          "TesterDiagPatternName":"MemoryBist_P1_5",
          "Pin":"tdo",
          "StartingSOEID": 1, // First SOE run number (default is 1)
          "FailureFormat" : "CommandID:BitOffsets",
          "FailData" : [….]
          }
     ]
```

```
},
{
     "TestConditions": { // Optional and free-form value pairs to specify
                         // test conditions for the failure collection
       "Voltage":"<voltage>",
       "Temperature":"<temperature>",
     },
"UserTestName":"MemoryBist2_P1",
"Failures": [
   {
    "TesterDiagPatternName":"MemoryBist2_P1_2",
    "Pin":"tdo",
    "StartingSOEID": 1,
    "FailureFormat" : "CommandID:BitOffsets",
    "FailData" : [….]
   }
  ]
 }
]
}
```

Converting Raw JSON Data Files to Bitmap Files

Use the Tessent process_memory_failures utility to generate an output bitmap file formatted in standard JSON. The tool uses the raw STIL or raw SVF failure log files and the CDP as inputs.

Prerequisites

- You have generated raw STIL or raw SVF failure log files as described in "Collecting and Converting Failure Data" on page 256.
- Your path is set up to access Tessent utilities in the *<tessent_software_tree_path>/bin* directory.

Note

The process_memory_failures utility is not a Tessent Shell command. You must run it outside of Tessent Shell. The process_memory_failures utility is installed in the *bin* directory of your Tessent software installation.

Procedure

Run the process_memory_failures utility:

```
process_memory_failures
   --cdp cdp_path
   --failData { failure_log_directory | failure_log_file }
   [ --outDir datalog_directory ]
```

Where valid values are as follows:

- **--cdp *cdp_path*** — A required switch that specifies the CDP directory name.

- **--failData** { ***failure_log_directory*** | ***failure_log_file*** } — A required switch that specifies a directory that contains failure log files or a specific failure log file. If you specify a <*failure_log_directory*>, the tool only considers files with the *.faillog* suffix.
- --outDir *datalog_directory* — An optional switch that specifies the directory in which to store the results files. If not specified, the tool writes the bitmap files to the current working directory.

Results

As an example, for the following raw STIL failure log file as input:

```
{
  "Version": 2,
  "Type": "ESOE",
  "TrackingInfo": {"Design": "test1.cdp"},
  "FailDataType": "RawSTIL",
  "RawSTIL": [{
      "UserTestName": "MemoryBist_P1",
      "Failures": [{
          "TesterDiagPatternName": "MemoryBist_P1_2",
          "Pin": "ijtag_so",
          "StartingSOEID": 1,
          "FailureFormat": "FailingCycles",
          "FailData": [

"1892,1893,1894,1895,1896,1897,1970,2273,2280,2400,2593,2600,2677,2760,27
67",

"18,23,1892,1893,1894,1895,1896,1897,1970,2272,2280,2400,2592,2600,2677,2
759,2767",

"18,23,1892,1893,1894,1895,1896,1897,1970,2272,2273,2280,2400,2592,2593,2
600,2677,2759,2760,2767",

"18,23,1892,1893,1894,1895,1896,1897,1970,2271,2280,2400,2591,2600,2677,2
758,2767",

"18,23,1892,1893,1894,1895,1896,1897,1970,2271,2273,2280,2400,2591,2593,2
600,2677,2758,2760,2767",

"18,23,1892,1893,1894,1895,1896,1897,1970,2271,2272,2280,2400,2591,2592,2
600,2677,2758,2759,2767",

"18,23,1892,1893,1894,1895,1896,1897,1970,2271,2272,2273,2280,2400,2591,2
592,2593,2600,2677,2758,2759,2760,2767",

"18,23,1892,1893,1894,1895,1896,1897,1970,2279,2400,2599,2677,2766",

"18,23,1892,1893,1894,1895,1896,1897,1973,2273,2279,2455,2593,2599,2677,2
760,2765,2766"
          ]
      }]
  }]
}
```

The tool outputs the following bitmap file. Notice that in the Failures block, failure 24 has an asterisk next to it. This indicates that this failure has a particular consideration associated with it. The FailuresInfo block provides informational messages about the failures with asterisks next to them.

```
{
  "CreatedWith": "test1.cdp generated by 2015.2-prerelease",
  "TrackingInfo": {"Design": "test1.cdp"},
  "Header": "FailureIndex ControllerDesignName Algorithm
PhaseOrInstruction MemoryDesignName TestPort PhysicalBank PhysicalRow
PhysicalColumn BitPosition PhysicalExpected LogicalAddress
LogicalExpected",
  "Failures": [
    "0 core_rtl_tessent_mbist_c3_controller_inst SMARCHCHKBVCD 2_W0R0
blockB_clkc_i1\/mem3 0 0x0 0x0 0x0 7 1 0x0 1",
    "1 core_rtl_tessent_mbist_c2_controller_inst SMARCHCHKBVCD 2_W0R0
blockA_clkb_i1\/mem2 0 0x0 0x0 0x0 5 1 0x0 1",
    "2 core_rtl_tessent_mbist_c1_controller_inst SMARCHCHKBVCD 2_W0R0
blockA_clka_i1\/mem1 0 0x0 0x0 0x0 3 1 0x0 1",
    "3 core_rtl_tessent_mbist_c3_controller_inst SMARCHCHKBVCD 2_W0R0
blockB_clkc_i1\/mem3 0 0x0 0x0 0x1 7 1 0x1 1",
    "4 core_rtl_tessent_mbist_c2_controller_inst SMARCHCHKBVCD 2_W0R0
.
.
.
    "24* core_rtl_tessent_mbist_c3_controller_inst SMARCHCHKBVCD 3_W1R1
blockB_clkc_i1\/mem3 0 0x0 0x0 0x0 7 1 0x0 1",
    "25 core_rtl_tessent_mbist_c2_controller_inst SMARCHCHKBVCD 2_W1R1
blockB_clkb_i1\/mem3 0 0x0 0x0 0x0 4 1 0x0 1",
    "26 core_rtl_tessent_mbist_c1_controller_inst SMARCHCHKBVCD 2_W1R1
blockA_clka_i1\/mem1 0 0x0 0x0 0x0 6 1 0x0 1"
  ]
  "FailuresInfo":[
    "24 Info: Possible serial datapath short and\/or write enable short if
the algorithm is only failing on Phase 3"
  ]
}
```

The possible FailuresInfo messages are:

```
The failing bit positions for this memory are ambiguous because multiple
comparators are sharing the same goid report. At least one but maybe more
of the following bit positions are possible failures.

The correction process for this failure is ambiguous .i.e it leads to more
than one possible solution. This is possible solution <i> out of <n>
possibilities.

SOE limitation: WARNING Failure was missed at the address tested prior to
the one being reported due to back-to-back strobes. You have to setup odd
and even address selection in separate Test Steps through the
data_compare_time_slots property to have complete coverage.

The multi-strobe operation used by this instruction does not allow to
identify which one of the tested memory locations has failed.
```

```
Row and Column Address multiplexing is used for this memory. In such
situation, two logical address values are provided: one for row addressing
and one for column addressing.

Info: Possible serial datapath short and/or write enable short if the
algorithm is only failing on Phase 3

Info: Possible read enable stuck-active fault if the algorithm is only
failing on Phase 3.6

Info: Possible chip select stuck-active fault if the algorithm is only
failing on Phase 3.7

Info: Possible leakage fault if the algorithm is only failing on Phase 15
and/or 18
```

Examples

The following examples show how to use the process_memory_failures utility.

Suppose you have two failure log files, *log1.faillog* and *log2.faillog*, located in the *membistTest* directory, and *top.cdp* is your CDP. They both reside in the current directory. The following example generates two bitmap files named *log1.faillog.bitmap* and *log2.faillog.bitmap* and stores them in the results directory.

process_memory_failures --cdp top.cdp/ --failData membistTest/ --outDir ./results/

The following example generates a bitmap file named *mbist_fail1.fail.bitmap* from the specified failure log file and stores it in the results directory.

process_memory_failures --cdp top.cdp/ --failData mbist_fail1.fail --outDir ./results/

Related Topics

Diagnostic Bitmap File

Diagnostic Bitmap File

Diagnostic bitmap files contain logical and physical failure results of memory tests.

Use the process_memory_failures utility to generate a diagnostic bitmap file. The following example shows the JSON format of the file.

Example 9-1. Diagnostic Bitmap File Format

```
{
   "CreatedWith": "<cdp_name_and_tool_version>",
   "TrackingInfo": {"Design": "<design_name>"},
   "UserTestName": "<test_name>",
   "Header": "<field_name1>, <field_name2>, …",
   "Failures": [
      "0 <field_name1_data>, <field_name2_data>, …",
      "1 <field_name1_data>, <field_name2_data>, …",
      …
   ],
   "FailuresInfo": [
      <FailureIndex> Info: "Informational message",
      <FailureIndex> Info: "Informational message",
      …
   ],
   "SummaryHeader": "MemoryDesignName, PassFail, NumberOfFails",
   "MemoryFailSummary": [
      "<mem1_instance>, {Pass|Fail|Unknown}, <fail_count>",
      "<mem2_instance>, {Pass|Fail|Unknown}, <fail_count>",
      …
   ]
}
```

The following table shows the field names you can specify for the process_memory_failures utility to write to the diagnostic bitmap file. The Field Type column defines whether writing the field is on by default or whether the field is a mandatory output. For example, the Algorithm field writes out by default, but it is an optional field to choose in a custom output specification. The ControllerDesignName and FailureIndex fields always write out to the diagnostic bitmap file.

Table 9-2. Failures Data Array Fields

Field Name	Description	Field Type
ControllerDesignName	Design name of the controller testing the failing memory.	Mandatory
FailureIndex	Failure number for each bitmap string, starting with 0. The failure counter increases by 1 for each found failure. Some failures may have more than one entry if the state of the reported SOE register values has more than one possible solution for correcting the latency. If this is the case, the failure number is appended with a decimal point value. For example, 2.1 and 2.2.	Mandatory
Algorithm	Name of the algorithm used to test the failing memory	Optional Default = On

Table 9-2. Failures Data Array Fields (cont.)

Field Name	Description	Field Type
BitPosition	Failing bit position	Optional Default = On
LogicalAddress	Failing logical address	Optional Default = On
LogicalExpected	Expected value (0 or 1) at the port data line	Optional Default = On
MemoryDesignName	Design name of the failing memory	Optional Default = On
PhaseOrInstruction	Failing algorithm phase or instruction	Optional Default = On
PhysicalBank	Failing physical bank	Optional Default = On
PhysicalColumn	Failing physical column	Optional Default = On
PhysicalExpected	Expected value (0 or 1) for the physical cell	Optional Default = On
PhysicalRow	Failing physical row	Optional Default = On
PhysicalX	Failing memory cell physical X coordinate. Only applicable when enabling the X-Y physical coordinate report.	Optional Default = On
PhysicalX_	Failing memory cell physical X_ coordinate. Only applicable when enabling the X-Y physical coordinate report.	Optional Default = On
PhysicalY	Failing memory cell physical Y coordinate. Only applicable when enabling the X-Y physical coordinate report.	Optional Default = On
PhysicalY_	Failing memory cell physical Y_ coordinate. Only applicable when enabling the X-Y physical coordinate report.	Optional Default = On
TestPort	Failing port number	Optional Default = On
AddressPort	The failing address port name	Optional Default = Off

Table 9-2. Failures Data Array Fields (cont.)

Field Name	Description	Field Type
ControllerName	icl name of the controller testing the failing memory	Optional Default = Off
CounterA	General purpose counter	Optional Default = Off
LogicalBank	Failing logical bank	Optional Default = Off
LogicalColumn	Failing logical column	Optional Default = Off
LogicalRow	Failing logical row	Optional Default = Off
Memory	Failing memory collar identification	Optional Default = Off
MemoryInstance	icl name of the failing memory	Optional Default = Off
MemoryMacroModule	Reserved for future development	Optional
MemoryModule	Name of the failing memory library module	Optional Default = Off
PhysicalAddress	Failing physical address	Optional Default = Off
PortDataLine	Failing port data line	Optional Default = Off
RepeatLoopA	Value of the Repeat Loop A counter when the failure occurred.	Optional Default = Off
RepeatLoopB	Value of the Repeat Loop B counter when the failure occurred.	Optional Default = Off
RunId	Identification of the ESOE run that detected the failure	Optional Default = Off

In Example 9-1, the values of the Header field define the data that occurs in the Failures data array. The values of the SummaryHeader field define the data that occurs in the MemoryFailSummary data array.

Customize the data that occurs in the Failures data array by adding a Header field to the raw STIL or raw SVF failure log file. When you add a Header field to the failure log file, the only

fields that occur in the output bitmap file are the FailureIndex and the ControllerDesignName fields (which always occur in the output bitmap file) and the fields you specify.

The MemoryFailSummary data array is not customizable. The process_memory_failures utility adds per-memory pass/fail and failure count summary information to the diagnostic bitmap file using the format shown in Example 9-1. For "Pass" and "Unknown" entries, the <*fail_count*> is 0.

Note

The diagnostic bitmap file also supports X-Y coordinate data if you have an LDB. The header fields are PhysicalX, PhysicalY, PhysicalX_, and PhysicalY_. See "Adding Physical X-Y Coordinate Data to Bitmap Reports" on page 268 for details.

For example, suppose you add the following Header field to your raw STIL failure log file:

```
{
   "Version": 3,
   "Type":"ESOE",
   "TrackingInfo": {"Design": "test1.cdp"},

   "Header" : "PhaseOrInstruction Memory PhysicalRow \
               PhysicalExpected TestPort BitPosition PhysicalBank \
               LogicalBank PhysicalRow LogicalRow",

   "FailDataType": "RawSTIL",
   "RawSTIL": [{
      "UserTestName":"membistpv_1",
      "Failures": [{
         "TesterDiagPatternName": "membistpv_1",
         "Pin": "ijtag_so",
         "StartingSOEID": 1,
         "FailureFormat": "FailingCycles",
         "FailData": [
"4884,4892,4908,4916,4924,367824,367840,369336,371352",
"4900,4884,4892",
"4908,4916,4924,367824,367840,369336,371352,371112"
         ]
      }]
   }]
}
```

The process_memory_failures utility creates the following diagnostic bitmap file:

```
{
   "CreatedWith": "process_memory_failures version: XXXX.Y",
   "TrackingInfo": {"Design": "test1.cdp"},
   "UserTestName": "MemoryBist_P1",
   "Header": "FailureIndex ControllerDesignName PhaseOrInstruction\
              Memory PhysicalRow PhysicalExpected TestPort BitPosition \
              PhysicalBank LogicalBank PhysicalRow LogicalRow",
   "Failures": [
      "0 BP0 1 MEM3 0 0 0 1 0 0 0 0",
      "1 BP1 1 MEM4 0 0 0 7 0 0 0 0",
      "2 BP1 1 MEM4 0 0 0 7 0 0 0 0"
   ],
   "FailuresInfo":[
   ],
   "SummaryHeader": "MemoryDesignName PassFail NumberOfFails",
   "MemoryFailSummary": [
      "my_mem\/u_mem3 Fail 1",
      "my_mem\/u_mem4 Fail 2"
   ]
}
```

Adding Physical X-Y Coordinate Data to Bitmap Reports

When you enable the X-Y coordinate feature for failing RAM memory cells, the output bitmap report automatically includes the X-Y coordinates if they are available. The tool retrieves the coordinates and orientation of all memories that are setup for diagnosis from the specified LDB and stores them in the CDP.

Restrictions and Limitations

- This feature supports the ad hoc industry ASCII format for the Vendor X-Y Map files.
- For a logical memory module containing a single memory macro, the physical X-Y coordinates of failing bit cells can be reported if the DEF file specifies the macro instance path and the macro's Vendor X-Y Map file is available.

Prerequisites

- A layout database (LDB) generated from Tessent Diagnosis, or the design LEF/DEF files from which you can generate a LDB. The tool uses these files to retrieve the coordinates and orientation of all memories that are set up for diagnosis.
- When creating an LDB within the silicon_insight context, you also need a layer LEF file (*tech.lef*).
- One or more directories containing the Vendor X-Y Map files in which each cell coordinate consists of a pair of X-Y coordinates, as shown in the following snippet. The names of the files must match the memory module names and include *.bitmap* extensions. Tessent SiliconInsight uses the map files to create a relative ordering of the

cells, so the exact alignment of which edge of the cell corresponds to the x- and y-coordinates is not required to be known.

```
#address  bit[0]          bit[0]_          bit[1]           bit[1]_       ...
00000     14.481,1.407    14.565,1.565     14.481,2.367     14.565,2.525  ...
...
0008E     31.653,1.407    31.737,1.565     31.653,2.367     31.737,2.525  ...
...
```

- In the PatternsSpecification/CharacterizationDebugOptions wrapper, cell_failure_collection_method auto_increment_stop_on_error RAM diagnosis method must be enabled.
- If the construction of the LDB comes from memory names matching the post-synthesis gate-level netlist, you must ensure that the TSDB memory names match the LDB names by applying the required flow to update them. Refer to Example 4 of the write_design command in the *Tessent Shell Reference Manual* to see an example of updating the TSDB for this situation.

Procedure

1. **LDB Preparation**. If you do not have an LDB generated from Tessent Diagnosis, then use the LEF/DEF files to generate one. If you do have an LDB, proceed to step two.

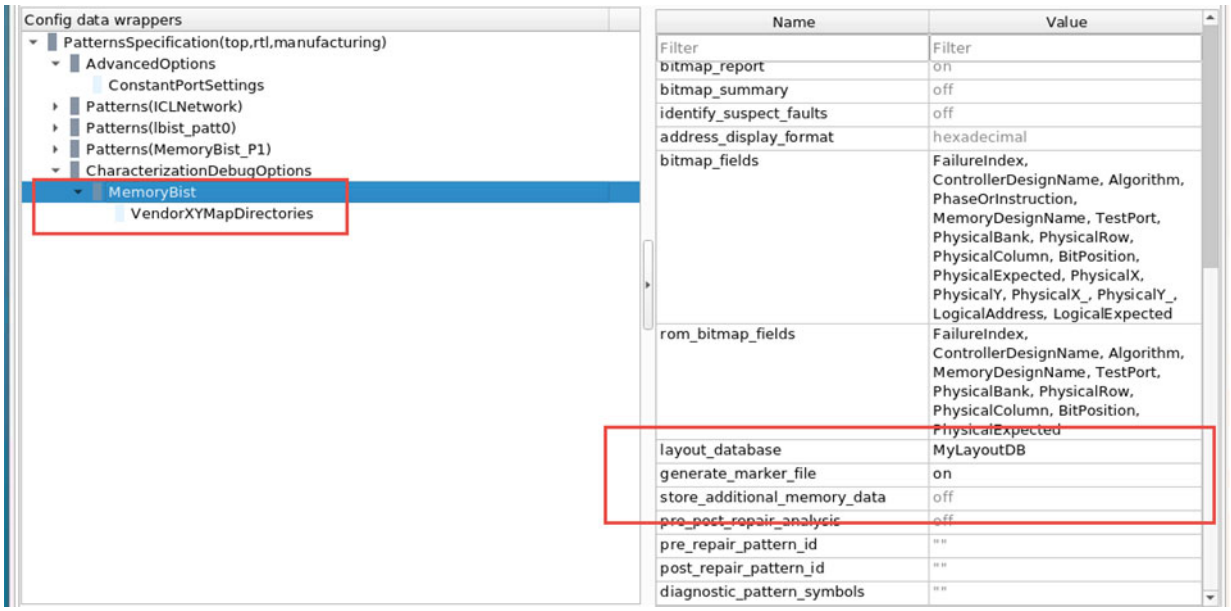
   Specify the create_layout command after entering the patterns silicon_insight context and before processing the patterns specification:

   ```
   create_layout layout_database_name [ -lef lef_file | -leflist
   lef_file ...] [ -def def_file | -deflist def_file ...]
   ```

2. **CDP Preparation**. Generate the CDP. In the GUI, at the PatternsSpecification level, ensure that you specify the following characterization debug options for MemoryBIST.

Note

If you need to generate a CDP so that you can perform failure collection on an ATE tester and convert the results through the process_memory_failures utility, refer to the Examples section that follows.



These CharacterizationDebugOptions options are briefly described below. See the cross-references for more detailed information.

- VendorXYMapDirectories — Specifies one or more directory paths to the Vendor X-Y Map files. Tessent SiliconInsight processes the applicable memory module bitmap files within the specified directories and saves them in Tessent XY data format within the CDP. These files have a *.tessent_xymap* extension.
- layout_database — Specifies the pathnames of the LDBs.
- generate_marker_file — Optionally enables the generation of a layout marker file with the ".lay" extension that is compatible for viewing with Calibre DESIGNrev. Layout marker files contain coordinate information that Calibre uses for displaying diagnosis results.
- store_additional_memory_data — Optionally specifies whether to save the Vendor X-Y Map files to the CDP.

Note

The tool adds memory layout information to the *summary.json* file when the memory has vendor data in a directory specified by the VendorXYMapDirectories property and the memory is present in the LDB paths specified by the layout_database property. When these conditions are met, the *summary.json* file contains the following memory information:

```
…
"Step": 0,
"TestedMemories": [
  {
    "CollarName": "M1",
    "MemoryInstancePath": "uUART_RAM1",
    "MemoryDesignName": "uUART\/RAM1",
    "LibraryModel": "SYNC_1R1W_16x8",
    "MacroModuleName": "SYNC_1R1W_16x8",
    "MemoryModuleBoundary":
      "[[0.000,0.000],[286.8400,0.0000],
        [286.8400,25.0000],[0.0000,25.0000]]",
    "PlacementLocationXY": "[100.0000,100.0000]",
    "PlacementOrientation": "N",
    "FinalBBox": "[[100.0000,100.0000],[386.8400,125.0000]]"
  },
…
```

Memory layout information in the *summary.json* file consists of the following:

- Memory module boundary polygon in a model coordinate system with a North orientation.

  "MemoryModuleBoundary": "[[px_1,py_1],…,[px_n,py_n]]"

- Memory instance placement location, with the origin assumed to be [0,0].

  "PlacementLocationXY": "[x,y]"

- Memory instance placement orientation.

  "PlacementOrientation": "{N|W|S|E|FN|FW|FS|FE}"

- DEF-style post-placement memory instance bounding box lower-left and upper-right corners.

  "FinalBBox": "[[Ll_x,Ll_y],[Ur_x,Ur_y]]"

3. **Diagnosis**. Proceed with the diagnosis flow.

Results

Tessent SiliconInsight—or the process_memory_failures utility when performing offline failure collection—generates a bitmap report that includes the physical X-Y coordinate data for the failures. The coordinate data are designated by the PhysicalX, PhysicalY, PhysicalX_, and PhysicalY_ headers.

In the GUI, you can use the corresponding PhysicalX, PhysicalY, PhysicalX_, and PhysicalY_ fields to customize the default report as needed.

Note

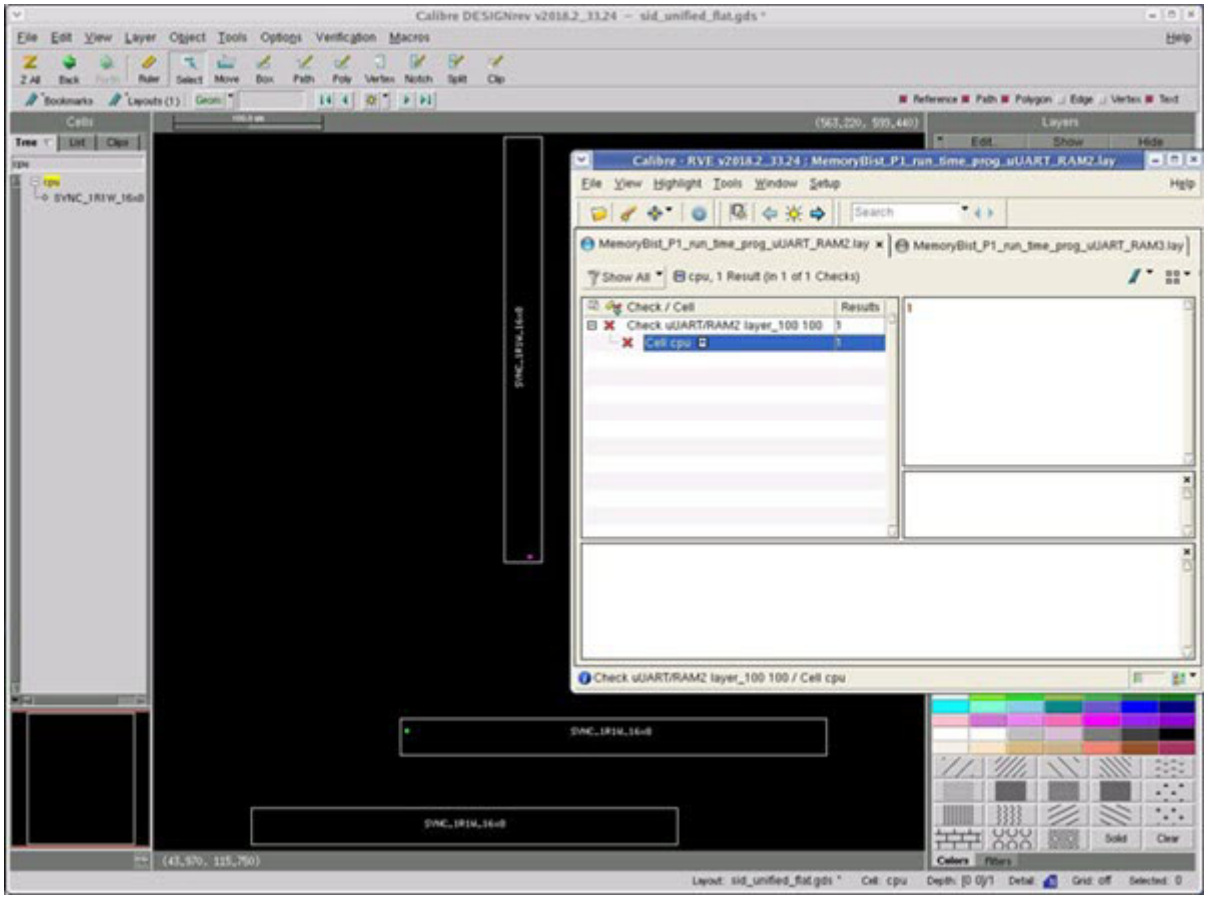
The PhysicalX, PhysicalY, PhysicalX_, and PhysicalY_ fields are automatically excluded from the report if you did not specify a valid LDB.

```
{
  "CreatedWith": "tool_name_and_version",
  "TrackingInfo" : {
       "Design" : "design_name",
       "DeviceId" : "id"
  },
  "Header": "FailureIndex ControllerDesignName Algorithm MemoryDesignName PhysicalBank
PhysicalRow PhysicalColumn PhysicalAddress BitPosition PhysicalExpected PhysicalX PhysicalY
PhysicalX_ PhysicalY_ ",
  "UserTestName": "test_name",
  "Failures": [
    "0 block_B_inst1\/block_B_gate_tessent_mbist_c1_controller_inst SMARCHCHKBCIL
block_B_inst1\/mem_1_1 0x0 0x0 0x0 0x0 1 1 1047.748 1003.0 1047.748 1003.504"
,    "1 block_B_inst1\/block_B_gate_tessent_mbist_c1_controller_inst SMARCHCHKBCIL
block_B_inst1\/mem_1_1 0x0 0x0 0x0 0x0 7 0 1047.748 1015.096 1047.748 1015.6",
  ],
}
```

Tessent SiliconInsight Desktop displays the following:

```
//  Failing Controller Verilog Instance: core_inst1/top_rtl_tessent_mbist_c1_controller_inst
//    Tested Controller Steps: 0-2
//    Step: 2
//      Algorithm: SMARCHCHKBVCD
//      Tested Memories:
//        M12:core_inst1/blockA_clka_i1/mem6
//        M13:core_inst1/blockA_clka_i2/mem6
//      Failing Memory Verilog Instance: core_inst1/blockA_clka_i2/mem6
//        Tested Ports: 0
//        Failing Port: 0
//          Bitmap:
//        -----------  ----------  …  --- ----  --------  --------  --------  --------
//        Phase Or     Physical    …  Bit Phys  Physical  Physical  Physical  Physical
//        Instruction  Bank        …  Pos Exp    X         Y         X_        Y_
//        ----------- -----------  …  --- ----  --------  --------  --------  --------
//        2_W0R0       0x0         …   3   1    1047.748  1007.032  1047.748  1007.536
//        2_W0R0       0x0         …   3   1    1047.748  1008.04   1007.536  1008.544
```

If you enabled generate_marker_file, you can view the diagnosis results in Calibre DESIGNrev. For details, refer to the *Calibre DESIGNrev Layout Viewer User's Manual*.



Examples

The following dofile example shows how to create a CDP with physical X-Y report capability when you are performing diagnosis from an ATE test program. After generating the CDP, you are ready to setup and perform failure collection on the ATE tester and convert the results through the process_memory_failures utility.

```
set_context patterns -silicon_insight -ijtag

# Create LDB if not available
create_layout MyLayoutDB -deflist { ../layout/top.def \
../layout/blockA.def } -leflist { ../data/14nmlpp_sp_rams  \
sram_sp_hse_256x16.lef ../data/14nmlpp_sp_rams/sram_sp_hse_512x16.lef }

open_tsdb tsdb_outdir
read_design top -design_id rtl -view interface
set_current_design top
create_silicon_insight_setup_specification
set_current_silicon_insight_setup NoTester(batch_cdp)
write_config_data si.setup -replace

create_patterns_specification manufacturing

start_silicon_insight

# Enable diagnosis, default is auto_increment_stop_on_error for MemoryBIST
set_config_value PatternsSpecification(top,rtl,manufacturing)/   \
CharacterizationDebugOptions/diagnose_on_failure on

# Set LDB path
set_config_value PatternsSpecification(top,rtl,manufacturing)/   \
CharacterizationDebugOptions/MemoryBist/layout_database MyLayoutDB

# Set Vendor X-Y files directory paths
set dirWrap [add_config_element -in_wrapper    \
 PatternsSpecification(top,rtl,manufacturing)/   \
CharacterizationDebugOptions/MemoryBist VendorXYMapDirectories]

# repeat the following if there is more than one directory
add_config_element -in $dirWrap ../data/14nmlpp_sp_rams -type property

process_patterns_spec -config_objects/    \
PatternsSpecification(top,rtl,manufacturing)

stop_silicon_insight
```

You can simulate the application of MemoryBIST patterns on a Tessent SiliconInsight Desktop tester. The simulator converts the responses to failures that you can then diagnose with Tessent SiliconInsight Desktop. The following example dofile shows how to use the previously created CDP in SimDUT mode.

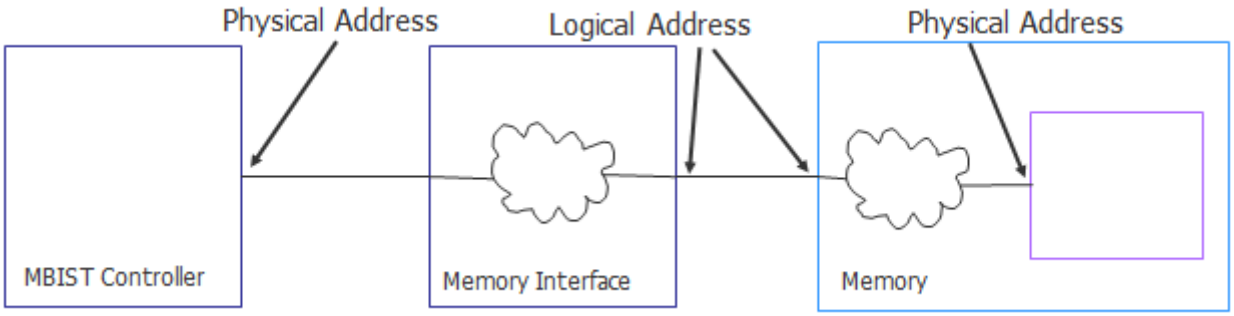
```
set_context patterns -silicon_insight
open_tsdb tsdb_outdir
read_design top -design_id rtl -view interface
set_current_design top

read_config_data si.setup
set_current_silicon_insight_setup Sid(simdut)

set_simulation_library_sources -v ../data/verilog/adk.v   \
-y ../data/14nmlpp_sp_rams -extension v

# Start the simulation with fault injection specific to the memory model
run_testbench_simulations -extra_verilog_files   \
../data/design/fi_simdut/fi.v -extra_top_modules fi -simdut_server

start_silicon_insight
```

Physical-Logical Address Mapping in the Bitmap Results

The bitmap results include the failing logical address and the failing physical address location (bank, row, column). The failing physical address location is reported by the MemoryBIST controller and is corrected for latency by the tool if needed. The tool also maps the physical locations to logical locations.

The physical address is the address seen in the MemoryBIST controller registers and inside the memory core. The physical address appears from least significant bit to most significant bit, going from column bits to row bits to bank bits. The logical address is the address seen on the output of the memory interface and the memory address port on the input of the memory core.

Figure 9-2. Physical and Logic Address Locations



Suppose you have a logical address map that looks as follows:

```
LogicalAddressmap {
ColumnAddress[1:0]: Address[1:0];
BankAddress[0:0]: Address[4:4];
RowAddress[1:0]: Address[3:2];
RowAddress[7:2]: Address[10:5];
}
```

And suppose the physical address map looks as follows:

```
PhysicalAddressMap {
BankAddress[0] : ( not ( r[7] ) ) ;
RowAddress[7] : ( ( not ( r[7] ) ) xor r[6] ) ;
RowAddress[6] : ( ( not ( r[7] ) ) xor r[5] ) ;
RowAddress[5] : ( ( not ( r[7] ) ) xor r[4] ) ;
RowAddress[4] : ( ( not ( r[7] ) ) xor r[3] ) ;
RowAddress[3] : ( ( not ( r[7] ) ) xor r[2] ) ;
RowAddress[2] : ( ( not ( r[7] ) ) xor b[0] ) ;
RowAddress[1] : ( ( not ( r[7] ) ) xor r[1] ) ;
RowAddress[0] : ( ( not ( r[7] ) ) xor r[0] ) ;
ColumnAddress[0]    : c[0];
ColumnAddress[1]    : c[1];
}
```

The MemoryBIST interpretation would be:

Table 9-3. Physical-Logical Address Mapping

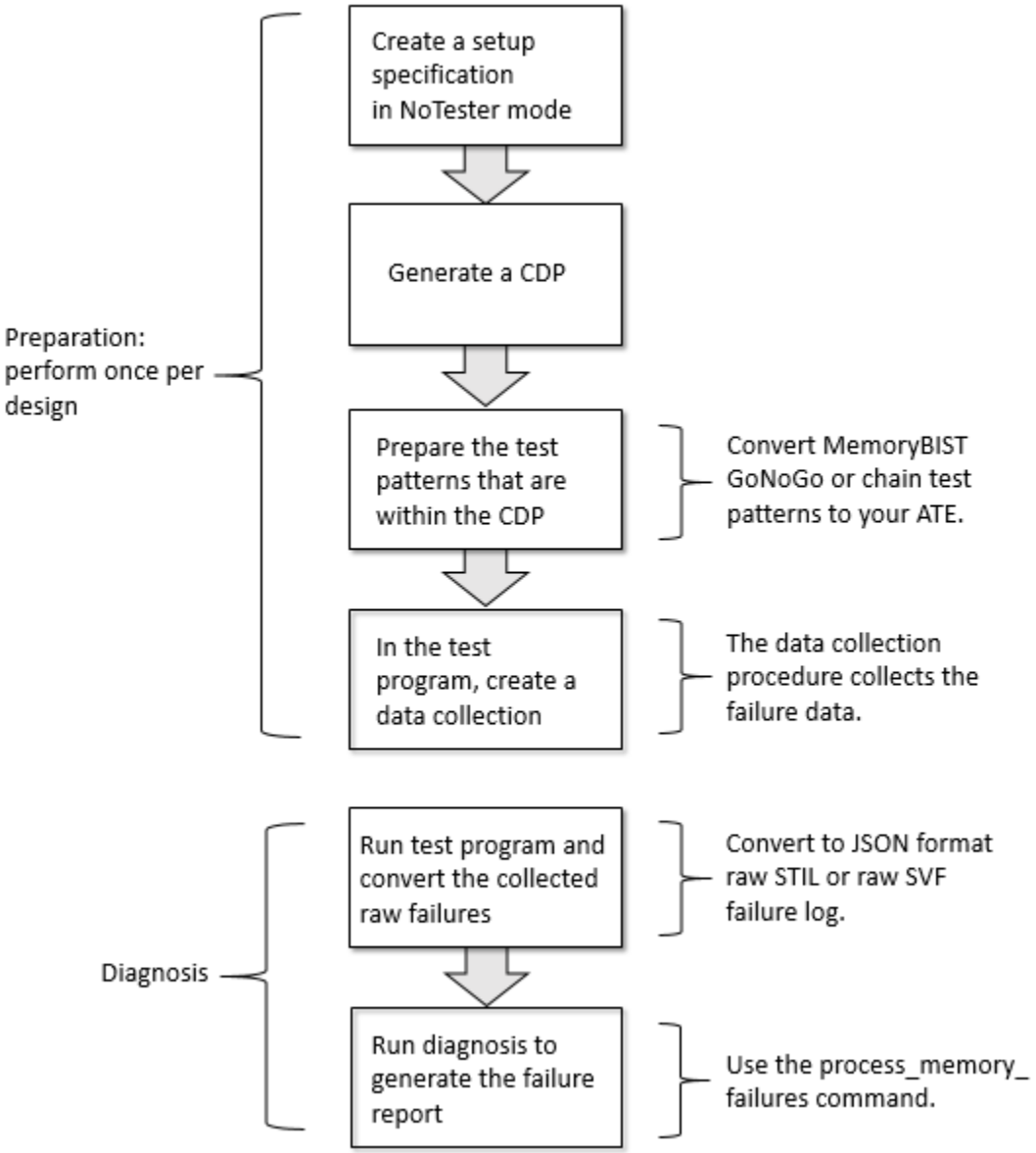
Address	Physical Address	Physical Value	Logical Translation	Logical Value	Logical Address
10	b[0]	0	((not r[7]) xor r[6])	1	R[7]
9	r[7]	0	((not r[7]) xor r[5])	1	R[6]
8	r[6]	0	((not r[7]) xor r[4])	1	R[5]
7	r[5]	0	((not r[7]) xor r[3])	1	R[4]
6	r[4]	0	((not r[7]) xor r[2])	1	R[3]
5	r[3]	0	((not r[7]) xor b[0])	1	R[2]
4	r[2]	0	not r[7]	1	B[0]
3	r[1]	0	((not r[7]) xor r[1])	1	R[1]
2	r[0]	0	((not r[7]) xor r[0])	1	R[0]
1	c[1]	0	c[1]	0	C[1]
0	c[0]	0	c[0]	0	C[0]
Physical address: 0			Logical address: 2044		.
Physical row: 0			Logical row: 255		.
Physical column: 0			Logical column: 0		.
Physical bank: 0			Logical bank: 1		.

Memory and Setup Chain Test Failure Reports

You can use Tessent SiliconInsight to generate failure reports for any MemoryBIST instrument for which you have the GoNoGo pattern data. In addition, you can generate failure reports for setup chain tests when you have enabled the diagnose_on_failure and include_setup_chain_test properties in the CharacterizationDebugOptions wrapper within the patterns specification.

The process for generating failure reports is similar to that used to generate diagnostic bitmap files, as shown in the following figure. Failure reports are simplified versions of the diagnostic bitmap report.

Figure 9-3. High-Level Flow for Reporting Failures From an ATE



Generating a Failure Report

The process for generating failure reports for memories is the same as for generating failure reports for setup chain tests. However, the unique test pattern names reside within different areas of the CDP directory as noted in step 2.

Prerequisites

- You have a signed-off chip-level TSDB.

Procedure

1. Generate a CDP as described in the Procedure section for "Generating a CDP" on page 247.

   For the memory failure report, regardless of the setting of the compare_go and compare_go_id properties in the DiagnosisOptions for a given controller in the main pattern, the MemoryBIST GoNoGo patterns are correctly generated for a memory failure report for a given controller if the diagnose_on_failure property in the CharacterizationDebugOptions is set to On and the resolution property is set to memory.

   For the setup chain failure report, patterns are automatically generated for a given controller if the include_setup_chain_test property is set to On in addition to the diagnose_on_failure property.

2. Within the CDP directory, locate the unique test patterns for each test.

   - For MemoryBIST GoNoGo test patterns:

     cdp_name/Tests/*user_test_name*/**main_patterns**

     The unique pattern name is:

     ```
     <unique_pattern_name>.<svf|stil>
     ```

   - For setup chain test patterns, use the summary.json file located in the test directory:

     cdp_name/Tests/*user_test_name/*

     The unique pattern names are indicated in bold. For example:

     ```
     {"<pattern_set_name>": {"CDPMemoryDiagnosticAvailability": [{
        "TestMethod": "setup_chain_test",
        "PatternDistribution": [{
           "PatternName": "<unique_pattern_name>_4.<svf|stil>,"
           "TestedControllers": [{ . . .
     ```

3. Prepare the test patterns and the test program:

   a. Use your pattern translator to convert the patterns to the format supported by your ATE. Ensure that there is a one-to-one mapping between the STIL pattern cycle

counts and the vector cycles of the translated ATE format. This mapping simplifies reporting of failing cycles in terms of the STIL pattern file.

Note

The CDP STIL pattern cycle starts at 1.

b. Collect the failing cycle data for each MemoryBIST wrapper configuration, perform the test, and scan out the failure data. The tool produces one pattern per MemoryBIST wrapper regardless of how many controllers each wrapper contains.

4. Load the test pattern and test program, and run the test program on your ATE, as described in the Procedure section for "Collecting and Converting Failure Data" on page 256.

   See "Examples of Raw Failure Files Converted to JSON Format" on page 282.

5. Use the process_memory_failures command to generate a failure report as described in the Procedure section for "Converting Raw JSON Data Files to Bitmap Files" on page 260.

   The failure report files are appended with the .memfail suffix for memory failures and the .setupfail suffix for setup chain test failures.

Results

As an example for setup chain test failures, for the following raw STIL failure log file as input:

```
{
        "Version": 2,
        "Type": "SETUPCHAIN",
        "TrackingInfo": {"Design": "test.cdp"},
        "FailDataType": "RawSTIL",
        "RawSTIL": [{
                "UserTestName": "MemoryBist_P1",
                "Failures": [{
                        "TesterDiagPatternName": "MemoryBist_P1_4",
                        "Pin": "ijtag_so",
                        "FailureFormat": "FailingCycles",
                        "FailData": [

"1981,1982,1983,2103,2106,2113,2186,2197,2328,2338,2431,2439,2486,2495,2524,2537,
2704,2710,2711,2712,2724,2727,2738"
                        ]
                }]
        }]
}
```

The tool outputs the following setup chain test failure report:

```
{
  "CreatedWith": "test.cdp generated by 2019.3",
  "TrackingInfo": {"Design": "test.cdp"},
  "UserTestName": "MemoryBist_P1",
  "Header": "FailureIndex ControllerName ControllerDesignName",
  "Failures": [
    "0 core_rtl_tessent_mbist_c3_controller_inst
core_rtl_tessent_mbist_c3_controller_inst",
    "1 core_rtl_tessent_mbist_c2_controller_inst
core_rtl_tessent_mbist_c2_controller_inst",
    "2 core_rtl_tessent_mbist_c1_controller_inst
core_rtl_tessent_mbist_c1_controller_inst
  ],
  "Header2": "FailureIndex RegisterName",
  "OtherRegisterFailures": [
    "0 block_rtl_tessent_mbist_bap_inst.FL_CNT_MODE0_tdr(1)",
    "1 block_rtl_tessent_mbist_bap_inst.CHAIN_BYPASS_EN_tdr(1)",
    .
    .
    .
  ]
}
```

As an example for memory failures, the following raw SVF failure log file:

```
{
        "Version": 2,
        "Type": "MEMORY",
        "TrackingInfo": {"Design": "default.cdp"},
        "FailDataType": "RawSVF",
        "RawSVF": [{
                "UserTestName": "MemoryBist_P1",
                "Failures": [{
                        "TesterDiagPatternName": "MemoryBist_P1",
                        "Pin": "tdo",
                        "FailureFormat": "CommandID:BitOffsets",
                        "FailData": [
"24:18,57;26:23,26,220,221,223,227,228,229,230,231,232,233,235,241,288,289,291,29
5,296,297,298,299,300,301,3
03,309,1182"
                        ]
                }]
        }]
}
```

Results in the following failing memory report. In the Failures block, failures 1 and 2 have asterisks next to them. This indicates that these failures have informational messages associated with them. The FailuresInfo block provides the informational messages.

```
{
  "CreatedWith": "default.cdp generated by 2019.3",
  "TrackingInfo": {"Design": "default.cdp"},
  "UserTestName": "MemoryBist_P1",
  "Header": "FailureIndex ControllerName ControllerDesignName Memory MemoryModule
MemoryInstance MemoryDesignName",
  "Failures": [
    "0 block_A_inst1.block_A_rtl_tessent_mbist_c1_controller_inst block_A_inst1\/
block_A_rtl_tessent_mbist_c1_controller_inst M1 SYNC_1RW_16x8
block_A_inst1.mem_1_1 block_A_inst1\/mem_1_1",
    "1* block_B_inst1.block_B_rtl_tessent_mbist_c22_controller_inst block_B_inst1\
/block_B_rtl_tessent_mbist_c22_controller_inst M1 SYNC_1RW_16x8
block_B_inst1.mem_1_22_1 block_B_inst1\/mem_1_22_1",
    "2* block_B_inst1.block_B_rtl_tessent_mbist_c22_controller_inst block_B_inst1\
/block_B_rtl_tessent_mbist_c22_controller_inst M2 SYNC_1RW_16x8
block_B_inst1.mem_1_22_2 block_B_inst1\/mem_1_22_2",
    "3 block_A_inst1.block_A_rtl_tessent_mbist_c6_controller_inst block_A_inst1\/
block_A_rtl_tessent_mbist_c6_controller_inst M2 ROM_1R_32X8
block_A_inst1.mem_1_6_2 block_A_inst1\/mem_1_6_2",
    "4 block_A_inst1.block_A_rtl_tessent_mbist_c7_controller_inst block_A_inst1\/
block_A_rtl_tessent_mbist_c7_controller_inst M1 ROM_1R_32X8 block_A_inst1.mem_1_7
block_A_inst1\/mem_1_7"
  ],
  "FailuresInfo": [
    "1 This failing memory is ambiguous because it is sharing the same goid report
with other memories. At least one but maybe more of the reported failing memories
are possible failures.",
    "2 This failing memory is ambiguous because it is sharing the same goid report
with other memories. At least one but maybe more of the reported failing memories
are possible failures."
  ]}
```

This example shows a FailuresInfo message indicating ambiguous results. For failing memories, you may receive this message when you have not specified per_memory for the DftSpecification/MemoryBist/Controller/DiagnosisOptions/go_status property and:

- The DftSpecification specifies using shared comparators (versus the default local comparators) with the DftSpecification/MemoryBist/Controller/Step/comparator_location property set to shared_in_controller.

  To overcome ambiguities relative to this selection, you can split the testing of the controller with shared comparator ambiguities across multiple patterns using the PatternsSpecification/Patterns/TestStep/Controller/AdvancedOptions/freeze_step property.

- In conjunction with specifying shared comparators, the DftSpecification/MemoryBist/Controller/AdvancedOptions/shared_comparators_per_go_id property specifies a value other than 1 while multiple memories are tested in the same controller step where some of the memories overlap on the same go_id register.

  To overcome ambiguities relative to this selection, you can split the testing of the controller with the shared comparator go_id register ambiguities across multiple patterns using the PatternsSpecification/Patterns/TestStep/MemoryBist/DiagnosisOptions/comparator_id_select property.

Examples

Examples of Raw Failure Files Converted to JSON Format

Example 9-2. STIL Failure Log File Example

```
{
"Version": 2,   // Failure log file version (Tessent Shell supports 2)
 "Type":"SETUPCHAIN", // Other types include Compstat, LBIST, Flop, SOE,
                      // ESOE, and MEMORY
 "TrackingInfo": { // Optional and free-form value pairs to identify
                   // failures and enable backmapping to a specific
                   // design, lot, or set of coordinates
                "Design":"<device_name>",
                "DeviceID":"<deviceId>",
                "XCoordinate":"<x_coord>",
                "YCoordinate":"<y_coord>"
 },
 "FailDataType": "RawSTIL",
 "RawSTIL": [
    {
     "TestConditions": { // Optional and free-form value pairs to specify
                         // test conditions for the failure collection
       "Voltage":"<voltage>",
       "Temperature":"<temperature>",
     },
     "UserTestName":"MemoryBist_P1", // Corresponds to the Patterns
                                     // wrapper name in the patterns spec
                                     // used to generate the CDP
     "Failures":[
      {
      "TesterDiagPatternName":"MemoryBist_P1_4",// Name of the applicable CDP
                                                // test pattern used
                                                // for failure extraction
       "Pin":"ijtag_so",            // Name of the scanout pin on which
                                    // the failing cycles were sampled
       "FailureFormat" : "FailingCycles",
       "FailData" :
["1981,1982,1983,2103,2106,2113,2186,2197,2328,2338,2431,2439,2486,2495,2524,2537
,2704,2710,2711,2712,2724,2727,2738"]
          // Actual failing cycles
     }
  ]
},
{
     "TestConditions": { // Optional and free-form value pairs to specify
                         // test conditions for the failure collection

       "Voltage":"<voltage>",
       "Temperature":"<temperature>",
     },
 "UserTestName":"MemoryBist2_P1",
 "Failures": [
   {
    "TesterDiagPatternName":"MemoryBist2_P1_4",
    "Pin":"ijtag_so",
    "FailureFormat" : "FailingCycles",
    "FailData" : [….]
```

```
    }
  ]
 }
]
}
```

Example 9-3. SVF Failure Log File Example

```
{
"Version": 2,      // Failure log file version (Tessent Shell supports 2)
 "Type":"MEMORY",    // Other types include Compstat, LBIST, Flop, SOE,
                     // ESOE, and SETUPCHAIN
 "TrackingInfo": { // Optional and free-form value pairs to identify
                     // failures and enable backmapping to a specific
                     // design, lot, or set of coordinates
                "Design":"<device_name>",
                "DeviceID":"<deviceId>",
                "XCoordinate":"<x_coord>",
                "YCoordinate":"<y_coord>"
 },
 "FailDataType": "RawSVF",
 "RawSVF": [
       {
     "TestConditions": { // Optional and free-form value pairs to specify
                         // test conditions for the failure collection
       "Voltage":"<voltage>",
       "Temperature":"<temperature>",
     },
        "UserTestName":"MemoryBist_P1",// Corresponds to the Patterns
                                       // wrapper name in the patterns spec
                                       // used to generate the CDP
       "Failures": [
         {
         "TesterDiagPatternName":"MemoryBist_P1",// Name of the applicable CDP
                                                 // test pattern used
                                                 // for failure extraction
          "Pin":"tdo",              // Name of the scanout pin on which
                                   // the failing cycles were sampled
          // SVF failures where the string
          // contains the list of failing SVF commands and a
          // comma separated list of failing bit offsets

          // Failing commands and their bits are separated using ";"

          // The following failures are for 2 SVF
          // commands (24 and 26), where the failing bits for SVF command
          // 24 are 18 and 57, while the second SVF command (26) has
          // 27 failing bits.
          "FailureFormat" : "CommandID:BitOffsets",
          "FailData":
["24:18,57;26:23,26,220,221,223,227,228,229,230,231,232,233,235,241,288,289,291,2
95,296,297,298,299,300,301,303,309,1182"]
          }
     ]
},
```

```
{
     "TestConditions": { // Optional and free-form value pairs to specify
                         // test conditions for the failure collection
       "Voltage":"<voltage>",
       "Temperature":"<temperature>",
     },
"UserTestName":"MemoryBist2_P1",
"Failures": [
    {
     "TesterDiagPatternName":"MemoryBist2_P1",
     "Pin":"tdo",
     "FailureFormat" : "CommandID:BitOffsets",
     "FailData" : [….]
    }
  ]
 }
]
}
```

Failure Report Formats

Failure reports produced by process_memory_failures are in JSON format.

Memory Failure Report

Example 9-4. Memory Failure Report Format

```
{
   "CreatedWith":"<tool_name_and_version>",
      "TrackingInfo:{"Design":"<design_name>"
   }
      "Header": "FailureIndex ControllerName ControllerDesignName
       Memory MemoryModule MemoryInstance MemoryDesignName",
      "Failures":[
       "0 controller_name controller_design_name
       memory memory_module memory_instance memory_design_name"'
       "1 controller_name controller_design_name
       memory memory_module memory_instance memory_design_name"
      .
      .
      .
      ]
      "FailuresInfo":[
      FailureIndex Info: "Informational message",
      FailureIndex Info: "Informational message",
      .
      .
      .
      ]
   }
```

The failing memory report contains a field named “Header” that defines the data that occurs in the mandatory and optional Failures data array. As with the diagnostic bitmap file, you can customize the data that occurs in the Failures data array by adding a Header field to the raw

STIL or raw SVF failure log file. Refer to the end of the section "Diagnostic Bitmap File" on page 263 for an example of how to customize the Header fields.

Table 9-4. Failures Data Array Fields for Memory Failure Reports

Field Name	Description	Field Type
FailureIndex	Failure number for each failing memory string, starting with 0. **Note:** The failure counter increases by 1 for each found failure.	Mandatory
ControllerDesignName	Design name of the controller testing the failing memory.	Mandatory
MemoryDesignName	Design name of a failing memory.	Optional Default = On
ControllerName	icl name of a controller testing the failing memory.	Optional Default = On
Memory	Failing memory collar identification.	Optional Default = On
MemoryInstance	icl name of a failing memory.	Optional Default = On
MemoryModule	Name of a failing memory library module.	Optional Default = On

Setup Chain Test Failure Report

Example 9-5. Setup Chain Test Failure Report Format

```
{
   "CreatedWith":"<tool_name_and_version>",
      "TrackingInfo:{"Design":"<design_name>"
   }
      "Header": "FailureIndex ControllerName ControllerDesignName",
      "Failures":[
       "0 controller_name controller_design_name"'
       "1 controller_name controller_design_name"
      .
      .
      .
      ]
      "Header2": "FailureIndex RegisterName",
      "OtherRegisterFailures": [
        "0 register_name",
        "1 register_name",
      .
      .
      .
      ]
   }
```

The setup chain test failure report contains two sections. The first section is the Header section that lists the data that occurs in the Failures data array.

Table 9-5. Failures Data Array Fields for Setup Chain Test Failure Reports

Field Name	Description	Field Type
FailureIndex	Failure number for each controller failure string, starting with 0. **Note:** The failure counter increases by 1 for each found failure.	Mandatory
ControllerDesignName	Design name of a failing controller.	Mandatory
ControllerName	icl name of a failing controller.	Mandatory

The second section is the Header2 section that defines that lists the data that occurs in the OtherRegisterFailures data array.

Table 9-6. OtherRegisterFailures Data Array Fields for Setup Chain Test Failure Reports

Field Name	Description	Field Type
FailureIndex	Failure number for each register failure string, starting with 0. **Note:** The failure counter increases by 1 for each found failure.	Mandatory

Table 9-6. OtherRegisterFailures Data Array Fields for Setup Chain Test Failure Reports (cont.)

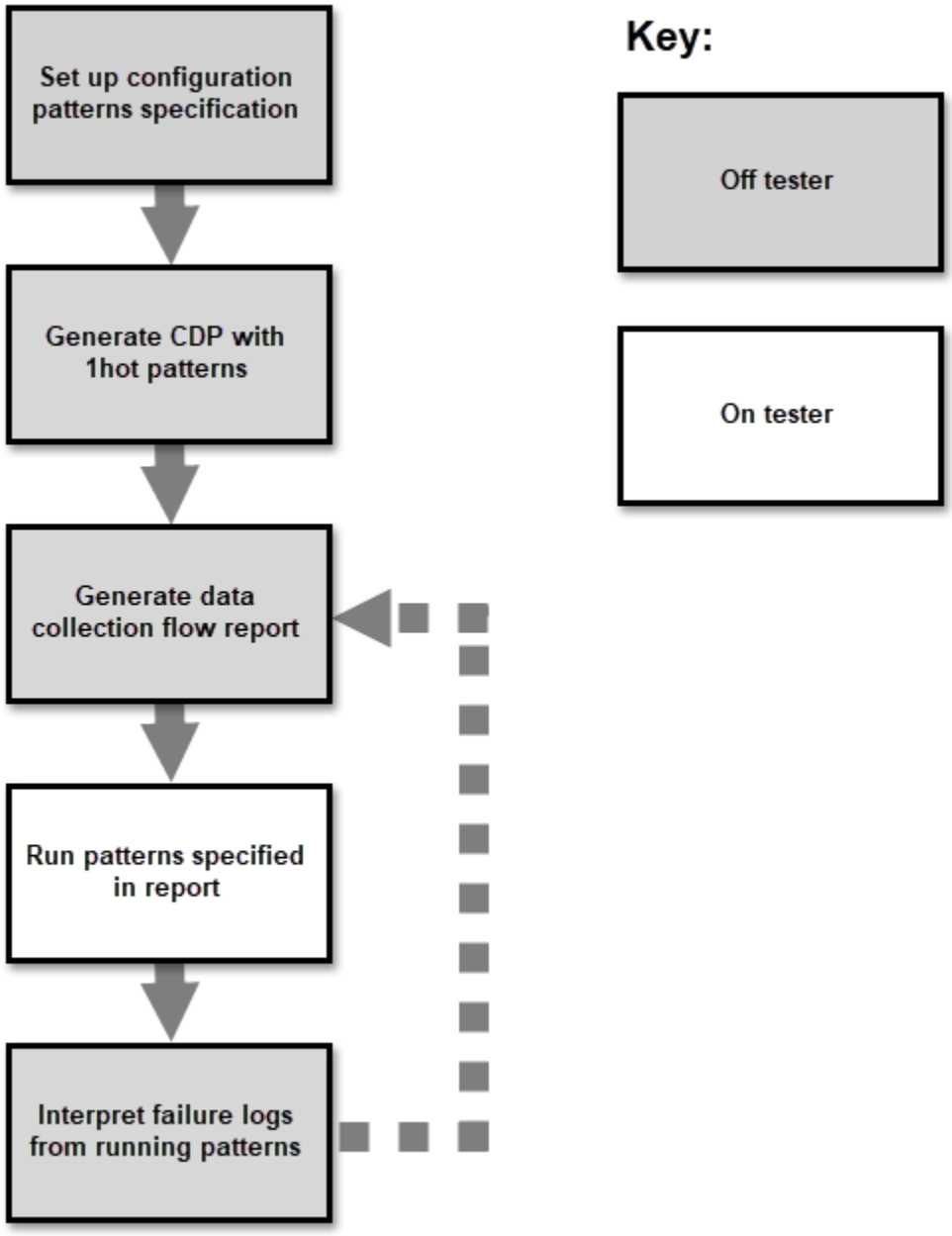
Field Name	Description	Field Type
RegisterName	Name of a failing register.	Mandatory

ATPG Offline Interpretation Flow

The process of diagnosing failing flops can take a large amount of time, depending on the design. The ATPG offline interpretation flow enables you to pregenerate 1hot patterns. Use these patterns on the tester to collect failure instances and generate failure logs. You can then process the failure logs while off the tester to produce failing flops data.

The following diagram provides a high-level overview of the interpretation flow:

Figure 9-4. ATPG Offline Interpretation Flow



The following topics describe the ATPG offline interpretation flow in more detail:

Pregenerating 1hot Patterns in the CDP

The first step in the ATPG offline interpretation flow is to pregenerate 1hot patterns in the CDP while off the tester. You can choose a range of patterns to pregenerate before generating the CDP.

Prerequisites

- You have already set up the rest of your pattern specification. You can add the pregenerate_1hot_patterns property at any time during your pattern specification setup, but you must set this property before generating the CDP.
- You do not need to be connected to the ATE to pregenerate patterns.

Procedure

1. Set up your configuration patterns specification as normal and include the following properties:
   - diagnose_on_failure : on ;
   - ATPG/pregenerate_1hot_patterns : <*pattern_range_list*> ;
   - ATPG/diagnosis_result : failing_flops ;
2. Generate the CDP from the pattern specification with the process_patterns_specification command.

Results

The generated CDP contains all collateral data needed for diagnosis, including 1hot pattern files, flat models, and so forth. Use these pregenerated 1hot patterns for offline diagnosis in the next step, "Generating and Interpreting ATPG Failure Logs."

Generating and Interpreting ATPG Failure Logs

The process_atpg_failures command provides instructions on how to generate the failure log files necessary to diagnose failing flops. After you follow the instructions, you run the command a second time to interpret the failure log files.

Prerequisites

- You have completed generating a CDP as described in "Pregenerating 1hot Patterns in the CDP" on page 289.

Procedure

1. Generate an instruction flow by running the "process_atpg_failures <*cdp*> <*test_name*> -report_ate_data_collection_flow" command.

   The command returns a list of instructions similar to the following:

```
//  1) On the tester:
//       a) Run pattern 'pat_glue_top.stil' from cdp directory
//          '<cdp>/Tests/<test_name>/main_patterns' on tester; save
//          failure log (i.e. 'pat_glue_top.flog')
//       b) If the above pattern fails, run the following 1hot
//          patterns from cdp directory '<cdp>/Tests/<test_name>/
//          default/atpg' and save the failure log:
// -----------------  -----------------  -----------------
// pattern name       covered pattern#s  example flog name
// -----------------  -----------------  -----------------
// <test_name>_1.stil 0:9,11             <test_name>_1.flog
//
//  2) Off the tester, run the following Tessent shell command
//     to interpret the failures to failing flops.
//      process_atpg_failures <cdp> <test_name> -interpret_fails
//      { pat_glue_top.stil pat_glue_top.flog <test_name>_1.stil
//      <test_name>_1.flog }
```

   The covered pattern numbers correspond to the pregenerated patterns you specified in "Pregenerating 1hot Patterns in the CDP".

2. Following the instructions provided in the results of the previous step, run the specified patterns on the tester. If any patterns fail, you must then also run the 1hot patterns for the failing patterns.

   A failure log file contains information similar to the following:

```
format cycle
failures_begin
34 SFRWE_c2_2 H L
67 SFRWE_c2_2 H L
84 edt_channels_out3_c2_2 L H
86 edt_channels_out3_c2_2 L H
…
2839 SFRWE_c2_2 H L
2872 SFRWE_c2_2 H L
2905 SFRWE_c2_2 H L
failures_end
failure_buffer_limit_reached none
failure_file_end
```

3. Once you have collected the failure log files for the failing patterns, continue to follow the instructions by running the "process_atpg_failures <*cdp*> <*test_name*> -interpret_fails { <*pattern_flog_pairs*> }[-learn_uncovered_1hot_patterns]" command off the tester.

The command provides a list of patterns with mismatched expected and tested values, similar to the following:

```
//pattern_id  chan/PO_name  chan_cyc_id  exp_val  tested_val  chain---cell_id---cell_path_name

scan_test
0             SFRWE_c2_2                H        L           // PO
1             SFRWE_c2_2                H        L           // PO
1             edt_channel3  15          L        H           // chain18---15---my_glue/…/ix1734
…
```

Note

The names in the table shown here have been abbreviated to fit onto a printed page.

The -learn_uncovered_1hot_patterns switch is optional. It enables the command to automatically learn uncovered patterns and generate additional 1hot patterns for them in the CDP for subsequent failure collection.

4. (Optional) Repeat steps 1 through 3 as needed, particularly if you have used the optional -learn_uncovered_1hot_patterns switch to add patterns to the CDP, to further refine the failing flop information you receive.

Chapter 10
ATE-Connect for IJTAG Debugging on Testers

ATE-Connect expands the Tessent SiliconInsight support for Tessent IJTAG-inserted instruments (MemoryBIST and LogicBIST) and ATPG by enabling you to perform debug on a tester rather than in Desktop mode. This enables you to debug your silicon in the same environment used for production testing. ATE-Connect supports SVF and STIL files.

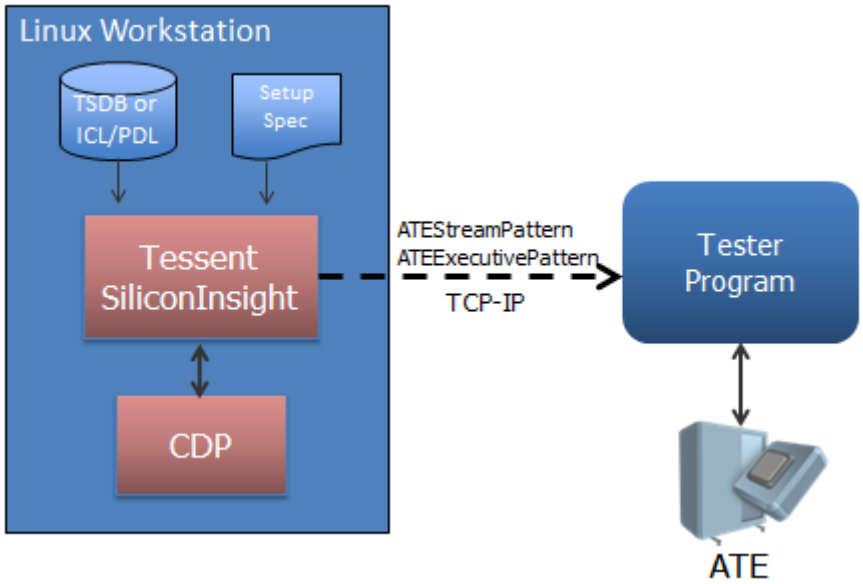
Note

If you are an ATE vendor, refer to "ATE-Connect Preparation for ATE Vendors" on page 393 for special considerations.

ATE-Connect Overview

ATE-Connect enables you to debug Tessent Shell IJTAG-inserted devices on an ATE. Tessent SiliconInsight and the tester communicate through a TCP-IP protocol, which enables Tessent SiliconInsight to send IJTAG command requests through the socket for the test program to run.

Figure 10-1. ATE-Connect Architecture



Tessent SiliconInsight runs on a Linux workstation within the Tessent Shell environment while the test program can run on any platform, such as Linux, UNIX, and Windows. You do not have to integrate the CDP into the tester software.

Performing IJTAG Debug With ATE-Connect

Once the tester and test program are configured, you can run IJTAG commands in Tessent SiliconInsight for execution on the tester. Refer to your ATE vendor documentation for information about configuring the test program and tester.

Prerequisites

- You have a signed-off TSDB.

Procedure

1. Generate a SiliconInsightSetupSpecification.

   To enable ATE-Connect, specify Ate(*tester_name*) as the selected setup. Modify the default Ate(*tester_name*) wrapper based on your design and ATE environment. Refer to "Ate(tester_name)" on page 130 for syntax details. For example:

   ```
   SiliconInsightSetupSpecification {
     selected_setup : Ate(tester1);
     Protocol {
       ...
       }
     }
     Ate(tester1) {
       cdp_integration_type : Partial
       cdp_directory : top.cdp;
       host_name : tester57;
       port_number : 6411;
       pattern_transmission_method : stream_file;
       initial_control_string: "@@@@";
       PinMap {
         P0 : pad_jtdi;
         P1 : pad_jtms;
         P2 : pad_jtdo;
         P3 : pad_jtck;
     }
   }
   ```

   Consult the ATE vendor documentation as needed. Ensure that you follow the ATE vendor specifications for the pattern transmission method and the initial control string (if any). In addition, the vendor may also specify the port number.

2. Prepare the test program.

   Prepare the test program following your ATE vendor's documentation. Ensure that the pin map as defined in the setup specification's PinMap wrapper matches the tester pin

map definition in the test program, and that the pins are mapped to the correct device pin.

3. Call the ATE vendor API that enables the test program to listen to commands from Tessent SiliconInsight through the specified port.

   To call the API, on the ATE tester controller bring up the test program and stop it at the correct point. Consider the following when deciding where to stop the program:

   - The device must be powered up such that the tester is delivering the correct voltage and current to all device pins.
   - The appropriate clocks must be running.
   - As applicable, run an initial sequence to grant Tessent SiliconInsight access to the TAP or to locking the PLLs.

4. Invoke Tessent Shell and proceed with debugging as described in the Procedure section of "PDL-Level Debugging for Tessent IJTAG-Inserted Devices" on page 149."

Chapter 11
Custom Test Flows

The Tessent SiliconInsight custom flow enables you to extend the diagnosis capabilities of SiliconInsight beyond the current instruments by enabling support for sophisticated diagnostic flows and third-party instruments (such as customer BIST or analog devices). In addition, some Mentor Graphics plug-ins require you to control the test flow for some documented flows, such as the repair flow.

The custom test flow enables you to include and access the following collateral data in the CDP:

- Test sequence control so that you can perform pattern file execution, pattern file result interpretation, pattern branching, and decision making.
- Diagnosis procedures associated with the specified custom test.
- Pattern files to be used by the specified custom test.
- Additional patterns wrappers also present in the Patterns Specification to be used by the specified custom test.
- Multiple pattern files that result from detecting pattern splits in the Patterns wrapper.

For an example of using a custom test flow, refer to "Interactive Desktop Standard Repair Flow Tutorial" on page 385.

Configuring and Performing a Custom Test

Configure custom test procedures in Tcl format using custom flow programming commands.

Refer to the section "Test Procedure Command Reference" on page 319 for more details about these commands.

Prerequisites

- This process assumes that you already have a Patterns wrapper in the patterns specification that you want to test with a custom test procedure.

  Note

  It is recommended to use the following naming convention for the Patterns wrapper name hosting a custom test. The tool does not enforce this naming scheme, but the scheme enables you to easily identify custom tests that use user-defined Tcl procedures to perform the test.

  ```
  CUSTOM[_<optional_keyword>]_PROCFILE_<tcl_filename>_PROCNAME_<tcl_procedure>
  ```

- You previously generated all patterns required for your custom test flow, and the pattern files are in either STIL or SVF format.
- You have generated and customized a setup specification as described in "Prepare the Design Under Test" on page 75.
- If your custom test includes Mentor Graphics instruments, you must have a signed-off TSDB. For more information, refer to the chapter "Tessent Shell Data Base (TSDB)" in the *Tessent Shell Reference Manual*.

Procedure

1. Develop a CDP test procedure file.

   A test procedure file may contain multiple procedures. For example, it may contain the custom test procedure and also a custom diagnostic test procedure.

   The CDP test procedure file defines how the SiliconInsight tool runs a custom test. It provides the CDP interface to access CDP functions and data structures that are required

to run the test. In general, generate a custom test so that it performs the following actions:

- Access collateral data associated with the custom test: custom data, pattern files, Mentor Graphics instruments specified in the custom test, other required tests.
- Run pattern files and other required tests.
- Access execution results for pattern files and other required tests.
- Run GPIB or any other commands to manipulate clocks, power supplies, or other GPIB instruments, and perform measurements.
- Set the custom test execution result.

Refer to "CDPProcedureFiles" on page 301 for example test procedures. Refer to the section "Test Procedure Command Reference" on page 319 for the custom flow programming commands that you can use to write the test procedures.

2. Add two new wrappers, PatternsSpecification/CDPProcedureFiles and Patterns/TestSequencing, to the patterns specification.

   The CDPProcedureFiles wrapper lists CDP test procedure files. The presence of the TestSequencing wrapper indicates that the given Patterns wrapper is for a custom test.

3. Set the tool context to enable Tessent SiliconInsight within Tessent Shell.

   ```
   SETUP>set_context patterns -silicon_insight
   ```

4. Run the process_patterns_specification command.

   The tool automatically detects the Patterns wrappers with TestSequencing wrappers and generates the required custom test collateral data in the CDP.

5. Run the execute_cdp_test command.

   The tool automatically sources the CDP test procedure file specified by the TestSequencing/test_source_code property and runs the specified custom test procedures within the test sequence.

Results

Each test returns 0 if the test passes and 1 if the test fails. The return value does not impact the returned data from the execute_cdp_test command. However, test failure could trigger extra diagnostic action when the following criteria are met:

- At least one of the instrument or the diagnosis keywords are specified with the execute_cdp_test -collect_data_type_list command.
- A TestSequencing/DiagnosticProcedure wrapper exists.

Patterns Specification Syntax for Custom Tests

To run a custom test, add the CDPProcedureFiles and TestSequencing wrappers to the patterns specification.

At a high level, the wrappers integrate as follows:

```
PatternsSpecification(design_name,design_id,pattern_id) {
  usage                          : usage       ;
  manufacturing_patterns_format  : format, ... ;
  compress_pattern_files         : boolean     ;
  CharacterizationDebugOptions { ...
  }
  CDPProcedureFiles {
    test_proc_file1 ;
    test_proc_file2 ;
    ...
  }
  Patterns(patterns_name) {
    CharacterizationDebugOptions { ...
    }
    TestSequencing {
      test_source_code : path_to_test_proc_file ;
      test_namespace   : custom_test_tcl_code_namespace ;
      TestProcedure(custom_test_proc) {
        Arguments { ... }
      }
      DiagnosticProcedure(custom_diagnostic_proc)
        Arguments { ... }
      }
      CustomPatterns { ...
      }
      CustomData { ...
      }
      OtherRequiredPatternsWrappers { ...
      }
    }
    ...
  }
}
```

CDPProcedureFiles

Specifies a list of CDP test procedure files that define custom test or diagnostic procedures.

Usage

```
PatternsSpecification(design_name,design_id,pattern_id) {
  ...
  CharacterizationDebugOptions { ...
  }
  CDPProcedureFiles {                    // Required for custom tests
    test_proc_file1 ;                    // Required
    test_proc_file2 ;
    ...
  }
```

Description

The tool accesses and runs a custom procedure defined in a test procedure file only when you have specified it within the Patterns/TestSequencing wrapper.

Arguments

- *test_proc_file1* ; ... ; ...

  Required list of pathnames to CDP test procedure files that define custom test procedures. You must specify at least one file, and the files must be in Tcl format (*.tcl*).

Examples

Refer to "TestSequencing" on page 309 for configuration syntax examples that include the CDPProcedureFiles wrapper in context with the TestSequencing wrapper.

Refer to "Test Procedure Command Reference" on page 319 for descriptions of the custom flow programming commands that you can use to write test procedures such as those shown in the following examples.

Example 1

The following example shows a simple test procedure.

```
namespace eval ::my_test_flow_namespace {

proc my_test_flow {} {
  log_message "my_test_flow: Entering ...\n"

  # Get the list of datalog selections selected with the
  # "execute_cdp_test -collect_data_type_list" command
  set datalog_mode [get_test_datalog_mode]
  log_message "my_test_flow: datalog_mode = '$datalog_mode'.\n"

  # Use the user-specified datalog selection to set the execution mode of
  # the pattern files (gonogo or get_failures)
  if { [lsearch $datalog_mode gonogo] >= 0 } {
    set mode gonogo
  } else {
    set mode get_failures
  }

  # Get the name of the custom test to be run for future access to related
  # data
  set current_test [get_active_test_name]

  # Get the list of pattern files to run for this test
  set pattern_names [get_test_data $current_test -main_pattern_names_list]
  log_message "my_test_flow: Pattern names = '$pattern_names'.\n"

  # Runs a pattern file in the appropriate execution mode, where the
  # -add_execution_result option indicates detailed failures from running
  # this pattern file are added to the custom test results
  foreach local_pat $pattern_names {
    log_message "my_test_flow: Executing pattern '$local_pat' ...\n"
    set exec_stat [execute_pattern_file $local_pat -mode $mode \
        -add_execution_result]
    if { $exec_stat } {
      log_message "my_test_flow: Pattern '$local_pat' failed.\n"
      # Get the list of pattern sets in this pattern file
      set participating_pat_sets [get_pattern_data $local_pat \
        -pattern_sets]
      log_message "\tParticipating pattern sets = \
        '$participating_pat_sets'\n"
      # Get the list of failing pattern sets from the latest run of this
      # pattern file
      set failing_pat_sets [get_pattern_data $local_pat \
        -failing_pattern_sets]
      log_message "\tFailing pattern sets = '$failing_pat_sets'\n"
      # Get the list of failing variables from the latest run of this
      # pattern file
      set failing_vars [get_pattern_data $local_pat \
        -failing_variable_names]
      log_message "\tFailing variable names = '$failing_vars'\n"

      # Set the custom test execution status to failed
      set_test_execution_status Failed
    }
  }
```

```
  log_message "my_test_flow: Exiting (Test execution status =
'[get_test_execution_status]') ...\n"

  return $exec_stat
}

}; #namespace my_test_flow_namespace
```

Example 2

The following example shows a more complex test procedure that includes diagnosis.

```
namespace eval ::custom_membist_test {

    variable preserved_controller_results {}

proc membist_custom_test { args } {
  variable preserved_controller_results
  set preserved_controller_results [dict create]
  log_message "membist_custom_test: Entering with the following arguments $args
...\n"
  # Get the name of the custom test to be run for future access to related
  # data
  set test_name [get_active_test_name]
  # Get the values for Param1, Param2, and Param3 that were specified in
  # the PatternSpecification under the TestSequencing/CustomData wrapper
  # for this test and then display them in the tester log

  log_message "membist_custom_test: Param1 is [get_test_data $test_name \
    -parameter_value -parameter_name Param1 -parameter_type dynamic], \
    Param2 is [get_test_data $test_name -parameter_value -parameter_name \
    Param2 -parameter_type dynamic] and Param3 is [get_test_data $test_name \
    -parameter_value -parameter_name Param3 -parameter_type dynamic]\n"

  set arg_list [join $args]
  set no_args [llength $arg_list]
  # Use the user-specified datalog selection to set the pattern file
  # execution mode (gonogo or get_failures)
  set datalog_mode [get_test_datalog_mode]
  if { [lsearch $datalog_mode gonogo] >= 0 } {
    set mode gonogo  } else {
    set mode get_failures
  }
```

```
set frequency 170
set f_index 0
set_test_diagnosis_result "" ;# initialize results.
# Get the list of pattern files to run for this test
set pattern_names [get_test_data $test_name -main_pattern_names_list]
log_message "main pattern names = '$pattern_names'\n"
foreach local_pat $pattern_names {
  log_message "membist_custom_test: Executing pattern '$local_pat' ...\n"
  if { $f_index < $no_args } {
    set frequency [lindex $arg_list $f_index]
  }
  # Mimics the execution of a gpib command that could manipulate some
  # frequency obtained from the TestProcedure input arguments, if specified
  # in the PatternsSpecification under the TestSequencing/TestProcedure/
  # Arguments wrapper for this test
  log_message "Faking gpib command at [execute_ate_command clock format \
      [clock seconds] -gmt true].\n"
  # Since you are executing user main patterns, make sure there is no
  # active pattern set and Mentor Graphics instrument (IP group) type.
  set_active_pattern_set ""
  set_active_ip_group_type ""

  # Runs a pattern file with the appropriate execution mode, where the
  # -add_execution_result option indicates detailed failures from the execution
  # of this pattern file are added to this custom test result
  set exec_stat [execute_pattern_file $local_pat -mode $mode \
    -add_execution_result]
  if { $exec_stat } {
    # The pattern failed
    # Set the custom test execution status to failed
    set_test_execution_status Failed
    # Perform more analysis and provide more results depending on specified
    # datalog selections
    if { [lsearch $datalog_mode instrument] >= 0 || \
        [lsearch $datalog_mode diagnosis] >= 0 } {
      # Get the list of failing pattern sets from the latest execution of this
      # pattern file
      set failing_pat_sets [get_pattern_data $local_pat -failing_pattern_sets]
      log_message "membist_custom_test: failing_pattern_sets([llength \
          $failing_pat_sets]) = '$failing_pat_sets'.\n"
```

```
foreach pat_set $failing_pat_sets {
  # Notice some commands related to instruments in the following sequence
  # only apply if the TestStep wrapper corresponding to this pattern set
  # had Mentor instruments in it and was part of the Patterns wrapper for
  # this test.
  #
  # Get the Mentor Graphics instrument type tested in this pattern set
  set ip_group_type [get_test_data $test_name -parameter_value \
      -parameter_name ${pat_set}::InstrumentTypeGroupsList]
  # Get the Mentor Graphics instruments list of the specified type tested
  # in this pattern set
  set ctrl_list [get_test_data $test_name -parameter_value \
      -parameter_name ${pat_set}::${ip_group_type}::InstrumentList]
  # Set this failing pattern set as the active pattern set for future
  # access to related data
  set_active_pattern_set $pat_set
  # Set the active Mentor Graphics instrument type for this failing
  # pattern set for future access to related data
  set_active_ip_group_type $ip_group_type
  set failing_ctrls [list]
  # Create huddle formatted list to accumulate failing instruments to be
  # reported in the diagnosis results
  set failing_ctrl_objs [create_result_list]
  set controller_results [list]
  foreach ctrl $ctrl_list {
    # Set this Mentor Graphics instrument tested by this failing pattern
    # set as the active instrument for future access to related data
    set_active_instrument $ctrl
    # Get the Mentor Graphics instrument type required to access Mentor
    # instrument callback procedures
    set ctrl_type [get_instrument_type $ctrl]
    # Get the execution status of the currently active Mentor Graphics
    # instrument (previously set through the set_active_instrument
    # command)
    set pass_fail [evaluate_instrument_callback $ctrl_type get_pass_fail]
    if { [huddle gets $pass_fail "ExecStatus"] != "PASS" } {
      # If you have requested to collect instrument data results,
      # accumulate this  instrument for display in the tester log and
      # for report in the diagnosis results
      if { [lsearch $datalog_mode instrument] >= 0 } {
        lappend failing_ctrls $ctrl
        # Append the failing instruments to the huddle formatted list
        append_result failing_ctrl_objs $ctrl
      }
      # If you have requested to collect diagnosis data results,
      # accumulate this instrument's non-pass result for future usage
      if { [lsearch $datalog_mode diagnosis] >= 0 } {
        lappend controller_results $ctrl $pass_fail
      }
    }
  }
```

```
            # If you have requested to collect diagnosis data results and some
             # instruments did not pass, preserve these results for future
             # diagnosis usage
             if { [lsearch $datalog_mode diagnosis] >= 0 && \
                  [llength $controller_results] } {
               dict set preserved_controller_results $pat_set [dict create \
                  $local_pat $controller_results]
             }
             # If you have requested to collect instrument data results, display
             # these failing instruments in the tester log and report them in the
             # diagnosis results
             if { [lsearch $datalog_mode instrument] >= 0 } {
               log_message "\tFailing pattern set = '$pat_set' has the following \
                   failing instruments '$failing_ctrls'\n"
               # Report the huddle formatted failing instruments in the diagnosis
               # results
               set ts_res [create_result_object PatternSetName $pat_set \
                   ParticipatingInstruments [create_result_list $ctrl_list] \
                   FailingInstruments $failing_ctrl_objs]
               add_test_diagnosis_result $ts_res
             }
           }
         }

       } elseif { [lsearch $datalog_mode instrument] >= 0 } {
         # The pattern did not fail and you have requested to collect instrument data
         # results
         #
         # Get the list of pattern sets in this pattern file and display them in the
         # tester log
         set passing_pat_sets [get_pattern_data $local_pat -pattern_sets]
         log_message "membist_custom_test: participating_pattern_sets([llength \
             $passing_pat_sets]) = '$passing_pat_sets'.\n"
         # Report the huddle formatted passing instruments in the diagnosis results
         foreach pat_set $passing_pat_sets {
           set ip_group_type [get_test_data $test_name -parameter_value \
              -parameter_name ${pat_set}::InstrumentTypeGroupsList]
           set ctrl_list [get_test_data $test_name -parameter_value -parameter_name \
              ${pat_set}::${ip_group_type}::InstrumentList]
           set ts_res [create_result_object PatternSetName $pat_set \
              ParticipatingInstruments [create_result_list $ctrl_list]]
           add_test_diagnosis_result $ts_res
         }
       }
       incr f_index
     }
     log_message "membist_custom_test: Exiting (Test execution status = \
        '[get_test_execution_status]') ...\n"

     return $exec_stat
   }

   proc membist_custom_diagnosis { args } {
     variable preserved_controller_results
     log_message "membist_custom_diagnosis: Entering with the following arguments
   $args ...\n"
```

```
set arg_list [join $args]
set no_args [llength $arg_list]
# Check if diagnosis was requested in the user-specified datalog section
set datalog_mode [get_test_datalog_mode]
if { [lsearch $datalog_mode "diagnosis"] >= 0 } {
  # Since you are running diagnosis patterns, ensure there is no active pattern
  # set, Mentor Graphics instrument (IP group) type, and instrument.
  set_active_pattern_set ""
  set_active_ip_group_type ""
  set_active_instrument ""
  # Get the name of the custom test to be diagnosed for future access to
  # related data
  set test_name [get_active_test_name]
  # Get the list of pattern files that were run for this test
  set pattern_names [get_test_data $test_name -main_pattern_names_list]

  # Diagnose the failing pattern files from the results preserved from the Test
  # Procedure (in this case, membist_custom_test)
  dict for { pattern_set data } $preserved_controller_results {
    set pattern_file [lindex [dict keys $data] 0]
    # Get the list of non-passing instruments to be diagnosed for this pattern
    # file from the results preserved from the Test Procedure
    set failing_ctrls [list]
    foreach { ctrl pass_fail } [dict get $data $pattern_file] {
      if { [huddle gets $pass_fail "ExecStatus"] != "PASS" } {
        lappend failing_ctrls $ctrl
        # Get the Mentor Graphics instrument type required to access Mentor
        # instrument callback procedures
        set ctrl_type [get_instrument_type $ctrl]
      }
    }

    # Must run preceding split pattern files if any
    # Since you could be performing diagnosis on multiple patterns in a loop,
    # make sure there is no active pattern set and Mentor Graphics instrument
    # (IP group) type
    set_active_pattern_set ""
    set_active_ip_group_type ""
    set frequency 1.5
    set f_index 0

    # If you have dependencies with previous pattern files, you must re-run
    # every pattern file until the one to undergo diagnosis, to ensure you
    # reproduce the same test conditions
    foreach local_pat $pattern_names {
      if { $f_index < $no_args } {
        set frequency [lindex $arg_list $f_index]
      }
```

```
        # Mimics execution of a gpib command that could manipulate some frequency
        # obtained from the DiagnosticProcedure input arguments, if specified in
        # the PatternsSpecification under the TestSequencing/DiagnosticProcedure/
        # Arguments wrapper for this test
        log_message "Faking gpib command at [execute_ate_command clock format \
            [clock seconds] -gmt true].\n"
        if { $local_pat eq $pattern_file } {
          # This is the one to be diagnosed
          # Set this failing pattern set and corresponding Mentor Graphics
          # instrument type as active for diagnosis
          set ip_group_type [get_test_data $test_name -parameter_value \
              -parameter_name ${pattern_set}::InstrumentTypeGroupsList]
          set_active_pattern_set $pattern_set
          set_active_ip_group_type $ip_group_type
          log_message "membist_custom_diagnosis: Diagnosing pattern set \
              '$pattern_set' ...\n"

          # Perform diagnosis and get results for the failed instruments of the
          # currently active pattern set (previously set through the
          # set_active_pattern_set command
          set callback_args "-failing_instruments { $failing_ctrls }"
          set ip_grp_diag_data [evaluate_instrument_callback $ctrl_type \
              collect_diagnostic_data $callback_args]
          break
        } else {
          # This is not the one to be diagnosed
          # Just run this pattern file to ensure you reproduce the same test
          # conditions until you reach the one to be diagnosed
          log_message "membist_custom_diagnosis: Executing pattern \
              '$local_pat' ...\n"
          execute_pattern_file $local_pat -mode get_failures
        }
        incr f_index
      }
      # Report the huddle formatted diagnosis results
      set ts_res [create_result_object PatternSetName $pattern_set \
          PatternSetData [create_result_object FailingInstruments \
          $failing_ctrls InstrumentResult $ip_grp_diag_data]]
      add_test_diagnosis_result $ts_res
    }
  }
  # Clear preserved results
  set preserved_controller_results [dict create]
  log_message "membist_custom_diagnosis: Exiting ...\n"

  return 0
}

}; #namespace custom_membist_test
```

TestSequencing

Specifies how to run a custom test for the given Patterns wrapper.

Usage

```
Patterns(patterns_name) {
  CharacterizationDebugOptions { ...
  }
  TestSequencing {
    test_source_code : path_to_test_proc_file ;                // Required
    test_namespace   : custom_test_tcl_code_namespace ;        // Required
    TestProcedure(custom_test_proc) {                          // Required
      Arguments {                                              // Optional
        arg1 : value ;
        arg2 : value ;
        ...
      }
    }
    DiagnosticProcedure(custom_diagnostic_proc)                // Optional
      Arguments {
        arg1 : value ;
        arg2 : value ;
        ...
      }
    }
    CustomPatterns {                                           // Optional
      pattern1_path ;
      pattern2_path ;
      ...
    }
    CustomData {                                               // Optional
      param1 : value ;
      param2 : value ;
      ...
    }
    OtherRequiredPatternsWrappers {                            // Optional
      patterns_wrapper1 ;
      patterns_wrapper2 ;
      ...
    }
  }
  ...
}
```

Description

The presence of the TestSequencing wrapper within a Patterns wrapper indicates that the pattern is for a custom test.

Arguments

- test_source_code : *path_to_test_proc_file* ;

  Required property that specifies the path to a test procedure file that contains custom test procedures. If the specified file is not listed within the CDPProcedureFiles wrapper, the tool returns an error.

- test_namespace : *custom_test_tcl_code_namespace* ;

  Required property that specifies the Tcl namespace associated with the custom test.

- TestProcedure(*custom_test_proc*){}

  Required property that specifies the name of the custom test procedure to run when you specify the execute_cdp_test command. This procedure must exist within the specified test_source_code file.

  You can optionally specify this wrapper with arguments. At minimum, you must specify this wrapper as empty ({}).

- Arguments {
  arg1 : *value* ;
  arg2 : *value* ;
  ... }

  Optional property used within the TestProcedure wrapper to list arguments to pass to the specified custom test procedure.

- DiagnosticProcedure(*custom_diagnostic_proc*)
  Arguments {
  arg1 : *value* ;
  arg2 : *value* ;
  ... }

  Optional wrapper that specifies a custom diagnostic test procedure to run when you specify the execute_cdp_test command. This procedure must exist within the specified test_source_code file.

  You can specify this wrapper as empty ({}) or with the Arguments wrapper. Use the Arguments wrapper to specify additional argument values to pass to the specified custom diagnostic procedure.

- CustomPatterns {
  pattern1_path ;
  pattern2_path ;
  ... }

  Optional wrapper that lists paths to previously generated custom pattern files associated with the specified custom test. STIL and SVF formats are supported.

For the CDP infrastructure to provide meaningful register results from the execution of custom pattern files, these files must include the following TESSENT_PRAGMA annotations that define the mapping of vector cycles to register values:

- TESSENT_PRAGMA pattern_set
- TESSENT_PRAGMA icl_checksum
- TESSENT_PRAGMA annotation
- TESSENT_PRAGMA variable

For details about these annotations, refer to "Comments and Annotations in Tessent IJTAG." This document is available through your Mentor Graphics support representative.

- CustomData {
  param1 : *value* ;
  param2 : *value* ;
  ... }

  Optional wrapper that lists parameter-value pairs associated with the specified custom test.

- OtherRequiredPatternsWrappers {
  patterns_wrapper1 ;
  patterns_wrapper2 ;
  ... }

  Optional wrapper that lists other Patterns wrappers associated with the specified custom test. These Patterns wrappers should also be present in the current patterns specification.

Examples

Example 1: Third-Party Instrument Test With Voltage Variations

The following example shows a simple third-party instrument setup with voltage variations and pre-generated patterns.

```
CDPProcedureFiles {
  my_test/user_test.tcl;
}
```

```
Patterns(MyTest) {
  TestSequencing {
    test_source_code : my_test/user_test.tcl;
    test_namespace : simple_user_instrument;
    TestProcedure(test_user_instrument) {
      Arguments {
        v1 : 1.6;
        v2 : 1.4;
        v3 : 1.8;
      }
    }
    CustomPatterns {
      my_test/my_voltage_test.stil;
    }
  }
}
```

Example 2: Customer Parallel Retention Test With Voltage Variations

The following example shows a custom parallel retention test setup with voltage variations, where patterns are generated from the TestStep wrappers included in the Pattern wrapper itself.

```
CDPProcedureFiles {
  my_test/my_membist_prt.tcl;
}

Patterns(MyMembistPRTTest) {
  TestSequencing {
    test_source_code : my_test/my_membist_prt.tcl;
    test_namespace : custom_prt_membist_test;
    TestProcedure(membist_custom_prt_test) {
      Arguments {
        v1 : 1.4;
        v2 : 1.8;
        v3 : 1.4;
      }
    }
    DiagnosticProcedure(membist_custom_prt_diagnosis) {
      Arguments {
        v1 : 1.4;
        v2 : 1.8;
        v3 : 1.4;
      }
    }
  }
```

```
CharacterizationDebugOptions {
    diagnose_on_failure : On;
    MemoryBist {
      resolution : cell;
      failure_limit_scope : controller;
    }
  }
  TestStep(start_to_pause) {
    MemoryBist {
      Controller(top_rtl_tessent_mbist_c1_controller_inst) { .. }
      Controller(top_rtl_tessent_mbist_c2_controller_inst) { .. }
      AdvancedOptions {
        retention_test_phase : start_to_pause;
        preserve_bist_inputs : on;
  .. }
  TestStep(pause_to_pause) {
    MemoryBist {
      Controller(top_rtl_tessent_mbist_c1_controller_inst) { .. }
      Controller(top_rtl_tessent_mbist_c2_controller_inst) { .. }
      AdvancedOptions {
        retention_test_phase : pause_to_pause;
        preserve_bist_inputs : on;
    .. }
    AdvancedOptions { split_patterns_file : on; // To split PRT phases }
  }
  TestStep(pause_to_end) {
    MemoryBist {
      Controller(top_rtl_tessent_mbist_c1_controller_inst) { .. }
      Controller(top_rtl_tessent_mbist_c2_controller_inst) { .. }
      AdvancedOptions {
        retention_test_phase : pause_to_end;
        preserve_bist_inputs : on;
    .. }
    AdvancedOptions { split_patterns_file : on; // To split PRT phases }
.. }
```

Example 3: Complex Test for Memory Repair With Looping on a Portion of a Pattern

The following example shows a memory repair setup, where patterns are generated from other Pattern wrappers (“OtherRequiredPatternWrappers”) specified in the PatternsSpecification. Furthermore, one of these other pattern wrappers is itself a custom test with a split.

Note

Refer to the section “Adding a Standard Repair Flow Custom Test” on page 317 for information about automatic insertion of the Repair Flow Test described in the pattern specification for the following example.

In complex test flows, you may need to repeat the execution of one or more pattern files in a loop after splitting the pattern. The repeated portion of the Patterns wrapper content (corresponding to one or more test steps) must also be configured to maintain the IJTAG network in an appropriate state. For details, refer to the TestStep/AdvancedOptions/network_end_state property in the *Tessent Shell Reference Manual*.

```
CDPProcedureFiles {
  membist_test/membist_bisr.tcl;
}

Patterns(MembistRepair) {
  TestSequencing {
    test_source_code : membist_test/membist_bisr.tcl;
    test_namespace : membist_bisr;
    TestProcedure(membist_repair) { }
    DiagnosticProcedure(membist_repair_diag) { }
    OtherRequiredPatternsWrappers {
      TP0_PowerUpEmulation;
      TP1_1_PreRepair; // TP1_n
      TP2_1_CheckRepairNeeded;
      TP2_MemoryBisr_AutonomousMode;
      TP3_MemoryBist_P1; // TP3_n
    }
  }
}
Patterns(TP0_PowerUpEmulation) {
  TestStep(PowerUpEmulation) {
    AdvancedOptions { network_end_state : reset; }
      MemoryBisr {
        run_mode : autonomous;
        Controller(chip_rtl_tessent_mbisr_controller_inst) {
          AutonomousOptions { operation : power_up_emulation; }
        }
.. }

Patterns(TP1_1_PreRepair) {
  AdvancedOptions { exclude_initial_ireset_from_patterns : on; }
  TestStep(PreRepair) {
    AdvancedOptions { network_end_state : reset; }
    MemoryBist {
      run_mode : hw_default;
      Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
        RepairOptions { check_repair_status : non_repairable; }
      }
      Controller(chip_rtl_tessent_mbist_c2_controller_inst) { }
        RepairOptions { check_repair_status : non_repairable; }
      }
.. }
Patterns(TP2_1_CheckRepairNeeded) {
  AdvancedOptions { exclude_initial_ireset_from_patterns : on; }
  TestStep(CheckRepairNeeded) {
    AdvancedOptions { network_end_state : reset; }
    MemoryBist {
      run_mode : check_repair_needed;
      Controller(chip_rtl_tessent_mbist_c1_controller_inst) { }
      Controller(chip_rtl_tessent_mbist_c2_controller_inst) { }
.. }
```

```
Patterns(TP2_MemoryBisr_AutonomousMode) {
  TestSequencing {
    test_source_code : membist_test/membist_bisr.tcl;
    test_namespace : membist_bisr;
    TestProcedure(fuse_programming) {
      Arguments {
        vddqh : 2.5;
        vddql : 1.5;
      }
    }
  }
  AdvancedOptions { exclude_initial_ireset_from_patterns : on; }
  TestStep(TP2_2_CaptureBiraRotate) {
    MemoryBisr {
      run_mode : autonomous;
      Controller(chip_rtl_tessent_mbisr_controller_inst) {
        AutonomousOptions {
          operation : rotate_bisr_chain;
          enable_bira_capture : on;
  .. }
  TestStep(TP2_2_SelfFuseBoxProgram) {
    MemoryBisr {
      run_mode : autonomous;
      Controller(chip_rtl_tessent_mbisr_controller_inst) {
        AutonomousOptions { operation : self_fuse_box_program; }
  .. }
  TestStep(TP2_3_VerifyFuseBox) {
    AdvancedOptions { split_patterns_file : on; }
    MemoryBisr {
      run_mode : autonomous;
      Controller(chip_rtl_tessent_mbisr_controller_inst) {
        AutonomousOptions { operation : verify_fuse_box; }
.. }

Patterns(TP3_MemoryBist_P1) {
  TestStep(clear_bisr) {
    MemoryBisr {
      run_mode : autonomous;
      Controller(chip_rtl_tessent_mbisr_controller_inst) {
        AutonomousOptions { operation : power_up_emulation; }
  .. }
  TestStep(hw_default) {
    MemoryBist {
      run_mode : hw_default;
      Controller(chip_rtl_tessent_mbist_c1_controller_inst) { }
      Controller(chip_rtl_tessent_mbist_c2_controller_inst) { }
.. }
```

Example 4: Complex Test With Split Patterns for MemoryBIST

The following example shows a complex MemoryBIST setup with frequency changes and split patterns, where patterns are generated from the TestStep wrappers included in the custom test Pattern wrapper itself.

In complex test flows, you may need to split the pattern into multiple pattern files so that you can perform intermediate actions—such as changing voltage or frequency—at different points

during the custom test. For details, refer to AdvancedOptions/split_patterns_file property in the *Tessent Shell Reference Manual*.

```
CDPProcedureFiles {
  membist_test/my_membist_frequency_test.tcl;
}

Patterns(MyMembistTest) {
  TestSequencing {
    test_source_code : my_test/my_membist_frequency_test.tcl;
    test_namespace : custom_membist_test;
    TestProcedure(membist_frequency_test) {
      Arguments {
        low_frequency : 50MHz;
        high_frequency : 88MHz;
      }
    }
    DiagnosticProcedure(membist_frequency_diag) {
      Arguments {
        low_frequency : 50MHz;
        high_frequency : 88MHz;
      }
    }
  }

TestStep(first_write) {
    MemoryBist {
      Controller(top_rtl_tessent_mbist_c1_controller_inst) { .. }
      AdvancedOptions { preserve_bist_inputs : on; }
    }
  }
  TestStep(first_read) {
    MemoryBist {
      Controller(top_rtl_tessent_mbist_c1_controller_inst) { .. }
      AdvancedOptions { preserve_bist_inputs : on; }
    }
  }
```

```
  TestStep(second_write) {
    MemoryBist {
      Controller(top_rtl_tessent_mbist_c1_controller_inst) { .. }
      AdvancedOptions { preserve_bist_inputs : on; }
    }
    AdvancedOptions { split_patterns_file : on; // Apply frequency change
                                                // at this TestStep }
  }
  TestStep(second_read) {
    CharacterizationDebugOptions {
      diagnose_on_failure : on; // To diagnose this TestStep on failure
    }
    MemoryBist {
      Controller(top_rtl_tessent_mbist_c1_controller_inst) { .. }
      AdvancedOptions { preserve_bist_inputs : on; }
    }
  }
  TestStep(third_read) {
    CharacterizationDebugOptions {
      diagnose_on_failure : on; // To diagnose this TestStep on failure
    }
    MemoryBist {
      Controller(top_rtl_tessent_mbist_c1_controller_inst) { .. }
      AdvancedOptions { preserve_bist_inputs : on; }
    }    AdvancedOptions { split_patterns_file : on; // Apply frequency
                                                     // change at this
                                                     // TestStep }
.. }
```

Adding a Standard Repair Flow Custom Test

Add a Standard Repair Flow Custom Test to the Patterns Specification using the create_memory_repair_flow command. This is useful for performing hard or soft repair using SID or soft repair through simulation (SIMDUT).

Prerequisites

- You have generated and customized a setup specification as described in “Prepare the Design Under Test” in the *Tessent SiliconInsight User’s Manual* for Tessent Shell.
- You must have a signed-off TSDB. For more information, refer to Chapter 9 in the *Tessent Shell Reference Manual*.
- You must have a freshly generated pattern specification for manufacturing usage from the “create_patterns_specification_manufacturing” command with the TESSENT_SI_ALLOW_DESTRUCTIVE_BISR environment variable set to 1.

Procedure

1. Set the tool context to enable Tessent SiliconInsight within Tessent Shell:

   SETUP> set_context patterns -silicon_insight

2. Run the create_memory_repair_flow command.

The tool automatically incorporates a hard Repair Flow Custom Test named "CUSTOM_PROCFILE_my_repair_tcl_PROCNAME_my_repair" in the patterns specification.

Results

The tool displays the following message:

```
PatternsSpecification /PatternsSpecification(<design_name>,<design_id>,manufacturing)
was modified for the CDP generation in order to support repair. If you want this
modification to be preserved in your PatternsSpecification for future sessions, please
use the write_config_data command to save it.
```

Test Procedure Command Reference

This chapter describes custom flow programming commands that you can use to create custom test procedures to control how the SID tester process runs tests when you specify the execute_cdp_test command. These commands enable you to create a platform for your designs to use custom flows to test and diagnose any supported or third-party instruments. They are intended primarily for use by IP designers.

Note

The ATE commands are not supported by the Tessent Shell environment through the "**tessent -shell**" command. The commands described in this chapter are used only within the SID tester process as an interface into the CDP.

The reference pages indicate the following usage contexts:

- Simple — Primary commands that support custom test flows that do not require diagnosis reports or patterns from other Patterns wrappers specified within the patterns specification. For simple custom test flows, familiarization with the following set of commands is sufficient:
  - execute_ate_command
  - execute_pattern_file
  - get_active_test_name
  - get_pattern_data
  - get_test_data
  - get_test_datalog_mode
  - log_message
  - set_test_execution_status

  The get_test_data command also supports additional syntax for complex usages.
- Complex — Additional commands that support complex custom test flows that require diagnosis reports or patterns from other Patterns wrappers specified in the patterns specification.

  This context also includes commands that facilitate handling of Huddle data structures. Tessent SiliconInsight execution results and diagnosis results are formatted in Huddle. Huddle provides a generic Tcl-based serialization/intermediary format. Familiarize yourself with the Huddle Tcl library if your custom test flow requires diagnosis reports or results interpretation from other required tests associated with the custom test.

For information about Huddle commands, see https://tools.ietf.org/doc/tcllib/html/huddle.html.

add_test_diagnosis_result

Context: complex

Adds diagnosis results to the current custom test.

Usage

add_test_diagnosis_result ***huddle_object***

Arguments

- huddle_object

  A required string that specifies a Huddle object.

Description

You can configure the Huddle object as needed, but the following Huddle keys require special consideration:

- PatternSetName and its value are mandatory.
- When specified, the DisplayResult string automatically displays on the Tessent Shell console.
- When Mentor Graphics instruments are specified in the patterns specification for the current custom test, the tool uses ParticipatingInstruments and FailingInstruments to update the Pass/Fail hierarchical tree.

Examples

The following command adds the specified Huddle object to the diagnosis results for the current custom test:

```
add_test_diagnosis_result [create_result_object PatternSetName
my_pattern_set ParticipatingInstruments [create_result_list $ctrl_list]
FailingInstruments $failing_ctrl_huddle_list]
```

append_result

Context: complex

Appends elements to either a Huddle object or a Huddle list.

Usage

create_result_object { ***huddle_object key value*** [*key value …*] |
huddle_list value [*value …*] }

Arguments

- ***huddle_object key value*** [*key value …*]

  Required choice when appending elements to a Huddle object. Specify the Huddle object you want to append and then the key and value pairs to include with the specified Huddle object.

- ***huddle_list value*** [*value …*]

  Required choice when appending elements to a Huddle list. Specify the Huddle list you want to append and then the values to include in the Huddle list.

Description

This command facilitates handling of Huddle data structures by enabling you to include additional elements to either a Huddle object or a Huddle list. The tool calls the “huddle append” command under the hood.

Examples

The following commands first create an empty Huddle list and then append a Huddle object to the list.

```
set diag_data [create_result_list]
append_result diag_data [create_result_object "DCParametericTests"
[create_result_object "MeasuredVoltage" "1.8V"] "FuncFail"
[create_result_object "FailingInstruments" [create_result_list
"blockA_inst.blockA_rtl_tessent_mbist_c1_controller_inst"]]]
```

create_result_list

Context: complex

Converts a specified Tcl list into a Huddle list.

Usage

create_result_list *list_name*

Arguments

- *list_name*

  An optional string that specifies the name of a Tcl list. The list can contain other Huddle objects.

Description

This command facilitates handling of Huddle data structures by enabling you to generate a Huddle list from a Tcl list. The tool calls the “huddle list” command under the hood.

The command generates an empty Huddle list if you do not specify *list_name*.

Examples

The following command creates a Huddle list from a Tcl list named ctrl_list.

```
create_result_list $ctrl_list
```

create_result_object

Context: complex

Creates a Huddle object of type *dict*.

Usage

create_result_object ***key value*** [*key value* …]

Arguments

- key

  A required string that specifies the key to later be used to access the corresponding value in the generated Huddle dict.

- value

  A required string that specifies the value to be stored in the Huddle dict. The value can be another Huddle object.

Description

This command facilitates handling of Huddle data structures and calls the "huddle create" command under the hood. You must specify one key-and-value pair, and you can optionally specify multiple key and value pairs.

Examples

The following command creates a Huddle object with three parameters: PatternSetName, DCParametericTests, and DisplayResult, where DCParametericTests is also a Huddle object.

```
create_result_object "PatternSetName" "DCMeasure" "DCParametericTests"
[create_result_object "MeasuredVoltage1" "1.8V" "MeasuredVoltage2" "1.9V"
"MeasuredVoltage3" "2.0V"] "DisplayResult" "\tMeasuredVoltage1=1.8V\n\
tMeasuredVoltage2=1.9V\n\tMeasuredVoltage3=2.0V\n"
```

evaluate_instrument_callback

Context: complex

Performs a Mentor Graphics instrument-specific callback when the instrument is present in the executed custom test or required test.

Usage

evaluate_instrument_callback ***mentor_instrument_type*** { get_pass_fail |
collect_diagnostic_data -failing_instruments *failing_mentor_instruments_list* }

Arguments

- ***mentor_instrument_type***

  A required string that specifies a valid Mentor Graphics instrument type. You can obtain the instrument type with the get_instrument_type command.

- get_pass_fail | collect_diagnostic_data -failing_instruments *failing_mentor_instruments_list*

  Choice between two Mentor Graphic instrument callbacks:

  - get_pass_fail — Returns a Huddle data structure, where the ExecStatus key value indicates the pass or fail status of the instrument. You can obtain the pass or fail status from the returned Huddle data structure by using the following command: huddle get *returned_data* "ExecStatus"

    Before using this callback, you must have previously set the active pattern set, Mentor Graphics IP group type, and Mentor Graphics instrument with the set_active_pattern_set, set_active_ip_group_type, and set_active_instrument commands.

  - collect_diagnostic_data -failing_instruments *failing_mentor_instruments_list* — Returms diagnostic results for the specified failing Mentor Graphics instruments list in the form of a Huddle data structure that you can add to the diagnostic test results for the custom test. This callback also generates diagnostic files, such as bitmap files, associated with failing instruments.

    You can obtain the list of failing instruments by issuing multiple calls to the get_pass_fail callback.

    Before using this callback, you must have previously set the active pattern set and Mentor Graphics IP group type with the set_active_pattern_set and set_active_ip_group_type commands.

Examples

Example 1

The following commands return the pass/fail status for the one_mentor_instrument_name instrument.

```
set_active_pattern_set my_failing_pattern_set
set_active_ip_group_type mentor_ip_type
set_active_instrument one_mentor_instrument_name
evaluate_instrument_callback [get_instrument_type
one_mentor_instrument_name] get_pass_file
```

Example 2

The following commands diagnose the instruments specified in the failing_ctrls list.

```
set_active_pattern_set my_failing_pattern_set
set_active_ip_group_type mentor_ip_type
evaluate_instrument_callback [get_instrument_type
one_mentor_instrument_name] collect_diagnostic_data -failing_instruments
{ $failing_ctrls }
```

execute_ate_command

Context: simple

Enables you to manipulate GPIB equipment for a custom test.

Usage

execute_ate_command ***gpib_command***

Arguments

- ***gpib_command***

  A required string that specifies a GPIB command and arguments. The GPIB command must be a valid GPIB command that is supported by the targeted GPIB instrument. Specify the command between quotes.

Description

Note

Using this command assumes that you have configured the equipment on the GPIB as described in General Purpose Interface Bus (GPIB) standard IEEE-488. It also assumes that the GPIB instrument supports the GPIB commands you specify.

You can use this command to control GPIB instruments—such as power supplies, clock generators, temperature controls, and digital IOs—as long as they support the GPIB standard.

Examples

execute_ate_command "exec my_gpib_command FREQUENCY 900Mhz"

execute_pattern_file

Context: simple

Runs pattern files associated to the current custom test.

Usage

execute_pattern_file ***pattern_name*** [-mode { gonogo | get_failures }] [-add_execution_result]

Arguments

- ***pattern_name***

  Required string that specifies the name of a pattern that is accessible through the get_test_data *test_name* -main_pattern_names_list command.

- -mode { gonogo | get_failures }

  Optional choice between preserving:

  - gonogo — The pass/fail status of the pattern file.
  - get_failures — The pass/fail status of the pattern file and the miscompare failures, if any.

- -add_execution_result

  Optional switch that specifies to add the pattern file execution results to the current custom test that you can view later by using the "report_config_data /CDPResult" command. This is the test name that is returned when you specify the get_active_test_name command.

  - Example for the gonogo mode:

```
CDPResult {
  Command : execute_test -test custom_test;
  ExecutionStatus : Success;
  ResultType : TestExecutionResult;
  TestExecutionResult {
    CustomResult(0) {
      ExecStatus : FAIL;
      Pattern : custom_test_1.stil;
    }
    CustomResult(1) {
      ExecStatus : FAIL;
      Pattern : custom_test_2.stil;
    }
    TestName : custom_test;
    TestStatus : Failed;
  }
  Version : 1;
}
```

- Example for the get_failures mode:

```
CDPResult {
  Command : execute_test -collect_failures variable -test
custom_test;
  ExecutionStatus : Success;
  ResultType : TestExecutionResult;
  TestExecutionResult {
    CustomResult(0) {
      ExecStatus : FAIL;
      Pattern : custom_test_1.stil;
      PatternSets(0) {
        PatternSetName : test1;
        Variables(0) {
          Actual : b1;
          Expected : b0;
          Pin : tdo;
          VarName : block1_I1.sib1.SIB(2);
        }
        Variables(1) {
        :
        :
        }
        Variables(7) {
          Actual : b0;
          Expected : b1;
          Pin : tdo;
          VarName : block1_I1.raw1_I1.done(1);
        }
      }
    }
    CustomResult(1) {
      ExecStatus : FAIL;
      Pattern : custom_test_2.stil;
      PatternSets(0) {
        PatternSetName : test1;
        Variables(0) {
          Actual : b1;
          Expected : b0;
          Pin : tdo;
          VarName : block1_I1.sib1.SIB(2);
        }
        Variables(1) {
        :
        :
        }
        Variables(7) {
          Actual : b0;
          Expected : b1;
          Pin : tdo;
          VarName : block1_I1.raw1_I1.done(1);
        }
      }
    }
    TestName : custom_test;
    TestStatus : Failed;
  }
  Version : 1;
```

```
}
```

Description

This command returns 1 if the pattern file fails because of miscompares, and it returns 0 if the pattern file passes.

Examples

The following command runs my_pattern_file, preserving all failures and adding them to the custom test's results.

```
execute_pattern_file my_pattern_file -mode get_failures
-add_execution_result
```

execute_required_test

Context: complex

Runs a required test associated with the current custom test. A required test ("OtherRequiredPatternsWrappers") is generated from other Pattern wrappers specified in the PatternsSpecification.

Usage

execute_required_test ***required_test*** [-collect_failures *data_type_list*]

Arguments

- required_test

  A required string that specifies the name of a required test associated with the current custom test. Allowable required test names are those returned when you specify the "get_test_data *test_name* -required_test_list" command.

- -collect_failures *data_type_list*

  An optional switch and string that specifies the type of failure data to collect. The possible data types are:

  - gonogo — pass/fail results
  - raw_fails — number of failures only; returns 0 if the test passed
  - variable — failing variables/registers
  - instrument — failing instruments
  - diagnosis — diagnosis datalog for failing instruments

Description

This command returns 1 if the required test fails because of miscompares, and it returns 0 if the required test passes. By default, the tool preserves the pass/fail status if you do not specify the -collect failure switch.

Examples

The following command runs the my_other_required_test required test and preserves failing instruments and diagnosis results:

```
execute_required_test my_other_required_test -collect_failures
{ instrument diagnosis }
```

get_active_test_name

Context: simple

Returns the name of the current custom test being executed.

Usage

get_active_test_name

Arguments

None.

Description

Within the Tessent SiliconInsight CDP context, each Patterns wrapper corresponds to a test. This command returns the name of the Patterns wrapper within which you have previously included the TestSequencing wrapper. The TestSequencing wrapper indicates that the Patterns wrapper is for a custom test.

get_instrument_type

Context: complex

Returns the Mentor Graphics instrument type required to access Mentor instrument callback procedures.

Usage

get_instrument_type ***mentor_instrument_name***

Arguments

- ***mentor_instrument_name***

  A required string that specifies the name of a Mentor Graphics instrument.

Description

Given the name of an instrument, this command returns its Mentor Graphics instrument type required to access Mentor instrument callback procedures, such as logic_bist, memory_bist, memory_bisr, and jtag_bscan.

Before using this command, you must have previously set the active pattern set with the set_active_pattern_set command and the Mentor Graphics IP group type with set_active_ip_group_type command.

get_pattern_data

Context: simple

Returns the pattern file data or failure results from the latest execution of the pattern file.

Usage

get_pattern_data ***pattern_name*** { **-pattern_sets** | **-failing_pattern_sets** | **-failing_variables_data** }

Arguments

- ***pattern_name***

  Required string that specifies the name of a pattern file associated with the current custom test. Allowable pattern file names are those returned when you specify the "get_test_data *test_name* -main_pattern_names_list" command.

- **-pattern_sets** | **-failing_pattern_sets** | **-failing_variables_data**

  Required choice between returning:

  - -pattern_sets — List of pattern sets for the specified pattern name.
  - -failing_pattern_sets — List of failing pattern sets for the specified pattern name.
  - -failing_variables_data — List of failing variables for the specified pattern name.

Description

Using this introspection command assumes that you ran the pattern file in get_failures mode. In addition, this command assumes that the pattern files include TESSENT_PRAGMA indications, which are automatically generated when you create pattern files using the Tessent Shell flow.

Examples

The following command returns a list of failing pattern sets that resulted from executing my_pattern_file:

```
get_pattern_data my_pattern_file -failing_pattern_sets
```

get_test_data

Context: simple and complex

Returns available collateral data for the current custom test.

Usage

Usage for Simple Custom Test Flows

get_test_data ***test_name*** {-main_pattern_names_list |
-parameter_value -parameter_name *parameter_name* [-parameter_type dynamic]}

Usage for Complex Custom Test Flows

get_test_data ***test_name*** { -main_pattern_names_list |
-parameter_value -parameter_name *parameter_name* [-parameter_type dynamic] |
-required_tests_list |{-execution_status | -execution_result }
[-for_required_test *required_test*] }

Arguments

- ***test_name***

  Required string that specifies the name of the current custom test, which is the name of the Patterns wrapper.

  In complex custom test flows, when specified with the "-parameter_value -parameter_name *parameter_name*" choice, the test name can be the name of another required test associated with the original custom test.

- -main_pattern_names_list

  Optional string that specifies to return a list of pattern files associated with the specified test.

  If you specified the pattern files with the TestSequencing/CustomPatterns wrapper, the files are listed in the same order as they appear in the CustomPatterns wrapper. If you generated the pattern files from the Patterns wrapper specification, the number of pattern files depends on the number of splits (#splits + 1) specified in the Patterns wrapper and the files appear in the same order as they appear in the Patterns wrapper.

  If pattern files exist in both sources, the files specified through the CustomPatterns wrapper are listed first.

- -parameter_value -parameter_name *parameter_name* [-parameter_type dynamic]

  Optional strings that together (with the "-parameter_type dynamic" option) specify to return the value of a custom test parameter named *parameter_name*. The parameter name is any user-specified parameter in the TestSequencing/CustomData wrapper. The option returns an empty string if the specified parameter is not in the wrapper.

  For complex custom test flows, you can also specify the parameter name (without the "-parameter_type dynamic" option) as:

  - *pattern_set_name*::InstrumentTypeGroupsList — Returns the Mentor Graphics instrument type present in the specified pattern set. In this context, a pattern set

corresponds to a TestStep or ProcedureStep wrapper under a Patterns wrapper. This is useful when introspecting custom flow patterns testing Mentor Graphics instruments.

- *pattern_set_name*::*instrument_type*::InstrumentList — Returns the list of Mentor Graphics instruments of the specified type present in the specified pattern set. This is useful when introspecting custom flow patterns testing Mentor Graphics instruments.

- -required_tests_list

  Optional string that specifies to returns a list of the required tests associated with the specified custom test as listed in the TestSequencing/OtherRequiredPatternsWrappers wrapper.

- { -execution_status | -execution_result } [-for_required_test *required_test*]

  Optional string that specifies to return either the execution status or the execution result for the specified custom test, or, if -for_required_test is specified, for the specified required test. The required test must be a valid required test associated with the specified custom test.

  - -execution_status — Returns the pass/fail status of the specified custom test, or the required test if you specified the -for_required_test option.
  - -execution_result — Returns the test result in Huddle format for the specified custom test, or the required test if you specified the -for_required_test option.

    From the returned Huddle data structure, you can retrieve a Huddle list of failing pattern sets by using the PatternSets key as follows:

    ```
    huddle get returned_data "PatternSets"
    ```

    Then, for each element in the list of failing pattern sets, you can return the name of the failing pattern set by using the "PatternSetName" key value as follows:

    ```
    huddle get returned_data_from_PatternSets_key index
    huddle get returned_data_from_previous_call "PatternSetName"
    ```

Description

This command enables access to collateral data required for running the current custom test.

Examples

Example 1: Simple Custom Test Flows

The following command retrieves the list of pattern files associated with the current custom test:

```
get_test_data [get_active_test_name] -main_pattern_names_list
```

The following command retrieves the value of the Param1 parameter if it was specified in the CustomData wrapper:

```
get_test_data [get_active_test_name] -parameter_value -parameter_name
Param1 -parameter_type dynamic
```

Example 2: Complex Custom Test Flows

The following command returns a list of other required tests associated with the specified custom test:

```
get_test_data [get_active_test_name] -required_tests_list
```

The following command returns the test result in Huddle format for the my_other_required_test required test associated with the specified custom test:

```
get_test_data [get_active_test_name] -execution_result -for_required_test
my_other_required_test
```

The following command retrieves the value of "my_pattern_set::mentor_ip_type::InstrumentList", which is the list of Mentor Graphics instruments of type mentor_ip_type that are tested in the my_pattern_set pattern set with the my_other_required_test required test:

```
get_test_data my_other_required_test -parameter_value -parameter_name
my_pattern_set::mentor_ip_type::InstrumentList
```

get_test_datalog_mode

Context: simple

Returns a Tcl list of the types of failure data to preserve, as specified with the “execute_cdp_test -collect_data_type_list” command/option pair.

Usage

get_test_datalog_mode

Arguments

None.

Description

You can request to preserve the following data types:

- gonogo — pass/fail results
- raw_fails — the number failures only; returns 0 if the test passed
- variable — failing variables/registers
- instrument — failing instruments
- diagnosis — diagnosis datalog for failing instruments

get_test_execution_status

Context: simple

Gets the pass/fail execution status of the current test.

Usage

get_test_execution_status

Arguments

None.

log_message

Context: simple

Outputs specified messages to the log file produced by the SID tester process.

Usage

log_message ***message_string***

Arguments

- ***message_string***

  A required string that specifies the message you want to appear in the log file. Use double quotes around the string and include \n to denote the end of a line.

Description

For custom test flows, you can include this command in the Tcl procedure that defines the test flow.

Examples

The following command outputs “my_test_flow: Entering ...” to the SID tester log file:

```
log_message "my_test_flow: Entering ...\n"
```

set_active_instrument

Context: complex

Sets the currently active Mentor Graphics instrument.

Usage

set_active_instrument ***mentor_instrument_name***

Arguments

- ***mentor_instrument_name***

  A required string that specifies the name of a Mentor Graphics instrument.

Description

You can obtain a list of instrument names with the "get_test_data *test_name* -parameter_value -parameter_name *pattern_set_name*::*mentor_ip_group_type*::InstrumentList" command. Mentor Graphics IP group types correspond to IP controllers that are inserted into designs using the Tessent Shell environment, such as MemoryBist, LogicBist, and MemoryBisr.

set_active_ip_group_name

Context: complex

Sets the currently active Mentor Graphics IP group type.

Usage

set_active_ip_group_type ***mentor_ip_group_type***

Arguments

- ***mentor_ip_group_type***

  A required string that specifies the name of a Mentor Graphics IP group type, such as LogicBist, MemoryBist, or MemoryBisr.

Description

You can obtain the Mentor Graphics IP group type with the “get_test_data *test_name* -parameter_value -parameter_name *pattern_set_name*::InstrumentTypeGroupsList” command.

set_active_pattern_set

Context: complex

Sets the currently active pattern set. A pattern set is marked by a “TESSENT_PRAGMA pattern_set” indication in the pattern file and corresponds to a TestStep or ProcedureStep wrapper under a Patterns wrapper if a PatternsSpecification exists for this pattern file.

Usage

set_active_pattern_set ***pattern_set_name***

Arguments

- ***pattern_set_name***

  A required string that specifies the name of a pattern set.

Description

You can obtain the pattern set name with the get_pattern_data command for pattern files or the get_test_data command for other required tests.

set_active_test_name

Context: complex

Sets the currently active test.

Usage

set_active_test_name ***test_name***

Arguments

- ***test_name***

  Required string that specifies the name of the custom test or required test that will be the currently active test, where a custom test is the name of the Patterns wrapper and a required test is specified within the TestSequencing/OtherRequiredPatternsWrappers wrapper.

Description

By default, the currently active test is the custom test being executed. In cases where you want to perform a Mentor Graphics instrument-specific callback on a required test that is part of the custom test (to get failing instruments or to diagnose them), you can use this command to temporarily change the currently active test to a required test. Ensure that you restore the custom test as the currently active test when you have completed the required test.

set_test_diagnosis_result

Context: complex

Sets the diagnosis results for the current custom test.

Usage

set_test_diagnosis_result ***huddle_list***

Arguments

- huddle_list

  A required string that specifies a list of Huddle objects. Specifying an empty string clears all accumulated diagnosis results.

Description

You can configure the Huddle objects within the Huddle list as needed, but the following Huddle keys require special consideration:

- PatternSetName and its value are mandatory.
- When specified, the DisplayResult string automatically displays at the Tessent Shell console.
- When Mentor Graphics instruments are specified in the patterns specification for the current custom test, the tool uses ParticipatingInstruments and FailingInstruments to update the Pass/Fail hierarchical tree.

Examples

The following command sets the diagnosis result of the custom test to a Huddle list, where one element is a Huddle object made of other Huddle objects or a list:

```
set_test_diagnosis_result [create_result_list [create_result_object
"DCParametericTests" [create_result_object "MeasuredVoltage" "1.8V"]
"FuncFail" [create_result_object "FailingInstruments" [create_result_list
"blockA_inst.blockA_rtl_tessent_mbist_c1_controller_inst"]]]]
```

set_test_execution_status

Context: simple

Sets the pass/fail execution status of the current custom test.

Usage

set_test_execution_status { passed | failed }

Arguments

- passed | failed

  Choice between setting the test execution status to passed or to failed. The default execution status is passed.

Chapter 12
Interactive Desktop Getting Started Tutorial

This tutorial is intended to teach you the basics of running tests and diagnosing logic and memory failures using Tessent SiliconInsight.

For the purposes of this tutorial, you use Tessent SiliconInsight Desktop in SimDUT mode because this mode does not require an adaptor. The only requirement is a signed-off chip-level TSDB for the design.

The provided TSDB is for a DUT with one Tessent MemoryBist controller and one LogicBist controller. This tutorial also uses a previously created patterns specification that is configured to collect a limited number of diagnostic failures.

Accessing the Tutorial

The data you need to perform the tutorial is located in your release tree.

1. Create a work directory and change to that directory:

   mkdir work_dir

   cd work_dir

2. In your work directory, untar the TSDB using the following command:

   tar xvfz <installed release tree>/share/SiliconInsight/Tutorial/tutorial1.tgz

3. Change directory to the tutorial1 directory.

   cd tutorial1

Running Tessent SiliconInsight and Performing Tests in the GUI

The tutorial uses two tabs of Tessent Visualizer to demonstrate how to run and diagnose tests. It uses the Config Data Browser tab, which is the main view of Tessent SiliconInsight, and it uses the Transcript tab to view results and run commands.

This tutorial instructs you in performing the following:

- Start Tessent Shell in the correct context and load all the necessary files.

- Run tests.
- Diagnose a failed memory test.
- Diagnose a failed logic test.
- Exit Tessent SiliconInsight.

Prerequisites

- You have accessed the tutorial1 directory.
- You have a licensed installation of the Siemens EDA ModelSim/Questa Verilog simulator with vsim and vlog specified in your environment path.

Procedure

1. Start Tessent Shell:

   ```
   % tessent -shell
   ```

2. Delete the CDP of any earlier run of the tutorial to force creation of a new one every time.

   ```
   SETUP> file delete -force top.cdp
   ```

3. Set the Tessent SiliconInsight context:

   ```
   SETUP> set_context patterns -silicon_insight
   ```

4. (Optional) If you want the memory bitmap report to include the physical X-Y coordinates, create a layout database:

   ```
   SETUP> create_layout MyLayoutDB -def layout/top.def \
       -leflist { layout/SYNC_1R1W_16x8.lef layout/tech.lef } -replace
   ```

5. Open the TSDBs. This design only has one TSDB:

   ```
   SETUP> open_tsdb top.tsdb
   SETUP> read_design top -design_id rtl -view interface
   SETUP> set_current_design top
   ```

6. Generate the default setup specification:

   ```
   SETUP> create_silicon_insight_setup_specification
   ```

   By default, the tool creates a setup specification named SiliconInsightSetupSpec.cfg.

7. (Optional) You can edit the SiliconInsightSetupSpec.cfg file to add or change startup settings. If you do, load those settings using the read_config_data command:

   ```
   SETUP> read_config_data SiliconInsightSetupSpec.cfg
   ```

8. Customize the default setup specification by changing the selected_setup property to Sid (simdut).

   ```
   SETUP> set_current_silicon_insight_setup Sid(simdut)
   ```

For more information about customizing setup specifications, see "Generating the Setup Specification for Desktop Mode (TSDB Flow)" on page 76.

Note

The set_current_silicon_insight_setup command only affects the current session. If you want to save a change to the setup specification so that you can read it in using the read_config_data command in a subsequent session, you must use the write_config_data command.

9. Load the patterns specification named *my_patspec.cfg*:

```
SETUP> read_config_data my_patspec.cfg
```

   You can save the patterns specification to a file using the write_config_data command and load it in subsequent sessions using the read_config_data command.

10. (Optional) If you want the memory bitmap report to include the physical X-Y coordinates, set up the appropriate Characterization Debug Options properties:

```
SETUP> set_config_value /PatternsSpecification(top,rtl,manufacturing)/ \
    CharacterizationDebugOptions/MemoryBist/layout_database MyLayoutDB
SETUP> set dirWrap [add_config_element -in_wrapper \
    /PatternsSpecification(top,rtl,manufacturing)/CharacterizationDebugOptions/ \
    MemoryBist VendorXYMapDirectories]
SETUP> add_config_element -in_wrapper $dirWrap vendor_data -type property
```

11. (Optional) If you want to generate a layout marker file (using the *.lay* extension) from the reported physical X-Y coordinates, set up the appropriate CharacterizationDebugOptions property. You can view the layout marker file in the Calibre DRC viewer; refer to the *Calibre DESIGNrev Layout Viewer User's Manual* for more information.

```
SETUP> set_config_value /PatternsSpecification(top,rtl,manufacturing)/ \
    CharacterizationDebugOptions/MemoryBist/generate_marker_file on
```

12. Set up the Verilog design files necessary for simulation:

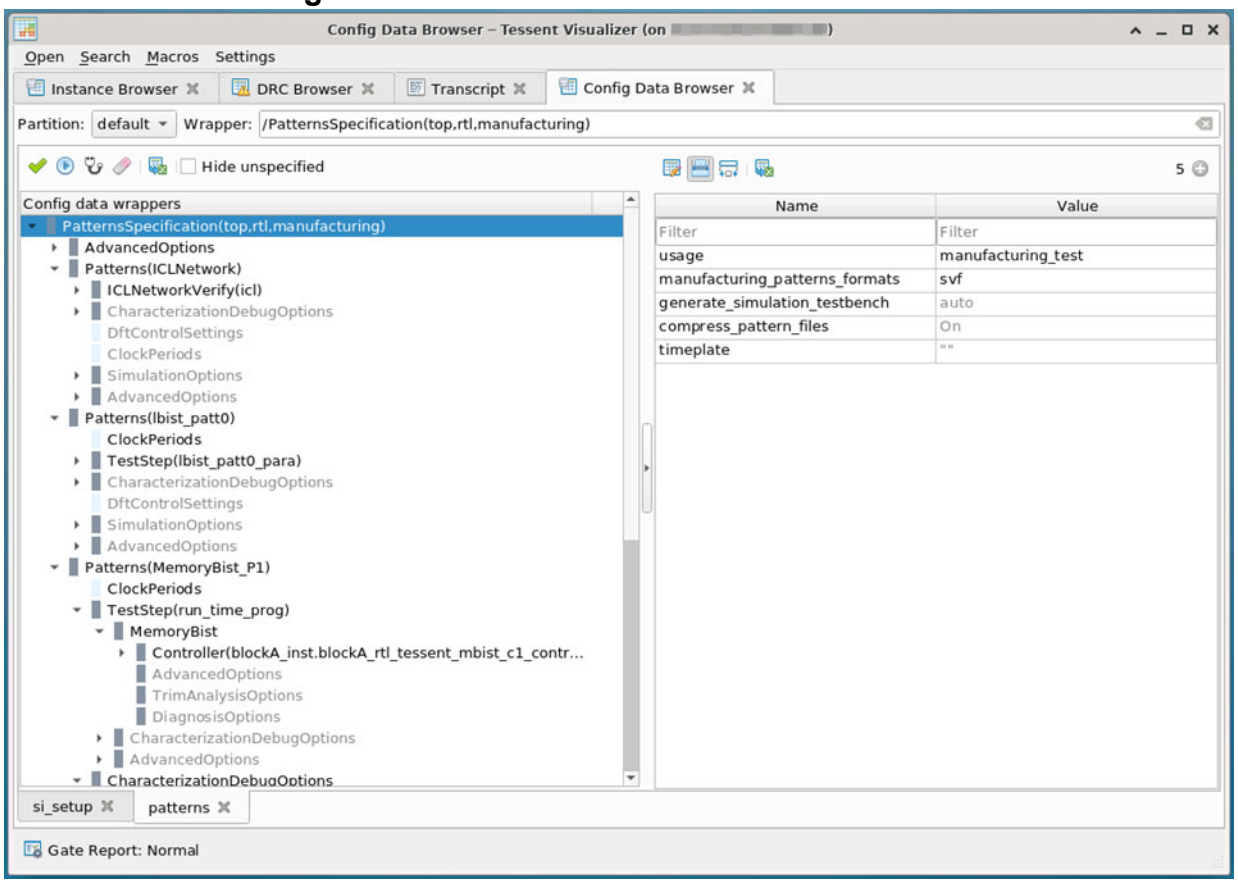
```
SETUP> set_simulation_library_sources \
    -v ../prerequisites/techlib_adk.tnt/current/verilog/adk.v \
    -v ../data/mem/*.v \
    -v ../data/picdram.v

SETUP> run_testbench_simulations -simulator_options "+nospecify" -simdut_server
```

    This step pertains to SimDUT mode only. For details, refer to "Simulating Desktop, ATE, and ATPG Behavior" on page 223.

13. Start Tessent Visualizer:

```
SETUP> start_silicon_insight -gui
```

Figure 12-1. Initial View of tutorial1 in the GUI



Each Patterns wrapper under the PatternsSpecification is a test. Selecting an individual MemoryBist controller saves test time in designs with many MemoryBist controllers. You can click any of these wrappers in the "Config data wrappers" pane and then run them by clicking **Process/Execute** ( ).

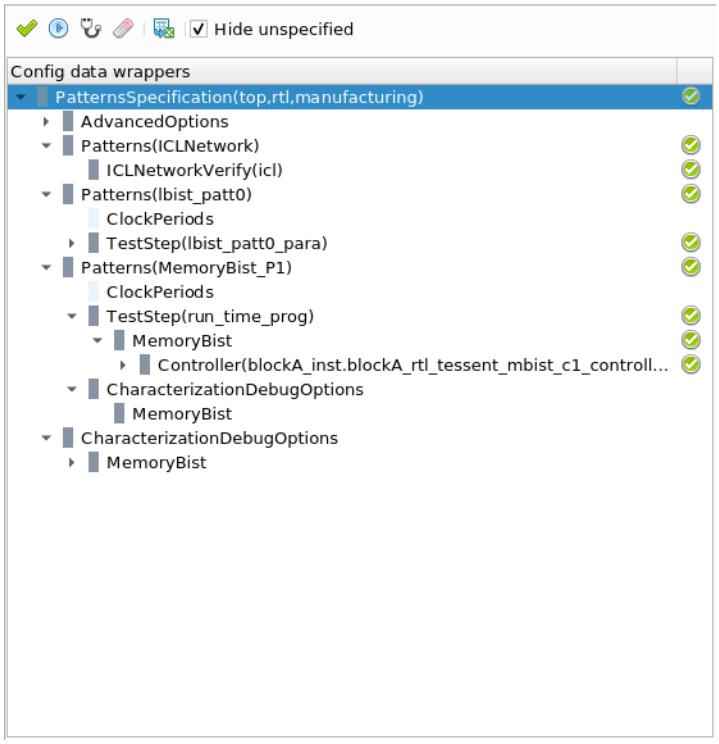
Green or red icons appear in the Config Data Browser tab depending on the result of the test: green for pass and red for fail.

The transcript of the test displays in the Transcript tab.

14. In the Config Data Browser tab, hide the unspecified wrappers to remove them from view in the "Config data wrappers" pane. Click the **Hide unspecified** box (Hide unspecified).

15. In the Config Data Browser tab, run all of the tests to ensure all tests pass. Ensure the PatternsSpecification wrapper is selected, and then click **Process/Execute**.

The “Config data wrappers” pane displays a summary of results that looks like the following, while the Transcript tab displays the detailed results:



Note

After running a test, you can introspect the results located in the CDPResult partition, which stores the results of the last performed test in memory. For more information about the CDPResult partition, see “CDP Result Structure” on page 207 and “CDP Result Introspection” on page 214.

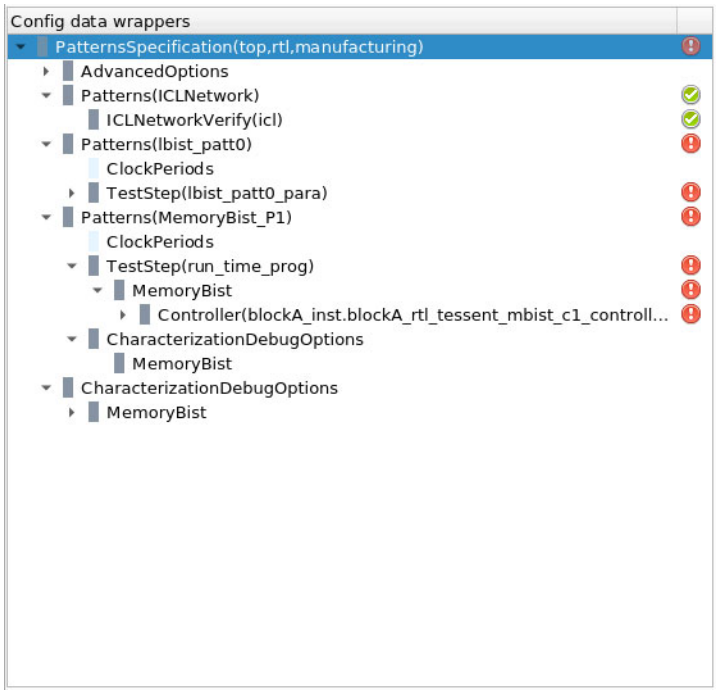
16. Because you are working in simulation mode, you can inject failures into the design to test it further. In the Transcript tab, inject one failure in the logic and one failure in the memory by entering the following commands in the command field beside the mode indicator label (SETUP>):

    add_simdut_fault -signal SIMDUT_TB.DUT_inst.piccpu_inst1.inst_reg_7.QB -fault 1

    add_simdut_fault -signal SIMDUT_TB.DUT_inst.blockA_inst.mem1.Q[0] -fault 1

17. In the Config Data Browser tab, select PatternsSpecification, and then click the **Process/ Execute** button.

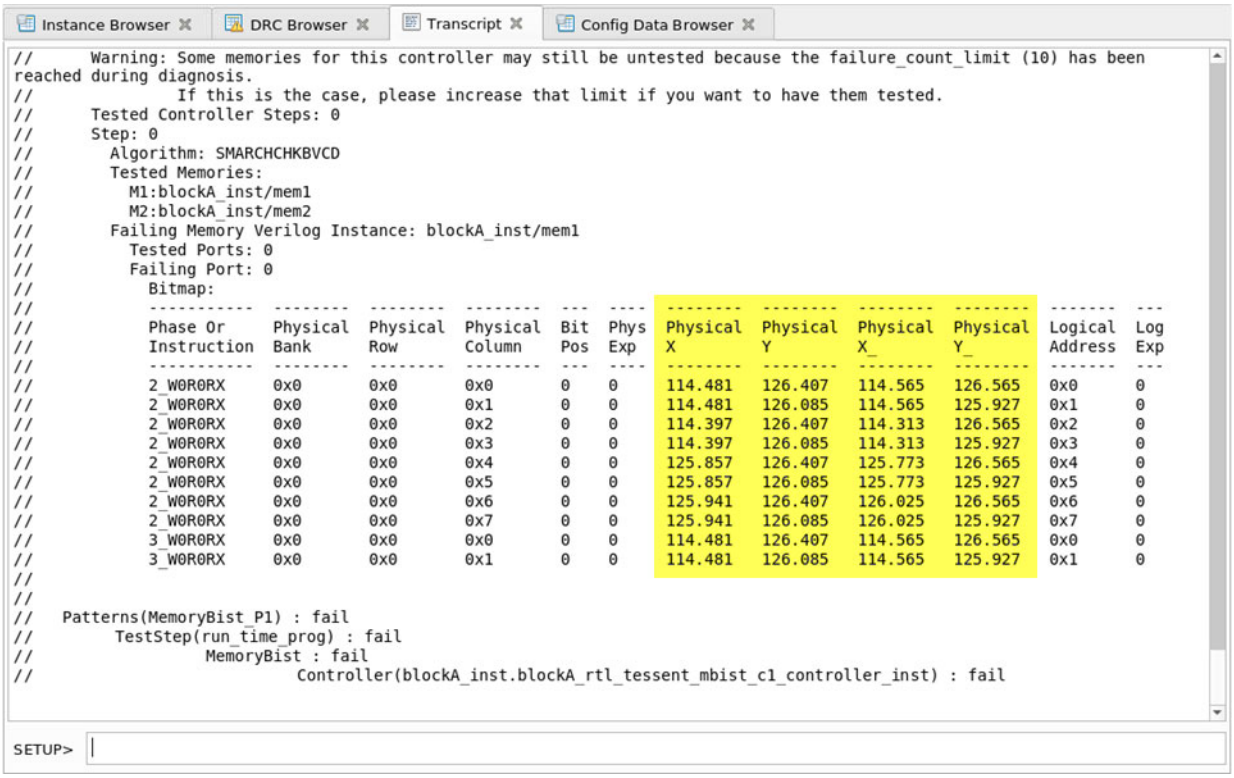
The GUI displays the following failures in the “Config data wrappers” pane as indicated by the red icons next to the wrappers. In addition, the Transcript tab displays the failing registers for each test.



18. In the Config Data Browser tab, click Patterns (MemoryBist_P1), and then click **Diagnose** ().

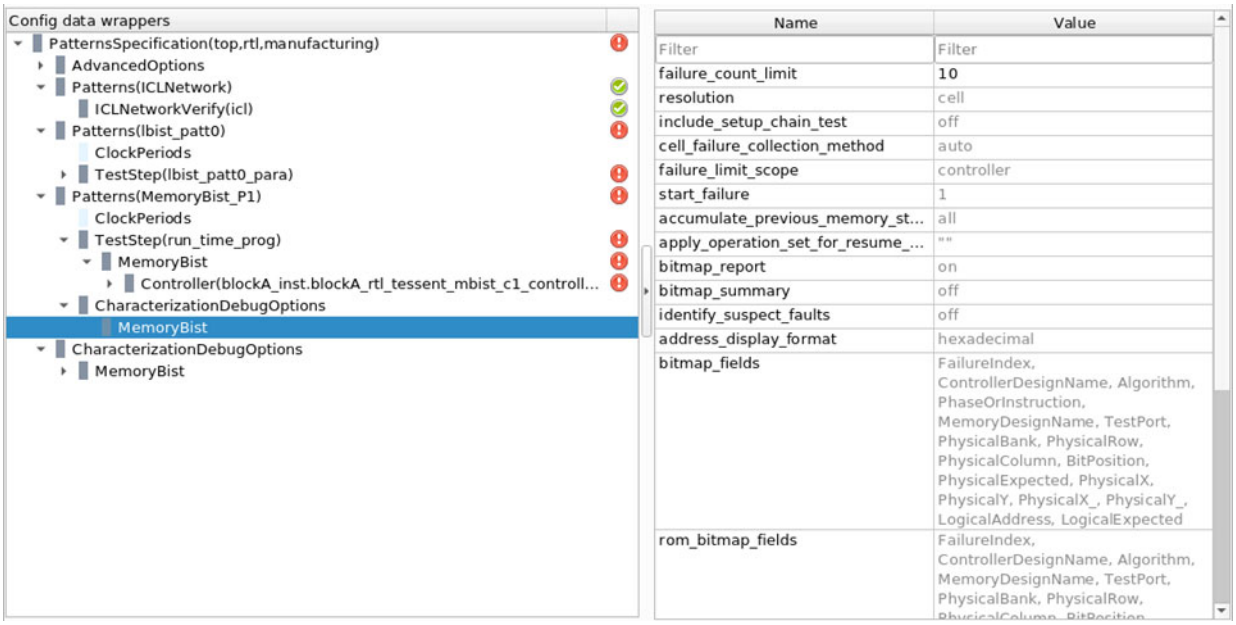
    The diagnosis results for this failed memory test display in the Transcript tab. These results include the optional physical X-Y coordinate results, which are highlighted in

yellow. If you do not follow the optional steps to produce these results, the results do not include the Physical X/Y/X_/Y_ columns.



19. In the Config Data Browser tab, view the characterization debug options for this test. In the “Config data wrappers” pane, find the Patterns (MemoryBist_P1) wrapper of this test and the CharacterizationDebugOptions wrapper underneath. Click MemoryBist.

The current characterization debug option settings of the MemoryBist test display:



The failure count limit is currently set to 10 to reduce diagnosis time because you are running in SimDUT mode.

You can adjust any of these settings as described in “Performing Manual Debug” on page 138 to control the way diagnosis is performed and to debug the failure.

20. In the Config Data Browser tab, click Patterns(lbist_patt0), and then click **Diagnose**.

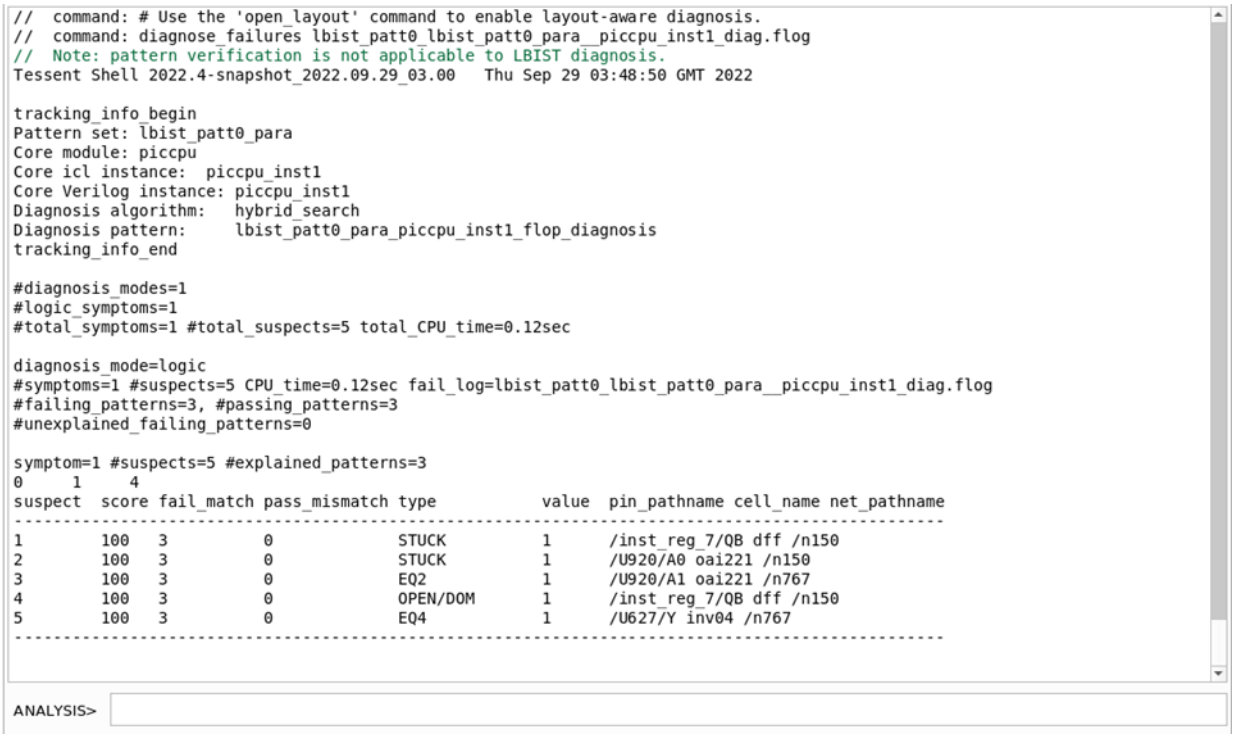
The diagnosis results for this failed logic test display in the Transcript tab:

Refer to "LogicBIST Diagnosis Results" on page 203 for details about the diagnosis results displayed in the transcript. The tool automatically saves the Tessent Diagnosis dofile as shown in the Transcript tab.

Run the dofile, and the tool diagnoses the logic down to the failing net. Enter the following command in the command field beside the mode indicator label (SETUP>):

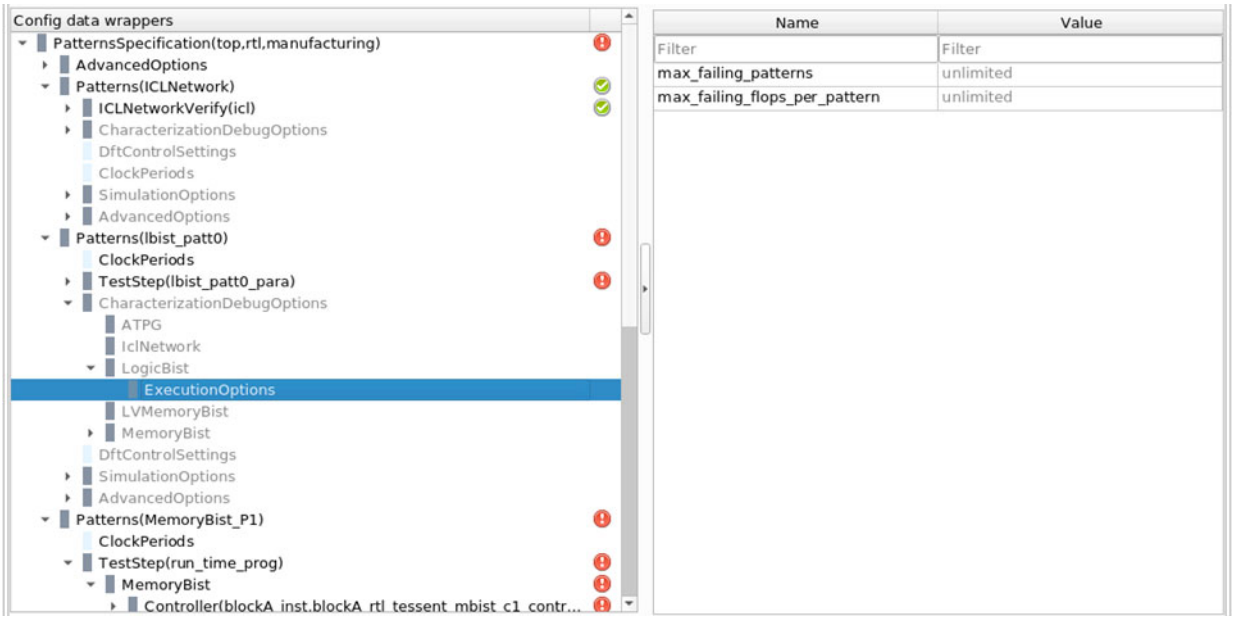
dofile lbist_patt0_lbist_patt0_para__piccpu_inst1_td_dofile.do

The dofile enters analysis mode and runs Tessent Diagnosis. The results display:



21. In the Config Data Browser tab, to view the execution options for this test, click the **Hide unspecified** box (☑ Hide unspecified) to clear it. Under Patterns(lbist_patt0), click the right arrow (▶) button next to CharacterizationDebugOptions, click the right arrow button next to LogicBist, and then click ExecutionOptions.

The current settings of the ExecutionOptions for this test display:



You can adjust any of these settings as described in "Performing Manual Debug" on page 138 to control the way diagnosis is performed and to debug the failure.

22. Exit the Tessent SiliconInsight session. In the Transcript tab, enter the following command in the command field beside the mode indicator label (ANALYSIS>):

    stop_silicon_insight

    The sid_tester shuts down.

23. Close Tessent Visualizer. From the menu, choose **Open > Quit Tessent Visualizer**.

24. Exit Tessent Shell:

    ANALYSIS> exit

Related Topics

Characterization Debug Options

CDP Result Introspection

CDP Result Structure

Diagnostic Bitmap File

Chapter 13
ATPG Diagnosis Tutorial

This tutorial is intended to teach you how to run tests and diagnose ATPG pattern failures.

Accessing the Tutorial

The data you need to perform the tutorial is located in your release tree.

1. Create a work directory and change to that directory:

   mkdir work_dir

   cd work_dir

2. In your work directory, untar the TSDB using the following command:

   tar xvfz <installed release tree>/share/SiliconInsight/Tutorial/tutorial2.tgz

3. Change directory to the tutorial2 directory.

   cd tutorial2

Diagnosing ATPG Failures

The tutorial uses the Tessent SiliconInsight GUI to demonstrate how to run and diagnose ATPG pattern failures.

This tutorial instructs you in performing the following:

- Start Tessent Shell in the correct context and automatically create a directory containing SimDUT collateral data (SIMDUT.v file).
- Restart Tessent Shell and load all the necessary files to set up the test in the GUI.
- Run pass/fail tests at the top level for the glue logic and at the core level for the retargeted chip-level patterns.
- Inject faults, run tests, and perform diagnosis for the top-level glue logic and core-level retargeted patterns.
- Run more experiments by running tests, collecting failing flop data, and performing iterative diagnosis.

Prerequisites

- You have accessed the tutorial2 directory.
- You have a licensed installation of the Siemens EDA ModelSim/Questa Verilog simulator with vsim and vlog specified in your environment path.

Procedure

1. Create the testbench wrapper required for SimDUT simulation, starting with invoking Tessent Shell:

```
% tessent -shell -log tsi.log -replace
SETUP> set_context patterns -silicon_insight
// remove working directories if any existing
SETUP> system rm -rf simulation_outdir simdut_outdir
// read in the verilog netlist
SETUP> read_verilog verilog/design_syn_scan.v
SETUP> read_verilog verilog/test_design_edt.v
SETUP> read_verilog verilog/core1.v -blackbox
SETUP> read_verilog verilog/core2.v -blackbox
SETUP> read_verilog verilog/glue.v
SETUP> read_verilog verilog/Hier4Core_top.v
// read in the ATPG library
SETUP> read_cell_library libs/adk.atpg
SETUP> set_current_design top_w_hier
SETUP> run_testbench_simulations -simdut_server \
      -generate_simdut_files_only
SETUP> exit -d
```

   The tool creates a simdut_outdir directory.

   The run_testbench_simulations command generates the SimDUT collateral data in a directory with the following message:

```
// The SimDUT files were generated in directory
// './simdut_outdir/top_w_hier.simdut_gate/SIMDUT.simulation'.
// To use this setup with run_testbench_simulation,
// point to it using the flag -simdut_output_directory.
```

2. Restart Tessent Shell:

```
% tessent -shell -log tsi_atpg.log -replace
```

3. Set the Tessent SiliconInsight context:

   SETUP> set_context patterns -silicon_insight

4. Read in the setup specification. For this tutorial, the selected_setup is Sid(simdut).

   SETUP> read_config_data simdut/si.setup

   The setup specification named si.setup uses a SimDUT startup dofile (simdut_startup.dofile) to automatically load the design files and run the simulator. The dofile contains the following commands. Do not enter these commands on the command line.

```
read_cell_library libs/adk.atpg
set_sim_lib_sources -v libs/macros.v -v libs/adk.v
run_testbench_simulation -simdut_server -simdut_output_directory \
   ./simdut_outdir/top_w_hier.simdut_gate/SIMDUT.simulation \
   -simulator_options {+nowarnTSCALE}
```

For more information about the setup specification, refer to "Prepare the Design Under Test" on page 75.

5. Create a patterns specification for ATPG by sourcing the following dofile, which is located in the dofiles directory (not in the tutorial2 directory):

   SETUP> dofile ../dofiles/create_pat_spec.do

   For ATPG, the patterns specification consists of two ATPG test step wrappers: UserDefinedATPG and UserDefinedRetargetedATPG. The UserDefinedATPG wrapper defines the flat ATPG pattern, and the UserDefinedRetargetedATPG wrapper defines the hierarchical ATPG pattern, which also includes the CoreModule wrappers.

   For details about the commands and format you use to generate a PatternsSpecification for ATPG, refer to "Generating a Patterns Specification for ATPG" on page 161.

6. Start Tessent SiliconInsight in GUI mode:

   SETUP> start_silicon_insight -gui

   a. Expand the PatternsSpecification wrapper by clicking the right arrow ( ) next to it.

   b. If necessary, activate (Hide unspecified) hiding unspecified wrappers by clicking the **Hide unspecified** box.

   Each Patterns wrapper (or node) under the PatternsSpecification is a test. You can select any of the Patterns wrappers in the "Config data wrappers" pane and then click the **Process/Execute** button ( ) in the toolbar above the "Config data wrappers" pane to run it. You can also select the PatternsSpecification wrapper at the top and then click the **Process/Execute** button to run all patterns.

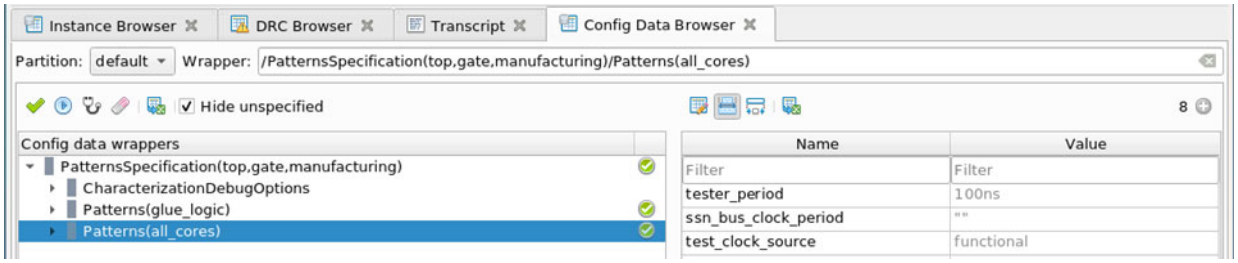
7. In the **Config Data Browser** tab, run the top-level flat test pattern (glue_logic) and then the retargeted hierarchical test pattern (all_cores).

   a. Select Patterns(glue_logic) in the "Config data wrappers" pane on the left.

   b. Click the **Process/Execute** button in the toolbar above.

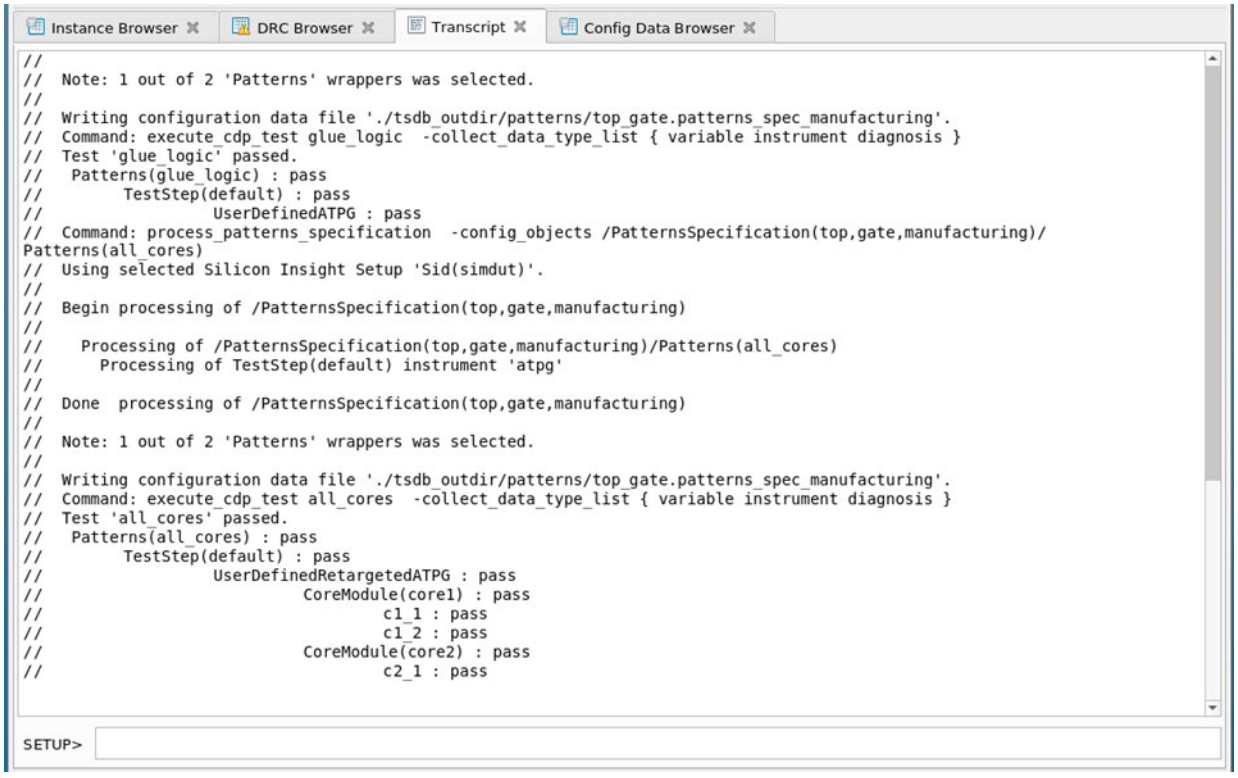
   c. Select Patterns(all_cores).

d. Click the **Process/Execute** button.



During test execution, the tool simulates each pattern using vsim. Green or red icons appear in the “Config data wrappers” pane depending on the result of the test: green check for pass and red exclamation for fail. The vsim results are also saved to the simdut output directory as defined by the simdut_startup.dofile. Click the right arrows beside Patterns(glue_logic) and Patterns(all_cores) and the other wrappers under them to reveal pattern details for a top-level test and two core tests.

The results of the tests also display in the **Transcript** tab as shown in the following. Beneath the transcript is a command field beside a mode indicator label (SETUP>).



Note

After running a test, you can introspect the results located in the CDPResult partition, which stores the results of the last performed test in memory. For more information about the CDPResult partition, see “CDP Result Structure” and “CDP Result Introspection”.

8. Because you are working in simulation mode, you can inject failures into the design to test it further.

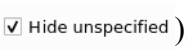
   a. In the **Transcript** tab, enter the following commands in the command field beside the mode indicator label (SETUP>):

   ```
   add_simdut_fault -signal SIMDUT_TB/DUT_inst/my_glue/cpu_i/uDMI/ix1019/Y \
     -fault 0

   add_simdut_fault -signal SIMDUT_TB/DUT_inst/c1_2/cpu_i/ix1232/Y -fault 0

   add_simdut_fault -signal SIMDUT_TB/DUT_inst/c2_1/cpu_i/uDMI/ix1019/Y -fault 0
   ```

   These commands add one fault each to the top level and two core tests.

9. In the **Config Data Browser** tab, run the Patterns(all_cores) test again. The test fails.

10. Run diagnosis on the test results.

    a. In the **Config Data Browser** tab, if the **Hide unspecified** box is active (), clear it by clicking the box to reveal unspecified wrappers.

    b. Select CharacterizationDebugOptions in the “Config data wrappers” pane on the left.

    c. Double-click the diagnose_on_failure property in the property table pane on the right.

    d. A small Set Property Value window opens. Choose “on” and click **Apply**.

    e. In the left pane, select ATPG under CharacterizationDebugOptions. If ATPG is not visible, click the right arrow beside CharacterizationDebugOptions to reveal it.

    f. Double-click the diagnosis_result property in the right pane.

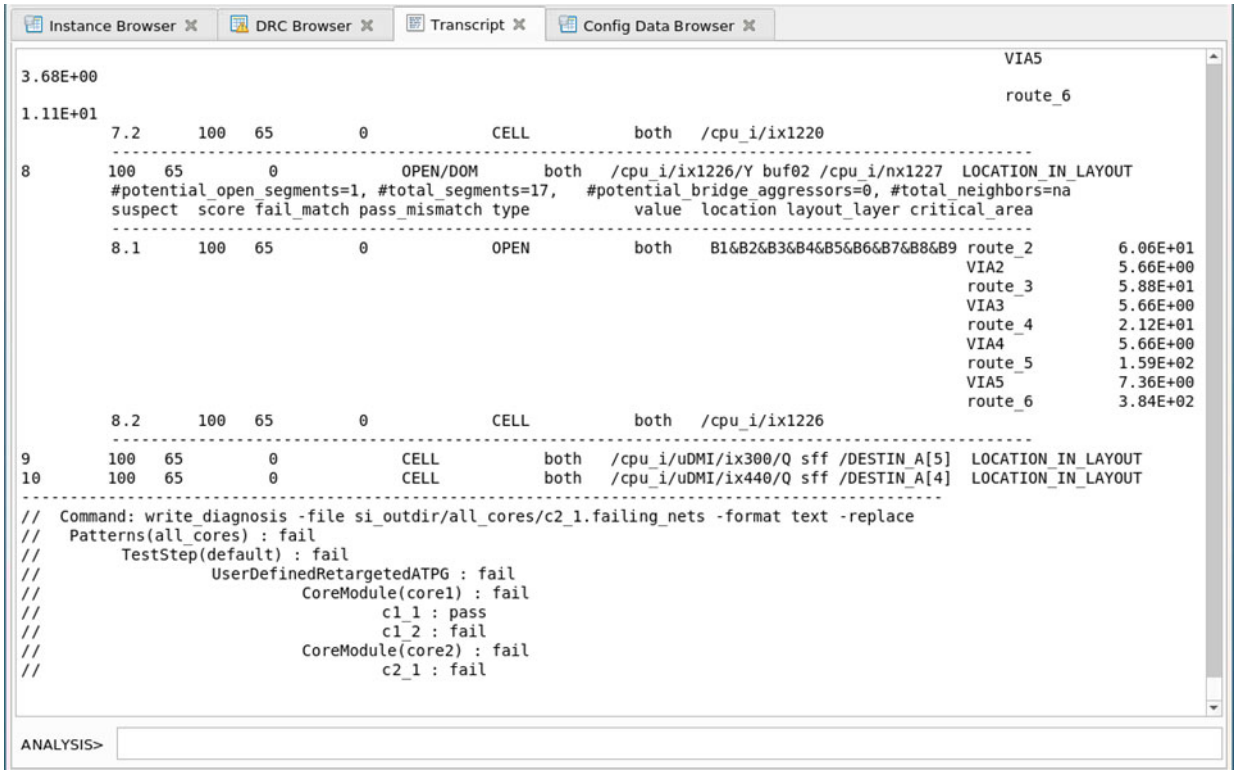
    g. A small Set Property Value window opens. Choose “failing_nets” and click **Apply**.

    h. Click the box beside **Hide unspecifed** to enable hiding unspecified wrappers again.

    i. Select the Patterns(all_cores) test.

    j. Click the **Process/Execute** button to run the test again.

k. View the results in the **Transcript** tab.



When you specify to perform diagnosis on the failure (in this case on the failing nets), the tool returns the failure information at the chip level. In the background, the Tessent Shell context resets to pattern -scan_diagnosis, and Tessent Diagnosis reverse maps the top-level failure files back to core-level failures files as described in "Reverse Mapping Top-Level Failures to the Core" in the *Tessent Diagnosis User's Manual*.

View the results in the **Transcript** tab. The tool creates two core failure files (one for the core1 instance named c1_2 and one for the core2 instance named c2_1) and performs layout-aware diagnosis.

Note

The mode changes from SETUP to ANALYSIS when the tool reads in the flat model. The flat model is generally archived in ANALYSIS mode during ATPG.

You can also run the Patterns(glue_logic) test and run diagnosis on its results. In this case, there is no reverse-mapping because the design (flat model) is already at the chip level.

11. Examine the failing flops of the Patterns(glue_test) test.

    a. In the **Config Data Browser** tab, select ATPG under CharacterizationDebugOptions in the "Config data wrappers" pane on the left.

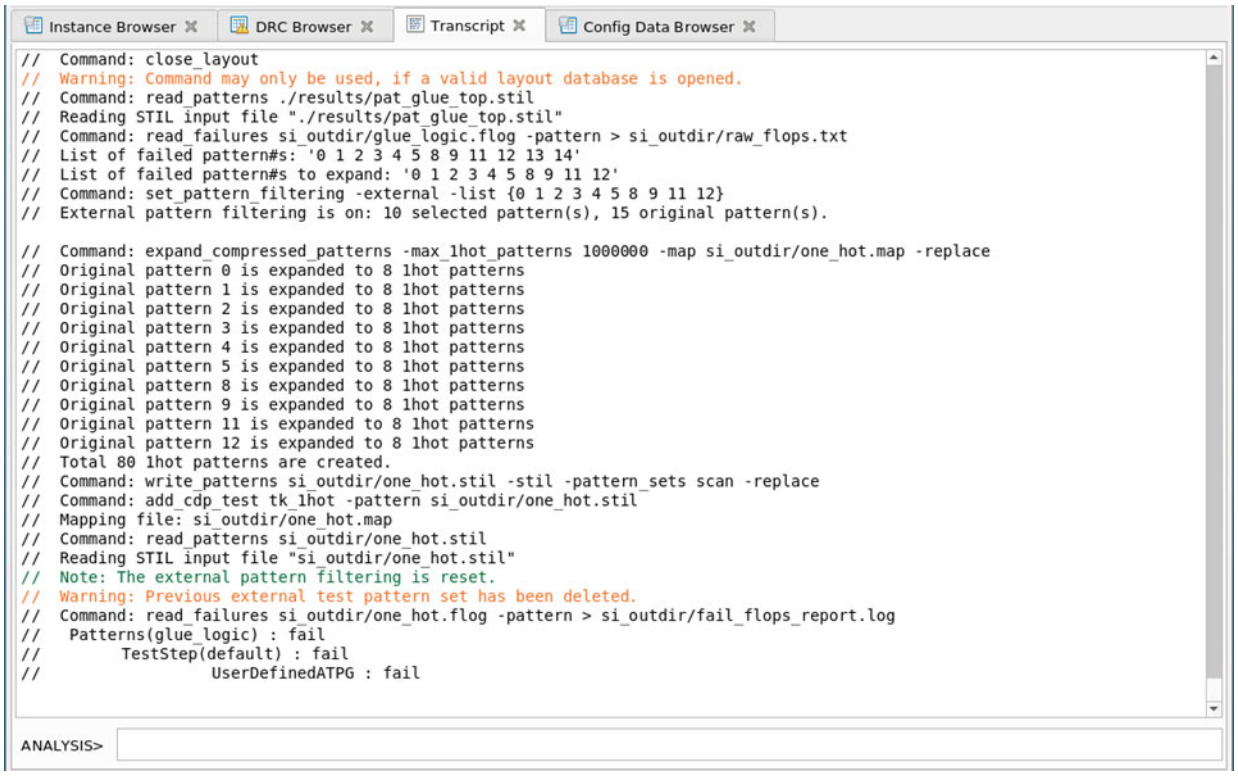
    b. Double-click the diagnosis_result property in the property table pane on the right.

c. A small Set Property Value window opens. Choose “failing_flops” and click **Apply**.

d. Select the Patterns(glue_logic) test in the left pane.

e. Click the **Process/Execute** button to run the test again.

f. View the results in the **Transcript** tab.



12. Summarize the flop information of the test for different core and failing patterns by specifying the failing_flops_summary option.

    a. In the **Config Data Browser** tab, select ATPG under CharacterizationDebugOptions in the “Config data wrappers” pane on the left.

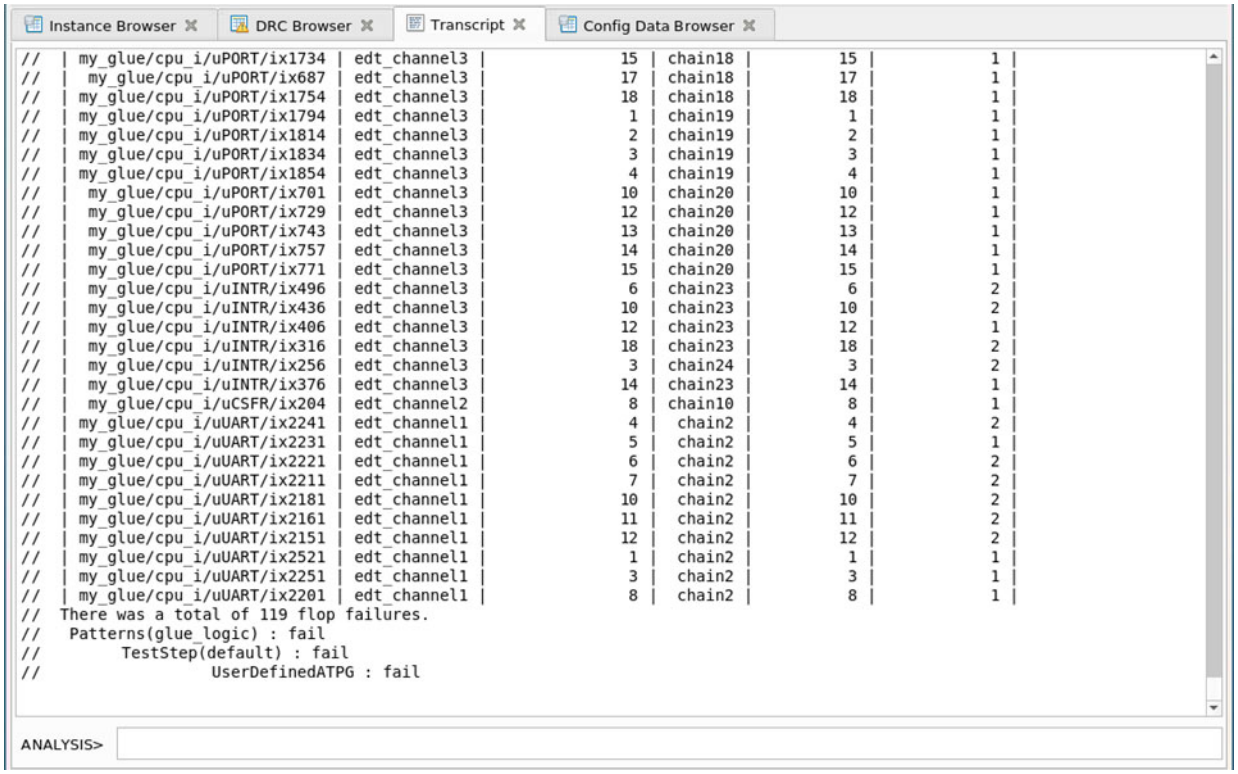
    b. Double-click the failing_flops_summary property in the property table pane on the right.

    c. A small Set Property Value window opens. Choose “scan_cells” and click the **Apply** button.

    d. Select a test, Patterns(glue_logic) or Patterns(all_cores), in the left pane.

    e. Click the **Process/Execute** button to run the test again.

f. View the results in the **Transcript** tab.



13. Perform iterative diagnosis on the Patterns(all_cores) test to refine the diagnosis results.

    a. In the **Config Data Browser** tab, select ATPG under CharacterizationDebugOptions in the “Config data wrappers” pane on the left.

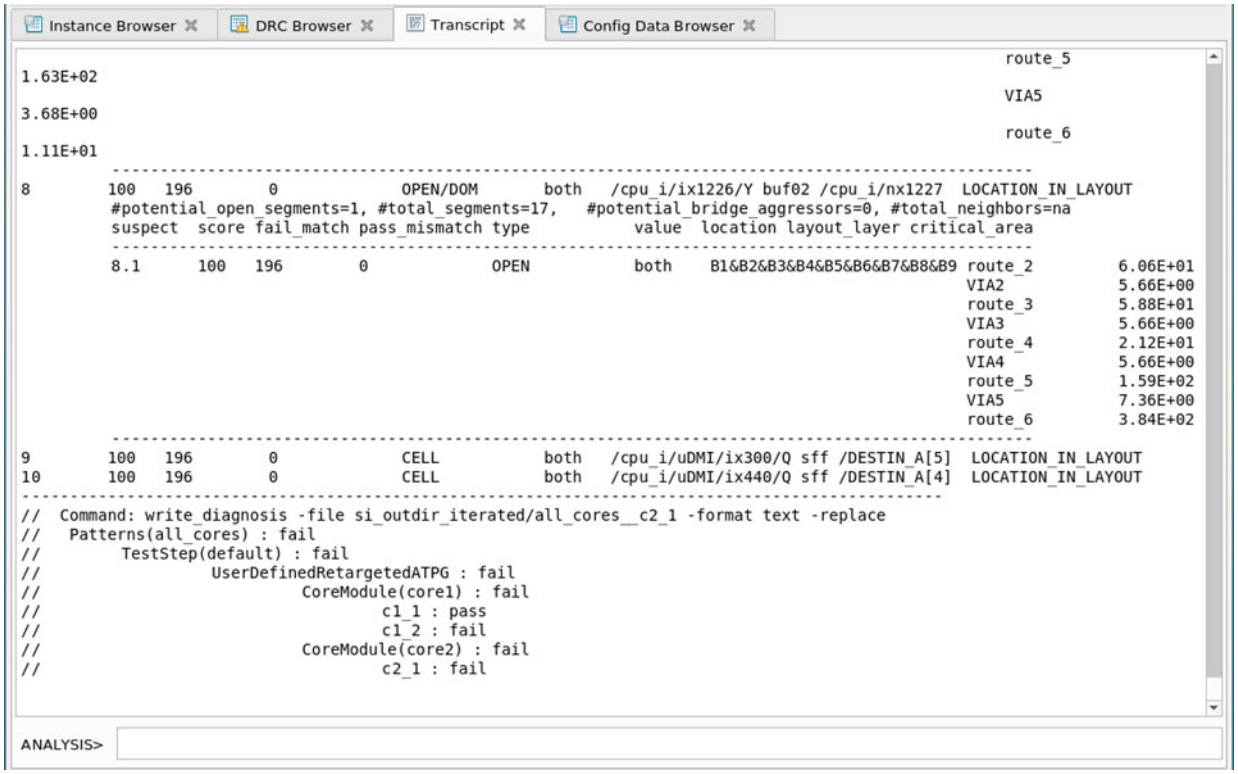
    b. Double-click the diagnosis_result property in the property table pane on the right.

    c. A small Set Property Value window opens. Choose “failing_nets_iterative_diagnosis” and click **Apply**.

    d. Select the Patterns(all_cores) test in the left pane.

    e. Click the **Process/Execute** button to run the test again.

f. View the results in the **Transcript** tab.



View the iterative diagnosis results by running the "report_silicon_insight_result -atpg_datalog" command. When you compare the new diagnosis results against the initial diagnosis results, you see that number of suspects after iterative diagnosis is reduced by roughly 50%.

For more information, refer to "Iterative Diagnosis" in the *Tessent Diagnosis User's Manual*.

Related Topics

CDP Result Introspection

CDP Result Structure

Chapter 14
MemoryBIST Offline ATE Diagnosis and Validation Tutorial

This tutorial is intended to teach you how to generate a CDP from Tessent Shell for a design that contains an ESOE MemoryBIST controller, use the patterns from the CDP to collect failures on the ATE, and use the process_memory_failures utility to convert the raw failures to a memory bitmap.

This tutorial serves dual purposes:

- Shows the test engineer how to perform data collection on the ATE.
- Shows the DFT engineer how to verify the CDP for a given device before first silicon.

The provided TSDB is for a DUT with one Tessent MemoryBist controller and one LogicBist controller. This tutorial also uses a previously created Patterns Specification that is configured to collect a limited number of memory failures.

Note

This tutorial uses Tessent SiliconInsight in SimDUT mode for the data collection task because it does not have access to an ATE. In a real-world context, you would use your ATE to collect your data, not SimDUT. As described in "Performing Diagnosis From an ATE Test Program in Offline ATE Mode," you implement your data collection loops in your test program using the test program language.

Access the Tutorial

The data you need to perform the tutorial is located in your release tree.

Access the tutorial as follows:

1. Create a work directory and change to that directory:

   mkdir work_dir

   cd work_dir

2. In your work directory, untar the TSDB using the following command:

   tar xvfz <installed release tree>/share/SiliconInsight/Tutorial/tutorial1.tgz

3. Change directory to the tutorial1 directory.

   cd tutorial1

Diagnosing ESOE Tessent MemoryBIST Failures in Offline Mode

The tutorial uses the Tessent SiliconInsight GUI to demonstrate how to generate the CDP.

This tutorial instructs you in performing the following:

- Start Tessent Shell in the correct context and load all the necessary files.
- Enable diagnosis for the MemoryBIST controller(s).
- Generate and examine the CDP.
- Find patterns of interest for data collection on the ATE.
- Implement data collection loops in the test program, and collect and save failure data in JSON format.
- Generate a bitmap file of the failures.
- Exit Tessent SiliconInsight.

Prerequisites

- You have accessed the tutorial1 directory.
- This tutorial does not require LogicBIST although the patterns specification contains a LogicBIST Patterns wrapper. To prevent an error from the tool stating that it requires a LogicBIST license to generate the CDP, for this tutorial only, delete the LogicBIST Patterns wrapper prior to generating the CDP.

Procedure

1. Start Tessent Shell:

   ```
   % tessent -shell -log tsi_create_cdp.log -replace
   ```

2. Clean up the previous CDP:

   SETUP> system rm -fr top.cdp

3. Set the Tessent SiliconInsight context:

   SETUP> set_context patterns -silicon_insight

4. (Optional) If you want the memory bitmap report to include the physical X-Y coordinates, create a layout database:

   **SETUP> create_layout MyLayoutDB -def layout/top.def **
   -leflist { layout/SYNC_1R1W_16x8.lef layout/tech.lef } -replace

5. Open the TSDBs. This design only has one TSDB:

```
SETUP> open_tsdb top.tsdb
SETUP> read_design top -design_id rtl -view interface
SETUP> set_current_design top
```

6. Generate the default setup specification:

```
SETUP> create_silicon_insight_setup_specification
```

   By default, the tool creates a setup specification named SiliconInsightSetupSpec.cfg.

7. Load the previously created pattern specification named “my_patspec.cfg”.

```
SETUP> read_config_data my_patspec.cfg
```

   You can save the patterns specification to a file using the write_config_data command and load it in subsequent sessions using the read_config_data command.

8. Specify to generate the patterns in STIL format:

```
SETUP> set_config_value \
    PatternsSpecification(top,rtl,manufacturing)/manufacturing_patterns_formats stil
```

9. (Optional) If you want the memory bitmap report to include the physical X-Y coordinates, set up the appropriate Characterization Debug Options properties:

```
SETUP> set_config_value /PatternsSpecification(top,rtl,manufacturing)/ \
    CharacterizationDebugOptions/MemoryBist/layout_database MyLayoutDB
SETUP> set dirWrap [add_config_element -in_wrapper \
    /PatternsSpecification(top,rtl,manufacturing)/CharacterizationDebugOptions/ \
    MemoryBist VendorXYMapDirectories]
SETUP> add_config_element -in_wrapper $dirWrap vendor_data -type property
```

10. (Optional) If you want to generate a layout marker file (using the *.lay* extension) from the reported physical X-Y coordinates, set up the appropriate CharacterizationDebugOptions property. You can view the layout marker file in the Calibre DRC viewer; refer to the *Calibre DESIGNrev Layout Viewer User’s Manual* for more information.

```
SETUP> set_config_value /PatternsSpecification(top,rtl,manufacturing)/ \
    CharacterizationDebugOptions/MemoryBist/generate_marker_file on
```

11. Start Tessent SiliconInsight:

```
SETUP> start_silicon_insight -gui
```

    The GUI displays as shown in Figure 12-1.

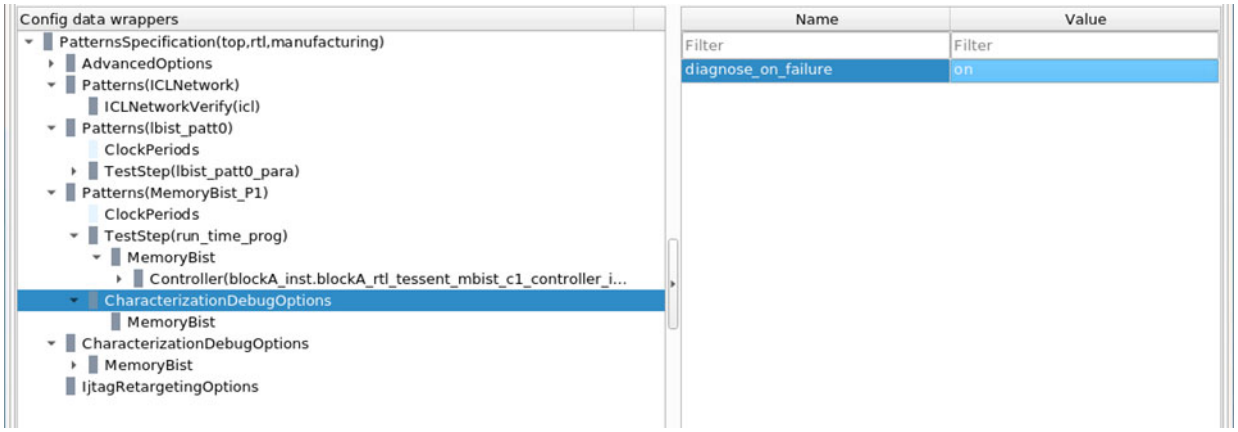
    In the **Config Data Browser** tab, each Patterns wrapper under PatternsSpecification is a test in the “Config data wrappers” pane on the left. Selecting an individual MemoryBist controller saves test time in designs with many MemoryBist controllers. You can click any of these wrappers and then click the **Process/Execute** ( ) button to run it. The result of the test displays in the **Transcript** tab. The icons in the “Config data wrappers” pane change color depending on the result of the test: green check for pass and red exclamation for fail.

12. Enable diagnosis for MemoryBIST as follows:

    a. In the **Config Data Browser** tab, select the CharacterizationDebugOptions wrapper under Patterns(MemoryBist_P1) in the left-hand pane. Click the Hide unspecified box if needed to activate (☑ Hide unspecified) or deactivate (☐ Hide unspecified) unspecified wrapper hiding.

    b. Double-click click the diagnosis_on_failure property in the property table pane on the right. See "Characterization Debug Options Display in the GUI" on page 172 for more information.

    c. Choose "on" and click **Apply**.



    d. Select the MemoryBist wrapper of the CharacterizationDebugOptions wrapper of Patterns(MemoryBist_P1).

    The failure count limit is set to 10. You can change this value as needed.

    You can also enable diagnosis for the MemoryBIST controller using the following command:

    ```
    SETUP> set_config_value \
        PatternsSpecification(top,rtl,manufacturing)/Patterns(MemoryBist_P1)/ \
        TestStep(run_time_prog)/MemoryBist/ \
        Controller(blockA_inst.blockA_rtl_tessent_mbist_c1_controller_inst)/ \
        CharacterizationDebugOptions/diagnose_on_failure  on
    ```

13. Generate the CDP by selecting PatternsSpecification in the "Config data wrappers" pane, and then click the **Process/Execute** button.

    The CDP named *top.cdp* is created in the current working directory.

    You can also generate the CDP by using the following command:

    ```
    SETUP> process_patterns_specification
    ```

14. Exit the session and exit Tessent Shell.

```
SETUP> stop_silicon_insight
SETUP> exit
```

15. Examine the CDP. The directory hierarchy looks as follows:

```
top.cdp/
   Design/
   Tests/
      ICLNetwork/
      lbist_patt0/
      MemoryBist_P1/
         summary.json
         PatternNameMap.tcl
         main_patterns/
            MemoryBist_P1.stil
            MemoryBist_P1.var_map
         run_time_prog/
            MemoryBist/
               MemoryBist_P1_1.stil
               MemoryBist_P1_1.var_map
               MemoryBist_P1_2.stil
               MemoryBist_P1_2.var_map
               MemoryBist_P1_3.stil
               MemoryBist_P1_3.var_map
         upgrade.tcl
         StaticParams.tcl
         DynamicParams.tcl
```

The Tests directory contains all of the tests that were defined in the PatternsSpecification. The MemoryBIST test (MemoryBist_P1) contains a directory called main_patterns, which contains the main (user-defined test) pattern for the test. It may also contain additional diagnostic patterns located under the corresponding pattern set (run_time_prog) and IP type (MemoryBist). In this case the diagnostic patterns are MemoryBist_P1_1.stil, MemoryBist_P1_2.stil, and MemoryBist_P1_3.stil.

16. Examine the unique pattern names in the summary.json file. You use these names to create data collection procedures to collect raw failure data, which you then save in JSON format.

    For each test containing diagnosis-enabled MemoryBist, the summary.json file contains the information describing the purpose of the ESOE diagnostic patterns and the memories that are tested in each controller step.

Here is the content of the summary file for purposes of this tutorial:

```
{"run_time_prog": {"CDPMemoryDiagnosticAvailability": [{           //The pattern set
   "TestMethod": "auto_increment_stop_on_error",
   "PatternDistribution": [{
    "InitPatternName": "MemoryBist_P1_1.stil",
    "ExecutePatternName": "MemoryBist_P1_2.stil",
    "IncrementPatternName": "MemoryBist_P1_3.stil",
      "TestedControllers": [{
         "ControllerName": "blockA_inst.blockA_rtl_tessent_mbist_c1_controller_inst",
         "ControllerDesignName": "blockA_inst\/blockA_rtl_tessent_mbist_c1_controller_inst",
         "failure_limit_scope":"controller",
         "failure_limit":10,
         "TestedControllerSteps": [{
            "Step": 0,
            "TestedMemories": [                    //The tested memories
             {
              "CollarName": "M1",
              "MemoryInstancePath": "blockA_inst.mem1",
              "MemoryDesignName": "blockA_inst\/mem1",
              "LibraryModel": "SYNC_1R1W_16x8"
             },
             {
              "CollarName": "M2",
              "MemoryInstancePath": "blockA_inst.mem2",
              "MemoryDesignName": "blockA_inst\/mem2",
              "LibraryModel": "SYNC_1R1W_16x8"
             }
            ]
           }]
         }]
       }]
   }]}}
```

17. Start Tessent Shell:

    ```
    % tessent -shell -log tsi_collect_failure.log -replace
    ```

18. In Tessent Shell, source the following dofile, which consists of a Tcl procedure that is called after data collecting to save the raw failure data in JSON format.

    SETUP> dofile save_raw_fails_to_json.tcl

    Note

    The *save_raw_fails_to_json.tcl* script is intended for use in single-tap scenarios. To save the data after scanning from multiple taps, refer to the *save_raw_fails_to_json_multi_tap.tcl* script included in the tutorial.

19. Run the following code in its entirety, one line at a time. The code uses a data collection loop to collect the raw failure data and calls the json_failure_log_version_2 procedure to save the collected data in JSON format in the failure log file.

Note

For data collection in this tutorial, you are using SimDUT with a temporary CDP named tmp_ate.cdp, so you must first prepare SimDUT, which is described in "Simulating Desktop, ATE, and ATPG Behavior" on page 223. On an ATE, you would proceed directly to the bolded instructions below, implementing your data collection loops in your test program using the test program language.

```
set_context pattern -silicon_insight

file mkdir tutorial1_outdir
set_tsdb_output_directory tutorial1_outdir

# Open TSDB
open_tsdb top.tsdb

read_config_data my_patspec.cfg

# Load setup specification with SimDUT mode and using cdp tmp_ate.cdp
# The setup specification named si.setup uses a SimDUT startup dofile
# name (simdut_startup.dofile) to automatically load the design files
# and run the simulator.

read_config_data si.setup
set_config_value \
   /SiliconInsightSetupSpecification/Sid(simdut)/cdp_directory  tmp_ate.cdp

########################################
# Start SiliconInsight and prepare SimDUT
########################################
read_design top -design_id rtl -view interface
set_current_design top
start_silicon_insight

add_cdp_test mem_go_nogo -pattern \
   top.cdp/Tests/MemoryBist_P1/main_patterns/MemoryBist_P1.stil

# First make sure it passes
execute_cdp_test mem_go_nogo

# Inject a fault
add_simdut_fault -signal SIMDUT_TB.DUT_inst.blockA_inst.mem1.Q[0] -fault 1

# Make sure it fails
execute_cdp_test mem_go_nogo

#####################################

# Now continue the offline ATE process

#####################################

# You can inspect the collect_ate_fails.tcl file to view the data collection loop
# the tool uses
dofile collect_ate_fails.tcl

stop_silicon_insight
```

Note

The *collect_ate_fails.tcl* script is intended for use in single-tap scenarios. To view the data collection loop after scanning from multiple taps, refer to the *collect_ate_fails_multi_tap.tcl* script included in the tutorial.

Note

After running a test, you can introspect the results located in the CDPResult partition, which stores the results of the last performed test in memory.

The resulting file, *ate.flog*, from this Tessent Shell session contains the following:

```
{
  "Version": 2,
  "Type": "ESOE",
  "TrackingInfo": {"Design": "top.cdp"},
  "FailDataType": "RawSTIL",
  "RawSTIL": [{
      "UserTestName": "MemoryBist_P1",
      "Failures": [{
          "TesterDiagPatternName": "MemoryBist_P1_2",
          "Pin": "tdo",
          "StartingSOEID": 1,
          "FailureFormat": "FailingCycles",
          "FailData": [
                  "352,417,463,470",
                  "352,417,462,470",
                  "352,417,462,463,470",
                  "352,417,461,470",
                  "352,417,461,463,470",
                  "352,417,461,462,470",
                  "352,417,461,462,463,470",
                  "352,417,469",
                  "352,417,468,470",
                  "352,417,463,468,470"
          ]
        }]
    }]
}
```

20. Generate the failure bitmap file using the process_memory_failures utility.

Note

The process_memory_failures utility is not a Tessent Shell command. You must run it outside of Tessent Shell. The process_memory_failures utility is installed in the *bin* directory of your Tessent software installation.

Assuming the name for the raw failure JSON file you generated previously is named *ate.flog*, the bitmap file is named *ate.flog.bitmap*.

```
process_memory_failures --cdp top.cdp  --faildata ate.flog --outDir ./
```

The “-outDir ./”switch tells the tool to put the bitmap file in the current directory. The tool stores the bitmap data in JSON format in a file called ate.faillog.bitmap.

Results

The process_memory_failures utility generates failure bitmap files. In the bitmap files, the Header fields describe how to interpret the bitmap data. The following bitmap file contains the

physical X-Y coordinates generated by the optional portions of this procedure, which are in green text. If you did not include the optional coordinates, the PhysicalX/Y/X_/Y_ headers and the corresponding coordinate information in the failure rows would not appear in this file.

```
{
  "CreatedWith": "top.cdp generated by 2020.4-snapshot_2020.09.21_03.00",
  "TrackingInfo": {"Design": "top.cdp"},
  "UserTestName": "MemoryBist_P1",
  "Header": "FailureIndex ControllerDesignName Algorithm
     PhaseOrInstruction MemoryDesignName TestPort PhysicalBank PhysicalRow
      PhysicalColumn BitPosition PhysicalExpected PhysicalX PhysicalY
      PhysicalX_ PhysicalY_ LogicalAddress LogicalExpected",
  "Failures": [
    "0 blockA_inst/blockA_rtl_tessent_mbist_c1_controller_inst
       SMARCHCHKBVCD 2_W0R0RX blockA_inst/mem1 0 0x0 0x0 0x0 0 0
       114.481 126.407 114.565 126.565 0x0 0",
    "1 blockA_inst/blockA_rtl_tessent_mbist_c1_controller_inst
       SMARCHCHKBVCD 2_W0R0RX blockA_inst/mem1 0 0x0 0x0 0x1 0 0
       114.481 126.085 114.565 125.927 0x1 0",
    "2 blockA_inst/blockA_rtl_tessent_mbist_c1_controller_inst
       SMARCHCHKBVCD 2_W0R0RX blockA_inst/mem1 0 0x0 0x0 0x2 0 0
       114.397 126.407 114.313 126.565 0x2 0",
    "3 blockA_inst/blockA_rtl_tessent_mbist_c1_controller_inst
       SMARCHCHKBVCD 2_W0R0RX blockA_inst/mem1 0 0x0 0x0 0x3 0 0
       114.397 126.085 114.313 125.927 0x3 0",
    "4 blockA_inst/blockA_rtl_tessent_mbist_c1_controller_inst
       SMARCHCHKBVCD 2_W0R0RX blockA_inst/mem1 0 0x0 0x0 0x4 0 0
       125.857 126.407 125.773 126.565 0x4 0",
    "5 blockA_inst/blockA_rtl_tessent_mbist_c1_controller_inst
       SMARCHCHKBVCD 2_W0R0RX blockA_inst/mem1 0 0x0 0x0 0x5 0 0
       125.857 126.085 125.773 125.927 0x5 0",
    "6 blockA_inst/blockA_rtl_tessent_mbist_c1_controller_inst
       SMARCHCHKBVCD 2_W0R0RX blockA_inst/mem1 0 0x0 0x0 0x6 0 0
       125.941 126.407 126.025 126.565 0x6 0",
    "7 blockA_inst/blockA_rtl_tessent_mbist_c1_controller_inst
       SMARCHCHKBVCD 2_W0R0RX blockA_inst/mem1 0 0x0 0x0 0x7 0 0
       125.941 126.085 126.025 125.927 0x7 0",
    "8* blockA_inst/blockA_rtl_tessent_mbist_c1_controller_inst
       SMARCHCHKBVCD 3_W0R0RX blockA_inst/mem1 0 0x0 0x0 0x0 0 0
       114.481 126.407 114.565 126.565 0x0 0",
    "9* blockA_inst/blockA_rtl_tessent_mbist_c1_controller_inst
       SMARCHCHKBVCD 3_W0R0RX blockA_inst/mem1 0 0x0 0x0 0x1 0 0
       114.481 126.085 114.565 125.927 0x1 0"
  ],
  "FailuresInfo": [
    "8 Info: Possible serial datapath short and/or write enable short if
       the algorithm is only failing on Phase 3",
    "9 Info: Possible serial datapath short and/or write enable short if
       the algorithm is only failing on Phase 3"
  ]
}
```

The process_memory_failures utility also generates layout marker files for each failing memory, with the *.lay* extension, that you can view later in the Calibre DRC viewer. Refer to the *Calibre DESIGNrev Layout Viewer User's Manual* for more information.

The tool also stores the layout marker files for each failing memory (in this case, *ate.flog_blockA_inst_mem1.lay*) if you previously set the generate_marker_file property.

Related Topics

CDP Result Introspection

CDP Result Structure

Collecting and Converting Failure Data

Converting Raw JSON Data Files to Bitmap Files

Diagnostic Bitmap File

Chapter 15
Interactive IJTAG Tutorial

This tutorial is intended to teach you how to use the interactive IJTAG functionality in Tessent SiliconInsight to test and diagnose your device at the PDL level. Using interactive IJTAG, you can debug your device faster than manually generating patterns and running them on the tester.

For the purposes of this tutorial, you use Tessent SiliconInsight Desktop in SimDUT mode because this mode does not require an adaptor. The only requirement is a signed-off TSDB for the design.

The provided TSDB is for a DUT with one Tessent MemoryBist controller and one LogicBist controller.

Accessing the Tutorial

The data you need to perform the tutorial is located in your release tree.

Access the tutorial as follows:

1. Create a work directory and change to that directory:

   mkdir work_dir

   cd work_dir

2. In your work directory, untar the TSDB using the following command:

   tar xvfz <installed release tree>/share/SiliconInsight/Tutorial/tutorial1.tgz

3. Change directory to the tutorial1 directory.

   cd tutorial1

Diagnosing and Debugging a Device in Interactive IJTAG Mode

The tutorial uses the Tessent SiliconInsight GUI to demonstrate how to diagnose and debug a device at the PDL level.

This tutorial instructs you in performing the following:

- Start Tessent Shell in the correct context and load all the necessary files.
- Generate tests.
- Enable interactive IJTAG.
- Run tests to confirm that the PDL instructions pass.
- Inject and debug failures.
- Exit Tessent SiliconInsight.

Prerequisites

- You have accessed the tutorial1 directory.

Procedure

1. Start Tessent Shell:

   ```
   % tessent -shell
   ```

2. Set the Tessent SiliconInsight context:

   SETUP>set_context patterns -silicon_insight

3. Open the TSDBs. This design only has one TSDB:

   SETUP>open_tsdb top.tsdb
   SETUP>read_design top -design_id rtl -view interface
   SETUP>set_current_design top

4. Read the setup specification for this design:

   SETUP>read_config_data si.setup

5. Load the previously created pattern specification named my_patspec.cfg.

   SETUP>read_config_data my_patspec.cfg

   You can save the patterns specification to a file using the write_config_data command and load it in subsequent sessions using the read_config_data command.

6. Start Tessent SiliconInsight:

   SETUP>start_silicon_insight

7. Generate the tests from the pattern specification:

   SETUP>process_patterns_specification

8. View the available tests:

   SETUP>get_cdp_test_list

9. Run the following ICLNetwork test to ensure the IJTAG network in the DUT is stable:

   ```
   SETUP>execute_cdp_test ICLNetwork -collect_data_type_list variable
   ```

   Note

   After running a test, you can introspect the results located in the CDPResult partition, which stores the results of the last performed test in memory.

10. Enable interactive IJTAG so that you can debug the device at the PDL level:

    ```
    SETUP>set_silicon_insight_option -interactive_ijtag on
    ```

11. Run the following PDL instructions.

    ```
    ANALYSIS>iReset
    ANALYSIS>iWrite top_rtl_tessent_tap_main_inst.instruction[3:0] 0b1000
    ANALYSIS>iRead  top_rtl_tessent_tap_main_inst.instruction[3:0] 0b0001
    ANALYSIS>iApply -end_in_pause
    ```

    You can generate all tests as a series of PDL commands ending with command iApply. After iApply runs, the tool packages all commands up to and including iApply, generates a test pattern, and applies it on the tester (ATE, Desktop or SimDUT). The tool displays the test results in the form of passing tests or failing registers.

    This sequence of commands tests the instruction register of the TAP, writes a “1000” pattern to the register, and scans it as is without doing a capture (as indicated by iApply -end_in_pause). The commands pass.

12. Inject a stuck-at-1 fault in the TAP instruction register as follows:

    ```
    ANALYSIS>add_simdut_fault -signal \
    SIMDUT_TB.DUT_inst.top_rtl_tessent_tap_main_inst.instruction[0] -fault 1
    ```

13. Run the PDL commands from Step 11. This time, the last iApply command fails as follows:

    ```
    //  +-----------------------------------------------------+-------------------+
    //  | ICL Object                                          | Expected          |
    //  |                                                     | Actual            |
    //  |                                                     | Fail Mask         |
    //  +-----------------------------------------------------+-------------------+
    //  | top_rtl_tessent_tap_main_inst.instruction           | 0001              |
    //  |                                                     | 1111              |
    //  |                                                     |  ^^^_             |
    //  +-----------------------------------------------------+-------------------+
    ```

    The failure indicates that one of the cells in the instruction register is stuck at 1.

14. Run the following command to receive the total number of failing cycles:

    ```
    ANALYSIS>iGetStatus

    3
    ```

In addition, the following command returns the actual value of the register (that is, 1111):

```
ANALYSIS> iGetReadData top_rtl_tessent_tap_main_inst.instruction
```

The following command returns the miscompares mask (that is, 1110):

```
ANALYSIS> iGetMiscompares top_rtl_tessent_tap_main_inst.instruction
```

15. Exit the Tessent SiliconInsight session:

```
ANALYSIS>stop_silicon_insight
```

Related Topics

CDP Result Introspection

CDP Result Structure

Chapter 16
Interactive Desktop Standard Repair Flow Tutorial

This tutorial illustrates the basics of setting up and running the standard repair flow using Tessent SiliconInsight.

This tutorial uses Tessent SiliconInsight Desktop in SimDUT mode, because this mode does not require an adaptor. The only requirement is a signed-off chip-level TSDB for the design.

The TSDB provided with this tutorial is for a DUT with one Tessent MemoryBist controller and one MemoryBisr controller. This tutorial also uses a previously-created design and the custom flow feature described in "Custom Test Flows" on page 297.

Accessing the Repair Flow Tutorial

The data required for the tutorial is located in an archive file in your release tree.

Procedure

1. Create a working directory for the tutorial and change to that directory:

```
% mkdir work_dir
% cd work_dir
```

2. In your working directory, extract the archive with the following command:

```
% tar xvfz <install_dir>/share/SiliconInsight/Tutorial/tutorialRepair.tgz
```

3. Change your directory to the *tutorialRepair* directory:

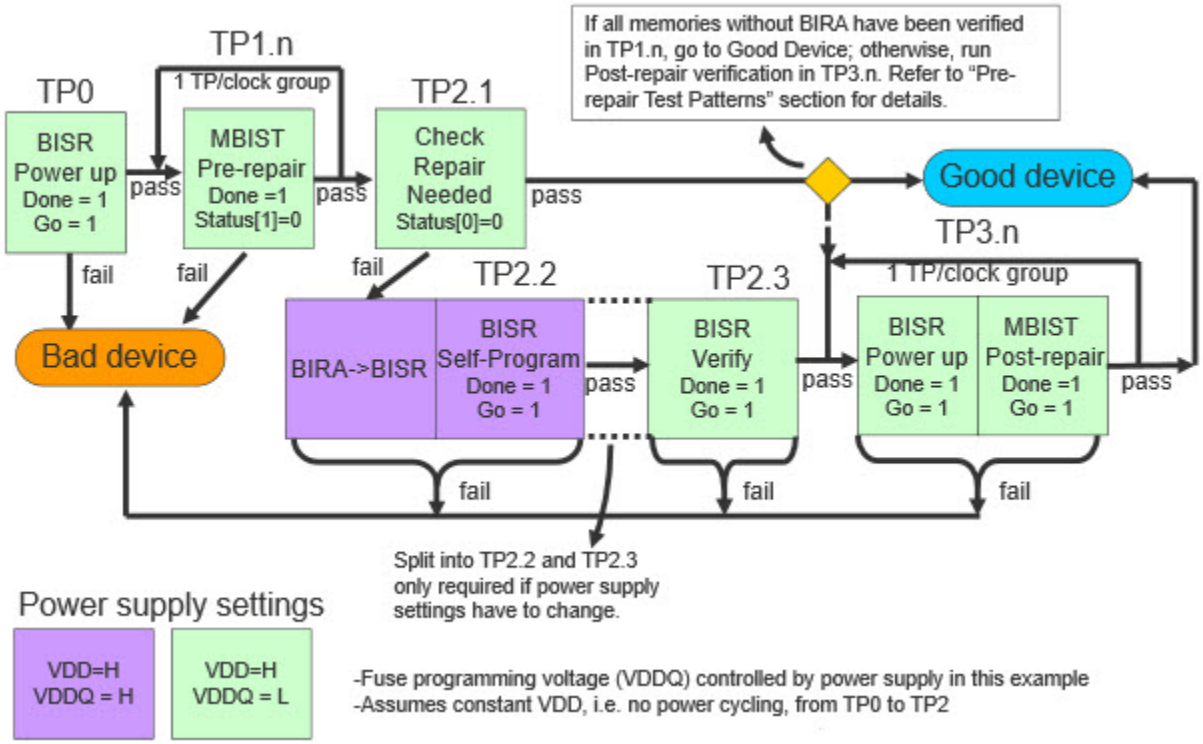
```
% cd tutorialRepair
```

Performing Soft Repair in the GUI With Tessent SiliconInsight

Follow the steps in this tutorial to use the Tessent SiliconInsight GUI to perform soft repair using the custom flow in a sample database.

This tutorial reproduces the repair flow shown in Figure 16-1. No split is required between TP2.2 and TP2.3, because no fuse programming is required when working with soft repair.

Figure 16-1. BISR Manufacturing Flow Example



Refer to the "Generating Your Manufacturing Test Patterns" topic in the "Top-Level Verification and Pattern Generation" chapter of the *Tessent™ MemoryBIST User's Manual For Use With Tessent Shell* for more details.

This tutorial procedure instructs you in performing the following tasks:

- Start Tessent Shell in the correct context and load all necessary files.
- Generate the standard soft repair flow.
- Repair a faulty repairable memory.
- Identify when you are incapable of repairing a faulty nonrepairable memory.
- Exit Tessent SiliconInsight cleanly after memory repair.

Prerequisites

- You have accessed the tutorialRepair directory, as described in "Accessing the Repair Flow Tutorial" on page 385.
- You have a licensed installation of the Siemens ModelSim or Questa Verilog simulator, with **vsim** and **vlog** specified in your environment path.

Procedure

1. Start Tessent Shell:

   ```
   % tessent -shell
   ```

2. Set the Tessent SiliconInsight context:

   ```
   SETUP> set_context patterns -silicon_insight
   SETUP> set_silicon_insight_option -ignore_uncontacted_pin_data on
   ```

3. Open the TSDBs.

   This tutorial design has only one TSDB:

   ```
   SETUP> open_tsdb top.tsdb
   SETUP> read_design top -design_id rtl -view interface
   SETUP> set_current_design top
   ```

4. Set your output directory:

   ```
   SETUP> file mkdir tutorialSoftRepair_outdir
   SETUP> set_tsdb_output_directory tutorialSoftRepair_outdir
   ```

5. Load the SiliconInsight setup specification:

   ```
   SETUP> read_config_data si.setup
   ```

6. Generate the default manufacturing patterns specification, which also includes the destructive BISR patterns that are required to create the repair flow:

   ```
   SETUP> set ::env(TESSENT_SI_ALLOW_DESTRUCTIVE_BISR) 1
   SETUP> create_patterns_specification manufacturing
   ```

7. Create the standard soft repair flow with the name “SoftRepairFlow”:

   Note

   You can check the contents of the PatternsSpecification when creating the software flow, as well as compare this with the previous figure that shows this repair flow, with one of the following commands to inspect the PatternsSpecification before and after using the create_memory_repair_flow command:

   - report_config_data /PatternsSpecification(top,rtl,manufacturing) [log view]
   - write_config_data {before.cfg|after.cfg} -wrappers / PatternsSpecification(top,rtl,manufacturing) [file view]

   ```
   SETUP> create_memory_repair_flow -name SoftRepairFlow -repair_type soft
   ```

   Tip

   The resulting PatternsSpecification refers to the file *<install_dir>/share/SiliconInsight/Utilities/CustomFlows/my_repair.tcl*. This is the Tcl source code that this tutorial automatically invokes to control the repair flow. It may help you to understand the processes in this tutorial to familiarize yourself with this code.

8. Disable splitting in non-custom flow patterns; this prevents unnecessary error messages during pattern generation. Do this by setting the appropriate MemoryBist properties:

   ```
   SETUP> set_config_value /PatternsSpecification(top,rtl,manufacturing)/
   Patterns(MemoryBisr_AutonomousMode)/TestStep(TP2_2_CaptureBiraRotate)/
   AdvancedOptions/split_patterns_file Off

   SETUP> set_config_value /PatternsSpecification(top,rtl,manufacturing)/
   Patterns(MemoryBisr_AutonomousMode)/TestStep(TP2_3_VerifyFuseBox)/
   AdvancedOptions/split_patterns_file Off
   ```

9. Save the PatternsSpecification as *manufacturing_patspec_soft_repair_flow.cfg* after adding the Repair flow:

   ```
   SETUP> write_config_data manufacturing_patspec_soft_repair_flow.cfg -wrappers \
     /PatternsSpecification(top,rtl,manufacturing) -replace
   ```

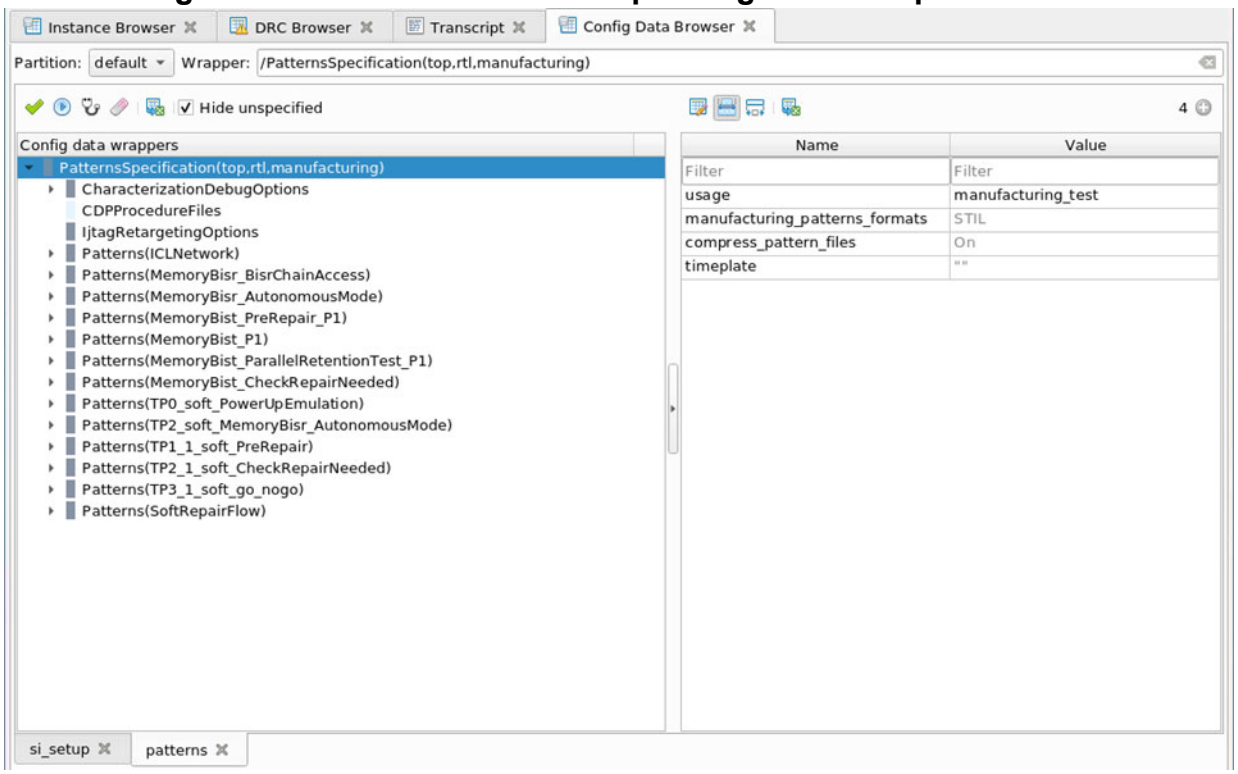
   Save the patterns specification to a file with the write_config_data command at any time during your session. You can then load it in a subsequent session with the read_config_data command.

10. Start SiliconInsight in GUI mode:

    ```
    SETUP> start_silicon_insight -gui
    ```

    In the Config data wrappers pane of the **Config Data Browser** tab, click the right arrow (▸) next to the PatternsSpecification wrapper to expand it.

Figure 16-2. Initial View After Expanding PatternsSpecification



Each Patterns wrapper under the PatternsSpecification is a test. In this instance, the test named "Patterns(SoftRepairFlow)" controls running the tests named "Patterns(TP*_soft_*)"; however, you can run them separately if you know the proper sequence for running them. To run any of the tests, click the corresponding wrapper in the configuration area and then click **Process/Execute** ( ).

The **Transcript** tab contains the results of the test. The icons next to the wrapper names indicate the results of the test: a green check indicates a passing test and a red exclamation point indicates a failing test.

Expand each Patterns wrapper further by clicking the right arrow ( ) to examine the tests contained in that wrapper.

11. In the **Transcript** tab, generate all patterns for the PatternsSpecification:

    SETUP> process_patterns_specification manufacturing

12. Ensure the validity of the IJTAG network by running the ICLNetwork test. Click "Patterns(ICLNetwork)" and then click **Process/Execute**.

    Tip

    After you run a test, you can introspect the results in the CDPResult partition ("report_config_data /CDPResult"), which stores the results of the most recently performed test in memory.

13. Check that the memory test without faults passes. Click "Patterns(MemoryBist_P1") and then click **Process/Execute**.

14. Try the repair flow on a good device. It should pass. Click "Patterns(SoftRepairFlow)" and then click **Process/Execute**.

    The **Transcript** tab displays the results of the test:

```
// Test 'SoftRepairFlow' passed.
// Diagnostic Result: CDPResult/TestExecutionResult/DiagnosticResult/PatternSets(0)
// PatternsSpect Path: Patterns(SoftRepairFlow/TestStep(TP2_1_soft_checkRepairNeeded)
// Part is good
//  Patterns(SoftRepairFlow) : pass
```

15. Working in simulation mode enables you to inject failures into the design to further test it. Use the following commands to inject repairable failures into the memory:

    SETUP> add_simdut_fault -signal simdut_fi.mem6_fi[0] -fault 1
    SETUP> add_simdut_fault -signal simdut_fi.mem6_fi[1] -fault 1
    SETUP> add_simdut_fault -signal simdut_fi.mem6_fi[2] -fault 1

    The affected memory (mem6) in the membist controller has two spare rows and one spare column. These simulate stuck-at faults and trigger full spare usage, as shown in the

following table. (This assumes that you encounter first failures in this order with the test algorithm.)

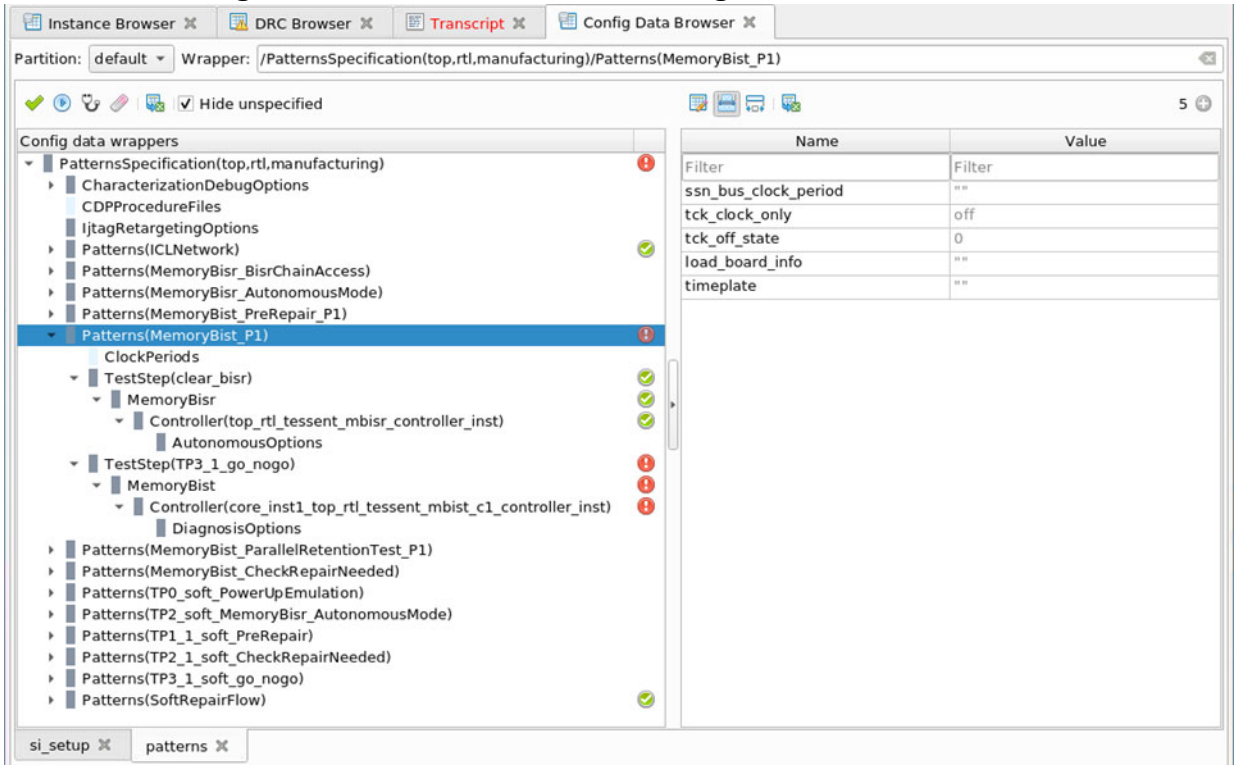
Table 16-1. Spare Usage Triggering by Injected Faults

Fault	RowAdd	ColAdd	I/O	Spare usage
mem6_fi[0]	1	0	3	Column spare (because single I/O always prioritizes vertical spare usage)
mem6_fi[1]	1	1	3	Row spare (because no more column spares are available)
mem6_fi[2]	2	1	3,4	Row spare (multiple I/Os force a row spare even if a column spare had been available)

16. Check that the memory test now fails. Click “Patterns(MemoryBist_P1”) and then click **Process/Execute**.

    The GUI displays failures indicated by red exclamation points next to the wrapper names. Expand the wrappers fully to see all the failure information.

Figure 16-3. Results After Running Tests With Failures



Additionally, the **Transcript** tab displays the failing registers for each test:

```
//  Test 'MemoryBist_P1' failed.
//  Variable failures and unmapped failures of pattern 'MemoryBist_P1.stil' :
//
//  PatternSet     Variable                                                    Pin           Expected
//                                                                                             Actual
//  -------------  ----------------------------------------------------------  ---  ----------------
//  TP3_1_go_nogo  core_inst1_top_rtl_tessent_mbist_c1_controller_inst.MBISTPG_GO(1)  tdo                b1
//                                                                                             b0
//  -------------  ----------------------------------------------------------  ---  ----------------
//  TP3_1_go_nogo           core_inst1_blockA_clka_i1_mem6_interface_inst.GO_ID_REG  tdo  b0000000000000000
//                                                                                     b0000000000011000
//  -------------  ----------------------------------------------------------  ---  ----------------
//   Patterns(MemoryBist_P1) : fail
//        TestStep(clear_bisr) : pass
//                 MemoryBisr : pass
//                          Controller(top_rtl_tessent_mbisr_controller_inst) : pass
//        TestStep(TP3_1_go_nogo) : fail
//                 MemoryBist : fail
//                          Controller(core_inst1_top_rtl_tessent_mbist_c1_controller_inst) : fail
```

17. Try the repair flow on a repairable device. It should also pass. Click Patterns(SoftRepairFlow) and then click **Process/Execute**.

    The **Transcript** tab displays the results of the test:

```
// Test 'SoftRepairFlow' passed.
// Diagnostic Result: CDPResult/TestExecutionResult/DiagnosticResult/PatternSets(0)
// PatternsSpec Path: Patterns(SoftRepairFlow)/TestStep(TP2_1_soft_CheckRepairNeeded)
// Part is repairable and the memories on the following controllers are being repaired:
//       core_inst1_top_rtl_tessent_mbist_c1_controller_inst
// Diagnostic Result: CDPResult/TestExecutionResult/DiagnosticResult/PatternSets(1)
// PatternsSpec Path: Patterns(SoftRepairFlow)/TestStep(TP3_1_soft_go_nogo)
// Part was successfully repaired
//  Patterns(SoftRepairFlow) : pass
```

18. Check that running the memory test without resetting soft repair passes. Click "Patterns(TP3_1_soft_go_nogo)" and then click **Process/Execute**.
19. Check that memory test after resetting soft repair fails. Click "Patterns(MemoryBist_p1)" and then click **Process/Execute**.
20. Inject another failure into the memory. This failure will make the memory non-repairable:

    SETUP> add_simdut_fault -signal simdut_fi.mem6_fi[3] -fault 1

    This simulates a stuck-at fault as shown in the following table. To repair this fault would require one more spare row, but no additional spare rows are available.

Table 16-2. Spare Usage Triggering by Non-Repairable Injected Fault

Fault	RowAdd	ColAdd	I/O	Spare usage
mem6_fi[0]	3	0	3,4	Row spare (multiple I/O forces row spare, but no more spares are available)

21. Try the repair flow on this non-repairable device. It should fail. Click "Patterns(SoftRepairFlow)" and then click **Process/Execute**.

The Tessent Shell console displays the results of the test:

```
// Test 'SoftRepairFlow' failed.
// Diagnostic Result: CDPResult/TestExecutionResult/DiagnosticResult/PatternSets(1)
// PatternsSpec Path: Patterns(SoftRepairFlow)/TestStep(TP1_1_soft_PreRepair)
// Pre-Repair failed (TP1_1_soft_PreRepair): Part is not repairable.
//      Pattern set PreRepair failed on the following controllers:
//        Controller core_inst1_top_rtl_tessent_mbist_c1_controller_inst failed on Go
//  Patterns(SoftRepairFlow) : fail
```

22. Check that the memory test continues to fail without resetting soft repair, because the tool did not find a soft repair solution. Click “Patterns(TP3_1_soft_go_nogo)”, and then click **Process/Execute**.
23. Exit the Tessent SiliconInsight session:

    SETUP> stop_silicon_insight
24. Exit Tessent Shell:

    SETUP> exit

Appendix A
ATE-Connect Preparation for ATE Vendors

To provide integration between Tessent SiliconInsight and an ATE through a TCP-IP connection, ATE vendors and engineers using custom in-house testers must enhance their environments to support a series of APIs that Tessent SiliconInsight uses to send instructions to the tester and receive results from the tester.

The minimum command set for ATPG is:

- ATESetupParameters
- ATEDefinePinmap
- ATELoadPattern or ATEStreamPattern
- ATEExecutePatternGetFailures
- ATEExecutePatternGoNo

The minimum command set for MemoryBIST (ESOE, SAR) is:

- ATESetupParameters
- ATEDefinePinmap
- ATELoadPattern or ATEStreamPattern
- ATEExecutePatternGetFailures or ATEExecutePatternsGetSVFFailures
- ATEExecutePatternGoNo
- ATEDefineTAPPins (for SVF patterns)

The minimum command set for LogicBIST is the minimum command set for MemoryBIST plus:

- ATEPatchPattern or ATEPatchSVFPattern

In addition, you can use the optional ATELog and ATECommand commands.

Example Test Program

The following Tcl example shows a sample test program using the commands described in this appendix.

```
variable tp_out ""
variable num_clients 0
variable verbosity_level 0
variable initial_control_string ""
variable loaded_patterns [list]

proc get_cmd_from_si { si_chan cmd_name_p cmd_args_p } {
  upvar $cmd_name_p cmd_name
  upvar $cmd_args_p cmd_args
  variable initial_control_string

  set data ""
  set cmd_name ""
  set cmd_args ""
  set full_str [gets $si_chan]
  if { $full_str eq "" } {
    return
  }
  if { $initial_control_string ne "" } {
    set ctrl_str_len [string length $initial_control_string]
    if { [string range $full_str 0 [expr $ctrl_str_len - 1]] eq   \
      $initial_control_string } {
    log_msg "Detected initial control string '$initial_control_string'."
    # Skip the initial control sequence
    set full_str [string range $full_str $ctrl_str_len end]
    set initial_control_string ""
    }
  }
  set dash [string first - $full_str]
  if { $dash < 0 } {
    return -code error "Couldn't find '-' in string '$full_str'.  \
      Expected a string of format <string length>-<string>"
  }
```

```
  set data_len [string range $full_str 0 [expr $dash - 1]]
  set data [string range $full_str [expr $dash + 1] end]
# log_msg "get_cmd_from_si: data_len=$data_len data='$data'"
  set actual_data_len [string length $data]
  if { $actual_data_len > $data_len } {
# log_msg "Warning: String '$data' is longer than noted length $data_len"
# log_msg "\tDiscarding the additional data."
    set data [string range $data 0 [expr $data_len-1]]
  }
  set space [string first " " $data]
  set cmd_name [string range $data 0 [expr $space - 1]]
  set cmd_args [string range $data [expr $space + 1] end]

  return
}

proc send_result_to_si { si_chan msg } {
  set msg_len [string length $msg]
  set full_res "${msg_len}-${msg}"
# log_msg "send_result_to_si: Returning to si: '$full_res'"
  puts $si_chan $full_res
  flush $si_chan
}

proc log_msg msg {
  variable tp_out
  variable verbosity_level

  if { $tp_out ne "" } {
  puts $tp_out $msg
  flush $tp_out
  }
  if { $verbosity_level > 0 } {
  puts $msg
  }
return
}proc process_silicon_insight_command { ate_cmd cmd_args } {
  variable tp_out
  variable loaded_patterns
```

```
# log_msg "process_silicon_insight_command: ate_cmd = '$ate_cmd'"
  switch $ate_cmd {
  ATEExitTestProgram {
    log_msg "Test program exiting based on request from SI."
    If { $tp_out ne "" } {
    close $tp_out
    }
    set result "pass"
  }
  ATESetupParameters {
    log_msg "Got Parameters from SI: '$cmd_args'"
    set params "TP:sample_tcl_tp;"
    set result "pass $params"
    log_msg "Returning params to SI: '$params'"
  }
  ATELoadPattern {
    set pattern_path $cmd_args
    log_msg "ATE pattern '$pattern_path' loaded successfully."
    lappend loaded_patterns [file tail $pattern_path]
    set result "pass"
  }  ATEDefineTAPPins {
    log_msg "Tester successfully setup TAP pins '$cmd_args'."
    set result "pass"
  }

  ATEDefinePinmap {
    log_msg "Tester successfully setup pinmap pins '$cmd_args'."
    set result "pass"
  }
  ATEStreamPattern {
    # Don't want to use tcl command 'split' here to avoid splitting the
    # pattern content too.
    set space1 [string first " " $cmd_args]
    set space2 [string first " " $cmd_args [expr $space1 + 1]]
    set pattern_name [string range $cmd_args 0 [expr $space1 - 1]]
    set eol_marker [string range $cmd_args [expr $space1 + 1]   \
      [expr $space2 - 1]]
    set pattern_content [string range $cmd_args [expr $space2 + 1] end]
    regsub -all $eol_marker $pattern_content "\n"   \
      updated_pattern_content; # replace all eol_makers with '\n'
    # save the pattern content in a file
    set out [open ${pattern_name}.tp w]
    puts -nonewline $out $updated_pattern_content
```

```
  close $out
  set result "pass"
  lappend loaded_patterns $pattern_name
  log_msg "ATE pattern '$pattern_name' was saved in file
  ${pattern_name}.tp."
}
ATEExecutePatternGoNoGo {
  set pattern_name $cmd_args
  log_msg "ATEExecutePatternGoNoGo: pattern_name = '$pattern_name'"
  if { [lsearch $loaded_patterns $pattern_name] < 0 } {
  set result "error pattern '$pattern_name' doesn't exist."
  } else {
  switch $pattern_name {
    MemoryBist_P1.svf {
    # fake miscompares (based on tutorial1 test case)
    set result "fail"
    }
    ICLNetwork.svf {
    # fake error of missing pattern:
    set result "error pattern '$pattern_name' doesn't exist."
    }
    default {
    # for all the rest of the tests report pass
    set result "pass"
    }
  }
  }
}


ATEExecutePatternGetSVFFailures {
  set pattern_name $cmd_args
  log_msg "ATEExecutePatternGetSVFFailures:    \
    pattern_name = '$pattern_name'"
  if { [lsearch $loaded_patterns $pattern_name]< 0 } {
  set result "error pattern '$pattern_name' doesn't exist."
  } else {
  switch $pattern_name {
    MemoryBist_P1.svf {
    # fake miscompares (based on tutorial1 test case.)
    set result "fail 21@4@tdo 23@18@tdo"
    }
      BadResponseFromATE.svf {
        set result "fail"
      }
    default {
    # for all the rest of the tests report pass
    set result "pass"
    }
  }
  }
}
```

```
ATEExecutePatternGetFailures {
  set pattern_name $cmd_args
  log_msg "ATEExecutePatternGetSVFFailures:    \
    pattern_name = '$pattern_name'"
  if { [lsearch $loaded_patterns $pattern_name] < 0 } {
  set result "error pattern '$pattern_name' doesn't exist."
  } else {
  switch $pattern_name {
    MemoryBist_P1.stil {
    # fake miscompares (based on tutorial1 test case.)
    set result "fail 21@4@tdo 23@18@tdo"
    }
      BadResponseFromATE.stil {
        set result "fail"
      }
    default {
    # for all the rest of the tests report pass
    set result "pass"
    }
  }
  }
}

ATECommand {
  set cmd_elems [split $cmd_args]
  set ate_specific_cmd [lindex $cmd_elems 0]
  switch $ate_specific_cmd {
  SetVoltage {
    log_msg "Successfully performed ATE command '$cmd_args'"
    set result "pass"
  }
      ReadVoltage {
        log_msg "Successfully performed ATE command '$cmd_args'"
    set result "pass 1.234V"
  }
  Ping {
    log_msg "Successfully performed ATE command '$cmd_args'"
    set result "pass Alive & Kicking"
  }
  HelloWorld1 {
    log_msg "Successfully running Hello World command '$cmd_args'"
    set result "pass Succeeded in running Hello World 1"
  }
  HelloWorld2 {
    log_msg "Failed in running Hello World command '$cmd_args'"
    set result "fail Failed in running Hello World 2"
  }
  HelloWorld3 {
    log_msg "Error in running Hello World command '$cmd_args'"
    set result "error Error in running Hello World 3"
  }
```

```
    default {
      log_msg "problems running ATE command '$ate_specific_cmd'"
      set result "error Unknown ATE command '$ate_specific_cmd'"
    }
    }
  }

  ATELog {
    log_msg $cmd_args
    set result "pass"
  }
  default {
    set result "pass"
  }
  }
  return $result
}


# Make a proc to receive communications
proc handle_silicon_insight_commands client_chan {
  variable num_clients
  variable tp_out


  get_cmd_from_si $client_chan cmd_name cmd_args

# log_msg "Got command from SI: '$cmd_name $cmd_args'"
  if { $cmd_name eq "" } {
  log_msg "SiliconInsight exited. Remove this SI client from the   \
    list....."
  fileevent $client_chan readable ""
  incr num_clients -1
  log_msg "Total clients = $num_clients"
  if { $num_clients <= 0 } {
    log_msg "Test program exiting because no more clients are connected."
    if { $tp_out ne "" } {
    close $tp_out
    }
    exit
  }
  return
  }
  set res [process_silicon_insight_command $cmd_name $cmd_args]
# log_msg "Returning following result to SI: '$res'"
  send_result_to_si $client_chan $res
}
```

```
# Make a proc to accept connections
proc accept_silicon_insight_connection {chan addr port} {
  variable num_clients

  fconfigure $chan -buffering none
  log_msg "Accepted client from host $addr on port $port. Listening   \
    to it on chan '$chan'...." ;# Receive a string
  fileevent $chan readable [list handle_silicon_insight_commands $chan]  \
    ;# set up to handle incoming data when necessary
  incr num_clients
  log_msg "Total clients = $num_clients"
  return
}


array set ARGS {
  -port_number 0
  -log_file ""
  -verbosity 0
  -initial_control_string ""
}
log_msg "got argv = '$argv'"


array set ARGS $argv

set ate_port_num $ARGS(-port_number)
set tp_log_file $ARGS(-log_file)
set verbosity_level $ARGS(-verbosity)
set initial_control_string $ARGS(-initial_control_string)

if { $tp_log_file ne "" } {
  set tp_out [open $tp_log_file w+]
}

# Create a server socket to accept SI connection
socket -server accept_silicon_insight_connection $ate_port_num;# Create a
server socket

log_msg "Waiting on requests from SI on port $ate_port_num ...."
vwait forever;# Enter the event loop socket -server
```

API Command Line Interface

Implement the API commands described in this section to support ATE-Connect on specific ATE platforms. These commands are transparent and non-relevant for ATE-Connect users.

The API commands must be sent through the TCP-IP ports from Tessent SiliconInsight to the tester. The following command format is used to transmit instructions from Tessent SiliconInsight:

```
<payload_length>-<command_and_arguments>
```

The payload length is the length of the command in ASCII plus all of its arguments, excluding the hyphen. The command and arguments are the API commands and arguments described in this appendix. In the following example, “25” is the payload length, “ATELog” is the API command, and “Hello Test Program” is the argument.

```
25-ATELog Hello Test Program
```

Results transmitted from the ATE use the following general format:

```
<payload_length>-<pass | fail | error> [<result_string>]
```

For example:

```
4-pass
23-fail 18@4@tdo 20@18@tdo
38-error this is an example error message
```

Note

If the ATE response string to the SiliconInsight tool contains newline (“\n”) characters, they must be replaced by two at symbols (“@@”) before the ATE sends the response.

ATECommand

Context: Optional

Runs the specified ATE command.

Usage

ATECommand ***ATE_command_and_arguments***;

Arguments

- ***ATE_command_and_arguments***;

  Required string that specifies an ATE command and its arguments.

Description

Specify a supported ATE command with arguments to access and control tester resources, such as clocks, power supplies, and PMUs. The transmitted command does not include end-of-line characters, such as "\n".

Upon receiving the command, the ATE:

- Runs the command on the ATE and returns the result of the command.
- Returns "pass" if execution completed successfully, "fail" with an optional message or result if it failed, or "error" with an appropriate message if the command could not run.

Examples

```
ATECommand SetVoltage PWR4 2.0V

ATECommand MeasureCurrent Pin56
```

ATEDefinePinmap

Context: Required for ATPG, MemoryBIST (ESOE, SAR) and LogicBIST

Maps design pin names to ATE pin names on the tester.

Usage

ATEDefinePinmap ***pin_name***:***ate_name***;***pin_name***:***ate_name***;...;

Arguments

- ***pin_name*:**

  Required string that specifies a design pin name.

- ***ate_name*;**

  Required string that specifies an ATE pin name that maps to a specified design pin name.

Description

Specify design pin names and associated ATE pin names in pairs separated by colons (:). You can specify multiple pairs by separating them with semicolons (;).

When you are configuring the tester for MemoryBIST or LogicBIST usage, all TAP pins declared as contacted with ATEDefineTAPPins must be mapped to ATE pin names.

Upon receiving the command, the ATE:

- Verifies that the pins match the data specified in the ATE pin table.
- Returns “pass” if there is a 100% match.
- If not a pass, returns “fail” with an optional message if there are mismatches.

Examples

```
ATEDefinePinmap
my_trst:ate_pin1;my_tms:ate_pin2;my_tdi:ate_pin2;my_tdo:ate_pin3;my_tck:a
te_pin4;additional_design_pin1:ate_pin5;

ATEDefinePinmap
my_pin15:ate_pin3;my_pin18:ate_pin8;my_pin21:ate_pin6;my_pin32:ate_pin9;
```

ATEDefineTAPPins

Context: Required for MemoryBIST (ESOE, SAR) and LogicBIST for SVF patterns

Defines which design pins are TAP pins.

Usage

ATEDefineTAPPins **TRST**:{***design_pin*** | **-**};**TMS**:***design_pin***;
TDI:***design_pin***;**TDO**:***design_pin***;**TCK**:***design_pin***;

Arguments

- **TRST:**{***design_pin*** | **-**}**;**

  Required contact designation for the TRST pin. Specify a design pin name or a dash if the TRST pin is uncontacted.

- **TMS:*design_pin*;**

  Required design pin contact for the TMS pin. This pin must be contacted.

- **TDI:*design_pin*;**

  Required design pin contact for the TDI pin. This pin must be contacted.

- **TDO:*design_pin*;**

  Required design pin contact for the TDO pin. This pin must be contacted.

- **TCK:*design_pin*;**

  Required design pin contact for the TCK pin. This pin must be contacted.

Description

Upon receiving the command, the ATE:

- Translates the SVF pattern from Tessent SiliconInsight.
- Returns “pass.”

Examples

```
ATEDefineTAPPins
TRST:my_trst;TMS:my_tms;TDI:my_tdi;TDO:my_tdo;TCK:my_tck;

ATEDefineTAPPins TRST:-;TMS:my_tms;TDI:my_tdi;TDO:my_tdo;TCK:my_tck;
```

ATEExecutePatternGetFailures

Context: Required for ATPG, MemoryBIST (ESOE, SAR) and LogicBIST for STIL patterns

Runs the loaded ATPG pattern and collects all failures.

Usage

ATEExecutePatternGetFailures ***pattern_name***;

Arguments

- ***pattern_name*;**

  A required string that specifies an ATPG pattern.

Description

Upon receiving the command, the ATE:

- Runs the loaded pattern and collects all failures.
- Returns “pass” if execution completed successfully, “fail” if it failed, or “error” with an appropriate message.

When a pattern fails, the failure response includes a list of failures in the format:

```
<failing_cycle>[@<pin_name>]
```

For example:

```
10-fail 18 20
38-fail 33@so_1 33@so_2 100@so_1 105@so_2
```

The failing cycle corresponds to the failing tester cycle (which starts at 1) relative to the STIL pattern cycles. The optional pin name specifies the pin where the mis-compare was observed; when not included in the response, it is assumed to be the TDO pin for the design.

Examples

```
ATEExecutePatternGetFailures ICLNetwork.stil;

ATEExecutePatternGetFailures MemoryBist_P1.stil;
```

ATEExecutePatternsGetSVFFailures

Context: Required for MemoryBIST (ESOE, SAR) and LogicBIST for SVF patterns

Runs the loaded ATPG pattern and collects all failures.

Usage

ATEExecutePatternGetSVFFailures ***pattern_name***;

Arguments

- ***pattern_name*;**

  A required string that specifies an ATPG pattern.

Description

Upon receiving the command, the ATE:

- Runs the loaded pattern and collects all failures.
- Returns “pass” if execution completed successfully, “fail” if it failed, or “error” with an appropriate message.

If a pattern fails, the failure response includes a list of failures in the format:

```
<svf_command_number>@<bit_offset>[@<pin_name>]
```

For example:

```
23-fail 18@4@tdo 20@18@tdo
27-fail 18@4 10@18 15@18 16@18
```

The SVF command number is the index number of the SVF command in the SVF pattern. The bit offset corresponds to the failing bit offset for the SVF command, where bit offset 0 equals the least significant bit, which is the first out of TDO. The optional pin name specifies the pin where the mis-compare was observed; when not included in the response, it is assumed to be the TDO pin.

Examples

```
ATEExecutePatternGetSVFFailures ICLNetwork.svf;

ATEExecutePatternGetSVFFailures MemoryBist_P1.svf;
```

ATEExecutePatternGoNoGo

Context: Required for ATPG, MemoryBIST (ESOE, SAR) and LogicBIST

Runs the loaded ATPG pattern in go-nogo mode.

Usage

ATEExecutePatternGoNoGo ***pattern_name***;

Arguments

- ***pattern_name*;**

  A required string that specifies an ATPG pattern.

Description

The command returns pass or fail results only, which enables the tester to stop at the first failure, minimizing tester time.

Upon receiving the command, the ATE does the following:

- Runs the loaded pattern in Go-NoGo mode, stopping at the first fail.
- Returns “pass” if execution completed successfully, “fail” if it failed, or “error” with an appropriate message.

Examples

```
ATEExecutePatternGoNoGo ICLNetwork.svf;

ATEExecutePatternGoNoGo MemoryBist_P1.svf;
```

ATELoadPattern

Context: Required for ATPG, MemoryBIST (ESOE, SAR) and LogicBIST if not specifying ATEStreamPattern

Loads an ATPG pattern stored in the CDP.

Usage

ATELoadPattern ***pattern_path***;

Arguments

- ***pattern_path*;**

  A required string that specifies the pathname of a ATPG pattern.

Description

Specify a pattern path relative to the same CDP to be specified in the setup specification as described in “Ate(tester_name)” on page 130. Configure the ATE to correctly interpret the forward slashes (/) in the path. For example, Windows-based ATEs need to interpret forward slashes as backward slashes (\).

Upon receiving the command, the ATE does the following:

- Converts the pattern to ATE ASCII format.
- Compiles the ATE ASCII pattern to binary ATE pattern format.
- Loads the pattern onto the tester.
- Returns “pass” or “error” with appropriate message.

Examples

```
ATELoadPattern top_ate.cdp/Tests/ICLNetwork/main_patterns/ICLNetwork.svf;

ATELoadPattern top_ate.cdp/Tests/MemoryBist_P1/main_patterns/
MemoryBist_P1.svf;
```

ATELog

Context: Optional

Returns the specified message string.

Usage

ATELog ***message_content***;

Arguments

- ***message_content*;**

  A required string that specifies a message that the ATE returns.

Description

Specify a message string that you want to display. The displayed message does not include end-of-line characters, such as "\n".

Upon receiving the command, the ATE does the following:

- Logs the message in the ATE console.
- Returns "pass" or "error" with appropriate message.

Examples

```
ATELog Hello ATE;
```

ATEPatchPattern

Context: Required for LogicBIST

Modifies a loaded STIL pattern by patching in new bit values at specified pins.

Usage

ATEPatchPattern ***pattern_name*** ***pin_name*** ***new_value*** ***vectors_per_bit*** ***start_address***;

Arguments

- ***pattern_name***

  A required string that specifies an ATPG pattern that was previously loaded by either ATELoadPattern or ATEStreamPattern.

- ***pin_name***

  A required string that specifies the name of a pin in the pinmap whose value you want to modify.

- ***new_value***

  A required binary string that specifies the new value to patch into the pin specified by *pin_name*.

- ***vectors_per_bit***

  A required integer string that specifies the number of vectors to be modified for each bit in the new value.

- ***start_address***;

  A required integer string that specifies the first vector address for the specified pin to patch with the new value. The specified integer indicates how many positions in the pin's bit string to offset. Offsetting begins with the most significant bit in the string.

Description

When performing LogicBIST diagnosis and debug on an ATE, the linear_search, binary_search, and hybrid_search algorithms require you to patch in new bit values on pins. This command provides the patching capability for STIL pattern files.

Upon receiving the command, the ATE:

- Modifies the loaded pattern within ATE vector memory at the specified pin with the specified new value starting at the specified start address. For each bit in the new value, the tool modifies as many vectors as specified by the *vectors_per_bit* parameter.
- Returns "pass" if execution completed successfully or "error" with an appropriate message.

Examples

```
ATEPatchPattern ICLNetwork.stil tdo 00001111 4 1580;
```

```
ATEPatchPattern MemoryBist_P1.stil tdi 1010 1 1692;
```

ATEPatchSVFPattern

Context: Required for LogicBIST

Modifies a loaded SVF pattern by patching in new bit values at specified pins.

Usage

ATEPatchPattern ***pattern_name pin_name new_value svf_command_id start_bit_offset***;

Arguments

- ***pattern_name***

  A required string that specifies an ATPG pattern that was previously loaded by either ATELoadPattern or ATEStreamPattern.

- ***pin_name***

  A required string that specifies the name of a pin in the pinmap whose value you want to modify.

- ***new_value***

  A required binary string that specifies the new value (translated to hex) to patch into the pin specified by *pin_name*. See example that follows.

- ***svf_command_id***

  A required string that specifies the index number of an SVF command in the SVF pattern that includes the pin you want modify.

- ***start_bit_offset***;

  A required integer string that specifies which bit in the bit string for the specified pin to patch with the new value. The specified integer indicates how many positions in the pin's bit string to offset. Offsetting begins with the most significant bit in the string. See the example that follows.

Description

When performing LogicBIST diagnosis and debug on an ATE, the linear_search, binary_search, and hybrid_search algorithms require you to patch in new bit values on pins. This command provides the patching capability for SVF pattern files.

Upon receiving the command, the ATE:

- For the specified SVF command, modifies the loaded pattern within ATE vector memory at the specified pin by patching in the specified new value at the specified bit offset position. The tool modifies exactly one vector for each bit in the new value.
- Returns "pass" if execution completed successfully or "error" with an appropriate message.

Examples

```
ATEPatchSVFPattern ICLNetwork.svf tdo 00001111 4 15;

ATEPatchSVFPattern MemoryBist_P1.svf tdi 1010 13 3;
```

Suppose you have the following SVF pattern named pat.svf:

```
.
.
!svf_cmd 8
SDR 34
    TDI(280800000)
    TDO(000000000)
    MASK(3FFFF0000);
.
.
```

The following command:

```
ATEPatchSVFPattern pat.svf tdo 11 8 16
```

Causes the tool to patch the pattern as follows:

```
.
.
!svf_cmd 8
SDR 34
    TDI(280800000)
    TDO(00003 0000)          // Value "3" replaces "0"
    MASK(3FFFF0000);
.
.
```

The pins values in SVF are hex values, but the command options you specify are binary. Each hex character corresponds to four binary bits. Therefore, in binary, the TDO pin translates to:

```
00000000000000000000000000000000
```

Starting from the least significant bit, the tool offsets the patched value by 16 places and updates the bit in that position with the new value:

```
00000000000000000011 0000000000000000
```

Binary value “11” corresponds to hex value “3”. Converting the binary value back to hex results in:

```
00003 0000
```

In the following command, the binary value “10” corresponds to hex value “2”:

```
ATEPatchSVFPattern pat.svf tdo 10 8 16
```

Causes the tool to patch the pattern as follows:

```
.
.
!svf_cmd 8
SDR 34
    TDI(280800000)
    TDO(00001O000)          // Value "1" replaces "0"
    MASK(3FFFF0000);
.
.
```

ATESetupParameters

Context: Required for ATPG, MemoryBIST (ESOE, SAR) and LogicBIST

Lists the initial parameters and their values to be used and displayed by the ATE.

Usage

ATESetupParameters ***parameter***:***value***;***parameter***:***value***;...;

Arguments

- ***parameter*:**

  A required string that specifies a setup parameter for the ATE.

- ***value*;**

  A required string that specifies the value of the associated setup parameter.

Description

Specify setup parameters and associated values in pairs separated by colons (:). You can specify multiple pairs by separating them with semicolons (;).

Upon receiving the command, the ATE:

- Processes the parameter-value pairs.
- Returns “pass” and an optional list of parameter-value pairs that Tessent SiliconInsight parses and displays. A returned parameter-value pair may be “PreloadedPatternFiles:<*pattern_name*>[+<*pattern_name*>...]”. This informs Tessent SiliconInsight that the ATE has already loaded the listed patterns.
- IReturns “error” with an applicable message if setup does not pass.

Examples

```
ATESetupParameters Protocol:svf;SiliconInsightVersion:2017.3-
prerelease;ATECommandsVersion:1;

ATESetupParameters ...;PreloadedPatternFiles:pat1.stil+pat2.stil+pat3.stil;...
```

ATEStreamPattern

Context: Required for ATPG, MemoryBIST (ESOE, SAR) and LogicBIST if not specifying ATELoadPattern

Streams an ATPG pattern stored in the CDP.

Usage

ATEStreamPattern ***pattern_path eol_marker pattern_content***;

Arguments

- ***pattern_name***

  A required string that specifies the name of a ATPG pattern.

- ***eol_marker***

  A required string that specifies the end-of-line (eol) marker.

- ***pattern_content*;**

  A required string that specifies the pattern data.

Description

Specify a pattern name followed by an eol marker, such as "@@", and then the pattern content. Include spaces before and after the eol marker.

Configure the ATE to replace all instances of the eol marker in the pattern content with the appropriate eol marker, such as "\n".

Upon receiving the command, the ATE:

- Processes the pattern content to the appropriate format and loads it onto the tester.
- Returns "pass" or "error" with appropriate message.

Examples

```
ATEStreamPattern ICLNetwork.svf @@ !TESSENT_PRAGMA icl_checksum
d82a0b4f1c20aca8ff8d00801e2090eb_1@@!svf_cmd 0@@PIOMAP ( IN RST IN reset
IN edt_clock );@@

ATEStreamPattern MemoryBist_P1.svf &&@@ !TESSENT_PRAGMA icl_checksum
d82a0b4f1c20aca8ff8d00801e2090eb_1&&@@!svf_cmd 0&&@@PIOMAP ( IN RST IN
reset IN edt_clock );&&@@
```

Appendix B
Validating the SiliconInsight Desktop Environment

The following sections provide information about validating the SiliconInsight Desktop environment and troubleshooting problems you may encounter during the validation process.

Pre-Silicon Validation

Once the setup data is ready for use with your USB adaptor and board, you can validate the data by running the adaptor in simulation mode. This mode generates patterns targeting the specified adaptor and applies them to the simulator through the procedural language interface (PLI) and SimDUT. This validation process is highly recommended.

For information about getting the setup data ready, refer to the chapters "Prepare the Design Under Test" on page 75 and "Support for ATPG (Non-TSDB Flow)" on page 157.

In particular, for using ATPG with SiliconInsight with an OpalKelly adaptor, running the first several (for example, 20) patterns in adaptor simulation mode enables you to confirm that complicated timing sets and bidirectional switching are properly accommodated to the adaptor.

Validation With SimDUT

Running SimDUT before running adaptor simulation mode is recommended. This helps isolate any issues with the original pattern from issues specific to the conditions and constraints used to retarget the pattern to the USB adaptor.

Validation With Adaptor Simulation Mode

The example in this section uses ATPG and illustrates how to modify the original setup data configured for online usage to run the adaptor in simulation mode

PatternsSpecification Wrapper

The ATPG test requires mode constraints when using the SiliconInsight Desktop (SID) than on ATE, because SID has a smaller tester pin count. This is generally the case regardless of whether you run online or in simulation mode. Because it is impractical to run a large number of simulated patterns, you must use a smaller version of the ATPG test pattern:

```
PatternsSpecification(cpu,SignOff,atpg) {
  usage : manufacturing_test; Patterns(at_speed) {
    TestStep(default) {
      UserDefinedATPG {
        scan_pattern_file : ./data/pat1_small.stil
        flat_model_file : ./data/cpu_edt.flat.gz;
        layout_database : ./data/cpu_top.ldb;
      }
    }
  }
}
```

SiliconInsightSetupSpecification Wrapper

The only difference in the SiliconInsightSetupSpecification wrapper between USB adaptor usage and adaptor simulation mode is the value of the operation_mode property, which must be set to "simulation" for simulation mode:

```
Sid(opalKellyXem6310) {
  cdp_directory : cpu_top.cdp;
  overwrite_cdp_on_startup : on;
  opalKellyXem6310 {
    // operation_mode : online;
    operation_mode : simulation;
    io_standard : 1.80;
    PinMap {
      P0 : Dxa_32;
      P1 : Dxa_31;
      P2 : Dxa_30;
      …
```

SimDUT.v File

The run_testbench_simulations command generates the SimDUT files, including *SimDUT.v*, in the *SIMDUT.simulation* directory. The system board configuration may require you to modify the Verilog file. For example, some DUT pins may be set on the board, or an extra circuit (such as FPGA) may be on the board, and the device initial sequence is applied through that circuit.

Dofile

The following dofile uses the assumption that the SimDUT files have been generated in the simdut_outdir directory:

```
set_context pattern -silicon_insight -design_id SignOff
set curr_si "Sid(opalKellyXem6310)"
set_simulation_library_sources -ext v vb \
  -y ../../../library/memory \
  -y ../../../library/verilog \
  -v ../../../library/adk.v
run_testbench_simulations -simulator_options "+nospecify" \
  -simdut_server -simdut_output_dir simdut_outdir \
  -store_simulation_waveforms on
read_config_data ./data/si.setup set_current_silicon_insight_setup \
  ${curr_si} read_config_data ./data/atpgpat.spec
start_silicon_insight -gui
```

The last line in the dofile starts the SiliconInsight GUI. Select a Patterns wrapper and click **Process/Execute** () to confirm whether the pattern passes. If it fails and you have specified the "-store_simulation_waveforms on" option (as in the preceding example), the tool writes a simulation dump file, such as *ysim.wlf*, into the *simulation_outdir* directory. Use this file to help you find the root cause of the failure.

SiliconInsight Desktop Online Environment Validation

The following sections provide troubleshooting to help you narrow down the cause of problems running the SiliconInsight Desktop online mode.

Bypass Shift-Only Patterns

Bypass shift-only patterns vary depending on the environment in which you use them. By bypassing the DUT, you can address the external test environment for SID.

The following describe bypass shift-only patterns for various environments:

- **Accessing the JTAG TAP interface for IJTAG test or BIST** — Running the JTAG BYPASS instruction sets the TAP to bypass mode. The TDI is connected to the TDO through the bypass register. Because the bypass register is a single-bit register, TDO outputs are delayed by one cycle from the TDI inputs. This enables you to create arbitrary patterns. Refer to the section “TAP Access Check With BYPASS Test” on page 428.
- **Accessing scan-in/scan-out pins for TestKompress** — In this case, there are several ways to create an arbitrary bypass shift-only pattern. One example is to generate a bypass chain pattern or patterns with one or more seeds. The following example dofile generates a bypass chain pattern with six seeds. (Six bypass chain tests with “111 …1”, “000 … 0”, “1010 … 10”, “0101 … 01”, “11001100 … 1100” and “00110011 … 0011” are generated in a single chain test pattern.)

```
set_context pattern -scan -design_id bypass
set_tsdb_output_directory ../tsdb_outdir
read_design cpu_top -design_id tk -verbose
read_cell_library ../library/adk.tcelllib read_cell_library ../
library/memory.lib
set_design_sources -format tcd_memory -Y ../library \
  -extension memlib
set_current_design cpu_top
set_current_mode bypass
add_core_instance -instance [get_name_list \
  [get_instance *tessent_edt*]] -parameter_value {edt_bypass on}
report_core_instance
add_clocks 0 shift_clock_p
add_input_constraints clock_mode_p -C1
add_input_constraints ijtag_reset -C1
add_input_constraints ijtag_sel -C0
set_procfile_name cpu_top.testproc
set_chain_test -sequence 1
set_chain_test -sequence 0 -append
set_chain_test -sequence 10 -append
set_chain_test -sequence 01 -append
set_chain_test -sequence 1100 -append
set_chain_test -sequence 0011 -append
set_system_mode analysis
report_scan_chains
report_scan_cells > scancells
create_pattern
write_patterns bypass_chain.stil -chain -stil -rep
```

When you use bypass shift-only patterns, output signals (from the TDO or scan-out) are controllable and become easy to expect. This, in turn, helps you determine the causes of the failures.

Troubleshooting sa0 and sa1 Failures

There are three primary causes for stuck-at-0 and stuck-at-1 faults.

Broken USB Adaptor

Although a broken USB adaptor is a rare event, it can cause sa0/sa1 failures. To check the adaptor, loop adaptor output pins back to input pins.

Note

Make sure to source the reference voltage according to your adaptor when you perform the loopback operation.

The following example output from a dofile shows the C0 output routed to the C2 input using a Singalyzer SHA40. In this case, “D” means “expect 0” and “U” means “expect 1”.

```
// command: set_context patterns -silicon_insight
// command: read_config_data si.setup
// command: set_current_silicon_insight_setup Sid(signalyzerSHA40)
// command: start_silicon_insight
// /SiliconInsightSetupSpecification is loaded in memory.
// Will use existing CDP cpu.cdp ...
// Note: Connected to sid_tester server.
// Note: Start sid_tester .......
// Note: Setting up SiliconInsight Desktop USB adaptor.
// Note: Setting up SVF tap pins.
// Note: Setting up pin maps.
// Note: Opening CDP at 'cpu.cdp'.
// Note: Connected to CDP socket server. 0
// command: execute_tester_command exec_vecs {C0 C2}
//    {0D 1U 0D 1U 0U 1D 0U 1D} {1 5} {1 6} {1 7}
// command: execute_tester_command exec_vecs {C0 C2}
//    {0D 1U 0D 1U 0D 1U 0D 1U}
// command: quit
// Note: sid_tester is shut down.
```

Since the first exec_vecs command injects failures at cycle 5 (0U: drive 0, expect 1), 6 (1D: drive 1, expect 0) and 7 (0U: drive 0, expect 1), it returns failures on pin 1 (C2).

Improper GND or Reference Voltage Setting

To detect a floating GND or incorrect reference voltage settings, you can measure known output levels from the DUT before investigating test mode setup issues (see the following section). This is an infrequent problem; it is more likely to have a weak GND connection leading to stability issues, which you can address while checking test mode setup issues. As with the broken USB adaptor case, you can measure DUT output values with the exec_vecs command through the execute_tester_command command.

Improper Test Mode or Device Initial Sequence Settings

Of the cases described in this topic, this is the most likely scenario. However, it is recommended to check for the other cases first to assist with this diagnosis.

If input pins of the USB adaptor running the bypass shift-only pattern give sa0 or sa1 results, carefully check the test control mode. It is possible that some of the following conditions may be true:

- The on-board jumper pins are incorrectly set.
- The on-board logic (such as FPGA) is not correctly controlled.

  You should physically check the signal levels having troubles instead of relying on the on-board logic.
- The validation may not have been performed in adaptor simulation mode.
- The design for adaptor simulation may not be identical to the DUT.

- The SiliconInsight setup may not be identical between simulated and actual adaptor.
- The initial sequence or test mode setting may not be applied identically.

  If the initial sequence or test mode setting is applied from adaptor pins, you must modify your *SimDUT.v* file to apply the extra sequence or setting.

Troubleshooting Pattern-Dependent Failures

In cases where pattern failures are not the result of stuck-at-0 or -1 faults, it is recommended to try various bypass shift-only patterns to determine the cause of the failures.

The following are some particular patterns that can help with diagnosis:

- Patterns with an isolated 1 (such as “00001000…0”)
- Patterns with an isolated 0 (such as “11110111…1”)
- Patters that toggle frequently (such as “010101…01”)
- Other toggling patterns (such as “00110011…0011” or “000111000111…000111”)

You can also use more random patterns to help with diagnosis.

If particular patterns fail such that values are not captured correctly from scan-out pins, there are two typical reasons.

Lower Adaptor Driver Slew Rate

The cabling/wiring may cause a lower slew rate. This results in errors in transferring values to DUT inputs. For the Opal Kelly XEM6310 adaptor, it may help to slow down the base frequency using the base_frequency property.

DUT Output Level Too Small For Adaptor Comparator

Although a USB adaptor may be capable of capturing high static output levels, it may not be able to correctly capture dynamic signals. This is particularly true for USB adaptors with fixed levels, such as Signalyzer adaptors. Even for USB adaptors with a reference voltage, capture levels could be marginal near the lowest reference voltage.

In this case, you can slow the signal frequency and adjust the reference voltage, which may produce better results.

Troubleshooting Fixed-Delay Failures

When you use a bypass shift-only pattern, captured signals may be delayed by a fixed number of cycles from the expected signal.

If the bypass shift-only patterns work correctly in validation with the adaptor simulation mode, and there is no shift timing issue on the DUT, the likely issue is that the design parameters (including the TSDB, flat model, and so forth) do not match the DUT. Check that the test design database (such as LVDB or TSDB) version is correct.

Debugging Pattern Failures From Stability Issues

It can be difficult to determine whether a stability issue results from problems within the DUT (for example, a shift timing issue) or outside the DUT (the SID environment). This section describes representative cases that can help you narrow down the cause.

Reproducing Intermittent Failures

The method for reproducing intermittent failures is similar to the method for diagnosing pattern-dependent failures.

As with pattern-dependent failures, you use patterns with an isolated bit and that toggle regularly, in addition to more random patterns:

- Patterns with an isolated 1 (such as “00001000…0”)
- Patterns with an isolated 0 (such as “11110111…1”)
- Patters that toggle frequently (such as “010101…01”)
- Other toggling patterns (such as “00110011…0011” or “000111000111…000111”)

However, in order to diagnose intermittent failures, you must use very long patterns and repeat them many times.

Typical Root Causes for Stability Issues

This section describes likely root causes for stability issues that are not DUT-dependent.

GND Problems

A floating GND or a weak connection between the USB adaptor GND and the GND of the DUT could cause a stability issue with intermittent failures. Make sure that you have a strong GND connection and that you avoid using large GND loops with multiple connections.

Pattern-Dependent Failures With Marginal Conditions

Sometimes, pattern-dependent failures become intermittent if they are associated with marginal conditions. Refer to the section “Troubleshooting Pattern-Dependent Failures” on page 423 for information on how to approach pattern-dependent failures.

Signal Integrity Issues With Reflection

This is the most likely issue for long ATPG patterns with an Opal Kelly adaptor connected with flat cables. In this case, the relatively faster base frequency and higher slew rate of the driver could cause higher harmonic waves. Additionally, flat cables can cause bigger impedance mismatches.

Because of this, it is recommended to mount Opal Kelly adaptors directly to the evaluation board with high-density connectors, enabling a 50-ohm on-chip termination resistor (XEM6310 only) with the on_chip_termination property.

If you connect an Opal Kelly adaptor to the evaluation board with flat cables otherwise, consider injecting Schmitt-trigger buffers to clock lines from the adaptor pins.

Once a board design is fixed, it is difficult to resolve this issue. Changing these settings is an experimental approach to altering the conditions causing signal reflection. For example, it may be the case that *disabling* the termination resistor can resolve the issue by changing the shape of the signal waveform, even if this worsens impedance matching.

Minimal simdut_outdir Directory Contents

When the TSDB is not available (for example, when using a third-party ATPG design), the tool cannot generate the *simdut_outdir* structure automatically. In this case, you can use the information in this section to manually create the directory.

Directory Structure

The simdut_outdir directory should contain a subdirectory called *<design_top>.simdut_<design_id>*, where *<design_top>* is the top module name for the design and *<design_id>* is the design ID as specified with the set_context command. Beneath this directory is the *SIMDUT.simulation* directory, which contains the following three files:

- *design.bookkeeping*
- *SIMDUT.design_source_dictionary*
- *SIMDUT.v*

design.bookkeeping Contents

The *design.bookkeeping* file contains the designtop module name and the design ID. This file ends with a space character and no EOL character; because of this, you should not use text editors such as **vi** that automatically add the EOL character to edit it.

The following is an example *design.bookkeeping* file.

```
design_name cpu_top
simdut_design_id bypass
```

SIMDUT.design_source_dictionary contents

The *SIMDUT.design_source_dictionary* file is a dictionary that contains the filepaths for Verilog files that are used in simulation. You can add Verilog files with the set_simulation_library_sources command before using run_testbench_simulations, so this dictionary file does not need to be exhaustive.

The following is an example *SIMDUT.design_source_dictionary* file:

```
set design_source_dictionary \
{
  read1 {
    format {
      "verilog_2001"
    }
    work_library {
      "work"
    }
    files {
      "../../../../tsdb_outdir/dft_inserted_designs/
cpu_top_tk.dft_inserted_design/cpu_top.vg" "SIMDUT.v"
    }
    libraries {
    }
    extensions {
    }
    defines {
    }
    incdirs {
    }
    options {
    }
  }
  logical_design_library_list {
    work
  }
  default_logical_design_library {
    work
  }
}
```

SIMDUT.v Contents

The *SIMDUT.v* file is the main testbench for the SimDUT. It uses the following simple rules:

- The testbench module name must be SIMDUT_TB
- The design top instance must be specified as DUT_inst

The wire descriptions are specified by referring to the top module.

```
`timescale 1ns/1ps
module SIMDUT_TB;
  wire [11:0] pdata_p;
  wire [7:0] portain_p, expdin_p;
  …
  cpu_top DUT_inst(
    .clk1_p(clk1_p),
    .clk2_p(clk2_p),
    …
  );
endmodule
```

TAP Access Check With BYPASS Test

Most JTAG devices use an instruction consisting of all 1s as the BYPASS instruction, even if the exact instruction length is unknown. Because of this, using a long string of 1s in Shift-IR could set the JTAG to BYPASS mode.

The following example illustrates a case for instruction bits less than 64. If the instruction bit length is larger than 64, or the total instruction bit length in the TAP daisy chain is much larger than 64, a much longer all-1s Shift-IR is required. With the SDR command, any longer data can be shifted into the TDI, and TDO output is delayed by one cycle (or by *N* cycles with the TAP daisy chain routes *N* TAPs).

Although this pattern could run on many designs without modification, it might be required to control the JTAG compliance enable (CE) or other device initial sequences before Shift-IR.

BYPASS test pattern example:

```
ENDIR IDLE;
ENDDR IDLE;
PIOMAP ( OUT TDO IN TMS IN TDI IN TRST);
PIO (XHLH );
RUNTEST 5 TCK ENDSTATE IDLE;
PIO (XLLH );
RUNTEST 1 TCK ENDSTATE IDLE;

// Set JTAG BYPASS mode SIR 64 TDI (FFFFFFFFFFFFFFFF) TDO
(FFFFFFFFFFFFFFFD) MASK (0000000000000003);

// Bypass shift-only test SDR 800 TDI
(554FFF42257FFF5555555555557FFF5555555555554FFF4225
554FFF42257FFF5555555555557FFF5555555555554FFF4225
554FFF42257FFF5555555555557FFF5555555555554FFF4225
554FFF42257FFF5555555555557FFF5555555555554FFF4225)

TDO (AA9FFE844AFFFEAAAAAAAAAAAAFFFEAAAAAAAAAAAA9FFE844A
AA9FFE844AFFFEAAAAAAAAAAAAFFFEAAAAAAAAAAAA9FFE844A
AA9FFE844AFFFEAAAAAAAAAAAAFFFEAAAAAAAAAAAA9FFE844A
AA9FFE844AFFFEAAAAAAAAAAAAFFFEAAAAAAAAAAAA9FFE844A)

MASK (FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE);
```

Dofile example:

```
set_context patterns -silicon_insight
read_config_data ./dofile/si.setup
set_current_silicon_insight_setup $curr_si
set_si_options -ignore_uncontacted_pin_data On
set_si_options -cdp_verification off
start_silicon_insight
add_cdp_test BYPASS_check -pattern ./BYPASS_check_with_TRST.svf;
execute_cdp_test BYPASS_check -collect_data_type_list variable
stop_silicon_insight
quit
```

Detecting Instruction Length

Basic JTAG tests may sometimes fail even after you have confirmed TAP access with a BYPASS test. A common reason for this failure is that the design data and, thus, resulting patterns do not match the actual device.

To check for this situation, confirm that the instruction register length of the master TAP for the DUT is the expected length (that is, identical with the design data). Use the “execute_tester_conmmand find_ir_length” command to learn the actual length of the instruction register.

The following examples show this for some different adaptors:

Olimex Adaptor Example

```
// command: set_context patterns -silicon_insight
// command: read_config_data ./data/si.setup
// command: set_current_silicon_insight_setup $curr_si
// command: start_silicon_insight
// /SiliconInsightSetupSpecification is loaded in memory.
// Note: Connected to sid_tester server.
// Note: Start sid_tester .......
// Note: Setting up SiliconInsight Desktop USB adaptor.
// Note: Setting up SVF tap pins.
// Note: Setting up pin maps.
// Note: Opening CDP at 'tmp.cdp'.
// Note: Connected to CDP socket server.
// command: set ir_length [execute_tester_command find_ir_length
//   $params(TRST) $params(TMS) $params(TDI) $params(TDO) $params(TCK)]
// sub-command: execute_tester_command find_ir_length - TMS TDI TDO TCK
// command: puts "INSTRUCTION_LENGTH: $ir_length";
INSTRUCTION_LENGTH: 22
```

Opal Kelly Adaptor Example

```
// command: set_context patterns -silicon_insight
// command: read_config_data ./data/si.setup
// command: set_current_silicon_insight_setup $curr_si
// command: start_silicon_insight
// /SiliconInsightSetupSpecification is loaded in memory. /
// Note: Connected to sid_tester server.
// Note: Start sid_tester .......
// Note: Setting up SiliconInsight Desktop USB adaptor.
// Note: Setting up SVF tap pins.
// Note: Setting up pin maps.
// Note: Opening CDP at 'tmp.cdp'.
// Note: Connected to CDP socket server.
// command: set ir_length [execute_tester_command find_ir_length
//   $params(TRST) $params(TMS) $params(TDI) $params(TDO) $params(TCK)]
// sub-command: execute_tester_command find_ir_length P23 P27 P25 P26 P24
// command: puts "INSTRUCTION_LENGTH: $ir_length";
INSTRUCTION_LENGTH: 4
```

Checking Connections of Test Mode Pins

Unlike an ATE load board, the performance board (evaluation board) for SID usually connects only a limited number of DUT pins to the USB adaptor pins. The remaining DUT pins are opened or pulled up or down with a jumper, DIP switch, or extra circuit (such as an FPGA), depending on the DUT requirements. This incomplete connection can result in an incorrect test mode if you are not careful in your setup.

In the case where the wrong setting causes a fatal result (such as when the TAP is not accessible because TST is not pulled up), this can be easy to troubleshoot. In cases where the test mode

appears to be set up properly (for example, when DUT responses match likely results initially), troubleshooting can be more difficult.

The following example illustrates how to check DUT pin connections, especially test mode pins, using a boundary scan test.

Prerequisites

- This method assumes I/O tests with boundary scan. You therefore must have a TAP on the DUT. (For example, this method does not work for a DUT with EDT and without a TAP.)
- To check the connections of test mode pins, you must have boundary scan cells for those pins. You cannot use test mode pins that are non-boundary scan pins.
- You should have verified TAP access before starting this method. Refer to "TAP Access Check With BYPASS Test" on page 428.

Example

Consider a case with the following test mode pins:

- **ENABLE_MEM** — Input pin (observed with BC_4), pulled up on the board
- **FI_RST_EN** — Input pin (observed with BC_4), connected to Opal Kelly pin P40
- **DATA_IN(7)** — Bidirectional pin (observed with LV_BC_7), pulled up on the board
- **CTRL_IN(6)** — Bidirectional pin (observed with LV_BC_7), connected to Opal Kelly pin P50
- **TMODE2** — Input pin (observed with BC_2), pulled down on the board
- **TMODE1** — Input pin (observed with BC_2), connected to Opal Kelly pin P30

The following setup and dofile provide an example for this case in running the input test in a BSDL-only flow.

BSDL File, BOUNDARY_REGISTER Section

```
attribute BOUNDARY_REGISTER of ROUTER: entity is
-- num cell port function safe [ccell disval rslt]
" 21 (BC_2    , *          , control      , 0 )              ,"&
" 20 (LV_BC_7 , DATA_IN(7) , bidir        , X , 21 , 0 , Z ),"&
" 19 (LV_BC_7 , DATA_IN(6) , bidir        , X , 21 , 0 , Z ),"&
" 18 (LV_BC_7 , DATA_IN(5) , bidir        , X , 21 , 0 , Z ),"&
" 17 (LV_BC_7 , DATA_IN(4) , bidir        , X , 21 , 0 , Z ),"&
" 16 (LV_BC_7 , DATA_IN(3) , bidir        , X , 21 , 0 , Z ),"&
" 15 (LV_BC_7 , DATA_IN(2) , bidir        , X , 21 , 0 , Z ),"&
" 14 (LV_BC_7 , DATA_IN(1) , bidir        , X , 21 , 0 , Z ),"&
" 13 (LV_BC_7 , DATA_IN(0) , bidir        , X , 21 , 0 , Z ),"&
" 12 (LV_BC_7 , CTRL_IN(7) , bidir        , X , 21 , 0 , Z ),"&
" 11 (BC_2    , *          , control      , 0 )              ,"&
" 10 (LV_BC_7 , CTRL_IN(6) , bidir        , X , 11 , 0 , Z ),"&
" 9  (LV_BC_7 , CTRL_IN(5) , bidir        , X , 11 , 0 , Z ),"&
" 8  (LV_BC_7 , CTRL_IN(4) , bidir        , X , 11 , 0 , Z ),"&
" 7  (LV_BC_7 , CTRL_IN(3) , bidir        , X , 11 , 0 , Z ),"&
" 6  (LV_BC_7 , CTRL_IN(2) , bidir        , X , 11 , 0 , Z ),"&
" 5  (LV_BC_7 , CTRL_IN(1) , bidir        , X , 11 , 0 , Z ),"&
" 4  (LV_BC_7 , CTRL_IN(0) , bidir        , X , 11 , 0 , Z ),"&
" 3  (BC_4    , ENABLE_MEM , observe_only , X )              ,"&
" 2  (BC_4    , FI_RST_EN  , observe_only , X )              ,"&
" 1  (BC_2    , TMODE2     , input        , X )              ,"&
" 0  (BC_2    , TMODE1     , input        , X )
```

SiliconInsightSetupSpecification File

```
Sid(opalKellyXem6310) {
  cdp_directory : tmp.cdp;
  opalKellyXem6310 {
    io_standard : 1.80;
    base_frequency : 10MHz;
    PinMap {
      P23 : TRST;
      P24 : TCK;
      P25 : TDI;
      P26 : TDO;
      P27 : TMS;
      P30 : TMODE1;
      P40 : FI_RST_EN;
      P50 : CTRL_IN[6];
    }
  }
}
```

PatternsSpecification File

```
PatternsSpecification(ROUTER, bscan, bsdl_only) {
  usage : manufacturing_test;
  manufacturing_patterns_format : stil;
  compress_pattern_files : off;
  Patterns(Input) {
    tester_period : 100ns;
    load_board_info : BoardA;
    TestStep(input) {
      BoundaryScan {
        bsdl_file : ./ROUTER.bsdl;
        RunTest(input) { }
      }
    }
  }
  LoadBoardInfo (BoardA) {
    TesterInterface {
      default_control : none;
      default_observation : none;
      default_pull_resistor : none;
      Port(TRST)   { control : three_state; observation : three_state; }
      Port(TCK)    { control : three_state; observation : three_state; }
      Port(TMS)    { control : three_state; observation : three_state; }
      Port(TDI)    { control : three_state; observation : three_state; }
      Port(TDO)    { control : three_state; observation : three_state; }
      Port(TMODE1) { control : three_state; observation : three_state; }
      Port(TMODE2) {
        control : tied_low ;
        observation : none ;
      }
      Port(FI_RST_EN) { control : three_state; observation : three_state; }
      Port(ENABLE_MEM) {
        control : tied_high ;
        observation : none ;
      }
      Port(CTRL_IN[6]) { control : three_state; observation : three_state; }
      Port(DATA_IN[7]) {
        control : tied_high ;
        observation : none ;
      }
    }
  }
}
```

Dofile

```
// Generate Input test STIL pattern
set_context pattern -silicon_insight
read_config_data ./bsdl_only.patterns_spec
set ps [get_config_element PatternsSpecification(*,*,*)]
set design_name [get_config_value $ps -id design_name]
set design_id [get_config_value $ps -id design_id]
set pattern_id [get_config_value $ps -id pattern_id]

set_context pattern -silicon_insight -design_id ${design_id}
set_config_value -in_wrapper $ps usage manufacturing_test
set_config_value -in_wrapper $ps manufacturing_patterns_format stil
set_config_value -in_wrapper $ps compress_pattern_files off
process_patterns_specification bsdl_only

// Start SiliconInsight
read_config_data si.setup
set_si_options -ignore_uncontacted_pin_data on
start_silicon_insight

add_cdp_test Input -pattern \
  ./tsdb_outdir/patterns/${design_name}_${design_id}.patterns_${pattern_id}/Input.stil
execute_cdp_test Input -collect_data_type_list variable

stop_silicon_insight
quit
```

Sample Results

The following results from the preceding input files indicate a problem with DATA_IN(7) and CTRL_IN(6):

- DATA_IN(7) should be pulled up (expect 1) but is always 0.
- CTRL_IN(6) should toggle (expect 1 and 0) with Opal Kelly pin P50, but it is always 0.

```
// command: execute_cdp_test Input -collect_data_type_list variable
// Test 'Input' failed.
// Variable failures and unmapped failures of pattern 'Input.stil' :
//
// PatternSet Variable Pin Expected
// Actual
// ------------ ---------------------- --- ----------------------
// input__input bsdl_tap.BOUNDARY(1)    TDO b01XXXXXXXX01XXXXXX1101
//                                          b00XXXXXXXX00XXXXXX1101
// ------------ ---------------------- --- ----------------------
// input__input bsdl_tap.BOUNDARY(2)    TDO b01XXXXXXXX00XXXXXX1000
//                                          b00XXXXXXXX00XXXXXX1000
// ------------ ---------------------- --- ----------------------
```

Appendix C
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

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.