SIEMENS EDA

Hybrid TK/LBIST Flow User's Manual

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
Introduction to the Hybrid TK/LBIST Flow

The hybrid TK/LBIST flow combines TestKompress (EDT) functionality with Tessent LogicBIST functionality in the Tessent Shell environment. Sharing Tessent EDT and Tessent LogicBIST IP functionality reduces hardware overhead. You can implement hybrid TK/LBIST as a dofile or specification-based flow, depending on how the command files are written, and as bottom-up or top-down, depending on the IP implementation.

This manual describes the flow to generate the hybrid TK/LBIST hardware, integrate it into the design, and perform scan insertion, fault simulation and pattern generation. This flow uses the configuration-based methodology. For details on using a dofile-based approach, refer to "The Dofile Flow" on page 263. Although the tool supports a dofile-based flow, it is recommended to migrate to the configuration-based flow to take advantage of its automation, seamless integration, and ease-of-use features.

Note

This manual uses the terms "LogicBIST" and "LBIST" interchangeably.

Hybrid TK/LBIST Implementation

When implementing the hybrid flow with the bottom-up method, you analyze each core in isolation. As soon as a core design is available, you can move on to the embedded insertion and simulation processes for that core without having to wait for the other cores, including the top-level core.

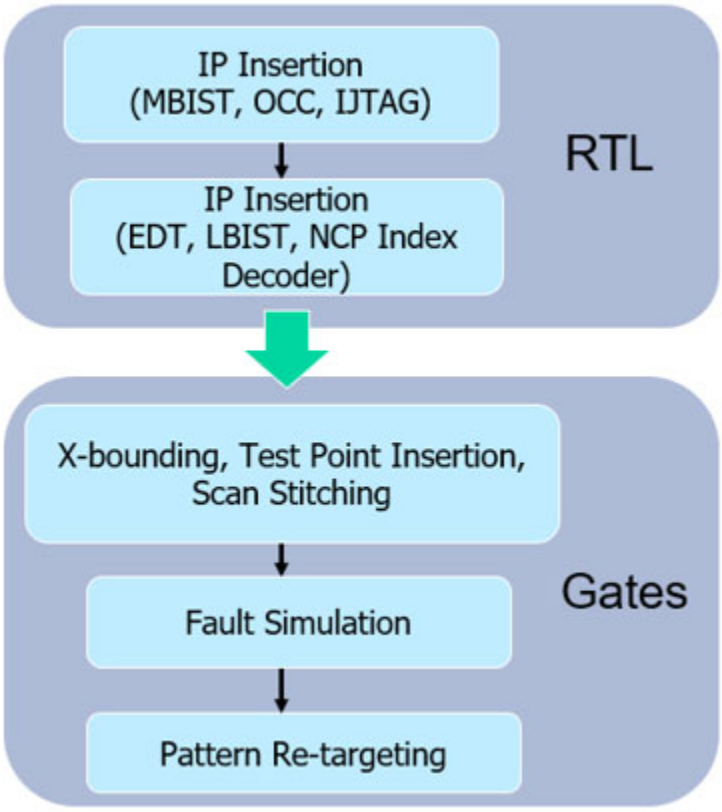
Figure 1-1 shows the five-step flow you perform on each of your cores. These steps are the same for the bottom-up and top-down flows.

The tool stores the output files for each step in the Tessent Shell Database (TSDB). For information about the TSDB, refer to "Tessent Shell Database" in the *Tessent Shell Reference Manual*.

Note

For the Tessent On-Chip Clock Controller (OCC) flow, the tool automatically generates the named capture procedures (NCPs) and test setups as inputs to fault simulation. For more information, refer to "Tessent OCC for Hybrid TK/LBIST" on page 149.

Figure 1-1. Basic Flow



Tessent Core Description

The Tessent Core Description (TCD) is a file that contains the description of the hybrid IP core. The TCD eliminates the need for multiple dofiles and test procedure files for pattern generation.

TCD files are created by the tool to represent specific instruments and apply to the following steps in the TK/LBIST hybrid flow:

- **EDT and LogicBIST IP Generation —** See "EDT and LogicBIST IP Generation" on page 23.
- **LogicBIST Fault Simulation and Pattern Creation —** See "LogicBIST Fault Simulation and Pattern Creation" on page 89.

The TCD file is used instead of dofile commands to describe the hardware between tool invocations. To use a TCD file, you read the file into Tessent Shell, bind it to a specific instance, and configure the instance with parameters.

The tool generates the TCD file and other hybrid logic files during IP generation and places them in the TSDB directory. The TCD file contains the description of the generated Tessent EDT IP and Tessent LogicBIST IP. With a TCD file, Tessent Shell can automatically extract the connectivity between the hybrid IP and the chip, adjust test procedures, and enable pattern generation.

If your hybrid IP can operate in multiple configurations (for example, low power, bypass, and so on), a single TCD file contains all the configurations in contrast to the multiple EDT IP dofile usage. During pattern generation, you can specify how you want those parameters of the EDT IP configured for that ATPG mode.

Note

Do not use instrument TCD files during pattern retargeting; instead, use the core-level TCD files.

Tessent EDT and LogicBIST IP Generation

As part of the hybrid IP generation step, you generate the shared Tessent EDT and LogicBIST RTL. The tool generates one LogicBIST controller for all EDT and LogicBIST blocks in a core.

You can also configure the low-power scheme to control the switching activity during "shift" to reduce power consumption.

There is no TAP controller at the core level. The tool integrates the access mechanism in the IJTAG network at the core level. This step of the flow creates new core-level pins corresponding to the Segment Insertion Bit (SIB) control signals, tck, and LBIST scan I/O. The core-level Verilog patterns operate these pins directly. These pins connect to the TAP controller at the top level of the design. See "Top-Level ICL Network Integration" on page 125 for more information.

As part of IP generation, the tool writes the following files to the TSDB:

- **ICL file —** Consists of the ICL module description for the LBIST controller, the NCP index decoder, and all EDT and LogicBIST blocks that the controller tests.
- **PDL file —** Contains iProcs at the core level that use the ICL modules.

During IP generation, the generated ICL file describes only the LogicBIST, NCP index decoder, and EDT modules. The extracted ICL file includes the core-level pin names and connectivity found from the core-level design netlist. The tool uses the extracted ICL file during top-level pattern generation. See "ICL Extraction and Pattern Retargeting" on page 131 for more information. You can write Verilog patterns in this step and simulate them to verify the test operation at the core level.

For complete information, see "EDT and LogicBIST IP Generation" on page 23. During integration with the top level, the tool adds new top-level test pins or uses existing top-level test pins controlled internally by the EDT and LogicBIST IP.

Logic Synthesis

You must synthesize all of the EDT and LogicBIST blocks and the common LogicBIST controller. Synthesis is fully automated. In the gate-level flow, you can use the run_synthesis command to synthesize the controllers and the test logic in the TSDB and integrate them into the gate-level design. When the run_synthesis command completes successfully, it creates a concatenated netlist of the design that contains the synthesized test logic and modified design modules and places them in the *dft_inserted_designs* directory of the TSDB.

In the RTL-level flow, you can use the run_synthesis command to synthesize the test logic inserted by the tool, but the netlists are not concatenated.

Test Point Analysis and Insertion

Tessent Shell can insert test points in the synthesized netlist.

Upon completion, you use Tessent Shell to write out the modified design and the scan setup files to the TSDB directory. Use these for scan insertion and X-bounding. See "Test Point Analysis and Insertion, Scan Insertion, and X-Bounding" on page 77 for more information.

Scan Insertion and X-Bounding

In this step of the flow, you perform scan insertion on the synthesized netlist. You can also insert test points along with the scan insertion and, optionally, perform wrapper analysis.

Subsequently, the tool writes the TCD file into the TSDB directory. This file contains the information for Logic BIST fault simulation and ATPG. See "Test Point Analysis and Insertion, Scan Insertion, and X-Bounding" on page 77 for more information.

LogicBIST Fault Simulation and Pattern Creation

During this step of the flow, you perform fault simulation and save the parallel LogicBIST patterns.

The core-level fault simulation run computes MISR signature, power consumption, and test coverage for the core. The following files are stored in the TSDB and are used later in top-level ICL extraction and pattern retargeting. Refer to logic_test_cores in the "*Tessent Shell Reference Manual*" for more information.

- **PatternDB file** — Contains the pattern data and the relevant LBIST register values per pattern, such as PRPG, MISR, and low-power registers.
- **Tessent Core Description file (TCD)** — Describes relevant information about the core for the LogicBIST mode.

For subsequent diagnosis, you can use Tessent Diagnosis to perform diagnosis with the EDT patterns. You can use the Single Chain Mode logic for LBIST diagnosis. See "Single Chain Mode Logic" on page 47

See "LogicBIST Fault Simulation and Pattern Creation" on page 89 for more information.

Pattern Generation

In this step of the flow, you generate core-level patterns for the bottom-up method and top-level patterns (including a Verilog testbench) for the LogicBIST controller for the top-down method. This step also generates chip-level serial patterns that you can apply from the tester.

See "Pattern Generation" on page 101 for complete information on all ATPG formats Tessent LogicBIST supports.

Top-Level ICL Network Integration

In this step of the bottom-up flow, you integrate the top-level netlist to instantiate all of the LogicBIST implemented cores.

- Insert IJTAG-compliant SIBs to provide access to cores with LogicBIST inserted and to connect these SIBs and cores to the top-level TAP controller.
- Shadow each core with a separate SIB to provide maximum flexibility for test scheduling.
- Connect EDT control signals from the core to the top level.

See "Top-Level ICL Network Integration" on page 125 for complete information.

ICL Extraction and Pattern Retargeting

In this step of the bottom-up flow, you use the fully integrated top-level netlist, the per-core LogicBIST files generated at the core level, and the top-level ICL/PDL files for your IJTAG instruments such as the TAP controller and clock controller.

You can extract the ICL description or manually create the top-level ICL. The ICL describes the SIB access network and connectivity between your instruments and the LogicBIST cores. You can write the final tester patterns out in any of the supported formats.

You must provide a pattern retargeting dofile that includes all LogicBIST cores and the required scheduling.

Complete Hybrid TK/LBIST Flow

- Per-core steps (repeat for each core in the design):
  - First RTL IP insertion step:
    - MBIST, OCC, and IJTAG
  - Second RTL insertion step:
    - EDT, LBIST, and NCP index decoder
  - Gates:
    - X-bounding, test point insertion, scan stitching, and wrapper analysis (optional)
    - Fault simulation
    - Pattern retargeting

- Top-level steps (perform once with the top-level netlist containing the cores):
  - ICL network integration:
    - Input files: core netlists and top-level interconnect between cores
    - Output: top-level netlist
  - ICL extraction and pattern retargeting:
    - Hybrid TK/LBIST insertion at the top level if required (steps similar to core-level insertion)
    - Input files: top-level netlist, ICL, and PDL; core TCD, patDB, ICL, and PDL; pattern retargeting dofile
    - Output: Verilog simulation patterns and tester patterns (WGL and STIL)

Related Topics

ICL Extraction and Pattern Retargeting

Considerations for Top-Down Implementation

When implementing the hybrid flow with the top-down method, the tool analyzes the entire pre-DFT-inserted view of your chip. The tool analyzes each core within the respective context of its parents.

For the top-down method, you use the same steps as when you implement the hybrid TK/LBIST flow on the core level except without top-level ICL network insertion and extraction or ICL pattern generation (Steps 6 and 7).

Figure 1-1 on page 16 illustrates the steps when using the top-down method. See "Hybrid TK/LBIST Implementation" for more information. You define blocks for insertion and run Steps 1-5 as follows:

- EDT and LogicBIST IP Generation — EDT/LogicBIST hybrid IP generation is similar to that performed with the bottom-up method except that the TAP controller is present at the top level.
- "Test Point Analysis and Insertion, Scan Insertion, and X-Bounding" on page 77 — These steps are similar to those performed using the bottom-up method.
- LogicBIST Fault Simulation and Pattern Creation and Pattern Generation — These steps are similar to those performed using the bottom-up method.

Limitations for the Hierarchical TK/LBIST Flow

When you use a LogicBIST controller at a child level for INTEST and another LogicBIST controller at a parent level for EXTEST, your setup must respect these limitations.

OCCs in the Core

An OCC in the core should not be defined with shift_only_mode=on when you do not create the top-level LBIST edt_clock DFT signal from other signals. During LBIST EXTEST, the core-level LBIST controller is off when the top-level LBIST controller is enabled. This limitation means that the shift_capture_clock_out of the core-level LBIST controller, sourced by test_clock, is unknown during top-level LBIST. If the core-level OCC is built with shift_only_mode, it shifts with shift_capture_clock during EXTEST shift. This propagates an X.

If you define the OCC without shift_only_mode, it shifts with the core's fast_clk, which is controlled by the parent-level OCC.

Embedded PLL and Wrapper Chain OCC

If your core includes an embedded PLL and the wrapper chains include the OCC scan chain, the core-level OCC must be active. Also, it must be accessible to both the core-level and the top-level LBIST controllers. This is automated if you are using the independent insertion flow; otherwise, you must manually insert custom muxing logic to enable this. See "Child-Level OCC Active During External Test" on page 251 for more information.

Mini-OCC in the Wrapper Chain

If you add a mini-OCC to the wrapper chain, the core-level lbist_i/shift_capture_clock must be functional. This is similar to the case of an OCC in the core, but there is no workaround such as turning off shift_only_mode.

Other Limitations

- The Hybrid TK/LBIST gate-level dofile flow is not supported in the independent Hybrid TK/LBIST insertion flow.
- The following DftSpecification/LogicBist/Controller properties have been deprecated:
  - pre_post_shift_dead_cycles: this property is replaced by the SetLoadUnloadTimingOptions wrapper properties and the set_load_unload_timing_options command.
  - ShiftCycles/counter_resolution: this property is not configurable. The bit setting is now the default.

Chapter 2
EDT and LogicBIST IP Generation

As part of the hybrid IP generation step, you generate the Tessent shared EDT and LogicBIST RTL. You can begin with RTL or a gate-level netlist.

The generated EDT and LogicBIST IP can be written out only in Verilog format.

EDT and LogicBIST IP Generation Overview

EDT and LogicBIST IP generation stores several files in the TSDB. The dofiles are used for LogicBIST fault simulation and pattern creation, while the ICL, PDL, and IP netlist files are used as inputs for pattern generation.

Hybrid TK/LBIST IP Generation Flow

This section describes the basic steps in IP generation.

Basic Steps in the Flow

- Load the design RTL and gate input files (libraries, configuration files, and so on).
- Specify constraints and clocks.
- Check design rules through system mode transition.
- Run create_dft_specification to EDT and LBIST requirements.
- Generate the IP with process_design_specification.
- Extract ICL.

These steps create the ICL, PDL, SDC, TCD, and a BIST-ready netlist. Then they store them in the TSDB with a unique design ID.

Overview Details

You use Tessent Shell to generate the shared EDT/LogicBIST RTL. Modular EDT is supported in this step, where both shared EDT/LogicBIST IP per block and LogicBIST controller are inserted at the same time.

The new core-level pins corresponding to the SIB control signals and TCK, and LogicBIST scan I/O are created at this step. The core-level Verilog patterns operate these pins directly. These pins are later connected to the TAP controller at the top level of the design (see "Top-Level ICL Network Integration").

As part of IP generation, the following files are written out:

- The ICL file consists of the ICL module description for the LogicBIST controller, the NCP index decoder, and all EDT blocks tested by this controller.
- The PDL file contains iProcs at the core level that use the ICL modules written out.

During the IP generation step, the generated ICL file describes only the LogicBIST and EDT modules. This extracted ICL includes the core-level pin names and connectivity found from the core-level design netlist. The extracted ICL is used during top-level pattern generation—see "ICL Extraction and Pattern Retargeting." Verilog patterns can be written out in this step and simulated to verify the test operation at the core level.

Integrating a Third-Party TAP in the Hybrid TK/LBIST Flow

When you use the DftSpecification wrapper and properties for the Hybrid TK/LBIST flow, the tool automatically inserts a Tessent TAP as part of the IJTAG network.

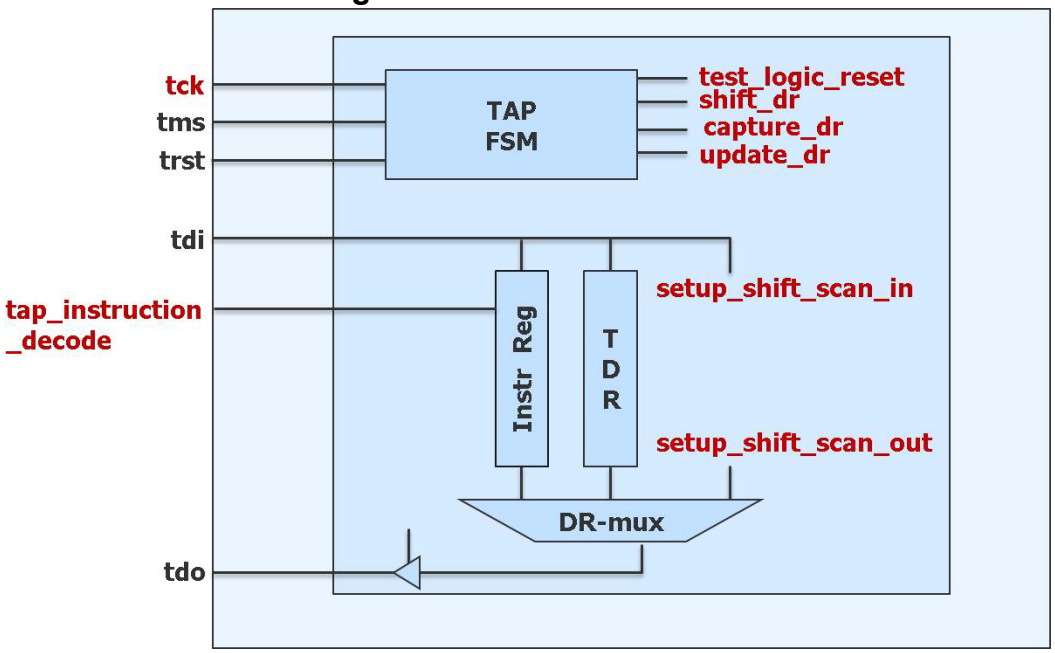
If you want to use a third-party TAP, see "Connecting to a Third-Party TAP" in the *Tessent Shell User's Manual* for complete information. When using a third-party TAP, ensure that the following signals are present.

- tck
- setup_shift_scan_in
- setup_shift_scan_out
- capture_dr
- shift_dr
- update_dr

- test_logic_reset
- tap_instruction_decode

Figure 2-1 shows the TAP controller with these signals highlighted in red.

Figure 2-1. TAP Controller



Note the following:

- The tool generates the ICL/PDL for the TAP—see "Pattern Generation."
- The tool automatically inserts a pipeline stage after the last SIB in the tool-produced ICL network for the LogicBIST controller to account for designs where the TAP already has a retiming flop on the TDO output pin. Consequently, you do not need to modify the ICL to account for the retiming flops at the output.

Clock Controller Connections to the EDT/LogicBIST IP

The clock controller interfaces only with scan_enable and shift_clock pins. You are responsible for creating and instantiating your clock controller in the input design. The clock controller should be pre-configured such that it is already usable for ATPG. The clock controller is expected to route shift_clock to its clkout pin when scan_enable is 1. It is expected to route the programmable internally-generated capture clock to its clkout pin when scan_enable is 0.

In this flow, the tool controls the scan_enable/shift_clock pins such that during ATPG mode the existing connections are used, but during LogicBIST mode they are driven by ~lbist_capture_en/lbist_shift_clock respectively.

Figure 2-2 shows the Clock Controller before the EDT/LogicBIST IP has been generated.

Figure 2-2. Clock Controller Before IP Generation

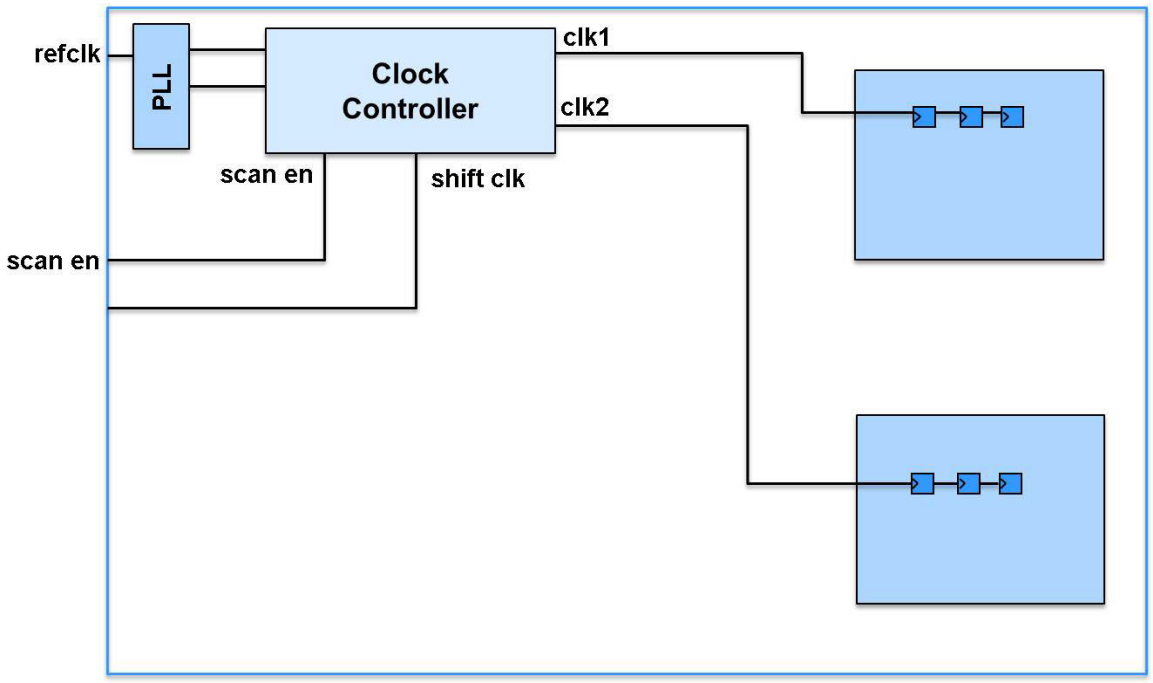
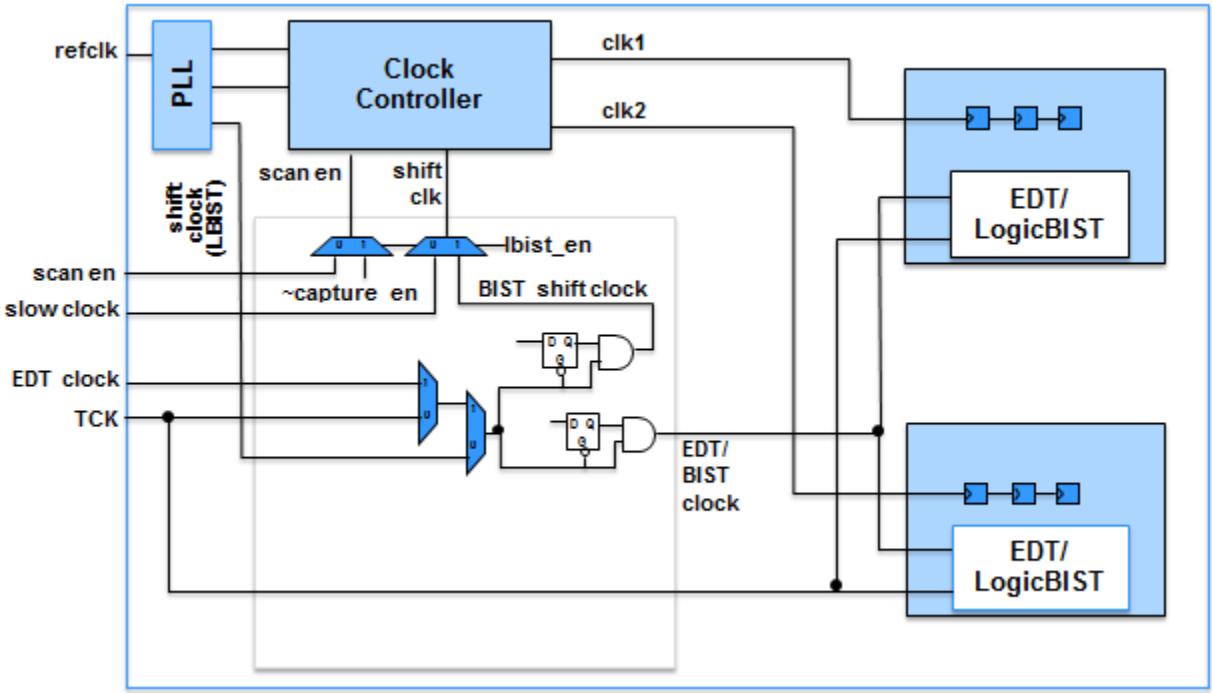


Figure 2-3 shows the clock controller connections to the BIST controller after EDT/LogicBIST IP generation.

Figure 2-3. Clock Controller After EDT/LogicBIST IP Generation



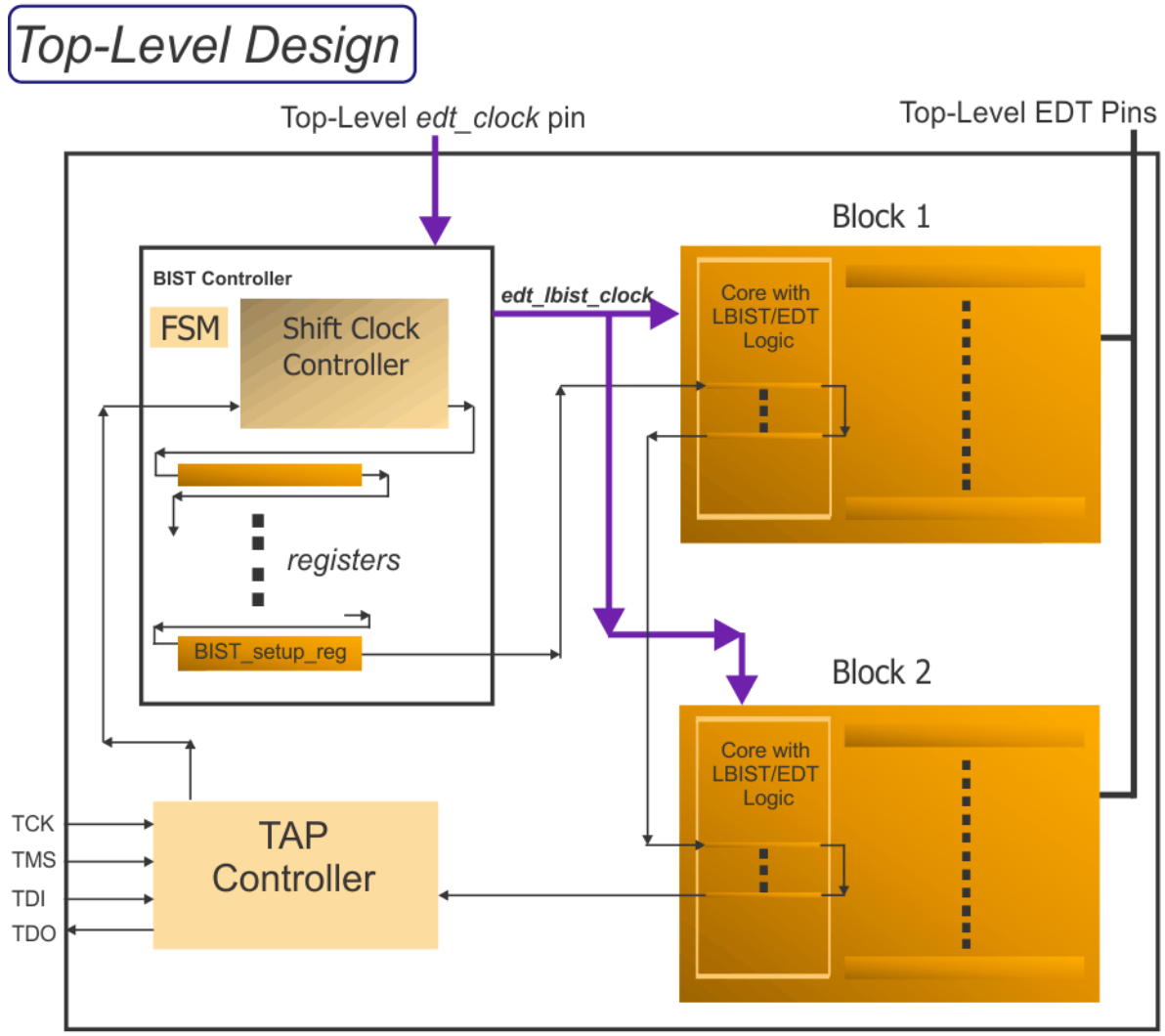
Alternatively, the clock controller could have scan_enable/shift_clock_en pins (instead of shift_clock). In this case, the tool muxes them with ~lbist_capture_en/lbist_shift_clock_en. The clock controller is responsible for using the lbist_shift_clock_en as the clock-gating signal to internally generate the shift clock signal.

EDT and LogicBIST IP

An EDT block refers to an independent set of scan chains that are driven by a decompressor and observed by a compactor. The EDT block organization can be different from the block organization used for physical implementation. In particular, there can be multiple EDT blocks inside a single physical block.

Figure 2-4 illustrates a high-level block view of the EDT and LogicBIST IP.

Figure 2-4. Block-Level View of the EDT/LogicBIST IP



The EDT/LogicBIST hybrid IP reduces the area overhead compared to a separate implementation of EDT and LogicBIST IP. This is done by re-using parts of the IP for both EDT and LogicBIST modes. This hybrid logic controls input stimuli generation and output response comparison, and is implemented separately for each EDT block. The top-level controller including the LogicBIST FSM is then connected to these hybrid IP inserted EDT blocks.

For detailed description of Hybrid EDT/LogicBIST IP, refer to the following sections:

- "Shared Logic" on page 137
- "Inserted Hybrid TK/LBIST IP" on page 138
- "Scan Chain Masking" on page 143
- "New LogicBIST Control Signals" on page 143

Figure 2-5 shows the final top-level netlist after IP generation. During IP generation, the tool inserts muxes at the shift clock and scan enable clock controller pins and drive them from the LogicBIST controller.

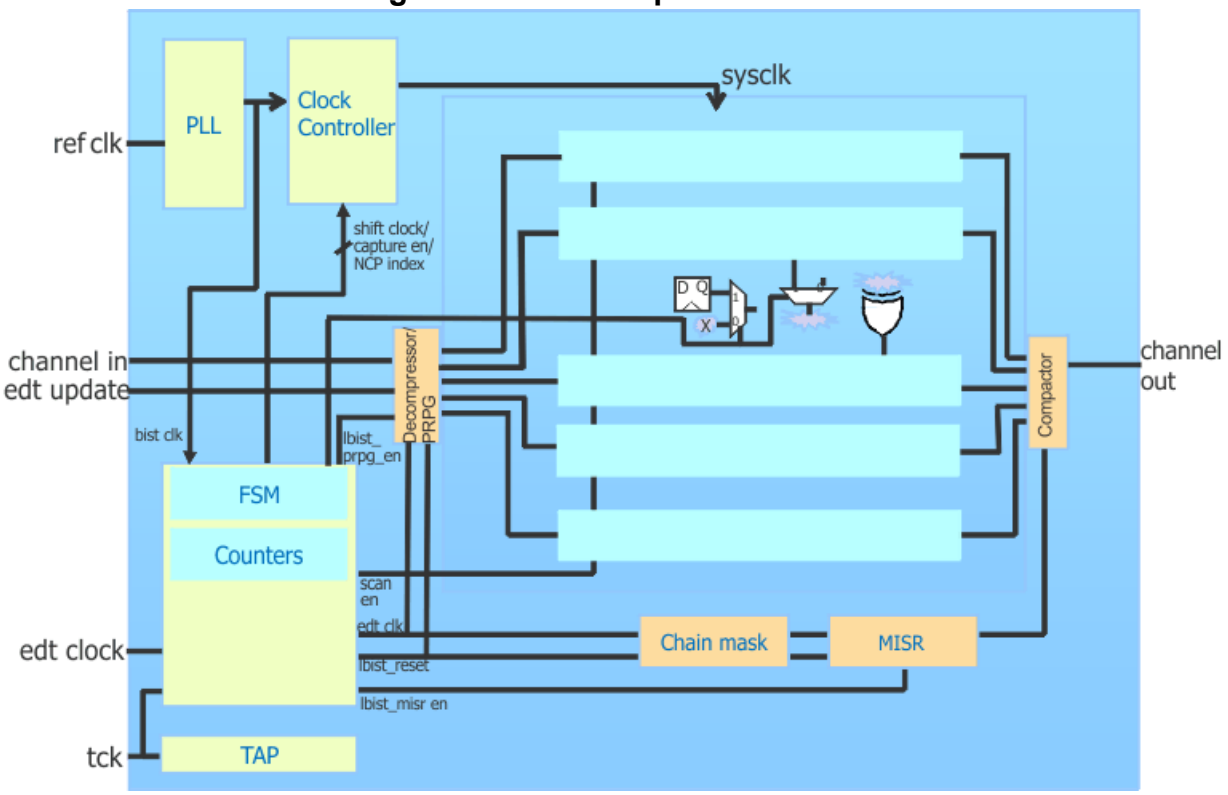
The clock controller's shift clock input is driven by:

- The LogicBIST shift clock in LogicBIST mode,
- The pre-existing EDT clock connection during EDT mode.

The clock controller's scan enable input is driven by:

- The inverted lbist_capture_en signal in LogicBIST mode,
- The pre-existing scan enable connection during EDT mode.

Figure 2-5. Final Top-Level Netlist



During EDT mode of operation, all EDT control signals except EDT clock, meaning update, bypass, and low power enable, are directly connected to the EDT logic without passing through the LogicBIST controller. The scan cells are shifted using a top-level shift clock. The Shift Clock controller chooses the EDT clock to be delivered to the EDT blocks.

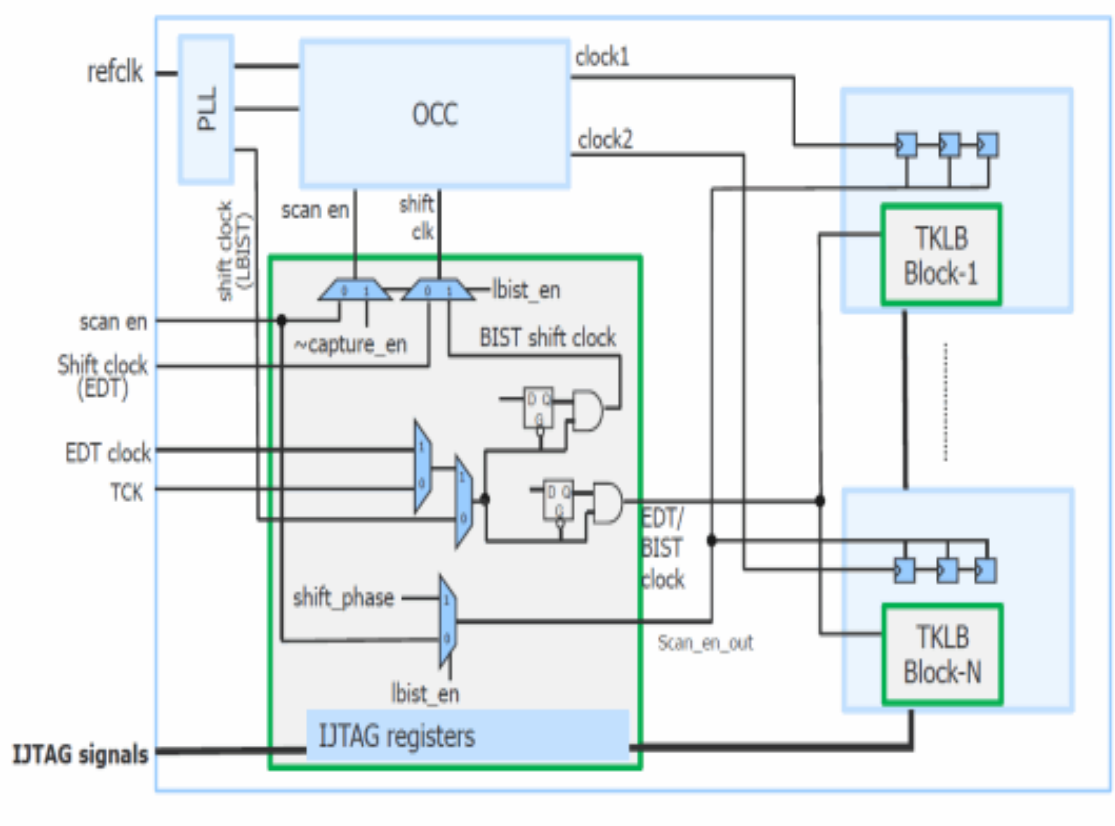
In addition, in EDT mode, the tool calls an automatically generated iProc procedure that initializes the hybrid IP.

During LogicBIST mode of operation, the scan cells are shifted using the shift clock output from the LogicBIST controller. The shift clock controller chooses a gated version of the LogicBIST clock to be delivered to the LogicBIST blocks. This gating ensures that the PRPG and MISR are not pulsed during capture as well as the transition between shift and capture modes.

The core scan cells receive the capture clock pulses from the clock controller logic. The LogicBIST clock input to the PRPG and MISR is turned off during capture.

Figure 2-6 shows the clock and EDT control signals in the LogicBIST IP.

Figure 2-6. Clocking During EDT Shift Mode of Operation



Clock Control Logic and Named Capture Procedures

In the EDT context, you must ensure the generation of proper shift and capture clocks. For LogicBIST mode, the controller generates the shift clock and capture enable signals, which are connected to the clock control logic. The scan enable output from the LogicBIST controller is de-asserted during capture.

Note

NCPs are always required for LogicBIST operation. Other clock control techniques such as external clocks and clock_control definitions are not compatible with BIST application.

EDT Mode

All existing EDT capabilities for controlling capture mode clocking can be used in the shared EDT/LogicBIST architecture. For example, suppose you have a clock pin that is a design top-level pin controlled by the tester. When using a PLL, clock control register values for functional clocks can be shifted in during scan shift. You can control the number of capture cycles on a per pattern basis, either through a named capture procedure or clock_control definition.

LogicBIST Mode

You can describe specific capture clock sequences to be applied using NCPs using the following commands:

- add_bist_capture_range
- set_lbist_controller_options, specifically, the -capture_procedures switch

In general, you associate the percentage of patterns for which each NCP is applied. This should reflect the number of faults that can be detected with the specified NCP. The number of cycles in capture should be uniform across all NCPs. You can specify the number of BIST clock cycles for capture. When not specified, the maximum value possible for the capture counter register in the hardware is used.

The LogicBIST controller consecutively applies the NCPs for the specified percentage of patterns, with the cycle repeating after 256 patterns. An additional output that identifies the index of the currently targeted NCP is available as a counter calculating the total number of NCPs. The NCP index is generated by decoding from the least significant bits of the pattern counter.

Again, you must ensure your design generates the correct clock sequence corresponding to the NCP based on this index signal bits.

If you define only a single NCP, then the NCP index output is not generated. You should do this in cases where the clock controller is initialized during test_setup to generate a particular capture procedure. All patterns in this session use the same capture procedure.

See "LogicBIST Fault Simulation and Pattern Creation Overview" on page 89 for more information on NCP generation.

LogicBIST Consideration When Using OCC for Fast Capture Mode

In the hybrid EDT/LBIST flow, you are responsible for ensuring that OCC generates capture clock pulses within the capture window (corresponding to the capture state of the LogicBIST FSM). When you configure Tessent OCC for fast capture mode, the OCC hardware requires one extra slow clock pulse to sample the capture_en trigger and two extra fast clock pulses for the synchronizer before the tool generates the capture pulses. When the fast clock and the slow clock have the same or similar frequencies, depending on clock balancing, it may take up to four pulses before the tool generates the capture pulses. This can result in clock pulses occurring outside the capture window, which can generate pattern mismatch errors.

To ensure that the tool accounts for the extra cycles, generate the LogicBIST capture width register with a bigger counter. For example, for an OCC with a four-bit shift register, configure the LogicBIST capture width counter to support up to eight cycles in the capture, as follows:

```
DftSpecification(mymodulename,mymoduleid) {
  LogicBist {
    Controller(id) {
      CaptureCycles {
        max : 8 ;
      }
    }
  }
}
```

Likewise, during fault simulation, OCC is configured as having up to four pulses through NCPs, but you can load eight pulses into the capture width register by specifying the following command:

```
set_external_capture_options -fixed_cycles 8
```

Programmable Registers

Programming of the BIST registers is the same for both TAP and non-TAP cases. The BIST controller and EDT blocks use the same control interface.

When using a TAP, these signals are generated by the TAP controller. When not using a TAP, these signals are presented as top-level pins that operate similarly to the TAP controller output signals. This is done automatically in the next step of the flow—see “Pattern Generation” on page 101.

Related Topics

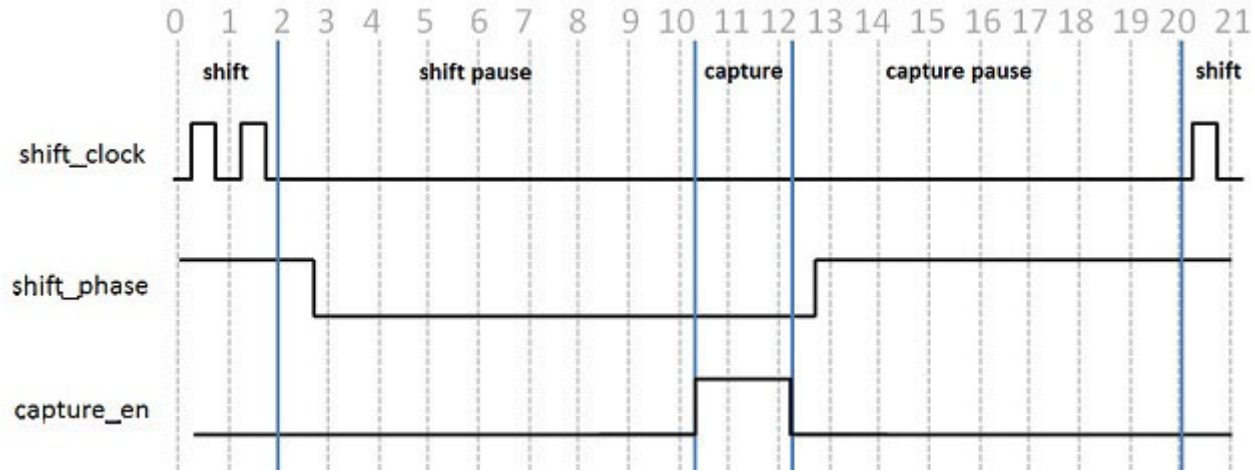
Programmable Registers Inside Hybrid IP

Programmable Shift and Capture Pause Cycles

During the LBIST controller operation, the tool applies a total of 16 dead cycles before and after the capture clocks are pulsed to help with the timing closure of the test logic. To reduce test overload due to the dead cycles, you can change the number of dead cycles by using the set_lbist_controller_options command.

The following figure shows the default cycle pause width for the dead cycles. The shift pause has 8 dead cycles from the last shift clock to the start of the capture, and the capture pause has 8 dead cycles from the end of the capture to the first clock edge in the shift. In addition, the scan enable (shift_phase) signal goes low after the last shift and goes high after the capture. This enables maximum setup time transition.

Figure 2-7. Default Dead Cycle Pause Width



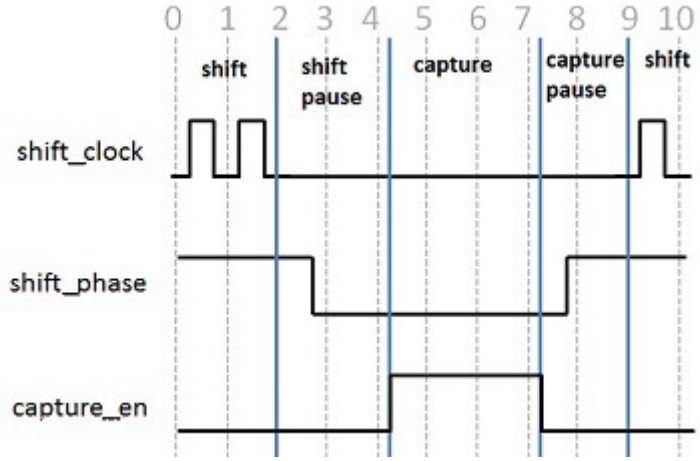
To change the number of dead cycles in the shift and capture pauses, change the number of dead cycles by specifying the pre_post_shift_dead_cycles parameter, as shown below:

```
LogicBist {
  Controller(id) {
    parent_instance           : name ;
    leaf_instance_name        : name ;
    pre_post_shift_dead_cycles : int ;  // default: 8
  }
}
```

The option accepts integer values from 2 to 8, with 8 being the default. When specified, the hardware changes so that the requested number of dead cycles is used for both the shift and the capture pauses. You cannot use a value of 1 because it might allow the capture enable and shift phase signals to toggle on the same clock edge.

Figure 2-8 shows the waveform when you have specified 2 for the dead cycle pause.

Figure 2-8. Dead Cycle Pause Width of 2



Low-Power Shift

During EDT and LogicBIST IP generation, you can configure the low-power scheme to control the switching activity during "shift" to reduce power consumption.

Use the DftSpecification/EDT/Controller/LogicBistOptions/ShiftPowerOptions/SwitchingThresholdPercentage wrapper to create the low-power shift controller for LogicBist. For example:

```
DftSpecification(module_name, id) {
  EDT {
    Controller(id) {
      LogicBistOptions {
        ShiftPowerOptions {
          SwitchingThresholdPercentage {
            hardware_default : 10 ;
          }
        }
      }
    }
  }
}
```

For a detailed description of the embedded structure inserted for this purpose, see "Low-Power Shift Controller."

Warm-Up Patterns

At the beginning of pattern generation, several patterns can elapse before the voltage in the chip stabilizes. This voltage droop can cause pattern mismatches. To remedy the impact of voltage droop, the tool inserts a warm-up pattern count register in the logic BIST controller. Warm-up patterns are those for which the scan chain unload values are not accumulated in the MISR, and for which no fault credit has been taken during fault simulation.

Figure 2-15 on page 56 shows the placement of the warm-up pattern count register within the logic BIST controller. By default, the tool creates an 8-bit register to enable up to 255 warm-up patterns. The tool loads the warm-up pattern count register with the value of the pattern count register when the warm-up is completed. For example, if there are 13 patterns, 4 of which are warm-up patterns, the value in the warm-up pattern count register is 10.

The logic BIST controller operates as follows: The first pattern that the tool uses for fault detection during fault simulation corresponds with begin_pattern=0. The PRPG value for this pattern is determined by the initial hardware PRPG seed, followed by applying the default number of warm-up patterns. When a non-default number of warm-up patterns is applied, the PRPG must be loaded such that it still reaches the same initial PRPG value for this first pattern. If begin_pattern is not zero, then the PRPG must be loaded such that it reaches the correct value that was specified for that pattern during fault simulation. When begin_pattern is zero, the tool loads the MISR with the value for that pattern (since the MISR does not update during warm-up

patterns). However, when begin_pattern is non-zero, the tool must load the MISR with the value that was computed during fault simulation for that pattern.

The tool masks the scan chain unload values for warm-up patterns to enable the voltage in the chip to stabilize. You can specify the number of patterns to mask by setting the warmup_pattern_count argument when executing the run_lbist_normal or scan_unload_register iCall. If this argument is not provided with the iCall command, the iProc procedure defaults it to 0.

Use the DftSpecification/LogicBist/Controller/WarmupPatternCount wrapper to change the maximum number of warm-up patterns, specify the number of hardware default warm-up patterns, or suppress warm-up pattern count register insertion.

Pattern Counts When Using Warm-Up Patterns

During fault simulation, the tool computes the PRPG value at the start of each warm-up pattern (without fault crediting and MISR accumulation) for the maximum number of warm-up patterns. Fault simulation is then performed for the number of patterns specified by set_random_patterns with fault crediting and MISR signature calculation.

Logic BIST patterns are typically applied starting with pattern 0. This is the default for the hardware, and is also the default for the run_lbist_normal ICL procedure. The most common reason for specifying a non-zero value for begin_pattern is during debug or diagnostic when you want to quickly apply a small range of patterns, or just a single failing pattern.

In the following example, the user wants to run LBIST starting from a specific pattern. The pattern specification instructs the logic BIST controller to run from patterns 100 to 199 with 16 warm-up patterns. Since there are 16 warm-up patterns, the tool seeds the PRPG with a seed corresponding to pattern 84, which is equal to:

begin_pattern - warmup_pattern_count

In this case, the controller executes 116 patterns but the MISR only observes the last 100 patterns.

```
PatternsSpecification(coreA,gate,signoff){
  Patterns(LogicBist_coreA){
    ClockPeriods {
      refclk : 10.00ns;
    }
    TestStep(serial_load){
      LogicBist{
        CoreInstance(.){
          run_mode : run_time_prog;
          begin_pattern : 100;
          end_pattern : 199;
          warmup_pattern_count : 16;
        }
      }
    }
  }
}
```

In the following example, the PRPG initial seed originates from the warm-up PRPG seed-only patterns. Beginning from pattern 0, the tool seeds the MISR with d‘0, and the PRPG is based on the number of specified warm-up patterns. Suppose your pattern specification states:

```
PatternsSpecification(coreA,gate,signoff){
  Patterns(LogicBist_coreA){
    ClockPeriods {
      refclk : 10.00ns;
    }
    TestStep(serial_load){
      LogicBist{
        CoreInstance(.){
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 99;
          warmup_pattern_count : 16;
        }
      }
    }
  }
}
```

Assuming a maximum of 255 warm-up patterns, the PRPG seed starts with the 240th warm-up pattern.

With respect to warm-up patterns and the PRPG seeding, scan unload pattern operations function the same as described above for logic BIST patterns.

For the hardware default mode, the tool performs the warm-up period within the hardware so no additional IJTAG access is required. The number of warm-up patterns, PRPG seed, and low-power register seeds are hard-coded in the iProc, and these values are checked against the PatternDB file to ensure they match. If the iProc values do not match the PatternDB, the tool generates an error and the hardware default patterns cannot be saved.

Fault Simulation

You can use the set_edt_options -decompressor_seed command to change the initial seed of the PRPG. This seed applies to both the warm-up patterns and the actual patterns. If you do not specify a PRPG seed, the tool uses the value from IP generation. Using the PRPG seed, the tool calculates the PRPG seed for the first regular pattern, which it then stores in the Tessent Core Description file and flat model.

If you use a different decompressor seed than that used during IP generation, then the hardware default mode cannot be used with the current pattern set.

The tool does not include warm-up patterns in the number of logic BIST patterns it simulates, which means these patterns do not contribute to fault coverage analysis. For example, if you specify set_random_patterns 32000 and you have a maximum of 255 warm-up patterns, then the tool creates 255 warm-up patterns followed by 32000 logic BIST patterns.

When creating parallel patterns with the write_patterns -parallel command, the tool excludes the warm-up patterns because the parallel patterns are primarily used for validating logic BIST operation. However, the PatternDB file includes the warm-up patterns so that the tool can apply them during pattern retargeting.

The following TCD file example shows a warm-up pattern register capable of 255 patterns.

```
Core(CoreA) {
  LbistMode(lbist) {
    Registers {
      PrpgRegister(CoreA_edt_i.lfsm_vec) {
        length : 31;
        type : prpg;
      }
      WarmupPatternRegister(CoreA_lbist_i.warmup_pattern_count) {
        length : 8;
      }
    }
  }
}
```

Example

In the following example, the logic BIST controller needs to support up to 300 warm-up patterns with 128 hardware default warm-up patterns. During IP generation, you would use the following values in the WarmupPatternCount wrapper:

```
DftSpecification(coreA, gate) {
  LogicBist {
    Controller {
      WarmupPatternCount {
        max : 300 ;
        hardware_default : 128 ;
      }
    }
  }
}
```

The Logic BIST controller generates a warm-up pattern counter register with 9 bits, enabling up to a maximum of 511 warm-up patterns.

The following example illustrates how you can apply the warm-up patterns for various scenarios.

```
PatternsSpecification(coreA,gate,signoff) {
  // Run the first 1K patterns with 300 warm-up patterns
  Patterns(warmup_300) {
    ...
    TestStep(serial_load) {
      LogicBist {
        CoreInstance(w2_A) {
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 1023;
          warmup_pattern_count : 300;
        }
      }
    }
  }

  // Unload the 274th pattern after applying 300 warm-up patterns
  Patterns(diag_warmup_300) {
    ...
    TestStep(diagnosis) {
      LogicBist {
        CoreInstance(w2_A) {
          run_mode : run_time_prog;
          begin_pattern : 274;
          end_pattern : 274;
          warmup_pattern_count : 300;
          DiagnosisOptions {
            extract_flop_data : on;
          }
        }
      }
    }
  }

  // Run the first 1K patterns with 16 warm-up patterns
  Patterns(warmup_16) {
    ...
    TestStep(serial_load) {
      LogicBist {
        CoreInstance(w2_A) {
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 1023;
          warmup_pattern_count : 16;
        }
      }
    }
  }

  // Run the HW default pattern, which contains 128 warm-up patterns
  Patterns(hw_default) {
    ...
    TestStep(hw_default) {
      LogicBist {
        CoreInstance(w2_A) {
          run_mode : hw_default;
        }
      }
```

```
    }
  }

  // Run two controllers in parallel with different
  // warm-up pattern ranges and number of warm-ups
  Patterns(parallel_warmup) {
    ...
    TestStep(serial_load) {
      LogicBist {
        CoreInstance(w2_A) {
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 1023;
          warmup_pattern_count : 64;
        }
        CoreInstance(w2_B) {
          run_mode : run_time_prog;
          begin_pattern : 1024;
          end_pattern : 2047;
          warmup_pattern_count : 128;
        }
      }
    }
  }
}
```

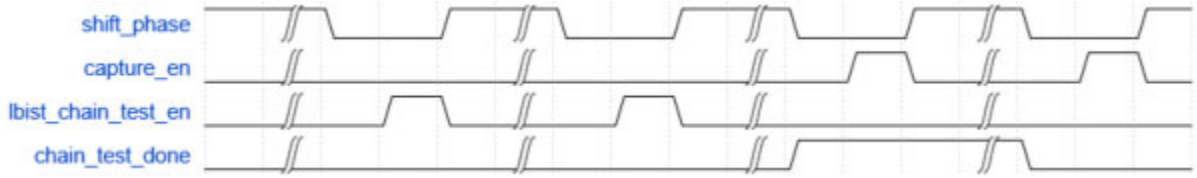
Chain Test Patterns

You can create the LBIST controller with dedicated hardware to integrate chain test patterns with the scan patterns. To add the chain test hardware, declare the maximum number of allowed chain test patterns with the DftSpecification/LogicBist/Controller/ChainTestPatternCount/max property.

During fault simulation, you can control how many chain test patterns are created using the "-parameter_values {chain_test_pattern_count integer}" option of the add_core_instances or set_core_instance_parameters command. If unspecified, the hardware default specified during IP generation is used.

When chain test patterns are enabled, they are applied after any warm-up patterns and before the asynchronous set/reset and scan patterns. The number of chain test patterns applied count towards the total number of random patterns applied. For example, if you request 2000 random patterns, and there are seven chain test patterns, then patterns 0:6 are chain test patterns and 7:1999 are normal scan patterns. During the chain test patterns, the capture procedure does not pulse any clocks, and the capture window is one shift cycle wide. The capture_en signal remains low during the chain test.

Figure 2-9. Chain Test Control Signals



Note

Fault simulation can create a dedicated chain test pattern set if the dedicated chain test hardware is not present. To do this, use the "-parameter_values {pattern_set chain_test} option of the add_core_instances or set_core_instance_parameters commands.

You can enable the chain test patterns with the set_bist_chain_test command when using the dofile flow where the add_core_instances or set_core_instance_parameters commands are not used.

Example

In the following example, the logic BIST controller needs to support up to 16 chain test patterns with seven hardware default patterns. Include the following in the DFT specification to generate the IP:

```
DftSpecification(coreA,dft) {
  LogicBist {
    Controller(1) {
      ChainTestPatternCount {
        max              : 16;
        hardware_default : 7;
      }
    }
  }
}
```

During fault simulation, the tool overwrites the default seven chain test patterns with 12 chain test patterns.

```
// command: add_core_instances ... -parameter_values
{chain_test_pattern_count 12}
...
//  command: set_random_patterns 2000
//  Note: First 12 patterns will test the chains.
```

When performing pattern retargeting, you do not need to specify additional properties. You can control the number of chain test patterns with the begin_pattern and end_pattern properties.

```
PatternsSpecification(coreA,gate,signoff) {
  // Run the first 1K patterns with 12 chain test patterns
  Patterns(first_1k_with_chain_test) {
    ...
    TestStep(serial_load) {
      LogicBist {
        CoreInstance(w2_A) {
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 1023;
        }
      }
    }
  }
  // Run the first 1K patterns without the 12 chain test patterns
  Patterns(first_1k_without_set_reset) {
    ...
    TestStep(serial_load) {
      LogicBist {
        CoreInstance(w2_A) {
          run_mode : run_time_prog;
          begin_pattern : 12;
          end_pattern : 1035;
        }
      }
    }
  }
  // Run the first 12 patterns to only perform the chain test patterns
  Patterns(set_reset_only) {
    ...
    TestStep(serial_load) {
      LogicBist {
        CoreInstance(w2_A) {
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 11;
        }
      }
    }
  }
}
```

Related Topics

add_core_instances [Tessent Shell Reference Manual]

set_core_instance_parameters [Tessent Shell Reference Manual]

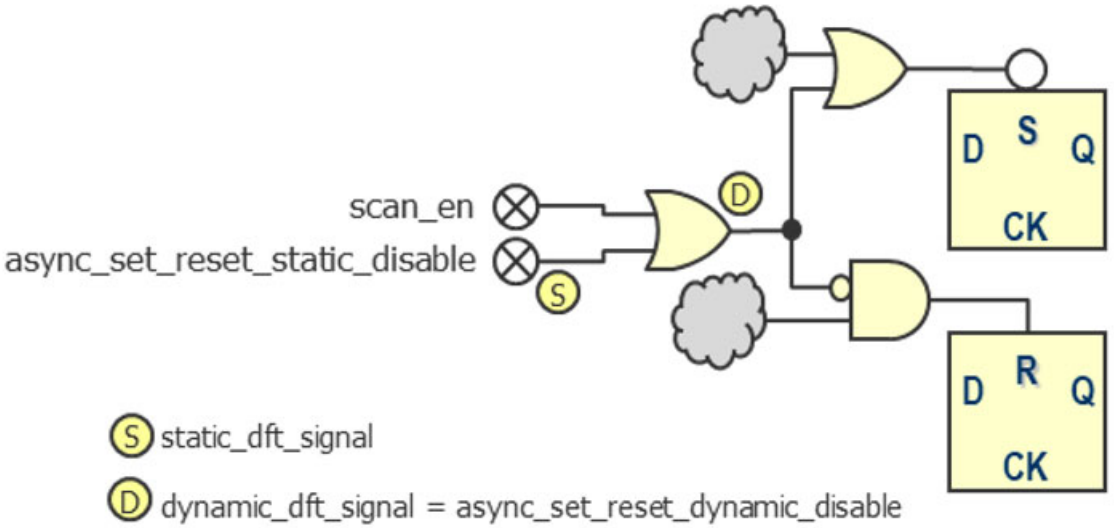
DftSpecification [Tessent Shell Reference Manual]

Asynchronous Set/Reset Patterns

When the design has scannable flip-flops with asynchronous set/reset pins, the tool adds the DFT signals async_set_reset_static_disable and async_set_reset_dynamic_disable. The static DFT signal enables all the asynchronous set/reset signals to be completely disabled.

The dynamic DFT signal, created by combining the static DFT signal with scan_en, disables the set and reset signals only during shift (when scan_en is high), as shown in Figure 2-10. The state of the functional circuit then determines whether the set/reset signals are active during the capture.

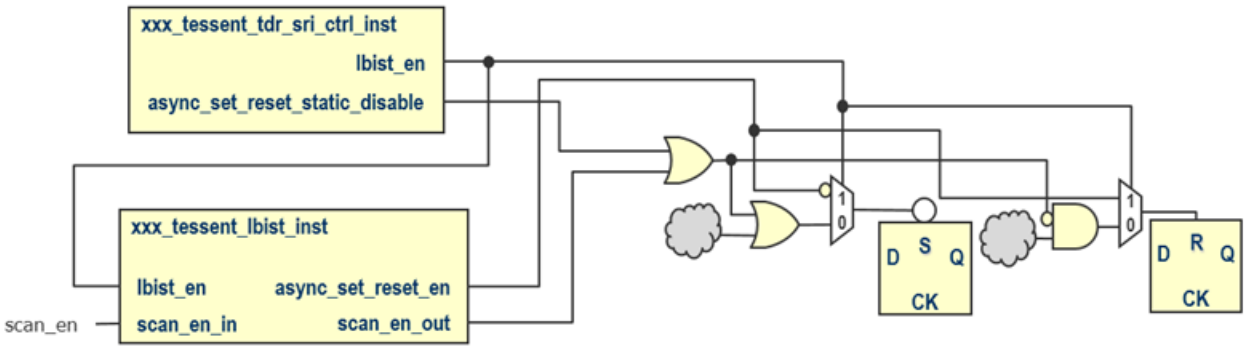
Figure 2-10. DFT Signal to Turn Off for Set/Reset Signals During ATPG



In most cases, it is sufficient to test only the set and reset signals during ATPG. However, if you want to test the set and reset signals with logic BIST, you can add hardware to achieve this. You must declare the dynamic DFT signal, lbist_async_set_reset_dynamic_enable to enable the logic BIST controller to pulse the set/reset signals. This DFT signal multiplexes with the async_set_reset_dynamic_disable DFT signal using the lbist_en static DFT signal as a select signal.

An output port on the logic BIST controller, async_set_reset_en, drives the DFT signal so that the tool pulses the set/reset signals during the logic BIST run modes (see Figure 2-11). An asynchronous set/reset pattern count register within the logic BIST controller determines which patterns are testing the set/reset pins. See Figure 2-15 on page 56 for register placement.

Figure 2-11. DFT Signal to Provide Set/Reset Controllability During LBIST



The tool applies the asynchronous set/reset patterns after the warm-up patterns, if any, and before the regular scan patterns. When the set/reset patterns are active, the async_set_reset_en port is low during the shift. Because these patterns are testing only the asynchronous set/reset pins, the capture procedure does not pulse any clocks. The capture window is two shift clock cycles wide to ensure sufficient propagation time for the set/reset pulse to reach the flip-flops.

Fault Simulation

The TCD for the logic BIST controller carries forward the size of the asynchronous set/reset pattern count register, so that during fault simulation, the default and maximum number of set/reset patterns are known. You can control the number of asynchronous set/reset patterns to simulate with the async_set_reset_pattern_count parameter of the add_core_instances command.

The number of asynchronous set/reset patterns applied count towards the total number of random patterns applied. For example, if you request 2000 random patterns, and there are seven asynchronous set/reset patterns, then patterns 0:6 test the set/reset pins and 7:2000 are normal scan patterns.

Example

In the following example, the logic BIST controller needs to support up to 16 asynchronous set/reset patterns with seven hardware default patterns. During IP generation, you would have the following in the DFT specification:

```
DftSpecification(coreA,dft) {
  LogicBist {
    Controller(1) {
      AsyncSetResetPatternCount {
        max             : 16;
        hardware_default : 7;
      }
    }
  }
}
```

During fault simulation, the tool overwrites the default seven asynchronous set/reset patterns with 12 asynchronous set/reset patterns.

```
// command: add_core_instances ... -parameter_values
          {async_set_reset_pattern_count 12}
...
//  command: set_random_patterns 2000
//  Note: First 12 patterns test the asynchronous sets and resets.
```

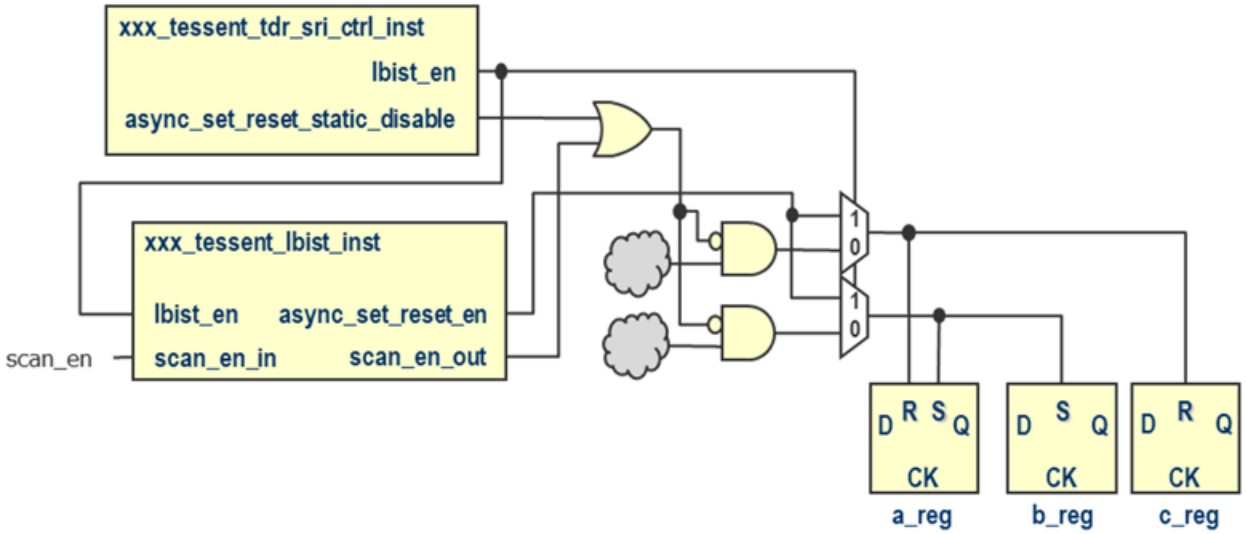
When performing pattern retargeting, no additional properties need to be specified. Using the begin_pattern/end_pattern properties, you can control the number of asynchronous set/reset patterns.

```
PatternsSpecification(coreA,gate,signoff) {
  // Run the first 1K patterns with 12 set/reset patterns
  Patterns(first_1k_with_set_reset) {
    ...
    TestStep(serial_load) {
      LogicBist {
        CoreInstance(w2_A) {
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 1023;
        }
      }
    }
  }

  // Run the first 1K patterns without the 12 set/reset patterns
  Patterns(first_1k_without_set_reset) {
    ...
    TestStep(serial_load) {
      LogicBist {
        CoreInstance(w2_A) {
          run_mode : run_time_prog;
          begin_pattern : 12;
          end_pattern : 1047;
        }
      }
    }
  }

  // Run the first 12 patterns to test only the sets/resets
  Patterns(set_reset_only) {
    ...
    TestStep(serial_load) {
      LogicBist {
        CoreInstance(w2_A) {
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 11;
        }
      }
    }
  }
}
```

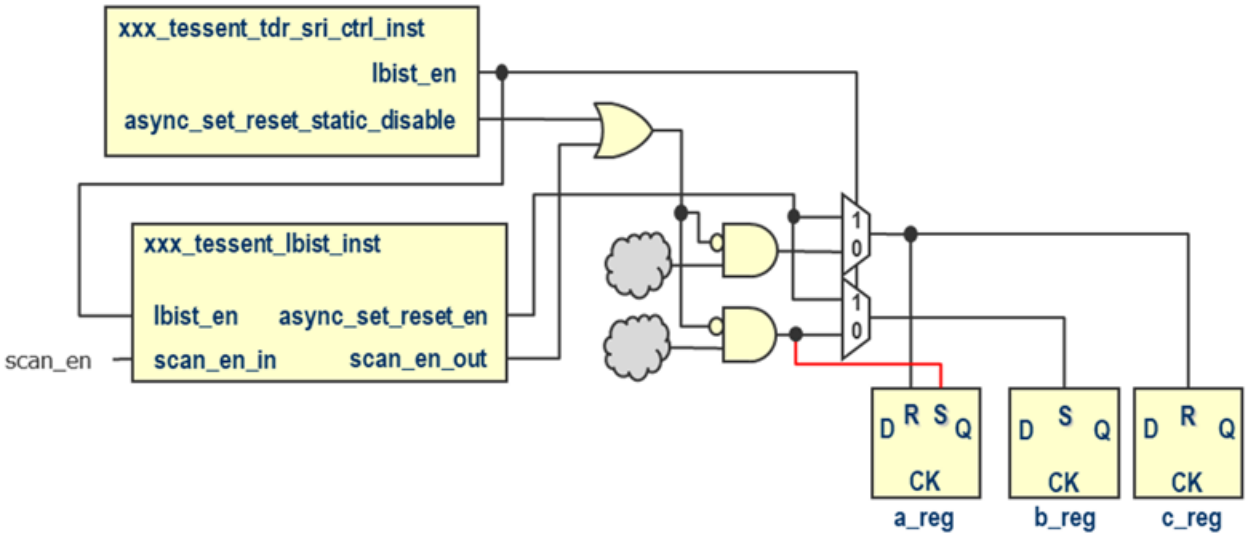
Limitations

The testing of asynchronous set/reset pins during logic BIST does not currently support flip-flops that have both a set and a reset pin. The testing drives both set and reset pins with the same enable signal on the logic BIST controller, which means that both pins are active at the same time. Therefore, the flip-flop model determines which signal has priority.

Consider the example below, where there are three flip-flops: a_reg has a set and reset, b_reg has only a set, and c_reg has only a reset. After running process_dft_specification, a_reg has both of its set and reset pins driven by the same source when logic BIST is running.



This results in errors during logic BIST fault simulation because the simulation sees both the set and reset pin of a_reg active at the same time. To prevent this error, edit the design to move the a_reg set or reset pin to be driven by the lbist_en mux i0 source, as shown below:



Single Chain Mode Logic

In the hybrid EDT/LBIST flow, LBIST pattern diagnosis is performed by loading values by the PRPG and shifting out the captured scan cell values. To enable this shift out, the tool inserts single chain mode logic that concatenates all the short scan chains across all the blocks. This logic is built by concatenating the EDT single bypass chain across all the EDT blocks.

By default, the tool inserts the logic for single chain mode during IP generation.

The single chain mode logic disables EDT bypass and single bypass pins to ensure that they do not interfere with LBIST operation. During diagnosis, a TDR control bit within the logic provides access to the single chain mode for scan chain unload.

Note

When you disable either EDT bypass or EDT single bypass in any EDT block, the tool does not generate the single chain mode logic. You can turn off bypass and single bypass by using the BypassChains wrapper:

```
DftSpecification(module_name, id)
  EDT {
    Controller(id) {
      BypassChains {
        present : off ;
        single_bypass_chain : off ;
      }
    }
  }
}
```

Caution

Setting these properties to off is not recommended because doing so also turns off LBIST pattern diagnosis.

Controller Chain Mode

For some safety-critical applications, you must test the test logic itself—that is, the LogicBIST and EDT IP blocks—in addition to testing the core design logic. Controller chain mode (CCM) enables you to generate ATPG patterns that target the hybrid EDT/LBIST blocks and LBIST controller in addition to the single chain mode logic and In-System Test controller.

You can generate CCM patterns within both the specification-based and dofile flows. The specification-based flow is described in this section.

Using tck as the Clock Source Rather Than Default test_clock

The hybrid EDT/LBIST controller has a 3-to-1 clock mux that sets shift_clock_src, test_clock, and tck as the runtime clock for LogicBIST test. By default, the tool configures this clock mux to use test_clock during scan insertion and fault simulation.

You can use either test_clock or tck as the clock when implementing CCM. Specify the clock at the time of IP generation. For example, select the tck clock as follows:

```
DftSpecification(module_name, id) {
  LogicBist {
    Controller (id) {
      ControllerChain {
        clock : tck ;
      }
    }
  }
}
```

When you configure CCM to use tck, perform scan insertion for the CCM mode chains in a separate run from regular scan cell insertion. When inserting the CCM mode chains, configure the LogicBIST clock mux to choose tck as the clock by setting the controller_chain_mode DFT signal to 1. This avoids a situation in which the source of the edt_lbist_clock at the LogicBIST controller output differs between scan insertion and pattern generation for CCM mode.

When you implement CCM with the default, test_clock, the scan insertion view remains consistent with the CCM pattern generation view, so a separate CCM mode scan insertion run is not required.

Usage Details

To enable the generation of CCM logic, specify the following property in the LogicBist wrapper:

```
DftSpecification(module_name, id) {
  LogicBist {
    Controller (id) {
      ControllerChain {
        present: on ;
      }
    }
  }
}
```

You can modify the names of CCM-specific ports using the DftSpecification/EDT/ ControllerChain wrapper, as shown in Figure 2-12 on page 51.

As described in "RTL and Scan DFT Insertion Flow With Hybrid TK/LBIST" in the *Tessent Shell User's Manual*, in the configuration-based flow, you insert the hybrid IP before performing scan chain insertion. By default, the tool inserts CCM scan segments for each instrument but does not connect them into one chain. This enables flexible scan chain stitching during scan insertion.

During scan insertion, configure the controller chains by creating a controller chain scan mode with the add_scan_mode command. This scan mode should only contain the CCM scan segments as valid scan elements.

The CCM scan segments in Tessent IP cores are inactive by default in order to not be confused with standard scan elements (design flops, segments on sub_blocks, and so on). Therefore, you must activate a controller_chain_mode scan mode on Tessent IP instances before adding them to the scan mode population. You should also reset the active child scan mode after adding CCM scan mode. You can perform these tasks using the DftSpecification/LogicBist/ ControllerChain wrapper.

Note

To benefit from flow automation, use the controller_chain_mode DFT signal as a scan mode enable signal. See add_dft_signals in the "*Tessent Shell Reference Manual*" for more information.

Example 2-1. DftSpecification Wrappers to Enable Controller Chain Mode

```
DftSpecification(module_name, id) {
  LogicBist {
    Controller(id) {
      ControllerChain {
        segment_per_instrument : on;
        present : on;
        clock : tck;
        max_segment_length : unlimited;
      }
    }
  }
}
```

You can generate multiple CCM chains within the EDT or LBIST controller, using a specified maximum length to guide the CCM scan chain stitching during IP creation.

Note

The minimum max_segment_length is 32.

For details about scan chain insertion when you are using the configuration-based flow, refer to "Perform Scan Chain Insertion" in the *Tessent Shell User's Manual*.

For information about generating CCM patterns in the configuration-based flow, refer to "Performing Pattern Generation for CCM in the TSDB Flow" on page 104.

To connect the CCM scan segments into one controller scan chain during IP generation, set the segment_per_instrument property to "off."

To enable generation of CCM logic, set the CCM mode to “on” in the ControllerChain wrapper:

```
DftSpecification(module_name, id) {
  LogicBist {
    Controller(id) {
      ControllerChain {
        present : on;
      }
    }
  }
}
```

By default, the tool inserts CCM scan segments for each instrument but does not connect them into one chain. You perform scan chain insertion prior to inserting the IP, which means that if you want to configure the controller chains, you must perform an incremental scan insertion to create the CMM scan chain from the segments.

To automatically generate the EDT/LBIST IP with a single RTL scan chain assembled from the CCM scan segments in each IP block, set the segment_per_instrument property to “off”:

```
DftSpecification(module_name, id) {
  LogicBist {
    Controller(id) {
      ControllerChain {
        segment_per_instrument : off ;
      }
    }
  }
}
```

Use the ControllerChain and Connections wrappers to define port names and other properties. The complete definition of these wrappers is as follows:

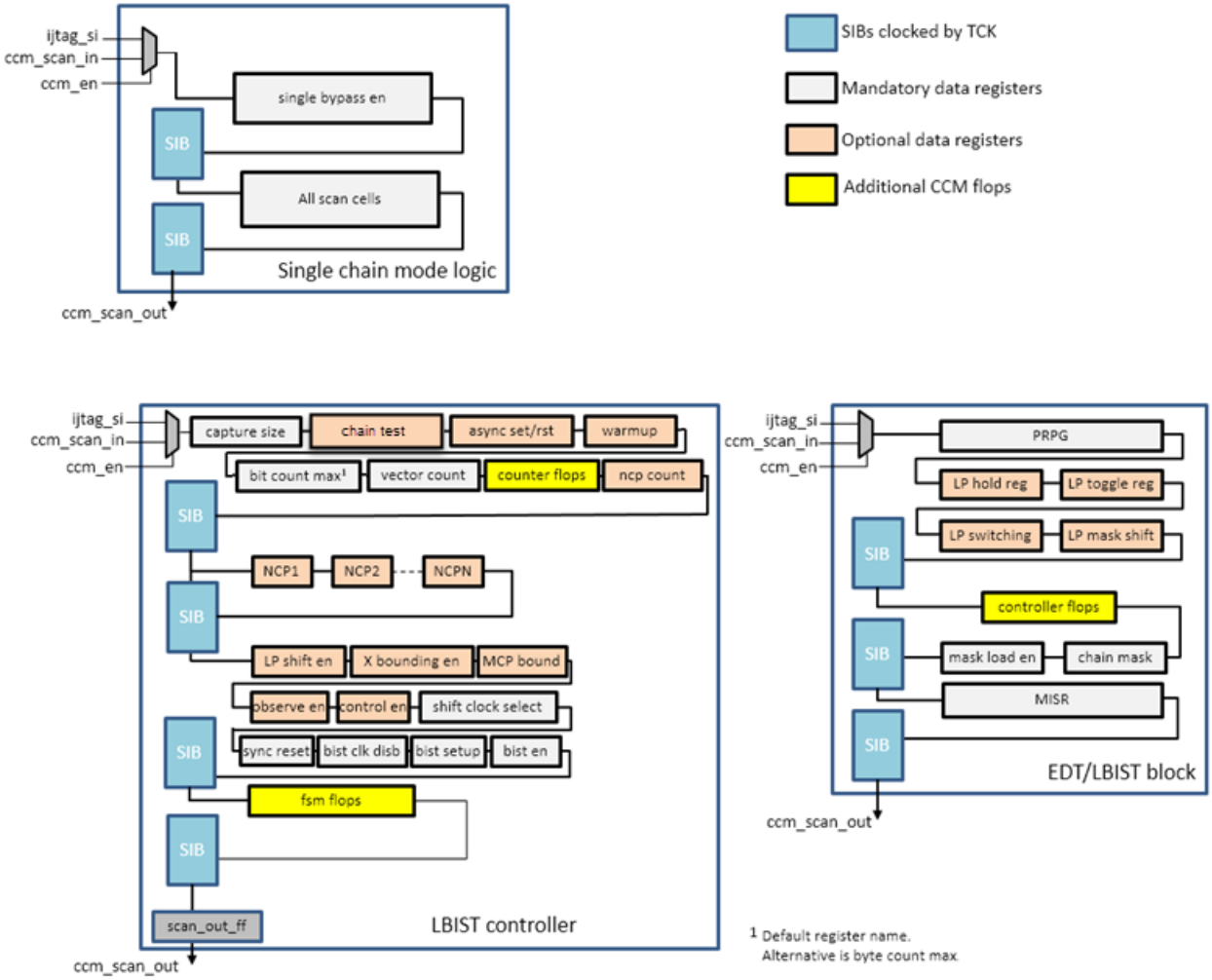
Figure 2-12. ControllerChain and Connections Wrappers

```
DftSpecification(module_name,id) {
  EDT {
    ControllerChain {
      present : on | off ;      // default: off
      clock   : enum ;          // legal: tck | edt_clock
      segment_per_instrument : on | off ; // default: on
      Interface {
        enable   : port_name ; // default: ccm_en
        scan_in  : port_name ; // default: ccm_scan_in
        scan_out : port_name ; // default: ccm_scan_out
        scan_en  : port_name ; // default: ccm_scan_en
      }
      Connections {
        scan_en                   : port_pin_name ;
                    // default: OptionalDftSignal(scan_en)
        controller_chain_enable   : port_pin_name ;
                    // default: OptionalDftSignal(controller_chain_mode)
        controller_chain_scan_in  : port_pin_name ;
                    // default: control_chain_%s_scan_in
        controller_chain_scan_out : port_pin_name ;
                    // default: control_chain_%s_scan_out
      }
    }
  }
  LogicBist {
    Controller(id) {
      ControllerChain {
        present                 : on | off ;
        clock                   : enum ; // legal: tck | edt_clock
        segment_per_instrument  : on | off ;
        max_segment_length      : int | unlimited; // int >= 32
      }
      Interface {
        ControllerChain {
          enable   : port_name; // default: ccm_en
          scan_in  : port_name; // default: ccm_scan_in
          scan_out : port_name; // default: ccm_scan_out
          scan_en  : port_name; // default: ccm_scan_en
        }
      }
      Connections {
        controller_chain_enable   : port_pin_name ;
        controller_chain_scan_in  : port_pin_name ;
        controller_chain_scan_out : port_pin_name ;
      }
    }
  }
}
```

Controller Chain Mode Architecture

The default inserted design architecture is as follows:

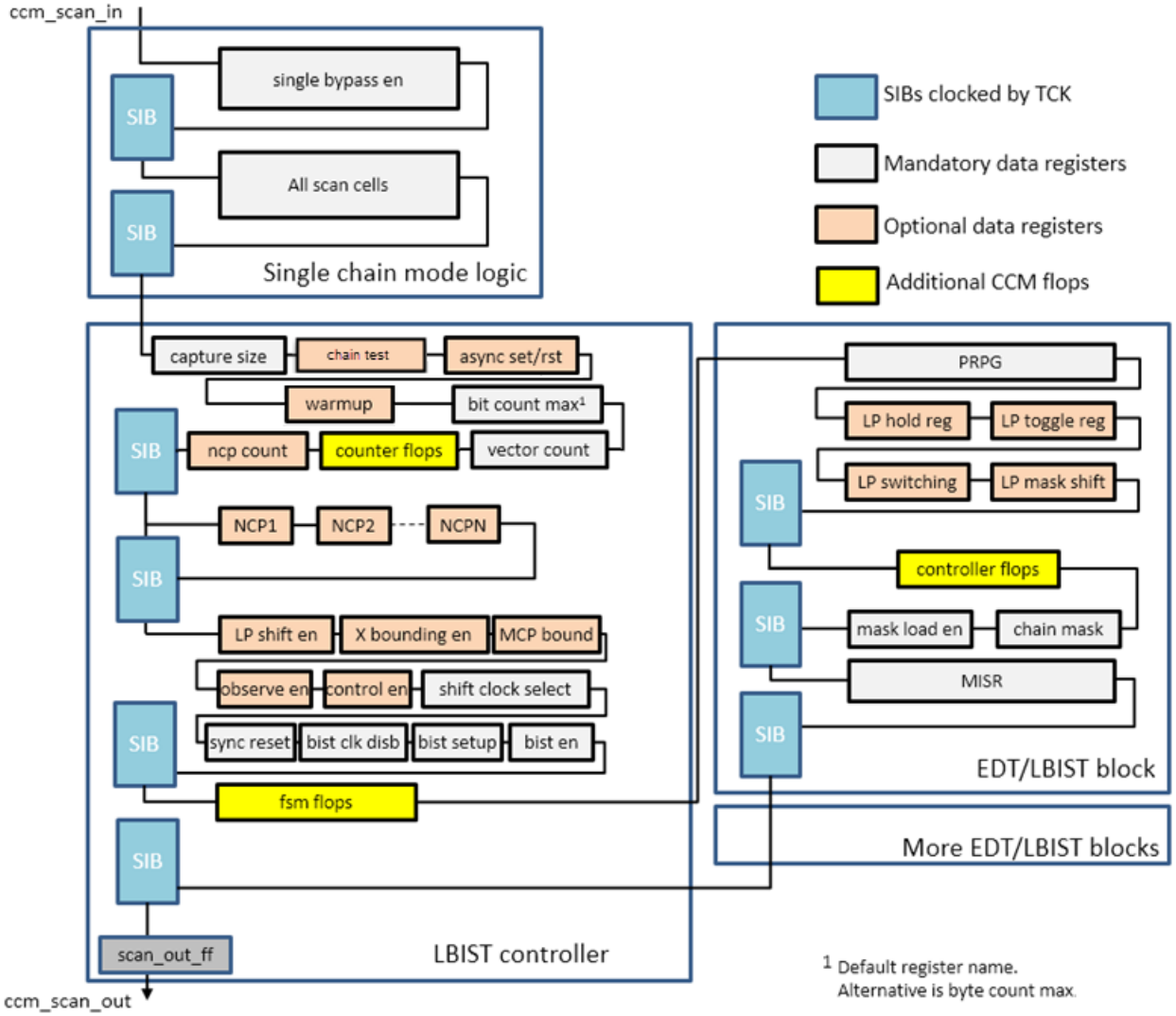
Figure 2-13. EDT/LogicBIST-Inserted Design for CCM, Segmented Scan Chains



The tool inserts CCM scan in and CCM scan out ports, and muxes controlled by the CCM enable signal, into each IP instrument that contains CCM scan segments. After IP insertion, the ccm_scan_in and ccm_en ports are tied to 0, and the ccm_scan_out is unconnected. The tool internally generates the ccm_scan_en signal within in each IP block by AND'ing the ccm_en and scan_en signals. The input port (ccm_en) enables the CCM. The tool reuses the existing ATPG scan enable for CCM.

When you set the property segment_per_instrument to “off” so that the tool automatically connects the scan segments into one controller scan chain, the inserted architecture design is:

Figure 2-14. EDT/LogicBIST-Inserted Design for CCM, Non-Segmented Scan Chains



CCM uses the existing ICL network chain that already contains most of the flops in the controllers. Flops that are not in the ICL network are multiplexed into it during the CCM shift. The result is a single chain that is accessible through a new scan input (ccm_scan_in) and output (ccm_scan_out).

Related Topics

Performing Pattern Generation for CCM in the TSDB Flow

Performing Pattern Generation for the Dofile Flow

IJTAG Network in EDT/LogicBist IP

SIBs are a mechanism to provide flexible access to data registers they control. For the Tessent version of a SIB, the data registers are accessible via the IJTAG network when the SIB controlling a data register is set to 1, and the data register is bypassed when the SIB is set to 0.

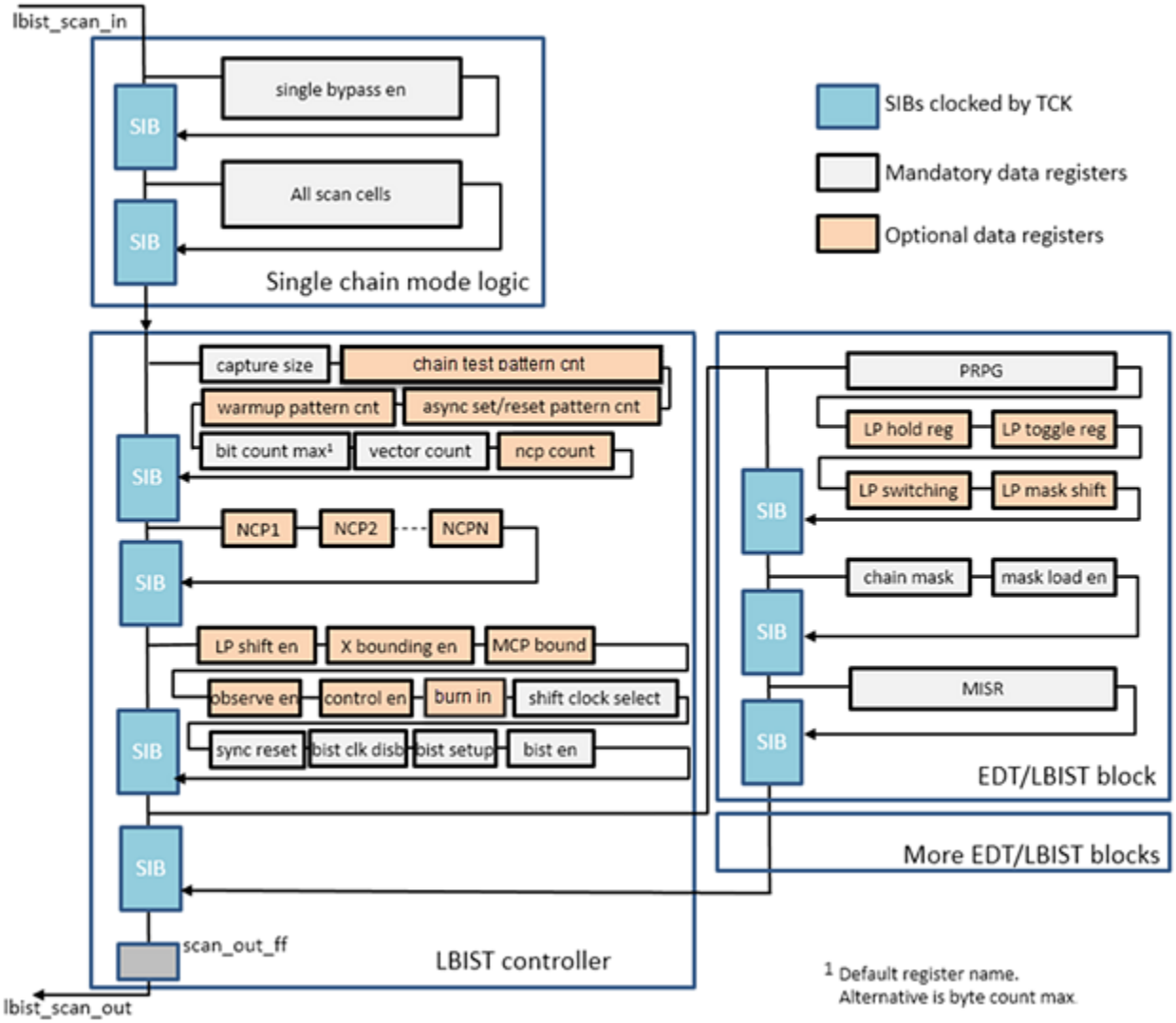
SIBs are clocked by TCK. To provide flexibility in accessing specified registers in the IJTAG network at different times, the SIBs are arranged in a hierarchical tree structure. The top-level SIBs use the lbist_tap_instruction signal from the JTAG LogicBIST instruction as the SIB enable signal. The SIB enable signals for the child SIBs are controlled by their parents. This is how all the SIBs inside the LogicBIST controller, EDT/LogicBIST blocks and single chain mode logic are configured.

The EDT/LogicBIST blocks contain SIB registers to provide access to various registers, such as the PRPG, the EDT chain mask register, MISR, and programmable NCP count registers. The EDT SIBs are clocked by tck, and the data registers they control are clocked by edt_clock. The tool adds a lockup cell to avoid clock skew between these two clock domains. A SIB inside the LogicBIST controller controls access to the EDT SIBs. The enable output of this LogicBIST controller SIB is used as the input enable for the EDT SIBs.

For each specified NCP, an 8-bit register is created and inserted on the ICL network. These registers are loaded at run time so that the number of patterns applied for the NCP is programmable. When an integer percentage is provided during IP generation, the NCP registers reset to those values when sib_reset is asserted. Otherwise, these registers are reset to equal percentages across all NCPs. For more information, see “Generating the EDT and LogicBIST IP” on page 64.

Figure 2-15 shows the IJTAG network in a hybrid EDT/LBIST-inserted design.

Figure 2-15. SIBs Insertion and Integration of Cores for Concurrent Flow



Burn-In Test Mode

The hybrid TK/LBIST flow supports wafer-level burn-in tests for use during board-level testing. When you enable burn-in test mode, the tool creates additional RTL to enable the LogicBIST IP to run for longer periods of time than is normally enabled by the size of the LogicBIST pattern counter.

Enable burn-in mode by specifying the burn_in property, as defined here:

```
DftSpecification(module_name, id) {
  LogicBist {
    Controller(id) {
      burn_in : bool ; // default: off {symbols: on off}
    }
  }
}
```

When using the pattern specification in the TSDB flow, enable burn-in mode by setting the run_mode property to burn_in and specifying the burn_in_time property, as defined here:

```
PatternsSpecification(design_name, design_id, pattern_id) {
  Patterns(patterns_name) {
    TestStep(step_name) {
      LogicBist {
        CoreInstance(icl_instance_name) {
          run_mode     : burn_in ;
          burn_in_time : tvalue ;     // a time value in seconds
        }
      }
    }
  }
}
```

In burn-in mode, the hardware prevents the LogicBIST controller from reaching an end state in its FSM, which enables the PRPG to run continuously for the specified amount of time. Because the burn-in runtime can easily exceed the pattern counter, no MISR comparisons are performed when the test completes.

Burn-In Only Fault Simulation

For fault simulation, you can create multiple burn-in configurations, such as different power configurations or decompressor seeds, by running a reduced fault simulation for the purposes of just creating the data needed to initialize/seed the LogicBIST registers.

To (re-)run fault simulation to generate a dedicated burn-in mode pattern set, do the following:

1. Change the current mode (set_current_mode).
2. Reduce the number of patterns to simulate (set_random_patterns).
3. Make any other changes as needed, such as changing the power configuration or using a different decompressor seed.

LBIST Controller Hardware Default Mode

The hardware default run mode of the LBIST controller uses the default (hardcoded) settings specified at IP generation (process_dft_specification). Because the settings are all built into the

hardware, this run mode requires the least amount of time to set up the controller for launching the test.

This section documents considerations for configuring hardware default mode.

Configuration of the Hardware Default Values

Use the DftSpecification/LogicBist/Controller wrapper to access sub-wrappers that specify hardware_default properties for the controller IP:

- ShiftCycles
- CaptureCycles
- PatternCount
- AsyncSetResetPatternCount
- WarmupPatternCount

Register Initialization

Hardware default mode uses the least initialization for manufacturing patterns of any of the run modes. When you run hardware default mode, the LBIST controller synchronously resets the PRPG, MISR, pattern_count, and other registers to their hardware_default values. The controller performs this reset as part of its “run” operation. Because the controller itself initializes these registers (by the synchronous reset), the pattern runtime is reduced because there is no need to load those registers with an initial value through the IJTAG network.

Compatibility Restrictions

Running hardware default mode is possible only when hardware_default values hardcoded in the IP match fault simulation settings. When incompatible, hardware default mode pattern generation is not possible, although regular LBIST pattern generation is always possible.

For example, if you specify the following for IP generation:

```
DftSpecification(module_name, id) {
  LogicBist {
    Controller(id) {
      ShiftCycles {
        max : 100;
        hardware_default : 30;
      }
    }
  }
}
```

But after scan insertion, perhaps due to an ECO, some scan chains have a length larger than 30 flops. In this case, you cannot run hardware default mode because the controller's hardcoded shift cycle length is 30.

Compatibility Reporting

You can detect compatibility problems with your controller's configuration that prevent the controller from being able to run in hardware default mode. Report compatibility with the "report_lbist_configuration -hardware_default_compatibility" command during fault simulation. The command compares your hardware_default settings with the values specified during fault simulation, and reports which settings are compatible. For some parameters, like NCP count, the values have to exactly match to be considered compatible. Other parameters, like pattern count, are compatible when more than the hardware_default number of patterns are fault-simulated.

The "shift length" setting is hard to predict during IP creation, but you can use a reasonable upper bound to set the hardware_default value. When the actual (Fault Simulation) value is less than this value, you can use the set_number_shifts command to make the actual value compatible with the hardware_default value. When the actual value is greater than the hardware_default value, running in hardware default mode is not possible.

Self-Test Mode

The hybrid TK/LBIST flow supports self-testing of the LogicBIST and EDT controllers. This enables you to improve latent fault metrics (LFM) in conformance with ISO26262 by generating an InSytemTest pattern set or a manufacturing pattern set to test the EDT/LBIST controller in a stand-alone mode. You can also use this pattern set to perform an RTL simulation and validation of the embedded EDT/LBIST controller when the design RTL is ready.

In Self-Test Mode, the LBIST controller tests itself by directly feeding PRPG data into MISRs. The FSM of the LBIST controller goes through all the possible states that it would during a normal LBIST run. This confirms that the controller is operating correctly whether or not your design is in place.

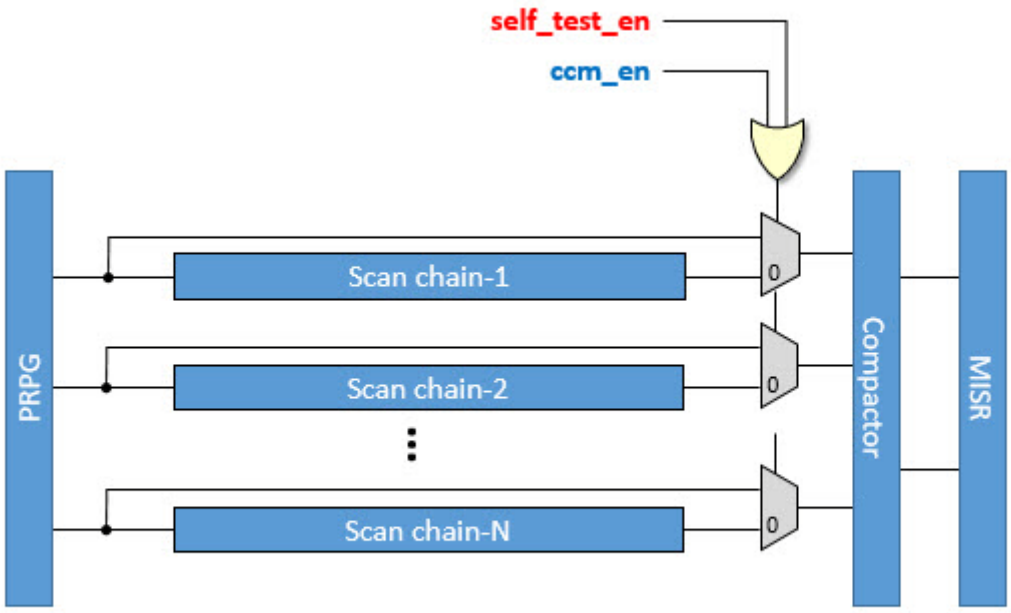
IP Generation for Self-Test Mode

To perform a fast and thorough test of the LBIST controller, the PRPG inputs are connected directly to the compactor inputs to bypass the scan chain structures. As in a regular LBIST run, the pattern data and signatures are generated from the PRPG and the MISR. The difference is that the internal scan chains are bypassed for fast operation (pattern data generated from the PRPG is fed directly into the MISR).

For Controller Chain Mode (CCM), the scan chains are bypassed under the control of the CCM enable signal (ccm_en in the figure below) using existing muxes.

The scan chains are also bypassed during self-test. When CCM is available, the same muxing can be reused with an additional enable signal to enable and disable the self-test, as shown in the following figure.

Figure 2-16. LBIST Controller Self-Test



When CCM is not available, the bypass muxes are added along with the self-test enable signal.

Note

Controller Chain Mode is not required for Self-Test Mode.

The self-test enable signal comes from a TDR. When the self-test signal is enabled, pattern data from the PRPG is fed directly into the MISR. The self-test runs in a manner similar to that of a normal LBIST run mode, except that the scan chains are bypassed. This requires a different run mode. During self-testing, the PRPG and MISR behave similarly to a regular LBIST run, and the LBIST controller's FSM goes through all normal LBIST tests. These tests cover most of the registers in the LBIST controller.

Note

Although NCP limits and NCP counters are covered during the operation of the LBIST controller FSM, the corresponding NCP for a specific pattern cannot be verified by the self-test. Any defect in the NCP index logic that propagates to the OCC is covered during Logic BIST and the monitoring of the clock outputs during system operation.

Also, this self-test is not capable of verifying the LBIST controller's output signals, such as scan_en, xbounding_en, mcp_bounding_en, control_point_en, observe_point_en, and lbist_async_set_reset_en. These signals must be separately targeted by adding redundancy and

ensuring that a Single Event Upset (SEU) doesn't affect the functional operation of the design. This is beyond the scope of this feature and is covered separately.

Self-Test Pattern Generation

Self-test patterns can be generated based on either an RTL view (during or after IP creation) or on a gate-level netlist.

Note

Self-test pattern generation can only be performed when the Self-Test hardware is available.

During specification-based pattern creation, the create_patterns_specification command creates the Self-Test mode if this feature is available in hardware. You can specify the information for processing pattern specifications for self-test patterns or use the default values for pattern count, shift length, and capture length. You can reconfigure this information, if needed. You can also specify the warmup pattern count, low_power settings, and chain masks for self-test patterns. All of this information can be specified in the pattern specification configurations.

When pattern specifications are processed, the Self-Test patterns are simulated and generated based on the given pattern specification configurations, and stored in the TSDB. You can create multiple sets of pattern data by using different configurations. Whenever the settings (such as shift length, capture length, low power settings, chain mask settings, or pattern count) are changed, new pattern data is created. For new settings specified with a lower pattern count, process_patterns_specification uses the existing pattern data.

Note

Self-Test pattern generation does not require fault simulation results.

See Performing Pattern Generation for the TSDB Flow for an example of how to generate self-test patterns during IP creation.

Related Topics

create_patterns_specification [Tessent Shell Reference Manual]

Performing Self-Test Pattern Generation During IP Creation

This example illustrates the generation of Self-Test patterns during IP creation in the -rtl context.

Procedure

1. Set the context:

   SETUP> set_context dft -rtl

2. Read the design RTL:

   SETUP> read_verilog ../data/piccpu_rtl.v

3. Read the library file:

   SETUP> read_cell_library ../library/adk_complete.tcelllib

4. Elaborate the design:

   SETUP> set_current_design piccpu

5. Define the DFT signals:

   SETUP> add_dft_signals ltest_en

6. Transition to analysis mode:

   SETUP> set_system_mode analysis

7. Create the DFTSpecification:

   ANALYSIS> create_dft_specification -sri_sib_list {occ edt lbist}

8. Apply commands to customize the DFTSpecification. For example, using read_config_data to enable the self-test hardware:

```
ANALYSIS> read_config_data -in_wrapper \
DftSpecification(piccpu,rtl) -from_string {
  LogicBist {
    ijtag_host_interface : Sib(lbist);
    Controller(c0) {
      ShiftCycles   { max : 40; }
      CaptureCycles { max : 4; }
      PatternCount  { max : 10000; }
      NcpOptions    { count : 3; }
      self_test : on;
    }
  }
}
```

9. Process the DFTSpecification for IP insertion:

   ANALYSIS> process_dft_specification

10. Find all modules and their ICL descriptions:

    SETUP> extract_icl

11. Define the top-level clocks. For example:

    SETUP> add_clocks 0 clk -period 10

12. Generate the test patterns:

    ```
    SETUP> create_patterns_specification
    ```

13. Specify or change any pattern specification requirements. For example, the self-test shift_length option can be specified as follows:

    ```
    SETUP> set_config_value /PatternsSpecification(piccpu,rtl,signoff)/
    Patterns(LogicBist_piccpu)/TestStep(self_test)/LogicBist/CoreInstance(.)/
    SelfTestOptions/shift_length 10
    ```

14. Process and validate the test patterns:

    ```
    SETUP> process_patterns_specification
    ```

15. Run and check the testbench simulations. For example:

    ```
    SETUP> set_simulation_library_sources -v ../library/Verilog/adk.v -v ../data/picdram.v

    SETUP> run_testbench_simulations

    SETUP> check_testbench_simulations -report_status
    ```

Generating the EDT and LogicBIST IP

To generate the hybrid EDT/LBIST IP, you must create a LogicBIST wrapper, which specifies the information used by the process_dft_specification command to build the hybrid IP and insert the IP into the design.

You can also use the dofile flow to generate the EDT and LogicBIST IP. See "The Dofile Flow" on page 263 for details.

Once you have nested the LogicBIST wrapper within the DftSpecification wrapper, you use Tessent Shell to validate and process the wrapper, and create the hybrid IP. The flow is the same as that used for EDT, with the addition of the LogicBIST wrapper. Refer to "Validating the EDT Specification and Creating the EDT IP" in the *Tessent TestKompress User's Manual* for details.

The high-level syntax for the LogicBIST wrapper looks as follows:

```
DftSpecification(module_name,id) {
   LogicBist {
    ijtag_host_interface : name
    Controller(id) {           // configure the LBIST controller
      ...
      ControllerChain {        // configure controller chain mode
        ...
      }
      SingleChainForDiagnosis { // configure single chain mode logic
        ...
      }
      ShiftCycles {             // configure shift cycles
        ...
      }
      CaptureCycles {           // configure capture cycles
        ...
      }
      PatternCount {            // configure the pattern counter
        ...
      }
      WarmupPatternCount {      // configure warm-up pattern count
        ...
      }
      AsyncSetResetPatternCount { // configure asynchronous set/reset
        ...                       // pattern count
      }
      NcpOptions {              // configure NCP-related hardware options
        ...
      }

      Interface {               // specify interface port names
        ...
      }
      Connections {             // configure controller connections
        ...
      }
    }
    NcpIndexDecoder {           // convert LogicBIST NCP index output into
      ...                       // clock sequences for Tessent OCC usage
    }
  }
}
```

Interactions

When you specify the LogicBist/Controller wrapper, the following interactions apply:

- **EDT**. By default, all EDT controllers are converted into EDT/LBIST hybrid controllers. To create both hybrid and non-hybrid controllers in the same process_dft_specification run, set the property DftSpecification/EDT/Controller/LogicBistOptions/present to “off.”

- **OCC**. OCC signals are intercepted by the LogicBIST controller. For proper LogicBIST support, generate the OCC with capture enable trigger and static clock control enabled. When you generate OCC in the same pass as the LogicBIST controller, these properties are automatically configured to support LogicBIST.
- **NCP Index Decoder.** When you specify the NCP Index Decoder and the LogicBIST controller within the same DftSpecification, the tool automatically populates the count for NcpOptions from the count specified by the NcpIndexDecoder/Ncp wrapper. You cannot manually specify the NcpOptions count parameter. You should use the count when the NCP Index Decoder is implemented in a different DftSpecification, or when not using a decoder (that is, when the number of NCPs is 1).

Limitations

The following limitations apply when you are using the LogicBist wrapper to generate the hybrid IP:

- When the NcpIndexDecoder and LBIST controller are generated in the same pass, they are automatically connected. When not inserted together, the user is responsible for the connections.
- LogicBIST does not support child OCCs.
- Pre-inserted mini-OCCs, such as those created during MemoryBIST insertion or with boundary scan, may not be automatically intercepted. Ensure that you have explicitly specified these OCCs by using the add_core_instances command.
- For scan-chain clocking, input design scan chains must have first scan cell clocked by an LE clock edge and last scan cell clocked by an TE clock edge.
- Does not support Low Pin Count Controller.
- You must specify the LogicBist/Controller wrapper with its associated EDT wrappers, and you can only specify one LogicBIST controller. You must use the LogicBIST controller in conjunction with EDT to generate hybrid IP.

Dual Compression Configurations for the Hybrid IP

Using two compression configurations when setting up the EDT logic enables you to easily set up and reuse the EDT logic for two different test phases. For example, wafer test versus package test.

You can use hybrid IP with dual configuration in EDT mode in either the low or high compression configurations (similar to non-hybrid EDT IP). For the LBIST mode of operation, the IP is used in the low compression configuration. Specifically on the output side, the MISR input taps are always taken from the low compression spatial compactor.

For more information about dual compression configurations, see "Dual Compression Configurations" in the *Tessent TestKompress User's Manual*.

The following example IP creation dofile enables dual compression configurations, where "LC" is the low compression configuration that uses 8 channels and "HC" is the high compression configuration that uses 2 channels. The add_edt_configurations command enables the dual compression functionality.

```
# Set the context to insert DFT into RTL-level design
set_context dft -rtl -design_id rtl1

# Set the location of the TSDB. Default is the current working directory.
set_tsdb_output_directory ../tsdb_outdir

# Read the design
read_verilog <design>
read_design
set_current_design gps_baseband

# Set the design level as physical_block
set_design_level physical_block

# Use this command to report all DFT signals
report_dft_signal_names

# Add DFT Signals : #Required  for hybrid TK/LBIST
add_dft_signals

# Add input constraints necessary for the design
report_dft_signals

# Specify pre-DFT DRC rules
set_dft_specification_requirements -logic_test on
add_clocks clk -period 2ns
add_clocks lbist_shift_clk -period 8ns

# System mode transition, run DRCs
check_design_rules

# Create and report a DFT Specification
set spec [create_dft_specification -sri_sib_list {edt occ lbist} ]
report_config_data $spec
```

```
# Use report_config_syntax DftSpecification/edt|occ to see full syntax
read_config_data -in $spec -from_string {

# Modify the below specification for your specific design requirements
read_config_data -in $spec -from_string {
  EDT {
    ijtag_host_interface : Sib(edt);
    Controller (c1) {
      longest_chain_range : 50, 65;
      scan_chain_count : 60;
      input_channel_count : 2;
      output_channel_count : 2;
      HighCompressionConfiguration {
        present : on | off ;
        input_channel_count : int ;
        output_channel_count : int ;
      }
      LogicBistOptions {
        misr_input_ratio : 1 ;
        ShiftPowerOptions {
          present : on ;
          default_operation : disabled ;
          SwitchingThresholdPercentage {
            min : 25 ;
          }
        }
      }
    }
  }
}
```

```
report_config_data $spec
read_config_data -in $spec -from_string {
  LogicBist {
    ijtag_host_interface : Sib(lbist);
    Controller(id) {
      burn_in : on ;
      pre_post_shift_dead_cycles : 8 ;
      SingleChainForDiagnosis {
        Present : on ;
      }
      ControllerChain {
         present : on;
         clock : tck;
      }
      Connections {
        shift_clock_src:lbist_shift_clk;
      }

      NcpOptions {
        count : 1;
      }

      ShiftCycles { max :200 ; }
      CaptureCycles { max : 7; }
      PatternCount { max : 1024; }
      WarmupPatternCount { max : 128; }
    }
  }
}

process_dft_specification

## The ICL for the entire design gets created
extract_icl

##====== Synthesis and simulation
write_design_import_script  for_dc_synthesis.tcl -replace

# Make patterns
create_pattern_specification
process_pattern_specification

# Run Simulation
run_testbench_simulations
```

Timing Constraints for EDT and LogicBIST IP

Tessent Shell generates timing constraints for the Hybrid EDT/LogicBIST IP during IP generation in Synopsys Design Constraints (SDC) format. The SDC file contains timing constraints and exceptions for all modes of operation of the IP.

For the TSDB flow, the tool creates the SDC for the current design when you execute the extract_icl command, which does the following:

- Stores the SDC in Tcl procs.
- Writes the SDC to a single file named design_name.sdc, where design_name is the name of the current design loaded in Tessent Shell.

The SDC file is located next to the extracted ICL, which is typically in the dft_inserted_designs directory as follows:

```
${tsdb}/dft_inserted_designs/${design_name}_${design_id}.dft_inserted_design
```

Every instrument type (for example, MBIST, IJTAG) of the current design and all sub-blocks that provide SDC constraints are represented by a separate proc in the SDC file. Constraints for logic test-related instruments such as OCC, EDT, or LBIST, including logictest-related DFT signals, are all grouped under similar placeholder “ltest” instrument procs.

Refer to “Timing Constraints (SDC)” in the *Tessent Shell User’s Manual* for details about specifying timing constraints for logic test instruments when you are using the TSDB flow.

The following sections describe how to generate timing constraints when you are using the dofile flow.

Timing Constraint Generation in the Specification-Based Flow

The tool generates timing constraints when you run the extract_icl command in the configuration-based flow.

The tool generates SDC for the following Hybrid IP modes:

- EDT — EDT shift, slow and fast capture. The capture mode constraints are the same for EDT and EDT-bypass.
- EDT-bypass — Bypass shift, slow and fast capture. The capture mode constraints are the same for EDT and EDT-bypass.

- LogicBIST — LogicBIST setup, shift, capture and diagnostic modes. The diagnostic mode constraints are generated only when single chain mode logic is synthesized in the IP.

LogicBIST Timing Constraints

In the dofile flow, the four LogicBIST modes have their own lists of timing constraints/ exceptions.

LogicBIST Setup Mode

This mode is for seeding the BIST registers in the EDT blocks like PRPG, MISR and EDT chain mask registers as well as those in the LogicBIST IP like vector counter, shift counter etc. The BIST controller is clocked by tck for this mode of operation.

The TDI and TDO pins of the TAP controller are used for seeding, hence IO delays are defined for those pins. These delays are defined with respect to virtual clocks named force_pi and measure_po that reflect the timing described in the test procedures.

The following constraints/exceptions are specified for the LogicBIST Setup mode:

- Set LogicBIST enable TDR bit to 1. Set the BIST setup registers to “001” corresponding to the LongSetup mode of operation for the BIST controller.
- Set false paths from EDT control signals that are not used like EDT reset and EDT update.
- Set false paths from EDT channel input pins and to EDT channel output pins.
- Set false paths through TAP controller's LogicBIST instruction enable and test logic reset outputs. The TAP controller paths for shift, capture and update are enabled.
- EDT chain mask registers are active in this mode. The paths from these registers to the design scan cells, specifically the hold paths, where the source and destination are on different clock domains are not explicitly disabled. This works correctly because the destination design scan cells are not clocked in this mode.
- Set variables for sequentially propagating case analysis constraints through the user clock controller.
- Disable clock gating checks on the shift controller clock muxes. This is because only the tck path is selected in this mode.

LogicBIST Shift Mode

This mode describes scan chain shifting. The scan cells are driven by the internally generated LogicBIST shift clock and the PRPG/MISR are driven by the internally generated LogicBIST

clock. The BIST controller is clocked by the free running shift clock source. IO pins are not involved in this, hence no IO pin delays are specified.

The following constraints/exceptions are specified for the LogicBIST Shift mode:

- Set LogicBIST enable TDR bit to 1.
- Constrain the internally generated LogicBIST scan enable that reaches the scan cells to ON. There is a 4-cycle window around the time the scan enable changes during which the design scan cells are not clocked, which enables the scan enable to be described as a constant.
- Constrain the prpg_en, misr_en, and scan_en signals to MCP 4, and constrain the lbist_reset signal to MCP 3.
- Constrain the internally generated clock controller scan enable to ON.
- Set the clock source for LBIST test. The shift mode constraints TCL procedure takes a clock_select parameter to choose from LBIST shift clock source, EDT clock or TCK. The default is LBIST clock. To analyze timing when either EDT clock or TCK is used for logic test, call the top-level procedure with the required clock parameter. For example, lbist_shift_mode edt_clock. Both shift and capture modes run with the same clock, so clock_select should be consistent for shift and capture mode.
- Constrain or exclude EDT pins like EDT clock, update, reset and other control signals.
- Exclude all paths from the TAP controller.
- EDT chain mask registers are static throughout test, so declare all paths from these registers as false.
- Exclude all paths from SIBs inside the EDT logic. These SIBs are clocked by the faster lbist shift clock and hence excluded in this mode.
- Set variables for sequentially propagating case analysis constraints through the user clock controller.
- Turn off clock gating checks on the shift controller clock muxes. This is because only the free running shift clock source is selected in this mode.

LogicBIST Capture Mode

This mode describes the capture. The clocking for the scan cells is described in the Named Capture Procedures. The BIST controller is clocked by the free running shift clock source. IO pins are not involved in this mode, hence no IO pin delays are specified.

The following constraints/exceptions are specified for this mode:

- Set LogicBIST enable TDR bit to 1.

- Constrain the internally generated LogicBIST scan enable that reaches the scan cells to OFF. There is a 4 cycle window around the time the scan enable changes during which the design scan cells are not clocked, which enables the scan enable to be described as a constant.
- Constrain the prpg_en, misr_en, and scan_en signals to MCP 4, and constrain the lbist_reset signal to MCP 3.
- Constrain the internally generated clock controller scan enable to OFF.
- Set the clock source for LBIST test. The shift mode constraints TCL procedure takes a clock_select parameter to choose from LBIST shift clock source, EDT clock or TCK. The default is LBIST clock. To analyze timing when either EDT clock or TCK is used for logic test, call the top-level procedure with the required clock parameter. For example, lbist_shift_mode edt_clock. Both shift and capture modes run with the same clock, so clock_select should be consistent for shift and capture mode.
- Constrain or exclude EDT pins like EDT clock, update, reset and other control signals
- Exclude all paths from the TAP controller
- EDT chain mask registers are static throughout test, so declare all paths from these registers as false.
- Set variables for sequentially propagating case analysis constraints through the user clock controller
- Turn off clock gating checks on the shift controller clock muxes. This is because only the free running shift clock source is selected in this mode

LogicBIST Single Chain Mode

This mode describes shifting through the concatenation of the design scan cells through the single chain mode logic, used primarily for LogicBIST diagnostics. The BIST controller and the design scan cells are clocked by tck for this mode of operation. The TDI and TDO pins of the TAP controller are used for seeding, hence IO delays are defined for those pins. The delays are defined with respect to virtual clocks named force_pi and measure_po that reflect the timing described in the test procedures.

The following constraints/exceptions are specified for the LogicBIST Single Chain mode:

- Set LogicBIST enable TDR bit to 1. Set the BIST setup registers to 100 corresponding to the SingleChain mode of operation for the BIST controller.
- Constrain or exclude EDT pins like EDT clock, update, reset and other control signals.
- Set false paths from EDT channel input pins and to EDT channel output pins.
- Set false paths through TAP controller's LogicBIST instruction enable and test logic reset outputs. The TAP controller paths for shift, capture and update are enabled.

- EDT chain mask registers are static throughout test, so declare all paths from these registers as false.
- Set the single chain mode TDR bit to 1.
- Set variables for sequentially propagating case analysis constraints through the user clock controller.
- Turn off clock gating checks on the shift controller clock muxes. This is because only the tck path is selected in this mode.

ECO Implementation in the Hybrid TK/LBIST Flow

As part of the IP creation step, the tool ensures that every scan chain starts with a leading edge (LE) flip-flop and ends with a trailing edge (TE) flip-flop. The retiming flip-flops are exposed as part of the scan chains. You must make sure that the clocking of the boundary flip-flops never changes; if you need to add a flip-flop into the design, you must add it in the middle of the chain, not at the beginning or end.

The same is true for deleting a flip-flop. Only remove a scan cell that is in the middle of a chain. Do not remove a scan cell that is at the boundary because it controls the retiming latches that have been added.

In addition, consider the following:

- During IP creation, the tool generates a shift counter that is used during LogicBIST mode. By default, the tool adds seven more clock cycles to the number of shift cycles. This enables additional flip-flops to be added later through an ECO. Note that this is across all scan chains, so many flip-flops can be added in ECO mode if you want. However, you should not exceed seven per chain.

  Note

  This is true only when DftSpecification/LogicBist/Controller/ShiftCycles/counter_resolution is set to “byte” (the default is “bit”).

- You should specify the size of the pattern counter. Although not directly related to the ECO process, you should specify the hardware in such a way that you can double the number of LogicTest patterns that you think are necessary during IP generation.
- Handle timing exception paths after placement and routing as follows:

  Note

  Make sure you declare only the timing exception paths that are discovered late. Do not provide the entire SDC or the tool inserts muxes for FP/MCPs that are already bounded.

a. Use the “set_xbounding_options -xbounding_enable *pin_name* -mcp_bounding_enable *pin_name*” command to declare the existing enable pin that the tool should use to drive the X-bounding and MCP bounding logic. The enable signal must be constrained to 1 so that it activates the logic that was inserted in a previous path. The xbounding_enable is used for bounding the static X-source like primary input and the mcp_bounding_enable is used for bounding the timing exception path, that is, the dynamic X-source.

b. Use the “read_sdc new_exception.sdc” command to read the new timing exceptions. The new_exception.sdc includes only the FP/MCP newly added by ECO.

c. Use the “add_scan_chains” command as per the defined scan mode. Also, set the DFT signal to configure that particular scan mode. You may see S6 violations if the scan chains are not defined. It is easier to add all the scan chain by loading the atpg setup file.

d. Use the “add_nonscan_instances -all” command to remove any non-scan cells from consideration that the tool sees as scannable. At this point, the netlist is completely scanned, so any non-scan cells should remain as is.

e. Use the “set_system_mode analysis” command.

f. Use the analyze_xbounding command to determine where the muxes should be inserted. While this command analyzes whether any X-sources have been missed, none should have been at this stage.

g. Use the report_xbounding command to show where the muxes should be inserted to block those paths.

h. Either insert the muxes yourself as part of the ECO or use the insert_test_logic command to let the tool insert them.

Chapter 3
Test Point Analysis and Insertion, Scan Insertion, and X-Bounding

In this step of the flow, you read in a gate-level non-scan Verilog netlist of your design, generate and insert test points, insert scan chains, and perform X-bounding. You can optionally perform these tasks in separate sessions.

Use the synthesized gate-level netlist with the inserted hybrid TK/LBIST IP from the previous step for these tasks.

Test Point Analysis and Insertion, Scan Insertion, and X-Bounding Overview

In the test point analysis and insertion step of the hybrid TK/LBIST flow, you generate and insert test points into the netlist to achieve high test coverage.

During test point analysis and insertion, you add random pattern test points to certain locations in your design. By adding these test points, you can increase the testability of the design by improving controllability or observability.

Test Point Analysis and Insertion and Scan Insertion Flow

Use the following process to insert test points and scan:

- Load the design with the read_design command. The inputs are the synthesized gate-level netlist and the TSDB from the hybrid TK/LBIST insertion step.
- Specify the X-bounding options and settings.

- Analyze X-bounding.
- Specify scan modes, scan chain families, or scan segments.
- Analyze scan chains.
- Insert test points, X-bounding, scan chains, and (optionally) wrapper chains with the insert_test_logic command. This command also automatically updates the TSDB with the output files.

At the conclusion of this step, the tool writes out a netlist with test points and scan chains inserted, and upon which X-bounding has been performed. Optionally, wrapper analysis may also have been performed. This netlist and a Tessent Core Description (TCD) file are available in the TSDB directory and are linked to the unique design ID associated with this step.

Requirements

Performing test point analysis and insertion has certain requirements. You must adhere to the following:

- Test point insertion needs a gate-level Verilog netlist and a Tessent Cell Library.
- You can read in functional SDC so the tool omits adding any test points to multicycle paths or false paths—see the read_sdc command.
- The tool identifies potential scan candidates for correct controllability/observability analysis. To ensure that eventual non-scan cells are not used as the destination for observe points or source for control points, you should declare all the non-scan memory elements during test point insertion using the add_nonscan_instances command.
- You should define black boxes using the add_black_boxes command so that test point analysis can incorporate this information.
- If you are performing test point analysis on a pre-scan netlist that has unconnected clock gaters, you should add the "set_clock_gating on" command to your dofile.

You can also insert Observation Scan Technology (OST) during this step. For more information, see "Observation Scan Technology" on page 177.

For more information on this step, refer to "Test Points for LBIST Test Coverage Improvement" in the *Tessent Scan and ATPG User's Manual*.

Scan Insertion and X-Bounding

You should adhere to the following during this step:

- **Inputs —** The modified design netlist with test points and the TCD file you have generated using the Tessent Shell insert_test_logic command.

- **MCPs and failing paths —** You read in SDC to identify MCP/FP, which enables X-bounding to add DFT logic that prevents the capture of any transition on such paths.

Requirements for Using a Third-Party Scan Insertion Tool

Tessent Shell must make certain assumptions about which flip-flops in the design are converted to scan and which cells remain non-scan. If you are using a third-party scan insertion tool, then these assumptions might not be correct.

Specifically, the primary factors in determining the conversion of non-scan flops to scan flops are the S-rule DRC violations and the availability of a suitable scan model. In general, flops with S-rule violations are not converted to scan cells unless you use the following command:

```
set_test_logic -clock on -set on -reset on
```

You might need to read the design into Tessent Shell and go through DRC. You can subsequently post DRC use the report_scan_elements command to report the exact status for each flop in the design.

Scan Insertion and X-Bounding Limitations

Be aware of the following:

- If low power is used in the hybrid TK/LBIST flow, the scan chain length must be equal to or greater than the decompressor size. The minimum decompressor size in the hybrid flow is 31. Therefore, the scan chain length cannot be less than 31. The reason for this requirement is that the size of the low-power register and the size of the decompressor are the same; to initialize the low-power register, the shift length must be equal to or greater than the decompressor size. If the tool generates a larger decompressor, say a size of 62 bits, the required minimum shift length is 62.
- X-bounding analysis does not take into account wrapper cells identified with the analyze_wrapper_cells command and inserts X-bounding multiplexers at the primary input pins that feed the wrapper cells. The workaround is to perform wrapper chain identification and insertion in a separate pass, prior to X-bounding. In addition, you must constrain the scan enable signal for the input wrapper chains to keep these chains in “shift mode” during the capture cycles. This pin constraint prevents insertion of extra X-bounding multiplexers at the primary input pins that drive the wrapper cells in functional mode.
- The tool performs X-bounding first, and targets each primary input with an X-bounding mux. However, when you issue the analyze_wrapper_cells command, the X-bounding results change because the primary input pins receive a dedicated wrapper cell, or the reachable scan flops become part of the input wrapper chains. This is sufficient to block the unknown value from reaching any scan cells during intest.

Note

The tool does not remove X-bounding logic for ports specifically excluded from wrapper analysis using the -exclude_ports switch of the set_wrapper_analysis_options command.

Example

The following example script performs test point analysis and insertion, scan insertion, and X-bounding.

```
# 1. Set merged subcontext to perform both scan and test point
#    insertion, specify unique design ID
set_context dft -scan -design_id gate_scaninv -test_points -no_rtl

# 2. Point to TSDB that contains outputs of previous steps (hybrid
#    TK/LBIST IP generation)
set_tsdb_output_directory ../1_dft/tsdb_outdir

# 3. Read synthesised netlist and DFT libraries
read_verilog ../2_synth/piccpu_gate.v
read_cell_library ../prerequisites/techlib_adk.tnt/4.3/tessent/ \
  adk.tcelllib ../data/picdram.atpglib
read_cell_library ../libs/mgc_cp.lib

# 4. Read all configuration files of previous step (IP insertion) using
#   read_design_command
read_design piccpu -design_id rtl -no_hdl

# 5. Elaborate the design
set_current_design

# 6. Specify test point analysis and insertion settings
set_test_point_analysis -mini_shift_length 16
set_test_point_type lbist_test_coverage
set_test_point_analysis -pattern_count_target 10 -test_coverage 99.9 \
  -total_number 10

# 7. Run design rule checks
set_system_mode analysis

# 8. Analyze test points
analyze_test_points

# 9. Analyze X-bounding
analyze_xbounding

# 10. Specify scan modes as required
add_scan_mode short_chains -edt [get_instance -of_module *_edt_lbist_c0]

# 11. Scan chain analysis
analyze_scan_chains

# 12. Insert all test logic that was analyzed in the previous steps -
#     insert_test_logic command automatically updates the TSDB as well
insert_test_logic
```

X-Bounding

You must perform the X-bounding to prohibit all X generators (that is, non-scan cells, black boxes, and primary inputs) from reaching a scan cell. The source of the X-bounding mux could either be existing scan cells in the design or newly inserted scan cells.

Tessent Shell performs the following operations during X-bounding:

- X-sources are identified as part of DRCs, specifically E5 violations.
- All identified X-sources are bounded from reaching scan cells.
- A multiplexer is introduced to block propagation of an X.
- To increase controllability, the second input of the mux is driven by scan cell.
- Clocks for destination cells are analyzed to choose clocks for new scan cell or choose an existing scan cell.

Note

There are several limitations you should keep in mind:

- When an X-source does not reach a scan flop or scannable flop, and feeds only a small number of combinational gates that may drive output pins, then this signal is not x-bounded, but all the affected logic is marked as no_control_point and no_observe point, with no_control_reason and no_observe_reason set to xbounding.
- Sometimes, it is not possible to bound the signal directly at the source. In that case, any combinational logic that is driven by the X-source and is upstream of the X-bounding mux is also marked as no_control_point and no_observe_point, with no_control_reason and no_observe_reason set to xbounding.
- If the tool needs to bound an unknown state that originates from a primary input with a pad, the X-bounding logic is inserted on the core side of the pad.

X-Bounding Control Signals (Existing or New Scan Cells)

By default, the tool identifies existing scan cells with the correct clock domain and uses the output from these flip-flops to drive the test mode input of the X-bounding multiplexer.

Details about how the tool selects the clocks are in the "Clock Selection" section.

You also can choose to drive the test mode input of the new multiplexers from new scan cells using the -connect_to new_scan_cell switch of the set_xbounding_options command. The clock signal that drives the new scan cell is selected using the same algorithm that applies when using existing scan cells. The new scan cells are merged into existing chains whenever possible. The output of the flip-flops directly drives the data input of the new flip-flops.

Clock Selection

The tool analyzes the location of each X-bounding multiplexer to identify the clocks if a new flip-flop is chosen to drive the multiplexer. The tool looks at clocks for the controlling flip-flops that feed into this location and also the clocks for the flip-flops that are fed from this signal.

When only a single clock domain is involved, that clock is used to drive the test points. However, when multiple clock domains are involved, the behavior depends on the definition of the false or multicycle paths. X-bounding could potentially introduce a new false path that would not be bounded because the false path did not exist in the original netlist. Therefore, when an SDC file is loaded, the tool uses static bounding (forcing a constant 0 or 1 via an AND or OR gate) to avoid creating a new false path. When no false or multicycle paths are defined, the tool chooses the clock domain that is most frequently used by the memory elements in the fanout of the X-source.

Multiple Clock Domain Handling

By default, X-bounding by the tool does not guard cross-clock domain paths. However, if you reference these paths in the SDC file (for example, set_false_path -from CLK1 -to CLK2), then the tool adds the appropriate bounding hardware at the destination flop.

For cases when you create at-speed capture procedures that do not explicitly exercise these paths, none of these paths need to be bounded. The following command and switch:

set_xbounding_options -exclude_sdc_cross_domain_path on

modify the tool's behavior such that paths that meet the following criteria are not X-bounded:

1. Identifies all the clock domains that feed into the test point location.
2. Identifies all the clock domains that observe the signal from the test point.

3. If there is no overlap between the clocks identified in (1) and (2), then this is considered a cross clock domain path that cannot be activated by pulsing only a single clock, and it is excluded from X-bounding.

False and Multicycle Paths Handling

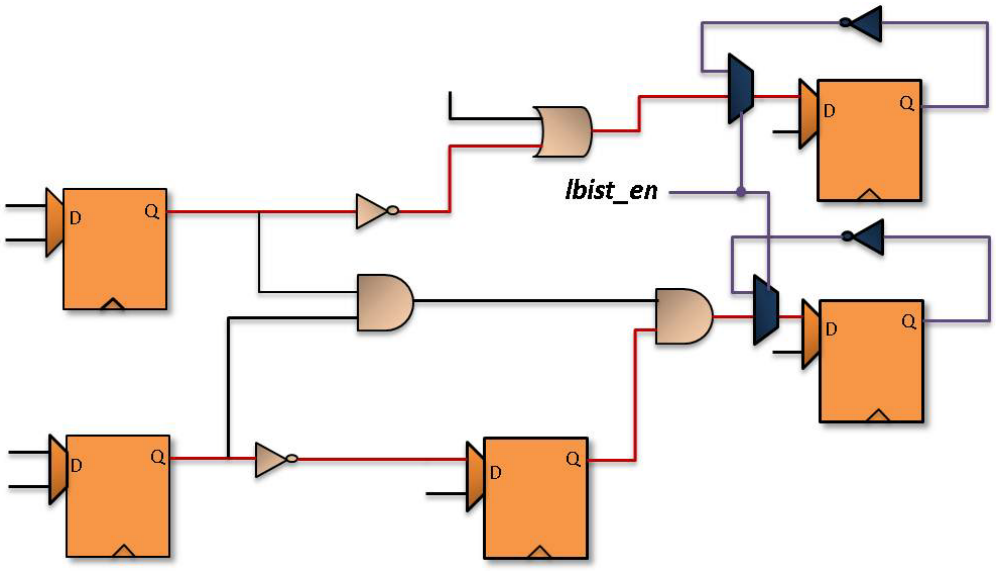
The tool can also insert logic to block false and multicycle paths to inhibit these from contributing Xs during LogicBIST as follows:

- All destination flops are identified from the SDC file you load into the tool using the read_sdc command.
- The destination flops are replaced with holding flops to prevent capturing Xs—see "Destination Flop Identification" in the following section.
- The holding mux has an inverter in the feedback loop to enable transition coverage for the downstream logic. In Figure 3-1, lbist_en is the signal to control the muxes. However, you can use a completely new enable signal to control the select line of these muxes. This allows enabling this logic independent of the LogicBIST run.

  The same enable signal (which defaults to lbist_en but can be overridden using the mcp_bounding_en dft signal) controls the X-bounding of both false paths as well as multicycle paths.

Figure 3-1 shows the holding muxes in black.

Figure 3-1. Inverted Feedback Muxes



Re-circulating muxes with an inverter in the feedback path are inserted on the D input of the destination scan cells of false and multicycle paths. The inversion ensures that the destination scan cell output has a transition during broadside test to achieve higher coverage.

Destination Flop Identification

Each false/multicycle path statement in the SDC file is analyzed in the region identified by that particular statement. The tool subsequently analyses the ability of signals from the marked region to propagate to an observation point. These gates on the propagation path are marked as being in the false or multicycle path effect cone.

From this information, the tool looks at the data input(s) of scannable flops and wrapper cells to determine if they are marked in any false or multicycle path effect cone. These gate inputs are bounded using a mux and an inverted feedback loop. In addition, if the wrapper cells are constructed of separate library cells (for example a separate DFF and MUX), then the tool searches the fan-in of the DFF to find a non-scan path input that can be bounded. Note: this only works if the wrapper cells are part of existing traced scan chains since that enables the tool to differentiate between the shift path and the capture path.

In addition, the tool checks the set and reset ports of each flop and disables any set/reset ports that are also marked as false or multicycle paths. In this case, however, a combinational gate (either an AND gate or an OR gate) is used to force the set/reset port into the off state.

The inverting loopback path inserted during X-bounding may still be considered as a false path during fault simulation if the data-input of the flop is defined as the "-to" gate for the false path. To avoid capturing an X at the input of this flop during fault simulation, the tool searches forward from the MCP bounding enable signal, identifying these inverting loopback paths, and modifying the simulation such that these paths are no longer treated as false paths. The source of the MCP bounding enable signal is typically specified as the mcp_bounding_en signal, added with add_dft_signals during the logic BIST insertion steps.

X-Sources Reaching Primary Outputs

By default, the tool doesn't guard an X source that only reaches one or more primary outputs but doesn't reach any scan cell.

If there is a considerable amount of logic that can be observed at a primary output then inserting an observe point at such a primary output is beneficial for improving the test coverage of LogicBIST. The tool analyzes the amount of logic that can be observed at a primary output. If there are enough library cells that can be observed at a primary output, then the tool guards an X source that can reach this primary output. Test point analysis can then safely insert an observe point at this primary output.

X-Bounding and no_observe_point and no_control_point Attributes

X-bounding adds logic to block the propagation of the X-sources to scan cells.

The X-bounding muxes are inserted as close to the X-source as possible. However, there may be combinational logic between the X-source and the input of the X-bounding mux. If the test point analysis algorithm inserts an observe point at any of these locations, this observe point would capture the signal from the X-source. Therefore, the X-bounding algorithm marks all gate-pins in the combinational fan-in cone of the functional mode input of the X-bounding mux and sets the no_observe_point attribute value to true at those locations. The no_observe_reason attribute for these gate-pins returns "xbounding" as the reason.

In addition to marking these locations with the no_observe_point attribute, the tool also sets the no_control_point attribute. Inserting a control point at any of these locations cannot improve test coverage during Logic BIST because all the logic in the fanout of the control point is blocked by X-bounding muxes and is unobservable. The no_control_reason attribute for these gate-pins also returns "xbounding" as the reason.

For more information, see DFT Test Logic Attributes no_control_point and no_observe_point in the *Tessent Scan and ATPG User's Manual*.

EDT IP Handling

When EDT and LogicBIST IP have been inserted in the design, the tool uses information about these cores to avoid guarding any "perceived" X-sources in this IP.

This usually happens when you are using the pre-synthesis flow where the EDT IP is inserted and synthesized together with the design. The X-bounding analysis happens only after scan stitching has been completed and the tool sees a gate-level design. For more information, see ICL Extraction and Pattern Retargeting.

In the absence of such information, the tool uses information about the scan chains to avoid inserting bounding logic in the scan path and in the paths from the scan chain outputs to the primary outputs that are used as channel outputs. In some cases, particularly when third-party IP is involved, the tool may conservatively treat a functional primary output as a channel output. In such situations it does not guard an X source that reaches that primary output (see X-Sources Reaching Primary Outputs). You should provide the TCD for the EDT IP so that the tool can easily identify these instruments and mark them as non-scan sources. Another way to identify the EDT instances in the design is to use the set_edt_instances command when the TCD flow is not used.

X-Bounding and the Tessent Memory BIST Controller

The Tessent Memory BIST controller includes a mechanism that disables all the clocks during capture. This is intended to avoid any requirement to add X-bounding hardware for false and multi-cycle paths.

When the memory BIST controller is generated with the DftSpecification AdvancedOptions/ use_multicycle_paths property set to “off”, X-bounding is not necessary. If the controller is generated with this property set to “on”, X-bounding is accomplished by doing the following:

- During the dft insertion stage (memory bist + logic bist), you must use "add_dft_signals" and specify the following:

  add_dft_signals async_set_reset_static_disable

- During dft insertion, the memory BIST controller is generated by default with a clock gating cell and includes the static dft signal mcp_bounding_en. This signal is used to disable all clocks during capture.
- During the scan insertion stage, do the following:
  - Setting mcp_bounding_en to “1” disables all clocks during capture:

    set_static_dft_signal_values mcp_bounding_en 1

Note

Scannable flops inside the MemoryBIST controller that hold state during capture are identified as constant, and the tool ignores them during X-bounding analysis.

Test Point Insertion, Scan Insertion, and X-Bounding Command Summary

Tessent Shell enables you to perform scan insertion and X-bounding.

Table 3-1 lists the Tessent Shell scan insertion and X-bounding commands.

Table 3-1. Test Point Insertion, Scan Insertion, and X-Bounding Commands

Command	Description
analyze_test_points	Specifies the test points analysis and generates a list of test points to be inserted by the tool.
analyze_xbounding	Performs X-bounding analysis.
insert_test_logic	Inserts the test structures you define into the netlist to increase the design’s testability. Writes the output files into the TSDB.
read_cell_library	Loads one or more cell libraries into the tool.
read_sdc	Reads in the SDC file that describes the false and multicycle paths that should be blocked during LogicBIST.
read_verilog	Reads one or more Verilog files into the specified or default logical library.
report_scan_elements	Reports information and testability data for the sequential instances in the design.

Table 3-1. Test Point Insertion, Scan Insertion, and X-Bounding Commands (cont.)

Command	Description
report_test_points	Displays test points inserted with the insert_test_logic command.
report_xbounding	Reports the X-sources and the scan cells used in bounding.
set_context	Specifies the current usage context of Tessent Shell. You must set the context before you can enter any other commands in Tessent Shell.
set_scan_signals	Sets the pin names of the scan control signals.
set_system_mode	Specifies the operational state you want the tool to enter.
set_test_point_analysis_options	Sets the maximum number of test points, the breakdown in control and observe points, the target fault coverage, and the number of pseudorandom patterns to be applied. You can also set some other parameters to be taken into account during test point analysis.
set_test_point_insertion_options	Sets parameters related to test point insertion.
set_xbounding_options	Enables X-bounding and sets X-bounding parameters.

Chapter 4
LogicBIST Fault Simulation and Pattern Creation

In this step of the flow, you perform fault simulation and save the parallel LogicBIST patterns.

LogicBIST Fault Simulation and Pattern Creation Overview

During fault simulation with Tessent Shell, the tool generates files in the TSDB for pattern retargeting, such as the PatternDB files. Tessent Shell performs fault simulation at the core level, generates the signatures, and computes test coverage. The tool also writes out a parallel testbench.

LogicBIST Fault Simulation and Pattern Creation Flow

- Load design (netlist, ICL, PDL, TCD, and so on) from the TSDB
- Specify fault simulation details
- Check DRCs with system mode transition
- Run fault simulation (simulate_patterns)
- Store patterns (patDB, TCD) and Verilog testbench

You must generate NCPs manually. To generate the NCPs and the NCP index decoder, specify the LogicBist/NcpIndexDecoder wrapper in the DftSpecification. The NCP description and hardware are automatically included in the TSDB. The tool writes out a TCD for the NCP index decoder, which is read back during fault simulation.

The core level fault simulation run computes the test coverage for the core, MISR signature, and power consumption. Running the write_tsdb_data command during this step generates the following output files:

- **PatternDB —** Contains the relevant LogicBIST register values per pattern like PRPG, MISR, and low-power registers.
- **Tessent Core Description (TCD) —** Contains the description of the core in the LogicBIST mode of operation.
- **Flat model —** Contains the flattened circuit model, the scan trace, and all DRC-related information to a specific binary file.
- **Fault list —** Contains the fault information from the current fault list.

Low-Power Shift Simulation

If you specified a low-power controller during EDT and LogicBIST IP generation, you can specify the amount of switching activity required during fault simulation with the following commands. For example:

```
set_system_mode analysis
set_power_control shift on -switching_threshold_percentage 15
...
```

NCP Order

For fault simulation, the order of the specified NCPs must match the NCP order in the design. The tool automatically ensures this ordering when you use an NcpIndexDecoder generated with the LogicBist/NcpIndexDecoder wrapper.

Two commands control ordering:

- **set_lbist_controller_options —** This command reflects the order of the NCPs in the hardware. You must specify the NCP list in the same order as implemented in the hardware.
- **set_lbist_controller_options -capture_procedures —** The second order is when you specify the activity percentage using this command. The specified NCPs do not have to reflect the hardware order as long as you previously specified the NCP list with the -programmable_ncp_list option so that the NCP order is known to the tool.

Initial Static DFT Signal Values

Prior to fault simulation, the tool initializes DFT signals created during IP insertion to the values required for fault simulation, unless they were user specified with the

set_static_dft_signal_values or set_test_setup_icall commands. This ensures that the signals are properly constrained during fault simulation.

Note

By default, the tool initializes DFT signals in child sub-blocks and child physical blocks. Sub-blocks in child physical blocks are not initialized. If you do not want to initialize DFT signals in the child physical blocks of a current design level, use "set_lbist_controller_options -initialize_dft_signals_in_child_physical_blocks off".

The following table lists the DFT signals that are required for fault simulation and their default initialization values for fault simulation.

Table 4-1. Initial Static DFT Signals for Fault Simulation

DFT Signal	Value
async_set_reset_static_disable	1
control_test_point_en	1
ext_ltest_en	0
int_ltest_en	1
ltest_en	1
mcp_bounding_en	0
memory_bypass_en	1
observe_test_point_en	1
se_pipeline_en	0
x_bounding_en	1

The x_bounding_en signal must be set to 1 during LogicBIST operation.

The mcp_bounding_en signal is set to 0 to obtain optimal test coverage when performing fault simulation for stuck-at faults. To maintain the disabled state for MCP bounding, the following criteria must be met:

- There are no NCPs that can generate sequential patterns.
- The clocks located in cross-domain paths are not pulsed at the same time.
- There are enough cycles from last shift and first capture.

When performing fault simulation for transition faults, the recommended mcp_bounding_en signal value is 1.

See "X-Bounding and the Tessent Memory BIST Controller" on page 86 for more information.

Performing LogicBIST Fault Simulation and Pattern Creation

For LogicBIST, the external mode for NCPs is not relevant and is discarded with a warning message. When generating parallel patterns during fault simulation, only the -mode_internal patterns are valid. These patterns are the default. An error message is issued when trying to save -mode_external patterns.

Prerequisites

- Use the following files that you have generated during EDT and LogicBIST IP Generation using the write_edt_files command and during Logic Synthesis:
  - tsdb_outdir/dft_inserted_designs/gpu_gate.dft_inserted_design/gpu.vg
  - tsdb_outdir/instruments/gpu_gate_edt_lbist.instrument/*.v
  - tsdb_outdir/instruments/gpu_gate_lbist.instrument/*.v

Procedure

1. Use this Tessent Shell procedure to perform fault simulation:
2. From a shell, invoke Tessent Shell using the following syntax:

   ```
   % tessent -shell
   ```

   After invocation, the tool is in unspecified setup mode. You must set the context before you use the fault simulation commands.
3. Set the tool context to fault simulation using the set_context command as follows:

   SETUP> set_context patterns -scan
4. Open the TSDB. For example:

   SETUP> open_tsdb ../tsdb_outdir
5. Load the design netlist using the read_design command. For example:

   SETUP> read_design gpu
6. Load one or more cell libraries into the tool using the read_cell_library command.

   SETUP> read_cell_library atpg.lib
7. Set the top design using the set_current_design command as follows:

   SETUP> set_current_design top_module
8. Import the configuration settings:

   SETUP> import_scan_mode

9. Specify the capture procedure names with the set_lbist_controller_options command as follows; be sure to include all of the NCPs that you want to use in the LBIST mode:

   ```
   SETUP> set_lbist_controller_options -capture_procedures
   {clkseq1 40 clkseq2 40 clkseq3 10 clkseq4 10}
   ```

10. Change the tool's system mode to fault using the set_system_mode command as follows:

    ```
    SETUP> set_system_mode analysis
    ```

    During the transition from setup to analysis mode, the tool creates NCPs according to the NcpIndexDecoder specification and performs design rule checking.

11. Add faults using the add_faults command as follows:

    ```
    ANALYSIS> add_faults -all
    ```

12. Specify the number of random patterns the tool simulates using the command as follows:

    ```
    ANALYSIS> set_random_patterns 100
    ```

13. Set the pattern source to LogicBIST and execute fault simulation using the simulate_patterns command as follows:

    ```
    ANALYSIS> simulate_patterns -source bist -store_patterns all
    ```

14. Save the TCD, PatternDB, flat model, and fault list needed for the next step, Pattern Generation, using the write_tsdb_data command as follows:

    ```
    ANALYSIS> write_tsdb_data -replace
    ```

    By default, the write_tsdb_data command only saves the scan chain data for the first 1024 patterns. To save more patterns—for example, if you discover during diagnosis that additional patterns are required—use the –max_scan_load_unload_size option.

15. Write out the parallel testbench using the write_patterns command as follows:

    ```
    ANALYSIS> write_patterns lbist_patt_parallel.v -verilog -parallel \
    -mode_internal -param ../data/paramfile
    ```

Results

Now you are ready to perform Pattern Generation.

Specifying Warm-Up Patterns During Fault Simulation

You can specify the number of warm-up patterns to use in fault simulation. This enables you to obtain a known starting seed and improve test program size reduction for InSystemTest applications.

Use the "set_lbist_controller_options -warmup_pattern_count" command to specify the number of warm-up patterns to use for fault simulation.

When you specify this value for fault simulation, the hardware default PRPG seed and low power mask shift register values are used as the starting point for the warm-up patterns. This provides a known starting seed and avoids serially loading the registers for InSystemTest applications. However, you also lose the ability to change the warm-up pattern count when specifying patterns.

To maximize the benefit of test program size reduction for InSystemTest, use a 2022.2 or later version of Tessent Shell that generates the decompressor with separate low-power SIBs.

Note

A Tessent Core Description (TCD) for the EDT IP creation is required when using this feature.

Fault Simulation When There Are Inversions

Third-party tools may add inversions between the PRPG/phase-shifter and scan chains on the input side, and between scan chains and the spatial compactor on the output side. When Tessent Shell is unaware of the inversions, simulation of the Verilog patterns may result in MISR signature mismatches. To make Tessent Shell aware of these inversions, there are a couple of tasks you can perform prior to LogicBIST fault simulation.

Automatically import the inversions from the TCD. To do this, first perform EDT DRC (K rules) so that Tessent Shell learns the inversions, and then specify write_core_description to save the information into the TCD. Prior to running fault simulation, issue the read_core_description command to import the inversions. You must specify read_core_description before import_scan_mode.

Tessent Shell attempts to match each EDT block declared in the tool (through import_scan_mode or by add_core_instances or add_edt_block) with those described in the TCD by matching the SCI and SCO pins. If at least one AtpgMode section is found for the current top module, but not all EDT blocks are successfully matched in the TCD, then the tool reports a warning:

```
//  Warning: Could not match the following EDT instance in the TCD file and,
//           as a result, the decompressor to scan chain input and scan chain
//           output to compactor inversions are not imported for them:
//             <instance_name1>
//             <instance_name2>
//           You can specify the inversions manually with the
//           set_scan_chain_options command or provide a TCD that includes
//           all EDT instances.
```

If you do not run the K rules, then you can specify the inversions manually using the set_scan_chain_options command. For example, the following example indicates that there is an inversion between the decompressor and SCI at scan chain “chain1”:

```
set_scan_chain_options -chain_name chain1 -decompressor_to_scan_in_inversion on
```

When an inversion for the same scan is specified by set_scan_chain_options as well as the TCD, the user-specified inversion has priority. If the values do not match, Tessent Shell issues a warning.

Fault Coverage Report for the Hybrid IP

The fault coverage report provides statistics about the total faults and the relevant faults found during LogicBIST fault simulation. The total faults are those found within the hybrid IP controller instances. The relevant faults are those found within the core logic, excluding the hybrid IP controller faults.

If the relevant coverage percentile is too low, you can improve the test by adding test points, running more patterns, and so on. Faults found within the hybrid IP controller cannot be detected or improved upon in this way.

As shown in the example below, the default fault classification for faults found within the hybrid IP controller are designated as ATPG untestable (AU) faults with sub-class hybrid LBIST (LBIST) if they have not already been identified with another designator, such as unused (UU), tied (TI), or blocked (BL).

```
//  command: report_statistics

                        Statistics Report
                         Stuck-at Faults
-----------------------------------------------------------------
Fault Classes                      #faults          #faults
                                   (total)      (total relevant)
---------------------------  ---------------- ------------------
  FU (full)                        37280             29117
  -------------------------  ---------------- ------------------
UC (uncontrolled)                    50  ( 0.13%)       same  ( 0.17%)
UO (unobserved)                     870  ( 2.33%)       same  ( 2.99%)
DS (det_simulation)               22790  (61.13%)       same  (78.27%)
DI (det_implication)               3444  ( 9.24%)       same  (11.83%)
PT (posdet_testable)                 53  ( 0.14%)       same  ( 0.18%)
UU (unused)                         106  ( 0.28%)       same  ( 0.36%)
TI (tied)                            11  ( 0.03%)       same  ( 0.04%)
BL (blocked)                          2  ( 0.01%)       same  ( 0.01%)
AU (atpg_untestable)               9954  (26.70%)       1791  ( 6.15%)
-----------------------------------------------------------------
Fault Sub-classes
-------------------------
AU (atpg_untestable)
   PC*  (pin_constraints)          616  ( 1.65%)       same  ( 2.12%)
   TC*  (tied_cells)                95  ( 0.25%)       same  ( 0.33%)
   MPO   (mask_po)                1039  ( 2.79%)       same  ( 3.57%)
   LBIST (hybrid_lbist)           8163  (21.90%)         deleted
   Unclassified                     41  ( 0.11%)       same  ( 0.14%)
   *Use "report_statistics -detailed_analysis" for details.
-----------------------------------------------------------------
Coverage
   -------------------------
   test_coverage                        70.67%              90.56%
   fault_coverage                       70.44%              90.19%
   atpg_effectiveness                   97.46%              97.46%
-----------------------------------------------------------------
#test_patterns                                             10000
   #clock_sequential_patterns                              10000
#simulated_patterns                                        10000
CPU_time (secs)                                              7.8
-----------------------------------------------------------------
```

The following example shows that these faults appear as AU.LBIST faults in the report generated after LogicBIST fault simulation completes.

```
FaultInformation {
  version : 1;
  FaultType (Stuck) {
    FaultList {
      FaultCollapsing : FALSE;
      Format : Identifier, Class, Location;
      Instance ("") {
        0,  DI.SCAN,    "/u1/STATD_reg_1_/Q";
        1,  DI.SCAN,    "/u1/STATD_reg_1_/Q";
        ...
        1,  UU,         "/occ/occNX2/clk_enable_latch_reg/Q";
        1,  EQ,         "/occ/occNX2/U7/B";
        0,  AU.LBIST,    "/m8051_single_chain_mode_logic_i/tdr_sib_i/sib_reg/Q";
        1,  AU.LBIST,    "/m8051_single_chain_mode_logic_i/tdr_sib_i/sib_reg/Q";
        ...
        0,  AU.LBIST,    "/m8051_single_chain_mode_logic_i/tdr_single_bypass_reg/Q";
        1,  AU.LBIST,    "/m8051_single_chain_mode_logic_i/tdr_single_bypass_reg/Q";
        0,  AU.LBIST,    "/m8051_edt_i/m8051_edt_decompressor_i/m8051_edt_sib_i/U8/A";
        0,  EQ,         "/m8051_edt_i/m8051_edt_decompressor_i/m8051_edt_sib_i/U8/B";
        ...
        0,  AU.LBIST,    "/m8051_edt_i/m8051_edt_misr_i/m8051_edt_sib_i/sib_latch_reg/Q";
        1,  AU.LBIST,    "/m8051_edt_i/m8051_edt_misr_i/m8051_edt_sib_i/sib_latch_reg/Q";
        0,  AU.LBIST,    "/m8051_lbist_i/m8051_lbist_ctrl_i/
m8051_lbist_capture_phase_size_reg_i/U9/Z";
        1,  EQ,         "/m8051_lbist_i/m8051_lbist_ctrl_i/
m8051_lbist_capture_phase_size_reg_i/U9/A";
        ...
        0,  AU.LBIST,    "/m8051_lbist_i/lbist_scan_out_reg/CP";
        1,  AU.LBIST,    "/m8051_lbist_i/lbist_scan_out_reg/CP";    }
      }
    }
  }
```

When you are not reading ICL during fault simulation, the single chain mode logic instance faults are not classified as AU.LBIST.

Optionally, specify the set_relevant_coverage -include AU.LBIST command if you want to include the AU.LBIST faults in the relevant fault coverage.

AU.LBIST faults are not reclassified when you use the reset_au_faults command to reclassify AU faults.

Fault Simulation and Pattern Creation Command Summary

Tessent Shell provides variety of fault simulation and pattern creation commands.

Table 4-2. Fault Simulation and Pattern Creation Commands

Command	Description
add_bist_capture_range	Associates the capture procedure that is used for a specific set of patterns.
add_chain_masks	Specifies the scan chains that are masked during fault simulation and their load and unload values. Used to work around design issues.
add_faults	Adds faults to the current fault list, discards all patterns in the current test pattern set, and sets all faults to undetected (actual category is UC).
read_cell_library	Loads one or more cell libraries into the tool.
read_sdc	Reads in the SDC file that describes the false and multicycle paths.
read_verilog	Reads one or more Verilog files into the specified or default logical library.
report_misr_connections	Reports the MISR connections.
set_bist_debug	Sets up a trace of PRPG and MISR values during a pattern's shift cycles.
set_context	Specifies the current usage context of Tessent Shell. You must set the context before you can enter any other commands in Tessent Shell.
set_current_design	Specifies the top level of the design for all subsequent commands until reset by another execution of this command.
set_dft_enable_options	Enables or disables the control or observe points.
set_edt_options	Sets options for EDT IP creation and LogicBIST fault simulation.
set_lbist_controller_options	Specifies global options to configure the LogicBIST controller.
set_lbist_power_controller_options	Specifies creation of the low-power shift controller for LogicBIST.
set_power_control	Specifies the switching threshold in the patterns for fault simulation.
set_random_patterns	Specifies the number of random patterns the tool simulates.
set_system_mode	Specifies the operational state you want the tool to enter.
simulate_patterns	Performs simulation by applying the specified pattern source.
write_patterns	Writes out parallel patterns and PatternDB files.

Table 4-2. Fault Simulation and Pattern Creation Commands (cont.)

Command	Description
write_tsdb_data	Writes the *design_name*.lbist_*mode_name* directories that contain all the files needed to use LogicBIST test mode during PatternsSpecification processing and to perform diagnosis of the failures.

Chapter 5
Pattern Generation

In this step of the flow, you generate core-level patterns for the bottom-up method and top-level patterns (including a Verilog testbench) for the LogicBIST controller for the top-down method.

Pattern Generation Overview

For the bottom-up flow, pattern generation creates core-level patterns. For the top-down flow, it creates top-level patterns (including a Verilog testbench) for the LogicBIST controller. These are the chip-level serial patterns that you can apply from the tester.

Pattern Generation Flow

- Core-level steps
  - Load the design configuration files: netlist, ICL, PDL, patDB, TCD, graybox signatures, and the TSDB from LogicBist fault simulation.
  - Run design rules through system mode transition.
  - Perform pattern generation. Output patterns are stored in the TSDB.
  - Perform testbench simulation.
- Top-Level steps
  - You can, optionally, have a top-level hybrid TK/LBIST controller. To do this, perform all of the core-level steps and then integrate the top-level controller into the core-level design.

- Perform top-level ICL network extraction, pattern retargeting, and integration, using the TSDB data from the core-level pattern generation step. The output extracted ICL, retargeted PDL, TCD, and PatDB are stored in the Top-level TSDB.

The tool supports all the formats currently supported for ATPG.

Required Inputs

To program the LogicBIST controller, you use Tessent Shell to retarget the patterns and create hardware default mode testbench/vectors and pattern_range specific vectors.

The information you need to program the LogicBIST controller is stored in the TSDB, specifically in the ICL and PDL files created during EDT and LogicBIST IP Generation:

- ICL — The ICL file consists of ICL module description for the LogicBIST controller and all EDT blocks tested by this controller.
- PDL — The PDL file contains iProcs at the core level that use the ICL modules written out.

Pattern Generation for the TSDB Flow

The TSDB stores the various files required for IJTAG and CCM pattern generation. These files include the ICL, PDL, and design netlist.

Performing Pattern Generation for the TSDB Flow

The procedure for generating chip-level serial patterns uses the extracted ICL files that are stored in the TSDB. Generate the patterns using the create_patterns_specification and process_patterns_specification commands.

Prerequisites

- Modified design netlist found in the TSDB.
- The LogicBIST instruments PDL data file found in the TSDB.
- The LogicBIST instruments ICL data file found in the TSDB.
- Top-level ICL describing how the signals at the interface of the LogicBIST controller are connected to chip-level pins.
- A PDL that describes the test setup at the chip level if there is any. For example, if there is a TAP controller at the top level, then the tool requires an ICL and, optionally, PDL for the TAP controller.

Procedure

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

   ```
   % tessent -shell
   ```

2. Set the tool context to IJTAG mode as follows:

   SETUP> set_context patterns -ijtag

3. Open the TSDB if it is not already open. For example:

   SETUP> open_tsdb tsdb_outdir

4. Unless it is already in memory, read the current design's extracted ICL. For example:

   SETUP> read_icl ./tsdb_outdir/dft_inserted_designs/m01_gate.dft_inserted_design/m01.icl

5. Set the current design.

   SETUP> set_current_design m01

6. Define the top-level clocks. For example:

   SETUP> add_clocks refclk -period 10 ns -free_running

   SETUP> add_clocks 0 tck

7. Define the pin constraints. For example:

   SETUP> add_input_constraints RST -c0

8. Change to analysis mode to generate patterns:

   SETUP> set_system_mode analysis

9. Create patterns specification:

   SETUP> create_patterns_specification

   Note

   When the Self-Test or burn-in features are available, pattern specification configurations can be specified here according to user requirements.

10. Process patterns specification:

    SETUP> process_patterns_specification

11. Point to the simulation library sources so all design files can be found. For example:

    SETUP> set_simulation_library_sources -y ./techlib -extensions { v }

12. Simulate the LogicBIST testbenches with the following command:

    SETUP> run_testbench_simulations

13. As needed, monitor or check the simulation with the following command:

    SETUP> check_testbench_simulations

Results

Upon completion, Tessent Shell outputs the testbench and vectors for the entire pattern set, range specific vectors, or hardware default mode as specified in the dofile.

- If you are using the Considerations for Top-Down Implementation, you finished all necessary steps in this flow.
- If you are using the Hybrid TK/LBIST Implementation, you are ready to perform the Top-Level ICL Network Integration.

Performing Pattern Generation for CCM in the TSDB Flow

Controller chain mode enables you to generate ATPG patterns that target the hybrid-IP logic so that you can test the test logic itself.

For details, refer to "Controller Chain Mode" on page 48.

Note

When generating CCM patterns, do not use add_core_instances for the EDT/LogicBIST/OCC instruments. The presence of this command infers non-hybrid EDT pattern generation or LogicBIST fault simulation, as applicable.

Prerequisites

- Modified design netlist found in the TSDB.
- The LogicBIST instruments ICL data file found in the TSDB.
- Top-level ICL describing how the signals at the interface of the LogicBIST controller are connected to chip-level pins.
- A PDL that describes the test setup at the chip level if there is any. For example, if there is a TAP controller at the top level, then the tool requires an ICL and, optionally, PDL for the TAP controller.

Procedure

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

```
% tessent -shell
```

2. Set the tool context to ATPG pattern generation:

```
SETUP> set_context pattern -scan
```

3. Read in the design and libraries. For example:

```
SETUP> read_design piccpu
SETUP> read_cell_library ../library/tessent/adk.tcelllib ../data/picdram.atpglib
SETUP> set_current_design
```

4. Import the controller scan chain mode that you created during scan insertion. For example:

```
SETUP> import_scan_mode controller_chain_mode
```

   Refer to the second example below for a dofile that shows the flow when you have turned off segmented controller chain generation in favor of connecting the controller chain scan segments into one chain during IP generation. See "Usage Details" on page 49 for details.

5. Turn off all core clock and reset activity. Set these constraints because the faults in the design are not targeted during CCM.

```
SETUP> add_input_constraints clk -c0
SETUP> add_input_constraints reset -c0
SETUP> add_input_constraints shift_capture_clock -c0
```

6. Change the system mode to analysis:

```
SETUP> set_system_mode analysis
```

7. Target the faults to the EDT/LogicBIST logic. For example:

```
ANALYSIS> add_faults piccpu_rtl_tessent_lbist   \
   piccpu_rtl_tessent_edt_lbist_c0_inst  \
   piccpu_rtl_tessent_single_chain_mode_logic
```

8. Create and save the CCM patterns. For example:

```
ANALYSIS> create_patterns
ANALYSIS> write_patterns ccm_patt.v -verilog -replace -serial
```

Examples

Example 1: Generating Patterns for Controller Chain Mode, Default Flow for TSDB

```
set_context pattern -scan
read_design piccpu
read_cell_library ../library/tessent/adk.tcelllib ../data/picdram.atpglib
set_current_design

import_scan_mode controller_chain_mode

add_input_constraints clk -c0
add_input_constraints reset -c0
add_input_constraints shift_capture_clock -c0

set_system_mode analysis
add_faults piccpu_rtl_tessent_lbist \
      piccpu_rtl_tessent_edt_lbist_c0_inst \
      piccpu_rtl_tessent_single_chain_mode_logic
create_patterns
write_patterns ccm_patt.v -verilog -replace -serial
```

Example 2: Generating Patterns for Controller Chain Mode, Non-Default Flow for TSDB

The following dofile example generates CCM patterns when you specify to connect the controller chain scan segments into one chain during IP generation.

```
set_context pattern -scan
read_design piccpu
read_cell_library ../library/tessent/adk.tcelllib ../data/picdram.atpglib
set_current_design

# Add edt_clock or tck as the primary clock source for CCM.
# Specify the same clock (edt_clock or tck) that you specified during
# IP generation with the set_lbist_controller_options command. If the
# edt_clock was derived from test_clock or previously specified as a
# DFT signal, then you do not need to specify it with the add_clock
# command
add_clocks 0 edt_clock
```

```
# Define a scan group. Assuming you defined scan_en as a DFT signal when
# you generated the IP, you can define a scan group without a test
# procedure file
add_scan_groups grp1

# Define scan chains on ports specified during IP creation
add_scan_chains chain1 grp1 control_chain_scan_in control_chain_scan_out

# Define the pin constraint for the ccm_en signal
add_input_constraints control_chain_enable -c1

# Turn off all core clock and reset activity.
add_input_constraints clk -c0
add_input_constraints reset -c0
add_input_constraints shift_capture_clock -c0

# For retargeting, specify to pulse the edt_clock during shift.
# The following command is only required when you use edt_clock for CCM
# and edt_clock is a top-level port, or when you use tck for CCM.
# The tool automatically generates a test procedure file that configures
# the scan_en and shift_capture_clock DFT signals. If the edt_clock is
# derived from test_clock, do not specify the
# set_procedure_retargeting_options command
set_procedure_retargeting_options -pulse_during_shift edt_clock

set_system_mode analysis

add_faults piccpu_rtl_tessent_lbist \
    piccpu_rtl_tessent_edt_lbist_c0_inst \
    piccpu_rtl_tessent_single_chain_mode_logic

create_patterns
write_patterns ccm_patt.v -verilog -replace -serial
```

Pattern Generation in Multiple, Shorter Sessions

During pattern generation, you can split LogicBIST tests into multiple, shorter tests for tasks such as continuous monitoring of safety critical parts of a design.

Consider the following patterns specification:

```
PatternsSpecification(top,gate,signoff) {
  Patterns(logicbist) {
    ClockPeriods {
      refclk : 5ns ;
    }
    TestStep(test1) {
      LogicBist {
        CoreInstance(chip) {
          run_mode : run_time_prog ;
          begin_pattern : 0 ;
          end_pattern : 4999 ;
        }
      }
    }
  }
}
```

When you run create_patterns_specification, the tool generates a patterns specification with one Patterns wrapper. To run pattern generation in shorter sessions, split the patterns into multiple Patterns wrappers.

As an alternative to performing fault simulation for 5000 patterns in a single session, do the following:

1. Use the design editing commands as described in the Configuration Data Editing and Introspection Commands table in the *Tessent Shell Reference Manual* to modify the patterns specification.
2. Use the begin_pattern and end_pattern properties to specify the pattern subsets. Rename the Patterns wrappers accordingly.

For example:

```
PatternsSpecification(top,gate,signoff) {
  Patterns(pat1) {
    ClockPeriods {
      refclk : 5ns ;
    }
    TestStep(test1) {
      LogicBist {
        CoreInstance(chip) {
          run_mode : run_time_prog ;
          begin_pattern : 0 ;
          end_pattern : 999 ;
        }
      }
    }
  }
```

```
Patterns(pat2) {
  ClockPeriods {
    refclk : 5ns ;
  }
  TestStep(test1) {
    LogicBist {
      CoreInstance(chip) {
        run_mode : run_time_prog ;
        begin_pattern : 1000 ;
        end_pattern : 1999 ;
      }
    }
  }
}

Patterns(pat3) {
  ClockPeriods {
    refclk : 5ns ;
  }
  TestStep(test1) {
    LogicBist {
      CoreInstance(chip) {
        run_mode : run_time_prog ;
        begin_pattern : 2000 ;
        end_pattern : 2999 ;
      }
    }
  }
}

Patterns(pat4) {
  ClockPeriods {
    refclk : 5ns ;
  }
  TestStep(test1) {
    LogicBist {
      CoreInstance(chip) {
        run_mode : run_time_prog ;
        begin_pattern : 3000 ;
        end_pattern : 3999 ;
      }
    }
  }
}
```

```
    Patterns(pat5) {
      ClockPeriods {
        refclk : 5ns ;
      }
      TestStep(test1) {
        LogicBist {
          CoreInstance(chip) {
            run_mode : run_time_prog ;
            begin_pattern : 4000 ;
            end_pattern : 4999 ;
          }
        }
      }
    }
  }
```

Pattern Generation for Low Power LBIST

You can generate patterns for low power LBIST during the fault simulation step.

The LBIST low power switching threshold is fully programmable during pattern simulation. The specified percentage can be any number from 1 to 50, independent of the SwitchingThresholdPercentage number specified during IP creation. By default during pattern simulation, if you have not specified a switching threshold, the tool uses the threshold defined during IP creation.

After generating LBIST low power hardware with LogicBistOptions/ShiftPowerOptions/ default_operation:disabled, you must manually enable low power for pattern generation:

- Enable the lbist_low_power_shift_en signal. (This signal is usually disabled based on the IP creation setting default_operation:disabled.)
- Specify the required switching threshold if it is different from the number used during IP creation. You must do this before the pattern simulation step “simulate_patterns -source lbist -store_patterns all”.

For more information, see the example below.

Example

Assume that you have generated LBIST low power hardware with the following settings:

```
read_config_data -in $spec -from_string {
  EDT {
    ijtag_host_interface : Sib(edt);
    Controller(c1) {
      longest_chain_range : 50, 65;
      scan_chain_count : 60;
      input_channel_count : 2;
      output_channel_count : 2;
      LogicBistOptions {
        misr_input_ratio : 1 ;
        ShiftPowerOptions {
          present : on ;
          default_operation : disabled ;
          SwitchingThresholdPercentage {
            hardware_default : 25 ;  // default 15
          }
        }
      }
    }
  }
}
```

Enable LBIST low power with the command below, which uses the edt instrument to set up the LBIST low power enable pin. This is because LBIST re-uses EDT IP in the hybrid TK/LBIST flow.

```
set_core_instance_parameter -instrument edt -parameter_values \
      {lbist_low_power_shift_en on}

set_system_mode analysis
```

Now set the low power threshold:

```
set_power_control shift on -switching_threshold_percentage 15
```

Any percentage is legal, independent of the percentage specified during IP creation (25% in the example above).

Single Chain Mode Diagnosis

Single chain mode diagnosis pinpoints the location of failed scan cells in a design under test.

For a design with one or more blocks, diagnosis for LogicBIST patterns consists of two steps:

1. Identify the first failing pattern on which a MISR mismatches.
2. Use the scan_unload_register iProc to shift in the failing pattern stimuli using the PRPG, capture the results, and shift out the results through the single chain mode logic.

By default, during IP generation the tool adds additional SIB logic to each EDT block so that their scan cells can be bypassed as needed. During fault simulation, you specify scan chains that should be masked. If the scan chain has a bad capture, you can mask only that scan chain with an -unload_value during the single chain diagnosis. If a chain has a shift problem, mask the chain with a -unload_value and -load_value. During pattern generation, the tool uses the added SIB to skip EDT blocks that contain at least one masked chain with a -load_value applied. The scan_unload_register iProc automatically skips those blocks during single chain mode diagnosis. Single chain diagnosis cannot be performed on those blocks, but the MISR signature can still be used even in the presence of masked scan chains. This enables the tool to achieve a functional diagnosis on the rest of the blocks in the design.

The tool issues an error message when all the EDT blocks in a design have at least one masked chain with a -load_value. You can turn off the default behavior with the set_lbist_controller_options -single_chain_mode_skip_edt_blocks switch or the DftSpecification/LogicBist/Controller/SingleChainForDiagnosis/skip_edt_blocks property.

Note

You cannot use single chain mode diagnosis on a design in which MISR mismatches occur in a block that contains masked scan chains with a -load_value. Because the scan_unload_register iProc does not have access to the MISR mismatches, this iProc cannot detect and report errors in this scenario. It is your responsibility to determine whether you can use single chain mode diagnosis based on the failing EDT block and masked scan chain information.

Pattern Mismatch Debugging

At the pattern generation stage, most simulation mismatches occur because the clocks were not configured properly during PDL retargeting. Rather than waiting through a time-consuming serial pattern simulation to verify that the clocks are working as expected, the simulation mismatch debugging flow provides a means to verify the clocks prior to running serial simulation.

Debug Based on MISR Signature Divergence

You can verify the BIST registers and clocks at the same time, which in turn enables you to identify the patterns corresponding to mismatches (such as MISR) as the mismatches occur.

Restrictions and Limitations

- Once the TCD file is created as described below, you cannot alter the core hierarchy (such as by ungrouping the LogicBIST controller). Altering the core hierarchy causes the list of monitor points in the TCD file to become out of sync.

Prerequisites

- You have performed the hybrid TK/LBIST flow through the pattern generation step.

Procedure

1. Verify the clocks.

   Enable clock verification by setting the SimulationOptions/logic_bist_debug property in the PatternsSpecification wrapper, as follows:

   ```
   PatternsSpecification(design_name, design_id, pattern_id) {
      Patterns(LogicBist) {
         ClockPeriods { ... }
         SimulationOptions {
           logic_bist_debug : setup_and_clock_verify;
         }
   ...
   ```

   The setup_and_clock_verify value exercises the full 256-pattern NCP count range. If you want to run a single pattern per NCP, specify the setup_and_clock_verify_one_per_ncp value instead. This is equivalent to specifying “one_pattern_per_ncp 1” in the dofile flow.

2. Run Verilog simulation with the LogicBIST debugging feature enabled as shown below, and identify any failing LogicBIST patterns.

Set the SimulationOptions/logic_bist_debug property in the PatternsSpecification wrapper as follows:

```
SimulationOptions {
  logic_bist_debug : bist_registers_and_clock_verify;
}
```

For both flows, the resulting transcript includes mismatch statements such as those shown in bold below. The statements tell you at which pattern the MISR signature started to diverge from the expected value.

Note

To display the passing data, specify "+show_passing_regs" when you start the simulator.

```
Setting up controller TLB_coreB_I1.coreB_lbist_i
     Number of patterns     : 5 (5 + 0 warm-up patterns)
     Pattern Length         : 40
     Shift Clk Select       : 0b01
     Capture Phase Width    : 0x3 Shift Clock Cycles
     PRPG Seed              : 0x66241da0
     MISR Seed              : 0x000000
Starting controller TLB_coreB_I1.coreB_lbist_i in Normal mode,
patterns 0 to 3
     Checking that the controller TLB_coreB_I1.coreB_lbist_i DONE
signal is NO at the beginning of the test
   Mismatch at pattern 2 for TLB.coreB_I1.coreB_edt_lbist_i.misr:
Expected = 83ab37 Actual = 7854d0
   Mismatch at pattern 3 for TLB.coreB_I1.coreB_edt_lbist_i.misr:
Expected = 9b96e3 Actual = 5e161a
Test Complete for controller TLB_coreB_I1.coreB_lbist_i
     Checking that signal DONE is YES for controller
TLB_coreB_I1.coreB_lbist_i
Checking results of controller TLB_coreB_I1.coreB_lbist_i
     Expected Signature for controller TLB_coreB_I1.coreB_lbist_i :
0x9b96e3
Turning off LogicBist controller TLB_coreB_I1.coreB_lbist_i
```

3. Re-run the simulation so that you can identify the failing flop associated with the particular pattern where the MISR started to diverge.

   Regenerate the LogicBist wrapper with the Diagnosis Options/extract_flop_data property enabled and execute the run_testbench_simulations command.

```
PatternsSpecification(CHIP,gate,signoff) {
  Patterns(LogicBist) {
    TestStep (diagnostic) {
      LogicBist {
        CoreInstance(CHIP) {
          run_mode : run_time_prog ;
          begin_pattern : 2;
          end_pattern : 2;
          DiagnosisOptions {
            extract_flop_data : on ;
          }
        }
      }
    }
  }
}
```

Examine the results of the re-run simulation to identify the failing flops, noting the Verilog simulation results against what fault simulation predicted. Use Tessent Visualizer to trace the flops to the cause of the failure. For detailed usage examples, refer to "Usage Examples."

Results

The following transcript example shows a mismatch at an lbist_scan_out pin.

```
   ...
   300ns:  piccpu MISR Seed           : 0x000000
 49300ns:  Starting controller TLB_coreB_I1.coreB_edt_lbist_i in Normal
mode, patterns 0 to 0
 51000ns:  Checking that the controller TLB_coreB_I1.coreB_edt_lbist_i
DONE signal is NO at the beginning of the test
 62800ns:  Test Complete for controller TLB_coreB_I1.coreB_edt_lbist_i
 69000ns:  Scanning out capture results of vector 0 for controller
TLB_coreB_I1.coreB_edt_lbist_i
180024ns: Mismatch at pin           0 name            lbist_scan_out,
Simulated x, Expected 0
180100ns: Corresponding ICL register:
TLB_coreB_I1.coreB_edt_single_chain_mode_logic_i.TLB_coreB_I1.coreB_edt_i
nternal_scan_registers_i.coreB_A_chain1[18]
180100ns: Corresponding design object:  coreB_A/u11/PRB_reg/DFF1
181700ns:  Turning off LogicBist controller TLB_coreB_I1.coreB_edt_lbist_i
```

Debug Based On Scan Cell Monitoring

You can have the tool monitor the scan chains and return information about scan cells associated with unexpected unload values.

Restrictions and Limitations

- Once the TCD file is created as described below, you cannot alter the core hierarchy (such as by ungrouping the LogicBIST controller). Altering the core hierarchy causes the list of monitor points in the TCD file to become out of sync.

Prerequisites

- You have performed the hybrid TK/LBIST flow through the pattern generation step.

Procedure

1. Verify the clocks as described in step 1 of "Debug Based on MISR Signature Divergence" on page 113.
2. If clock verification fails, investigate and fix possible causes as described in step 2 of "Debug Based on MISR Signature Divergence" on page 113.
3. Run Verilog simulation with the monitor_scan_cells LogicBIST debugging feature enabled as shown below. When specified, the tool monitors the scan chain output pins, detects when an unexpected value is unloaded, and reports which shift cycle and scan cell failed.

   Set the SimulationOptions/logic_bist_debug property in the PatternsSpecification wrapper as follows:

   ```
   SimulationOptions {
     logic_bist_debug : bist_registers_and_clock_verify;
   }
   ```

Results

When a mismatch occurs the tool first reports the scan chain output pin where the mismatch was observed, and then maps the mismatch to a pattern, shift cycle, and scan cell. For both messages it reports the simulated and expected values. If there is inversion between the scan cell and the scan out, the simulated/expected values on these two lines is different. If the failing scan cell is within a sub-chain of a hard module, then the message only reports the scan cell and not the pin of the scan cell that failed.

The following transcript example shows mismatches when the wrong values are observed on scan chain cells.

```
#ns: Pattern_set serial_load
#ns: Setting up controller xtea_tk_lbist_ip_tessent_lbist_i
#ns: Number of patterns : 3 (3 + 0 warm-up patterns)
#ns: Pattern Length : 2 #ns: Shift Clk Select : 0b00
#ns: Capture Phase Width : 0x20 Shift Clock Cycles
#ns: PRPG Seed : 0x3e0a
#ns: MISR Seed : 0x000000
#ns: Starting controller xtea_tk_lbist_ip_tessent_lbist_i in Normal mode,
patterns 0 to 2
#ns: Checking that the controller xtea_tk_lbist_ip_tessent_lbist_i DONE
signal is NO at the beginning of the test

#ns: Mismatch at pin xtea_tk_lbist_ip_tessent_edt_lbist_c0_inst/
tessent_persistent_cell_edt_scan_out_0_buf/Y, Simulated x, Expected 1
#ns: Corresponding scan cell for pattern 0 at shift cycle 0:
hard_mod2_inst1/OUT_R_reg_0_, Simulated x, Expected 0

#ns: Mismatch at pin xtea_tk_lbist_ip_dft_tessent_edt_lbist_c0_inst/
tessent_persistent_cell_edt_scan_out_0_buf/Y, Simulated x, Expected 0
#ns: Corresponding scan cell for pattern 0 at shift cycle 11:
hard_mod2_inst1/IN2_R_reg_3_, Simulated x, Expected 1

#ns: Mismatch at pin xtea_tk_lbist_ip_tessent_edt_lbist_c1_inst/
tessent_persistent_cell_edt_scan_out_2_buf/Y, Simulated x, Expected 1
#: Corresponding scan cell for pattern 1 at shift cycle 0: B_R_reg_2_/Q,
Simulated x, Expected 1
```

Usage Examples

Your debugging efforts may include debugging clock verification and MISR signature mismatches. In addition, to help with debugging, you can display passing capture clocks and BIST register values.

Example 1: Clock Verification Debugging

The following example illustrates how you can use the setup_and_clock_verification functionality to quickly find a problem with the clocking. Assume the design contains a PLL that generates three clocks. The tool inserts Tessent OCCs on each clock along with the following NCP Index Decoder that declares the NCPs, the clocks, and how many pulses occur in each NCP.

```
DftSpecification(cpu, gates2) {
   LogicBist {
      NcpIndexDecoder {
         Ncp(CLK1) {
            cycle(0): cpu_gates_tessent_occ_NX1_inst;
            cycle(1): cpu_gates_tessent_occ_NX1_inst;
         }
         Ncp(CLK2) {
            cycle(0): cpu_gates_tessent_occ_NX2_inst;
            cycle(1): cpu_gates_tessent_occ_NX2_inst;
         }
         Ncp(CLK3) {
            cycle(0): cpu_gates_tessent_occ_NX3_inst;
            cycle(1): cpu_gates_tessent_occ_NX3_inst;
         }
         Ncp(ALL) {
            cycle(0): cpu_gates_tessent_occ;
            cycle(1): cpu_gates_tessent_occ;
         }
         Ncp(ALL_1p) {
            cycle(0): cpu_gates_tessent_occ;
         }
      }
   }
}
```

During IP generation, the NCPs CLK1, CLK2, CLK3, ALL, and ALL_1p are specified to operate at 10%, 10%, 10%, 60%, and 10%, respectively.

```
set_lbist_controller_options -capture_procedures {CLK1 10 CLK2 10 CLK3 10
ALL 60 ALL_1p 10}
```

During pattern retargeting, you can enable the clock verification functionality using the logic_bist_debug property in the PatternsSpecification wrapper.

Example 2: MISR Signature Mismatch Debugging

The following example illustrates how to debug a MISR signature mismatch. For purposes of this example, a MISR signature mismatch is triggered by using a Verilog force statement on the gate pin A0, which fans into the register pin uINTR/SERVICE_LEVEL_0_reg/D. This emulates a stuck-at fault starting at pattern 97.

Suppose you have a testbench that runs from pattern 0 to 99 with LogicBIST debugging enabled with the logic_bist_debug property. For example:

```
PatternsSpecification(cpu,gates,signoff) {
  Patterns(lbist_normal) {
    SimulationOptions {
      logic_bist_debug : bist_registers_and_clock_verify;
    }
    TestStep(serial_load) {
      LogicBist {
        CoreInstance(.) {
          run_mode      : run_time_prog;
          begin_pattern : 0;
          end_pattern   : 99;
        }
      }
    }
  }
}
```

The following is a simulation transcript for this pattern set.

```
# Pattern_set lbist_normal
#  Setting up controller cpu_gate_tessent_lbist_i
#     Number of patterns  : 100 (100 + 0 warm-up patterns)
#     Pattern Length      : 72
#     Shift Clk Select    : 0b01
#     Capture Phase Width : 0x2 Shift Clock Cycles
#     PRPG Seed           : 0x597fc27a
#     MISR Seed           : 0x000000
#  Starting controller cpu_gate_tessent_lbist_i in Normal mode, patterns 0 to 99
#     Checking that the controller cpu_gate_tessent_lbist_i DONE signal is NO at the
beginning of the test
# Mismatch at pattern 98 for cpu_inst.cpu_gate_tessent_edt_lbist_i.misr: Expected = 0x11504d
Actual = 0x665059
# Mismatch at pattern 99 for cpu_inst.cpu_gate_tessent_edt_lbist_i.misr: Expected = 0xc71675
Actual = 0x33473f
#  Test Complete for controller cpu_gate_tessent_lbist_i
#     Checking that signal DONE is YES for controller cpu_gate_tessent_lbist_i
#  Checking results of controller cpu_gate_tessent_lbist_i
#     Expected Signature for controller cpu_gate_tessent_lbist_i: 0x397758
# Mismatch at pin           1 name      SIB_SCAN_OUT, Simulated 1, Expected 0
# Previous scan out : pin SIB_SCAN_OUT = cpu_gate_tessent_edt_lbist_i.misr[0]
# Mismatch at pin           1 name      SIB_SCAN_OUT, Simulated 1, Expected 0
# Previous scan out : pin SIB_SCAN_OUT = cpu_gate_tessent_edt_lbist_i.misr[1]
# Mismatch at pin           1 name      SIB_SCAN_OUT, Simulated 1, Expected 0
...
# Previous scan out : pin SIB_SCAN_OUT = cpu_gate_tessent_edt_lbist_i.misr[21]
# Mismatch at pin           1 name      SIB_SCAN_OUT, Simulated 1, Expected 0
# Previous scan out : pin SIB_SCAN_OUT = cpu_gate_tessent_edt_lbist_i.misr[22]
# Turning off LogicBist controller cpu_gate_tessent_lbist_i
```

You can now identify the failing flop associated with pattern 97 by creating a diagnostic LogicBIST pattern, as follows:

```
PatternsSpecification(cpu,gates,signoff) {
  Patterns(lbist_diag) {
    TestStep(diagnosis) {
      LogicBist {
        CoreInstance(.) {
          run_mode      : run_time_prog;
          begin_pattern : 97;
          end_pattern   : 97;
          DiagnosisOptions {
            extract_flop_data : on;
          }
        }
      }
    }
  }
}
```

The simulation transcript for this diagnostic pattern looks as follows:

```
# Pattern_set lbist_diag
#  Setting up controller cpu_gate_tessent_lbist_i
#     Number of patterns  : 1 (1 + 0 warm-up patterns)
#     Pattern Length      : 72
#     Shift Clk Select    : 0b01
#     Capture Phase Width : 0x2 Shift Clock Cycles
#     PRPG Seed           : 0x5c953748
#     MISR Seed           : 0x6f2d3a
#  Starting controller cpu_gate_tessent_lbist_i in Normal mode, patterns
97 to 97
#    Checking that the controller cpu_gate_tessent_lbist_i DONE signal is
NO at the beginning of the test
#  Test Complete for controller cpu_gate_tessent_lbist_i
#  Scanning out capture results of vector 97 for controller
cpu_gate_tessent_lbist_i
# Mismatch at pin          1 name      SIB_SCAN_OUT, Simulated 0, Expected
1
# Corresponding ICL register:
cpu_gate_tessent_single_chain_mode_logic_i.cpu_gate_tessent_edt_internal_
scan_registers_i.chain2[27]
# Corresponding design object:  uINTR/SERVICE_LEVEL_0_reg
#  Turning off LogicBist controller cpu_gate_tessent_lbist_i
```

The diagnostic pattern identifies uINTR/SERVICE_LEVEL_0_reg as the failing flop. You can find the cause of the failure by comparing the simulation waveform results against the LogicBIST fault simulation prediction in Tessent Visualizer.

Note

There are several items to consider in this example:

- Without enabling simulation debug, the only failure you would see is the final MISR signature that is scanned out and compared at the end of the pattern. To isolate the

pattern at which the MISR started to fail, you would have to rerun the simulation multiple times, possibly with a binary search. This can be time consuming for serial simulations for large designs.

- Simulation debug provides two mismatches on the MISR register, observed after the capture windows at patterns 98 and 99. Because the MISR comparison occurred before the pattern 98 scan chains were unloaded into the MISR, the MISR signature failure actually corresponds to the previous pattern, 97.

Example 3: Displaying Passing Capture Clocks and BIST Register Values

By default, the LogicBIST simulation debug feature only displays a message when it detects a capture clock or BIST register mismatch. Using two simulator plusargs, you can direct the simulator to display passing capture clock and BIST register values as well.

To show the passing BIST register comparisons, use the **+show_passing_regs** plusarg. As the simulation runs, the tool displays the MISR, PRPG, and low-power (if present) values for each pattern. For example:

```
# Pattern_set lbist_normal
#  Setting up controller cpu_gate_tessent_lbist_i
#     Number of patterns  : 3 (3 + 0 warm-up patterns)
#     Pattern Length      : 72
#     Shift Clk Select    : 0b01
#     Capture Phase Width : 0x2 Shift Clock Cycles
#     PRPG Seed           : 0x2070a3dc
#     MISR Seed           : 0x05aed3
#  Starting controller cpu_gate_tessent_lbist_i in Normal mode, patterns
25 to 27
#    Checking that the controller cpu_gate_tessent_lbist_i DONE signal is
NO at the beginning of the test
# Expected value 0x2070a3dc after initialization for
cpu_inst.cpu_gate_tessent_edt_lbist_i.lfsm_vec
# Expected value 0x11e23c4d after initialization for
cpu_inst.cpu_gate_tessent_edt_lbist_i.lbist_lp_mask_shift_reg
# Expected value 0x05aed3 after initialization for
cpu_inst.cpu_gate_tessent_edt_lbist_i.misr
# Expected value 0x4dd3bf15 at pattern 25 for
cpu_inst.cpu_gate_tessent_edt_lbist_i.lfsm_vec
# Expected value 0x67bb5c4f at pattern 25 for
cpu_inst.cpu_gate_tessent_edt_lbist_i.lbist_lp_mask_shift_reg
# Expected value 0x05aed3 at pattern 25 for
cpu_inst.cpu_gate_tessent_edt_lbist_i.misr
# Expected value 0x04b71551 at pattern 26 for
cpu_inst.cpu_gate_tessent_edt_lbist_i.lfsm_vec
# Expected value 0x434886a2 at pattern 26 for
cpu_inst.cpu_gate_tessent_edt_lbist_i.lbist_lp_mask_shift_reg
# Expected value 0x3a2db4 at pattern 26 for
cpu_inst.cpu_gate_tessent_edt_lbist_i.misr
# Expected value 0x7a9fb426 at pattern 27 for
cpu_inst.cpu_gate_tessent_edt_lbist_i.lfsm_vec
# Expected value 0x3f32bc25 at pattern 27 for
cpu_inst.cpu_gate_tessent_edt_lbist_i.lbist_lp_mask_shift_reg
# Expected value 0xf4968c at pattern 27 for
cpu_inst.cpu_gate_tessent_edt_lbist_i.misr
#  Test Complete for controller cpu_gate_tessent_lbist_i
#     Checking that signal DONE is YES for controller
cpu_gate_tessent_lbist_i
#  Checking results of controller cpu_gate_tessent_lbist_i
#     Expected Signature for controller cpu_gate_tessent_lbist_i:
0x268b4d#  Turning off LogicBist controller cpu_gate_tessent_lbist_i
```

To show the passing clock comparisons, use the **+show_passing_clocks** plusarg. As each pattern runs, the expected number of pulses for each clock displays, including the currently active NCP name. For example:

```
# Pattern_set lbist_normal
#  Setting up controller cpu_gate_tessent_lbist_i
#     Number of patterns  : 3 (3 + 0 warm-up patterns)
#     Pattern Length      : 72
#     Shift Clk Select    : 0b01
#     Capture Phase Width : 0x2 Shift Clock Cycles
#     PRPG Seed           : 0x2070a3dc
#     MISR Seed           : 0x05aed3
#  Starting controller cpu_gate_tessent_lbist_i in Normal mode, patterns
25 to 27#     Checking that the controller cpu_gate_tessent_lbist_i DONE
signal is NO at the beginning of the test
#  2 expected pulses at pattern 25 (NCP 'CLK1') for clock
'cpu_inst.cpu_gates_tessent_occ_NX1_inst.tessent_persistent_cell_clock_ou
t_mux.y'
#  0 expected pulses at pattern 25 (NCP 'CLK1') for clock
'cpu_inst.cpu_gates_tessent_occ_NX2_inst.tessent_persistent_cell_clock_ou
t_mux.y'
#  0 expected pulses at pattern 25 (NCP 'CLK1') for clock
'cpu_inst.cpu_gates_tessent_occ_NX3_inst.tessent_persistent_cell_clock_ou
t_mux.y'
#  0 expected pulses at pattern 26 (NCP 'CLK2') for clock
'cpu_inst.cpu_gates_tessent_occ_NX1_inst.tessent_persistent_cell_clock_ou
t_mux.y'
#  2 expected pulses at pattern 26 (NCP 'CLK2') for clock
'cpu_inst.cpu_gates_tessent_occ_NX2_inst.tessent_persistent_cell_clock_ou
t_mux.y'
#  0 expected pulses at pattern 26 (NCP 'CLK2') for clock
'cpu_inst.cpu_gates_tessent_occ_NX3_inst.tessent_persistent_cell_clock_ou
t_mux.y'
#  0 expected pulses at pattern 27 (NCP 'CLK2') for clock
'cpu_inst.cpu_gates_tessent_occ_NX1_inst.tessent_persistent_cell_clock_ou
t_mux.y'
#  2 expected pulses at pattern 27 (NCP 'CLK2') for clock
'cpu_inst.cpu_gates_tessent_occ_NX2_inst.tessent_persistent_cell_clock_ou
t_mux.y'
#  0 expected pulses at pattern 27 (NCP 'CLK2') for clock
'cpu_inst.cpu_gates_tessent_occ_NX3_inst.tessent_persistent_cell_clock_ou
t_mux.y'
#  0 expected pulses at pattern 28 (NCP 'CLK2') for clock
'cpu_inst.cpu_gates_tessent_occ_NX1_inst.tessent_persistent_cell_clock_ou
t_mux.y'
#  2 expected pulses at pattern 28 (NCP 'CLK2') for clock
'cpu_inst.cpu_gates_tessent_occ_NX2_inst.tessent_persistent_cell_clock_ou
t_mux.y'
#  0 expected pulses at pattern 28 (NCP 'CLK2') for clock
'cpu_inst.cpu_gates_tessent_occ_NX3_inst.tessent_persistent_cell_clock_ou
t_mux.y'
#  Test Complete for controller cpu_gate_tessent_lbist_i
#     Checking that signal DONE is YES for controller
cpu_gate_tessent_lbist_i
#  Checking results of controller cpu_gate_tessent_lbist_i
#     Expected Signature for controller cpu_gate_tessent_lbist_i: 0x268b4d
#  Turning off LogicBist controller cpu_gate_tessent_lbist_i
```

You can specify **+show_passing_clocks** and **+show_passing_regs** at the same time.

Chapter 6
Top-Level ICL Network Integration

In this step of the bottom-up flow, you use the top-level netlist that instantiates all of the LogicBIST implemented cores.

Note

You perform this step only when using the "Hybrid TK/LBIST Implementation" on page 16.

Chapter topics follow this sequence:

Top-Level ICL Network Integration Overview

For top-level ICL network integration, you use the top-level netlist that instantiates all the LogicBIST implemented cores. You insert IJTAG compliant SIBs to provide access to each of the LogicBIST cores as well as connect these SIBs and cores to the top-level TAP controller.

Network Integration Flow

- Load design configuration files (core level) using the TSDB from the LogicBist fault simulation

  Netlist, ICL, PDL, PatDB, TCD, Graybox signature

- System mode transition (DRCs)
- Pattern generation
- Testbench simulation
- Top-level ICL network integration
- Top-level ICL extraction and pattern retargeting

Performing Top-Level ICL Network Integration

It is recommended to shadow each core by a separate SIB to provide maximum flexibility for test scheduling. You can connect EDT control signals from the core to the top level as well at this time.

You can create a DftSpecification and use the Tessent Shell process_dft_specification functionality to automate this task.

Prerequisites

The following input is required for this step of the flow:

- The netlists for all your cores created in EDT and LogicBIST IP Generation.
- Top-level netlist with instantiation of cores and interconnect between them.

Procedure

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

   ```
   % tessent -shell
   ```

   After invocation, the tool is in unspecified setup mode. You must set the context before you can invoke the top-level SIB network Insertion commands.

2. Set the tool context to dft mode using the set_context command as follows:

   SETUP> set_context dft -no_rtl

3. Load the LogicBIST-ready design netlists using the read_verilog command. For example:

   SETUP> read_verilog top.v <tsdb_dft_inserted_designs_directory>/<core_name_1>.v <tsdb_dft_inserted_designs_directory>/<core_name_N>.v

4. Open the TSDB if it is not already open. For example:

   SETUP> open_tsdb tsdb_outdir

5. Load one or more cell libraries into the tool using the read_cell_library command. For example:

   SETUP> read_cell_library atpg.lib

6. Set the top design using the set_current_design command. For example:

   SETUP> set_current_design top

7. Implement an optional TCL proc named “process_dft_specification.post_insertion” that is executed after processing the DftSpecification and before writing out the top-level

IJTAG network-inserted design. This proc can be used to connect the EDT signals from the cores to top-level design pins. For example:

```
SETUP> proc process_dft_specification.post_insertion {root wrapper} {
  create_port edt_clock
  create_connection edt_clock [get_pins *_edt_i/edt_clock]
  …
}
```

8. Load the top-level IJTAG network description in DftSpecification format. For example:

```
SETUP> read_config_data top.dft_spec
```

9. Validate and implement the IJTAG network using the process_dft_specification command. This command generates the RTL description for the IJTAG network components and the top-level IJTAG network-inserted design.

```
SETUP> process_dft_specification
```

Results

Now you are ready to perform ICL Extraction and Pattern Retargeting.

Examples

The example in the following figure describes a design that has two cores, alu and cpu. The alu core has two EDT blocks named B1 and B2. Two instances of the alu core are in the final top-level design (/w2/A and /w2/B) and a single instance of the cpu core (/c1).

The example shows the Tessent Shell integration dofile for generating the IJTAG network and connecting the core-level EDT signals to the top level. The process_dft_specification.post_insertion TCL procedure connects the core-level EDT pins to the top level of the design. The example demonstrates the creation of shared top-level pins for all of the cores corresponding to the EDT control signals, such as edt_clock, edt_update, and edt_bypass. The example also shows the creation of dedicated top-level pins for channel inputs and outputs for each of the cores.

Figure 6-1. Top-Level ICL Network Integration Dofile Example

```
set_context dft -no_rtl

read_verilog top.v tsdb_outdir/dft_inserted_designs/alu_gate.dft_inserted_deisgn/alu.v
tsdb_outdir/dft_inserted_designs/cpu_gate.dft_inserted_deisgn/cpu.v

read_cell_library atpg.lib

read_icl {tsdb_outdir/dft_inserted_designs/alu_gate.dft_inserted_deisgn/alu.icl
tsdb_outdir/dft_inserted_designs/cpu_gate.dft_inserted_deisgn/cpu.icl}

set_current_design top

# TCL proc to connect EDT signals from block to top pins

proc process_dft_specification.post_insertion {root args} {

    foreach i {alu1_edt_channels_in1 alu1_edt_channels_in2
```

```
                alu2_edt_channels_in1 alu2_edt_channels_in2
                cpu_edt_channels_in1
                edt_clock edt_reset edt_update edt_bypass
                edt_single_bypass_chain} {
        create_port -direction input $i
    }
    foreach o {alu1_edt_channels_out1 alu1_edt_channels_out2
                alu2_edt_channels_out1 alu2_edt_channels_out2
                cpu_edt_channels_out1} {
        create_port -direction output $o
    }
    create_connection alu1_edt_channels_in1 w2/A/B1_edt_channels_in1
    create_connection alu1_edt_channels_in2 w2/A/B2_edt_channels_in1
    create_connection alu2_edt_channels_in1 w2/B/B1_edt_channels_in1
    create_connection alu2_edt_channels_in2 w2/B/B2_edt_channels_in1
    create_connection cpu_edt_channels_in1 c1/edt_channels_in1
    create_connection alu1_edt_channels_out1 w2/A/B1_edt_channels_out1
    create_connection alu1_edt_channels_out2 w2/A/B2_edt_channels_out1
    create_connection alu2_edt_channels_out1 w2/B/B1_edt_channels_out1
    create_connection alu2_edt_channels_out2 w2/B/B2_edt_channels_out1
    create_connection cpu_edt_channels_out1 c1/edt_channels_out1
    foreach i {edt_clock edt_reset edt_update edt_bypass
                edt_single_bypass_chain} {
        create_connection $i w2/A/$i
        create_connection $i w2/B/$i
        create_connection $i c1/$i
    }
}
# Insert IJTAG network using DFT specification
read_config_data top.dft_spec
process_dft_specification
```

The DftSpecification that is referenced in the dofile follows. In it, the HostScanInterface/Interface wrapper specifies the top-level TAP controller design pins for the IJTAG interface signals. Three SIBs are to be inserted, each of which controls a core whose instance path name

is specified in the DesignInstance wrapper. The SIBs are inserted one level above the core instance as specified in the parent_instance property for SIBs w2_A and w2_B. The SIB for core cpu (/c1) is inserted at the top level. The naming of the SIB instances is controlled using the leaf_instance_name property. The enable, control signal, and scan IO pin names (the IJTAG network interface at the core boundary) are taken from the ICL description of the cores in the *alu.icl* and *cpu.icl* files.

Figure 6-2. DftSpecification Example

```
DftSpecification(top, 3sibs_tap) {
  IjtagNetwork {
    HostScanInterface(3sibs_tap) {
      Interface {
        reset_polarity: active_high;
        tck: tck;
        reset: jtag/tlr;
        select: jtag/lbist_inst;
        capture_en: jtag/capture_dr;
        shift_en: jtag/shift_dr;
        update_en: jtag/update_dr;
        scan_in: tdi;
        scan_out: jtag/lbist_reg_out;
      }
      Sib(c1) {
        leaf_instance_name: piccpu_access_sib;
        DesignInstance(/c1) {}
      }
      Sib(w2_B) {
        parent_instance: /w2;
        leaf_instance_name: m8051_B_access_sib;
        DesignInstance(/w2/B) {}
      }
      Sib(w2_A) {
        parent_instance: /w2;
        leaf_instance_name: m8051_A_access_sib;
        DesignInstance(/w2/A) {}
      }
    }
  }
}
```

The RTL and ICL description of the generated IJTAG instruments is written out in the *tsdb_outdir/instruments/top_3sibs_tap_ijtag.instrument* directory. The top-level IJTAG network inserted design is written out as *tessent_outdir/dft_inserted_designs/top_3sibs_tap.dft_inserted_design/top.vg*. The final top-level netlist can be obtained by combining the *top.vg* file along with the gate-level synthesized netlists of the IJTAG instruments. The ICL files generated by this example can be used for downstream steps that use IJTAG, such as top-level LBIST pattern retargeting.

Note

For more information about ICL insertion using the DftSpecification, refer to the "IJTAG Network Insertion" chapter of the *TessentIJTAG User's Manual*.

Top-Level ICL Network Integration Command Summary

Tessent Shell provides a variety of commands that are used for top-level SIB network insertion.

Table 6-1. Top-Level ICL Network Integration Commands

Command	Description
create_connections	Connects, ports, pins, port objects.
create_instance	Instantiates an instance of a module mod_spec inside a design module of a current design.
create_port	Creates a port on a specified design module.
process_dft_specification	Validates and processes the content contained in a DftSpecification wrapper.
read_cell_library	Loads one or more cell libraries into the tool.
read_config_data	Loads a configuration data file into the Tessent Shell environment.
read_verilog	Reads one or more Verilog files into the specified or default logical library.
set_context	Specifies the current usage context of Tessent Shell. You must set the context before you can enter any other commands in Tessent Shell.
set_current_design	Specifies the top level of the design for all subsequent command until reset of another execution of this command.
write_design	Writes the current design to the specified file in Verilog netlist format.

Chapter 7
ICL Extraction and Pattern Retargeting

In this step of the bottom-up flow, you perform ICL extraction and pattern retargeting.

Note

You perform this step only when using the Hybrid TK/LBIST Implementation.

ICL Extraction and Pattern Retargeting Overview

You use the fully integrated top-level netlist, top-level ICL/PDL files for your IJTAG instruments such as the TAP controller and clock controller, and the per-core LogicBIST files generated at the core level. The final tester patterns can be written out in any of the supported formats.

ICL Extraction and Pattern Retargeting Flow

- EDT and LogicBIST IP Generation

  Inputs: PDL and ICL files created during EDT and LogicBIST IP Generation

- Pattern Retargeting

  Inputs: PatDB and TCD files created during Pattern Generation

Top-level PDL and ICL files with information about the TAP and SIBs are also required.

Performing ICL Extraction and Pattern Retargeting

ICL extraction can be used here instead of manually creating the top-level ICL describing the SIB access network and connectivity between your instruments and the LogicBIST cores.

Prerequisites

The required inputs for this step of the flow are as follows:

- PDL and ICL files for each core in your design created during EDT and LogicBIST IP Generation.
- PatternDB and TCD files for each core in your design generated during Pattern Generation.
- Top-level PDL and ICL files that include information about the TAP and SIBs.

Procedure

1. Set the context:

   ```
   set_context pattern -ijtag -design_id gate
   ```

2. Read the cell library:

   ```
   read_cell_library ./lib/tessent/adk.tcelllib ./lib/tessent/picdram.atpglib
   ```

3. Set the location of the TSDB (default: current working directory):

   ```
   set_tsdb_output_directory ./tsdb_outdir
   ```

4. Load and elaborate the design:

   ```
   read_design piccpu
   set_current_design piccpu
   ```

5. Add design constraints and define clocks:

   ```
   add_clocks 0 clk -period 100ns
   add_input_constraint scan_en -c0
   ```

6. Perform system mode transition and rule checks:

   ```
   set_system_mode analysis
   ```

7. Read the PatternsSpecification:

   ```
   read_config_data ./lbist_mbist_pattern.patspec
   ```

The PatternsSpecification is as follows:

```
PatternsSpecification(piccpu,gate,signoff) {
  AdvancedOptions {
    ConstantPortSettings {
      scan_en : 0;
    }
  }

  Patterns(LogicBist_piccpu) {
    ClockPeriods {
      clk : 100.00ns;
    }
    TestStep(serial_load) {
      LogicBist {
        CoreInstance(.) {
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 7;
        }
      }
    }
  }
}
```

8. Generate patterns:

   process_pattern_specification

9. Set any requirements for simulations and simulate the retargeted patterns:

```
set_simulation_library_sources -v \
  { ./lib/verilog/adk.v ./lib/verilog/picdram.v  }
run_testbench_simulation
```

Usage Examples for ICL Extraction and Pattern Retargeting

You can use pattern merging commands to process your core patterns in parallel or sequentially.

Example 1

This example merges all the core patterns to be run in parallel.

```
# Set context for pattern retargeting
set_context patterns -ijtag
# Read output design from the top-level ICL network integration step
read_verilog top_lbist_integrated.vread_cell_library atpg.lib
# Read ICL for cores extracted during block-level LogicBIST pattern
#    generation
read_icl {alu.icl cpu.icl}
# Read user-provided ICL for top-level components
read_icl {top_sib.icl jtag_controller.icl}
# Set current design and report ICL-matched modules
set_current_design top
report_module_matching -icl

# Define top-level clocks and pin constraints
add_clocks 0 refclk -pulse_always
add_clocks 0 tck
add_input_constraints RST -c0
add_input_constraints edt_reset -c0
# Change to analysis mode and perform ICL extraction
set_system_mode analysis
# Report user settings
report_clocks
report_input_constraints
# Create pattern specification, or you can read in a previously created
# patterns specification file by using the read_config_data command
create_patterns_specification
process_patterns_specification
```

Example 2

This example shows the pattern merging commands required for running 100 patterns of all cores sequentially. The initial setup is the same as the previous full example.

```
PatternsSpecification(top,gate,signoff) {
  Patterns(LogicBist_Top) {
    TestStep(cpu_serial_load) {
      LogicBist {
        CoreInstance(cpu) {
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 99;
        }
      }
    }    TestStep(alu1_serial_load) {
      LogicBist {
        CoreInstance(alu1) {
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 99;
        }
      }
    }
    TestStep(alu2_serial_load) {
      LogicBist {
        CoreInstance(alu2) {
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 99;
        }
      }
    }
  }
}
```

Example 3

The following example shows a pattern specification that you can use to run 100 patterns of cpu in parallel with 50 patterns of an alu core, followed by another alu core running 50 patterns by itself.

```
PatternsSpecification(top,gate,signoff) {
  Patterns(LogicBist_Top) {
    TestStep(cpu_and_alu1_serial_load) {
      LogicBist {
        CoreInstance(cpu) {
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 99;
        }
        CoreInstance(alu1) {
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 49;
        }
      }
    }
```

```
    TestStep(alu2_serial_load) {
      LogicBist {
        CoreInstance(alu2) {
          run_mode : run_time_prog;
          begin_pattern : 0;
          end_pattern : 49;
        }
      }
    }
}
```

ICL Extraction and Pattern Retargeting Command Summary

Tessent Shell provides a variety of commands for ICL extraction and pattern retargeting.

Table 7-1. ICL Extraction and Pattern Retargeting Commands

Command	Description
create_patterns_specification	Generates a pattern specification for the specified usage. The usage is either signoff or manufacturing.
process_patterns_specification	Validates and processes the content of the PatternsSpecification wrapper.
read_cell_library	Loads one or more cell libraries into the tool.
read_icl	Loads one or more ICL files into the tool.
read_verilog	Reads one or more Verilog files into the specified or default logical library.
set_context	Specifies the current usage context of Tessent Shell. You must set the context before you can enter any other commands in Tessent Shell.

Chapter 8
Hybrid TK/LBIST Embedded Structures

The EDT/LogicBIST hybrid IP reduces the area overhead compared to a separate implementation of EDT and LogicBIST IP. This is done by re-using parts of the IP for both EDT and LogicBIST modes. This hybrid logic controls input stimuli generation and output response comparison and is implemented separately for each EDT block. The top-level controller including the LogicBIST FSM is then connected to these hybrid IP inserted EDT blocks.

The hybrid TK/LBIST flow utilizes the following during generating and embedding the EDT/LogicBIST IP into your design.

- EDT and LogicBIST blocks (blocks and ELT cores)
- A decompressor
- Low-Power Shift controller
- LogicBIST controller
- EDT controller
- EDT compactor
- MISR
- Bypass logic

Shared Logic

The EDT/LogicBIST hybrid IP is shared in a number of ways.

The sharing is accomplished as follows:

- The EDT decompressor is re-configured as the PRPG by blocking the channel inputs during LogicBIST mode.
- Lockup cells (those placed in between the decompressor and the phase shifter) are re-used as hold cells when low-power LogicBIST is implemented.
- Biasing gates used when synthesizing EDT low-power hardware is shared with chain masking.
- The phase shifter network is used as is for driving scan chains from the PRPG.
- The spatial compactor XOR network is re-used for compacting scan chain outputs into MISR inputs.
- Lockup cells required between the EDT/LogicBIST IP and the design scan cells are shared between both modes.

Inserted Hybrid TK/LBIST IP

The hybrid TK/LBIST IP is inserted during the IP generation step. The same DFT architecture serves two functions: provide a mechanism for compression-based ATPG through EDT (decompressor, compactor, and bypass logic) and provide LogicBIST test capability.

For LogicBIST test, the inserted IP includes a:

- Pseudo Random Pattern Generator (PRPG)

  LogicBIST generates patterns internally using a PRPG. In the hybrid TK/LBIST architecture, the PRPG is shared with the decompressor, which internally contains a linear-feedback shift register (LFSR). When a pattern runs, the PRPG seed determines the value to be expected on the MISR signature. In LogicBIST mode, the PRPG (decompressor) is controlled by the lbist_en and the lbist_prpg_en signals, which trigger test input generation to the design under test (DUT).

- Multiple Input Signature Register (MISR)

  The MISR is a programmable register that connects to the output of the compactor; it is part of the shared EDT IP. During LogicBIST mode, the accumulate input (misr_accumulate_en) tells the MISR when to accumulate/capture the output of the compactor during the shift state. The size of the MISR varies depending on the DUT and the Tessent command options specified, but it is comprised of 24-bit and 32-bit segments.

The hybrid TK/LBIST flow generates hybrid IP that consists of the following three blocks:

- Hybrid EDT/LogicBIST controller

The hardware shared between EDT and LogicBIST consists of the decompressor/ PRPG, the compactor, and the bypass. You can identify this block by its module name or instance name with the prefix *designname_designID*_edt_lbist. This module is described in the instruments directory of the TSDB.

- LogicBIST controller

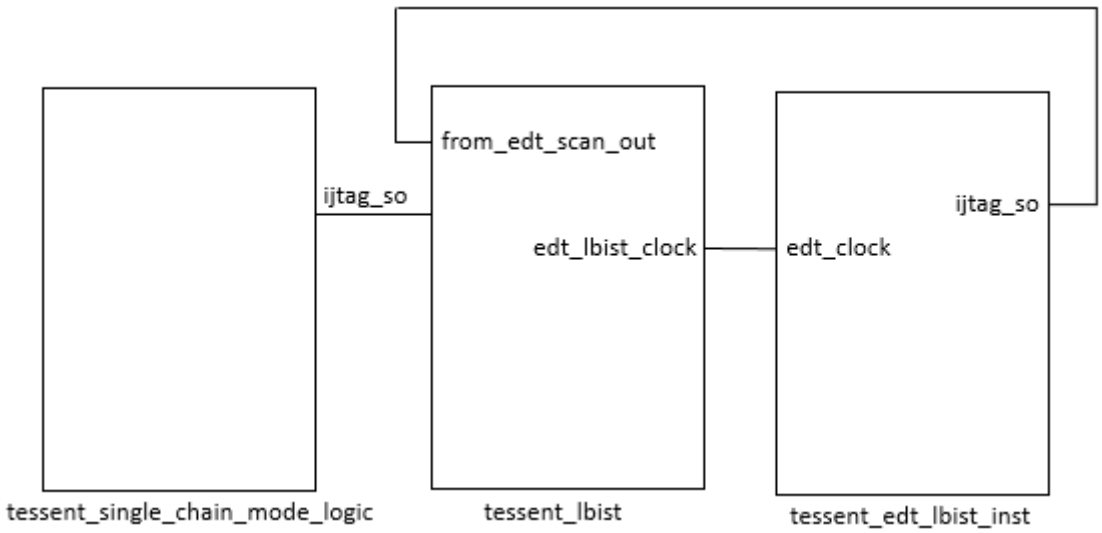
  This block contains hardware that is only used during LogicBIST mode and does not play a significant role during ATPG, except for some clock paths and multiplexed control signal lines that travel through this block. You can identify the controller in the netlist with the prefix *designname_designID*_lbist. This module is described in the instruments directory of the TSDB.

  The LogicBIST controller drives the shared hybrid EDT/LBIST block during LogicBIST mode and functions as a propagation path for the scan chain in the single chain mode.

- Single chain mode controller

  This module is responsible for setting the design mode to single chain mode for LogicBIST diagnosis. This block is optional.

The LogicBIST controller connects to the hybrid EDT/LBIST controller and the single chain mode controller as follows:



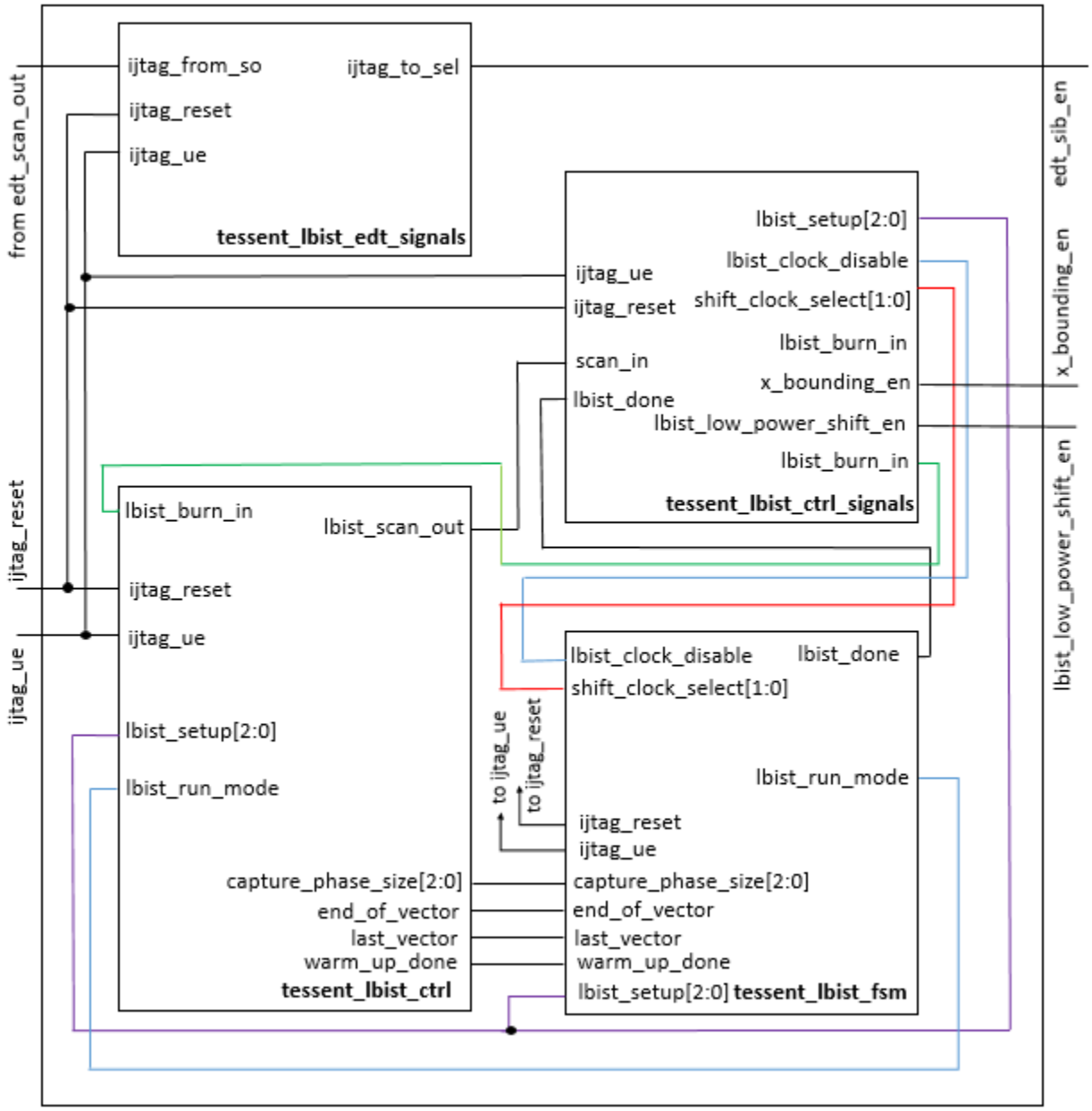
The LogicBIST Controller

Internally, the LogicBIST controller block (tessent_lbist) contains the following blocks:

- Control signals: tessent_lbist_ctrl_signals
- Finite state machine (FSM): tessent_lbist_fsm

- LogicBIST control: tessent_lbist_ctrl
- SIB that enables the shared EDT/LogicBIST logic: tessent_lbist_edt_signals
- Combinational cloud and some persistent cells feeding into the outputs. (Not discussed.)

The modules are connected as follows. Colored connections are for clarity only.



Control Signals Block

The control signals block generates all necessary control signals to the LogicBIST controller, the hybrid EDT/LBIST controller, and the NCP decoder block, and it generates the lbist_clock_disable signal.

The tool asserts the lbist_clock_disable signal during setup so that the values being shifted are not disturbed. This signal also suspends the clock to the MISR, PRPG, counters, and so on, when switching the clock from TCK to the chosen shift clock (shift_clock_src, by default), and vice versa. This prevents clock glitches stemming from the 3:1 mux from disturbing flops that were just loaded (and scan flops during a LogicBIST diagnostic scan out). The signal is de-asserted during shift.

The outputs primarily fan out and drive the FSM and the LogicBIST controller. The LogicBIST enable, burn-in controls, clock selection signal, and low-power shift enable are outputs of this block that control LogicBIST operation.

This block connects to the IJTAG network and is part of the LogicBIST scan chain.

FSM Block

The seven-state FSM ensures that signals generated by the control signals block reach the LogicBIST control in the required sequence. That is, the FSM shapes the control signals such as lbist_reset, lbist_run_mode, and lbist_enable so that they toggle correctly. The FSM also ensures that the prpg_en signal reaches the PRPG, and the misr_accumulate_enable signal reaches the MISR.

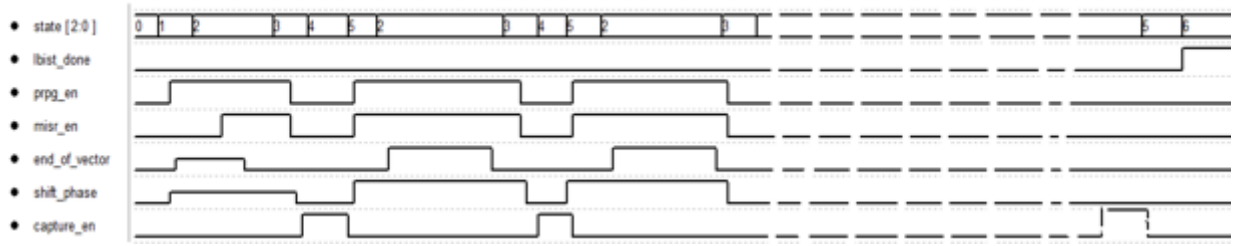
The state register in the FSM determines the state of the FSM. During LogicBIST, the FSM begins in the IDLE state and changes states from IDLE through CAPTURE_PAUSE. If there are pending patterns in the CAPTURE phase, the state returns to SHIFT. These iterations continue until the LogicBIST test is done.

Value of State Register	State Name	Description
0	IDLE	LogicBIST tests are not run in this state. The controller is at rest.
1	INIT	The FSM prepares the LogicBIST controller to start shift procedure.
2	SHIFT	Start shift procedure as part of LogicBIST. Test vectors are applied.
3	SHIFT_PAUSE	Shift procedure comes to a stop to prepare for capturing the responses.
4	CAPTURE	Once the test inputs are applied, the system captures the response stimulus.

Value of State Register	State Name	Description
5	CAPTURE_PAUSE	The captured response and the MISR are triggered to enable signature generation for comparison. If there are more patterns to be applied, the PRPG is enabled and the state returns to SHIFT. When all patterns are generated and responses captured, the state changes to DONE.
6	DONE	Marks the end of the LogicBIST run and what follows is signature recognition and whether it is as expected or not.

The following timing diagram applies to the seven values of the state register:

Figure 8-1. Timing Diagram for the FSM



LogicBIST Control

The LogicBIST control block contains the LogicBIST controller block, which is controlled by the FSM. The LogicBIST controller block contains a counter and the registers responsible for warm up and capture. It also controls which NCP is active.

The outputs of the LogicBIST controller block control the hybrid EDT/LBIST block.

Integrating the Hybrid TK/LBIST Two-Pin Serial Port Interface

The Two Pin Serial Port (TPSP) interface is an overlay of the TAP that may have two, three, or four top-level pins to access and initiate test with any instance on the IJTAG network. The TPSP does not require any special handling when integrated with the Hybrid TK/LBIST flow. See "DftSpecification/IjtagNetwork/HostScanInterface/TwoPinSerialPort" in the *Tessent Shell Reference Manual* or "How to Avoid Simulation Issues When Using the Two-Pin Serial Port Controller" in the *Tessent Shell User's Manual* for more information on the TPSP.

Note

TCK is generated inside TPSP with a frequency three times lower than the TPSP clock. To create one cycle of pure IJTAG data, you need three cycles of TPSP. When TCK is a BIST clock, this is not an efficient solution because the test clock is slow.

Scan Chain Masking

The hybrid TK/LBIST IP consists of a LogicBIST controller interfacing with multiple hybrid EDT/LBIST blocks connected to scan chains. The EDT logic includes a chain mask register that you can use to mask certain scan chains from reaching the MISR. For example, you can mask chains that contribute Xs, perhaps because they contain scan cells that fail timing or observe unknown values.

Specified chain masking is always applied to the scan outputs and can also be used to optionally mask the scan inputs. This is in addition to the per-patterns output masking performed by the EDT logic (either 1-or-all or Xpress compactors). The per-pattern masking only applies when the hybrid IP is operated in EDT mode. Chain mask registers are supported in both EDT and LogicBIST modes.

By default, scan chain masking occurs on a per-chain basis with a 1:1 ratio of chain mask register bits to number of scan chains. Specify per-chain scan masking by loading the chain masking register with the add_chain_masks command. Per-chain masking is applicable to all patterns in the test set.

For large designs, the area impact of the chain mask register can be more than the controllers themselves. It can also significantly increase the number of setup cycles to start BIST test. Optionally, you can use a single chain mask register bit to mask a group of chains rather than only one chain by specifying the set_edt_options -chain_mask_register_ratio option or the EDT/Controller/LogicBistOptions/chain_mask_register_ratio wrapper property.

You can specify the ratio on a per EDT block basis, and the ratios can be different, with some blocks using single scan chain masking and others using shared masking.

When you specify a chain mask register ratio greater than 1, the EDT logic changes so that you have a smaller chain mask register. The ICL, PDL, TCD and patdb files reflect the smaller register size. Converting masking information from individual scan chains to masking bit groups occurs automatically during LogicBIST fault simulation and EDT pattern generation.

New LogicBIST Control Signals

The new LogicBIST mode control signals are added to the EDT IP when using the hybrid hardware.

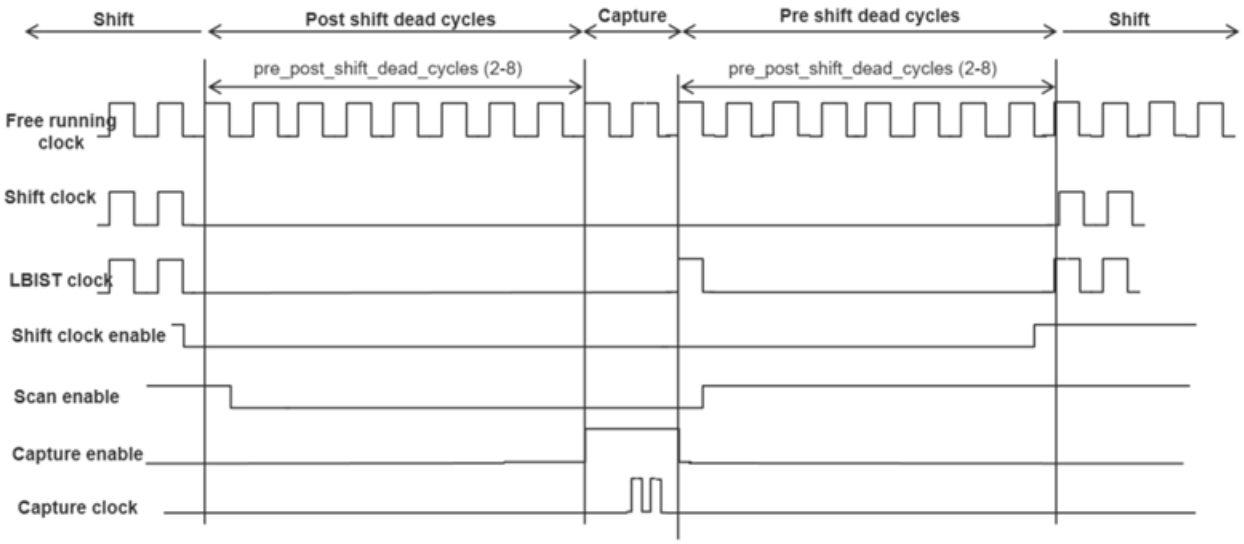
The lbist_en control signal determines whether the hybrid EDT operates in EDT mode (lbist_en=0) or LogicBIST mode (lbist_en=1). There are several other LogicBIST control signals like lbist_reset, lbist_prpg_en, and lbist_misr_en added to the Hybrid IP, which are driven by the common LogicBIST controller. These control signals have no impact during EDT mode of operation, and they are driven appropriately during LogicBIST mode by the LogicBIST controller.

Clocking

The input clock to the LogicBIST controller is a free-running clock, which could be a fast PLL output clock.

Figure 8-2 presents a timing diagram for EDT/LogicBIST IP.

Figure 8-2. Timing Diagram for LogicBIST



To enable the control signals to change between shift and capture, *8* empty (pause) cycles are introduced in between transitions, which helps with timing closure of the test logic.

- SE is a scan enable signal that goes to all the scan flops in the design. It transitions half-way through the pause states.
- shift_clock_en is a gating signal that could be gated with the free-running input clock to generate the shift-clock for the scan cells. This gating logic can either be implemented by the tool (when using set_clock_controller_pins shift_clock), or you can generate it as part of your clocking logic (when using set_clock_controller_pins shift_clock_en).
- The capture enable signal indicates the start of the capture cycle. This is intended as a trigger for the logic that generates programmable capture sequences. This signal is connected to the scan enable pin of your clock controller.
- The capture clock signal is shown here for illustration purposes only; this signal is generated inside your clock controller.
- The BIST clock signal is supplied to all hybrid EDT/LogicBIST blocks as well as used internally by the BIST controller. This clock is pulsed during shift and OFF during capture. It also has a pulse during capture pause state to operate the low-power BIST logic in the hybrid IP as well as to reset certain registers for each pattern in the BIST controller.

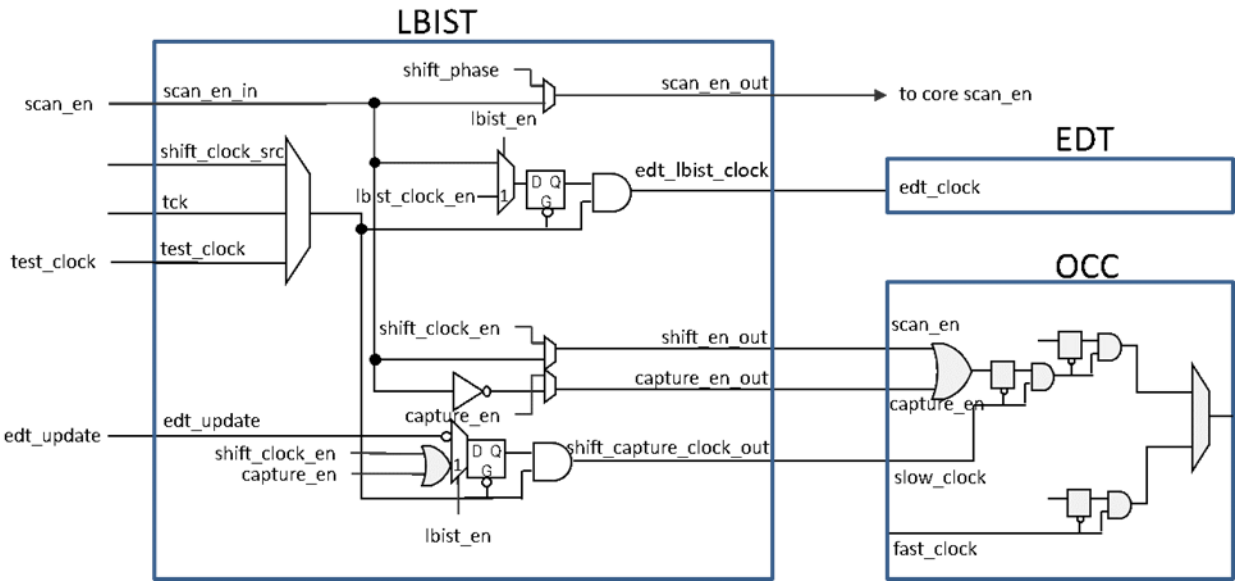
You specify the clock controller signals using the set_clock_controller_pins command.

As stated previously, the LogicBIST clock is typically a free-running clock, but the EDT clock should be controllable during test as it is pulsed during load_unload but held at off-state during capture. When the test clock is a top-level clock, it can be shared for both EDT and LogicBIST modes. When an internal clock, such as a free-running output of a PLL, is to be used for LogicBIST, then separate clock sources are required for EDT and LogicBIST modes, where the EDT mode clock is still controllable during test. An example of such a configuration is when shifting during LogicBIST is to be done at a higher speed than during EDT mode. TCK is used for seeding of specific values in the LogicBIST registers.

When internally generated functional clocks are used in the design, a top-level shift clock is required for shifting in EDT mode. Typically, a clock controller is used to generate the exact sequence of capture clocks and also to switch between shift-mode and capture-mode clocks. The clock controller takes a free-running clock, shift clock, and scan enable as inputs.

When you are using Tessent OCCs and have specified the add_dft_signals edt_clock -create_from_other_signals command, the clocking hardware after LogicBIST and EDT insertion looks as follows.

Figure 8-3. Hybrid TK/LBIST Clocking



Programmable Registers Inside Hybrid IP

Some of the registers inside the per-block hybrid IP are programmable.

The following registers are programmable:

- PRPG
- Low-power control registers (toggle, hold, switching, and mask shift)

- Chain mask register
- MISR

The following registers inside the top-level BIST controller are programmable.

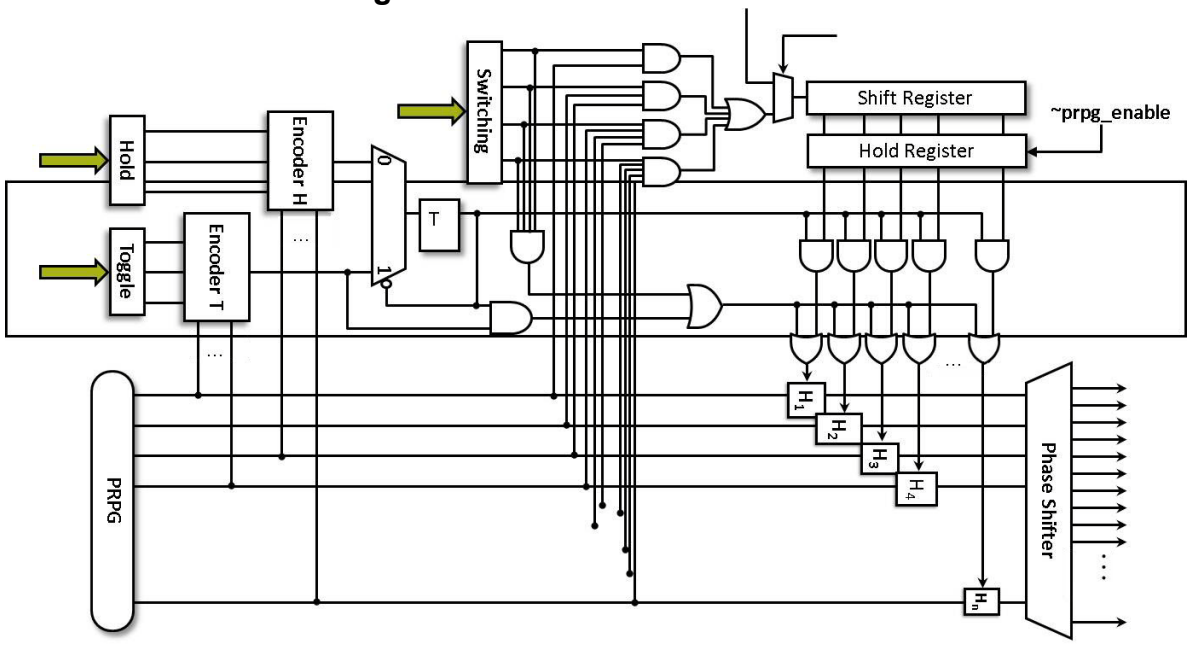
- Capture phase size
- Shift clock select
- Scan length counter (bit or byte counter depending on how you specify the set_lbist_controller_options -shift_counter_resolution command)
- Pattern counter (vector counter)
- BIST enable signal registers

Low-Power Shift Controller

The low-power scheme controls the switching activity during “shift” to reduce power consumption.

The following figure shows the overall architecture of the low-power shift controller. The existing Linear Feedback Shift Machine (LFSM) lockup cells are replaced by a hold register (denoted by Hi) which load select signal is controlled by the low-power control logic. By controlling when these hold registers update versus when they hold previous cycle values, the overall switching at the scan chain inputs can be controlled.

Figure 8-4. Low-Power Controller



The performance of the Low-Power BIST controller depends on the following factors:

- The switching code (SC)
- The Hold value (HV)
- The Toggle value (TV)

The SC, HV, and TV values are automatically calculated by the tool based on the switching threshold you set using the set_power_control command during fault simulation.

The 4-bit switching code might assume one of 15 different binary values ranging from 0001 up to 1111. The last value can be alternatively used to disable the control register and enter the poorly pseudo-random test pattern generation (unless you activate the Hold mode). All codes are used to enable certain combinations of AND gates forming biasing logic, and, hence, to produce 1s with probabilities 0.5 (0001), 0.25 (0010), 0.125 (0100), 0.0625 (1000), plus their combinations obtained due to an additional OR gate. The resultant 0s and 1s are shifted into the mask shift register, and, subsequently, they are reloaded to the mask hold register at the beginning of a test pattern to enable/disable the hold latches placed between the ring generator and its phase shifter.

The duration of how long the entire generator remains in the Hold mode with all latches temporarily disabled regardless of the hold register content.

In the Toggle mode (its duration is determined by TV), the latches enabled through the control register can pass test data moving from the ring generator to the scan chains. In order to switch between these two modes, a weighted pseudo-random signal is produced by a module Encoder H/T based on the content of different stages of the ring generator.

Chapter 9
Tessent OCC for Hybrid TK/LBIST

The Tessent On-Chip Clock Controller (OCC) can be used in the Hybrid TK/LBIST flow. Tessent Shell can generate and insert the Tessent OCCs with programmable capture clock sequences for use with Hybrid TK/LBIST applications.

During LBIST mode, the actual clock sequence is parallel loaded into the Tessent OCC. You can configure the Tessent OCC so that the values can be loaded through OCC module input ports or through a TDR inside the OCC. When an LBIST test uses only one NCP at a time, this value can be loaded through the TDR or be available as a constant at the module inputs. If the LBIST test uses multiple NCPs, then the tool generates the parallel load clock sequence for the currently active NCP, the index for which is provided by the LBIST controller using the NCP Index Decoder (NCPID). The NCPID hardware is generated during Hybrid TK/LBIST insertion. For additional information, see "NCP Index Decoder" on page 151.

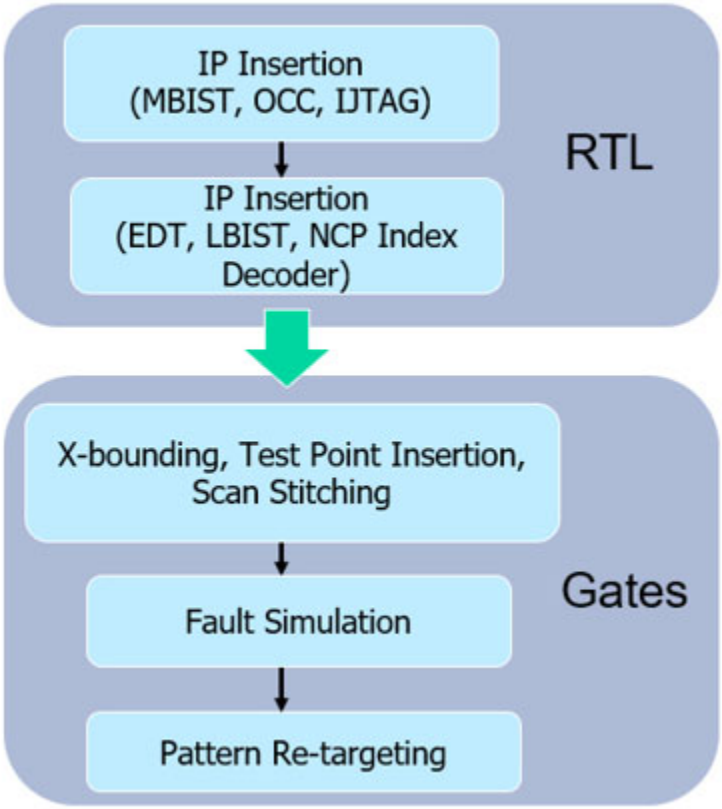
When Tessent OCC is generated with internal IJTAG control (that is, you have specified the Occ/ijtag_host_interface property), the static signals for controlling the OCC for LBIST mode are included within the OCC. Additionally, when static_clock_control is either internal or both, a TDR is included for generating the LBIST capture clock sequence. However, you can use this internal TDR only when LBIST test uses only one active NCP.

For additional Tessent OCC-specific information, see "Tessent On-Chip Clock Controller" in the *Tessent Scan and ATPG User's Manual*.

Tessent OCC TK/LBIST Flow

You can perform Tessent OCC insertion during either MBIST insertion or hybrid TK/LBIST IP insertion. Use the information in this chapter as a guide to configure and insert Tessent OCC and to interface it with the hybrid TK/LBIST controller.

Figure 9-1. Modified TK/LBIST Flow for Tessent OCC



Tessent OCC for TK/LBIST Flow Configuration

The clock controller can be used in two modes for LBIST: using a static sequence loaded through an ICL network for all patterns in a session; or, using a set of sequences that are cycled through every 256 patterns.

A Tessent OCC configured with *capture_en* as the capture trigger is used for hybrid TK/LBIST flow The slow clock input of the Tessent OCC is connected to TK/LBIST reference input clock in LBIST mode. The scan enable input of the Tessent OCC is connected to LBIST shift enable output in LBIST mode. The capture enable input of the Tessent OCC is connected to the LBIST capture enable output in LBIST mode. The tool adds muxes inside the LBIST controller to choose between LBIST and ATPG mode signals.

TCD for the Clock Controller

The tool writes out a Tessent Core Description file after you run the process_dft_specification command when processing the clock controller wrapper. This file is written out in the instrument directory for the clock controller with a *.tcd* filename extension. This file is one of the inputs to Tessent Scan to perform scan insertion.

NCP Index Decoder

The NCP index decoder is a simple combinational logic block that decodes the NCP index output of the Hybrid TK/LBIST controller into clock sequences to be generated by the Tessent OCCs across all capture procedures.

The LBIST controller generates an NCP index that cycles through the NCPs based on the active percentage for each NCP. This NCP index is decoded to provide the actual clock sequence that is parallel loaded to the OCC.

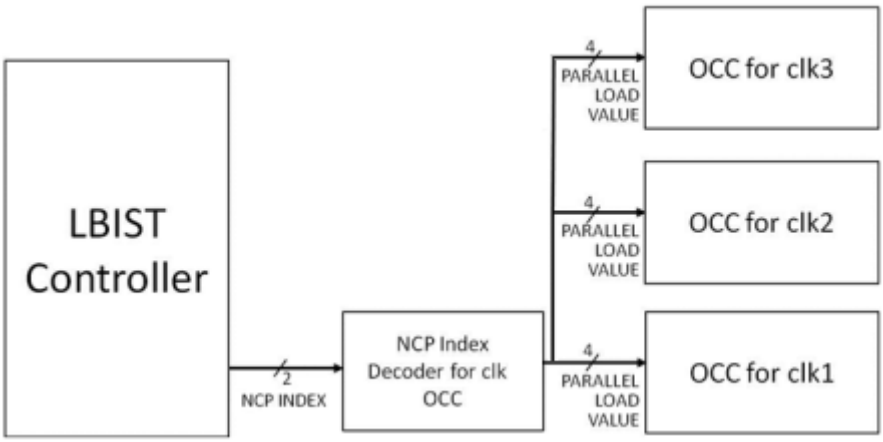
You must specify the clocking combinations to be used during TK/LBIST test. The tool synthesizes the NCP index decoder and generates named capture procedures based on this description.

You can use the NCP index decoder with only a single clock domain. The NCP index decoder is based on the number of unique clocking waveforms, not on the number of clocks. For example, with a single clock you can generate two NCPs (a single pulse and double pulse).

To reduce the test time and achieve high coverage, it is possible to activate multiple clock domains at the same time. This is a trade-off between test time and hardware cost: the cost comes from adding bounding logic for paths crossing clock domains. You may need bounding for both stuck-at and transition patterns. Coverage is lost in all blocked paths, but you can control the blocking with the McpBoundingEn dft_signal. It is possible to disable blocking at run time, but the NCPs can only pulse compatible clocks.

One NCP index decoder is synthesized for each LogicBIST controller and can be used for controlling all the OCCs involved in LogicBIST. In Figure 9-2, there are three OCCs configured for four cycles each. There are two input binary values to the NCP index decoder (indicating a maximum of four NCPs), which is decoded as a single control signal per OCC per cycle that reflects the required clocking waveform.

Figure 9-2. NCP Index Decoder Connections



The tool generates only one index decoder for all OCCs. The NCP index decoder is instantiated by default at the top level, or as controlled with the parent_instance property of the NcpIndexDecoder specification.

NCP Index Decoder Creation

The tool inserts the NCP index decoder when the hybrid TK/LBIST IP is generated. Specify the NCP index decoder when you create the hybrid TK/LBIST IP, using the DftSpecification/LogicBist/NcpIndexDecoder wrapper. For a complete description and usage, see NcpIndexDecoder in the *Tessent Shell Reference Manual*.

If you are using only one NCP, you cannot use the NcpIndexDecoder wrapper because it is supported only for external static clock controls and two or more NCPs. Refer to "Considerations When Only Using One NCP" on page 159 for fault simulation considerations.

When the NcpIndexDecoder is generated in the same run as the LogicBist IP, the NCP count is automatically inferred from the number of Ncp() wrappers in the NcpIndexDecoder wrapper. When NcpIndexDecoder is generated in a different run, you must specify the LogicBist/Controller/NcpOptions/count property = 1.

NCP Index Decoder Creation at the Gate Level

To create the NCP index decoder at the gate level, use the following flow. You can use the existing DftSpecification for the gate-level design used to insert the Hybrid TK/LBIST logic. When you generate the NCP index decoder, synthesize it separately and read the resulting logic in later steps such as fault simulation and pattern generation.

Note

The NCP index decoder generation requires an elaborated design. For the RTL flow, you need the TCD of the OCC. For the gate-level flow, add the Tessent OCC instances with the add_core_instances command.

Examples

In the following example, assume the design has two top-level Tessent OCC instances named *m8051_gate_tessent_occ_clk1_inst* and *m8051_gate_tessent_occ_clk2_inst* of the same Tessent OCC module *m8051_gate_tessent_occ*.

```
set_context dft -no_rtl
set_tsdb_output_directory ../tsdb_outdir
# You may need to read the synthesized design separately using
# read_verilog content if it is synthesized outside of Tessent
# shell using third-party synthesis tools.
read_design rtl2 -design_id htklb_insertion
read_cell_library atpg.lib
set_current_design
set_design_level physical
add_core_instances -module m8051_gates_tessent_occ
read_config_data -from_string {
  DftSpecification(m8051,rtl2) {
    LogicBist {
      NcpIndexDecoder {
        Connections {
          NcpIndex: m8051_lbist_i/ncp;
        }
        Ncp(stuck) {
          cycle(0): m8051_gate_tessent_occ;
          //Specified using module name, refers to both the OCC instances.
        }
        Ncp(clk1_double_pulse) {
          cycle(0): m8051_gate_tessent_occ_clk1_inst;
          cycle(1): m8051_gate_tessent_occ_clk1_inst;
        }
        Ncp(clk2_double_pulse) {
          cycle(0): m8051_gate_tessent_occ_clk2_inst;
          //Note - cycle(1) is omitted, so no clock activity
          cycle(2): m8051_gate_tessent_occ_clk2_inst;
        }
      }
    }
  }
}
process_dft_specification
```

OCC Generation and Insertion

You can insert OCC either during MBIST insertion or during logic test insertion. Create the OCC-specific DftSpecification to interface the hybrid controller to the OCC.

You should add a separate OCC DftSpecification with controller wrappers for each different clock that needs to be programmable during capture. Additionally, you may add OCCs for asynchronous reset signals declared as a clock.

Tessent Shell creates TCD files, Verilog RTL, ICL, PDL, and tcd_scan describing the Tessent OCC instrument, as well as a Verilog netlist that instantiates the OCC in the user design. Many generation and insertion options are available in the DftSpecification to control this process. The ICL and TCD outputs are used in later steps like scan insertion, fault simulation, and pattern generation to describe the configuration of the generated OCCs as well to identify the port functions. The Tessent OCC RTL should be synthesized to a gate-level design, along with other logic to be inserted at the RTL level before be used for downstream steps that require a gate-level netlist.

The Tessent OCC can be inserted in a design either at RTL or gate level. When inserted at RTL level or before EDT IP, the Tessent OCC shift registers can either be merged with design scan cells or stitched up into dedicated Tessent OCC scan chains. When you stitch them into dedicated chains, these can be either compressed or uncompressed.

Static Clock Control

By default, the Tessent OCC generated by Tessent Shell provides programmability only through the clock control shift register suitable for ATPG. To use the OCC on a LBIST design, you should add static clock control. Static clock control, which refers to clock sequence not decided by ATPG, can be one of the following options:

- **Internal —** The Tessent OCC is statically programmable using an internal TDR for both LBIST and ATPG modes. When using this option, the LBIST test can use only one NCP at a time. When multiple NCPs are to be used, it needs to be done in multiple pattern sets.
- **External —** The Tessent OCC is statically programmable through OCC module ports for the LBIST mode. This enables use of multiple NCPs for LBIST test in a single pattern set. An NCP index decoder is synthesized to provide the clock sequence for the different NCPs based on the ncp_index output from the LBIST controller. The Tessent OCC external clock control module port is available only for the LBIST mode and unavailable for ATPG.
- **Both —** This combines both the internal and external options described above. ATPG can use the TDR for static clock control. LBIST can use either the TDR or the OCC module ports.

Capture Trigger

To use the Tessent OCC for TK/LBIST operation, you should set the capture trigger to capture enable. In this case, scan enable is replaced by the LBIST capture enable signal as the trigger. To enable either fast capture or slow capture to be used during LBIST, the slow clock signal is

connected to the LBIST controller's output shift capture clock, which pulses on all capture cycles.

The capture enable signal should to be tied to constant-0 or connected to inverted scan enable during OCC insertion.

Connection to OCC External Clock Ports

By default, when configuring the Tessent OCC with static_clock_control as either "external" or "both', the tool ties the OCC external clock control module ports to constant-0. These ports are not described in the OCC ICL and are not referenced in the OCC setup iProc. In an LBIST application with multiple NCPs, these OCC pins would be eventually connected to the NCP index decoder.

Scan Insertion

During Tessent OCC insertion, the clock control shift register IO of the Tessent OCCs are left unconnected. Integrating these Tessent OCC shift register sub-chains into the design is performed outside of the Dft Specification. Scan insertion can be performed using third-party tools.

You can insert the Tessent OCC into either a non-scan design or a scan design.

Non-Scan Design Scan Insertion

When the Tessent OCC is inserted in a non-scan design, the OCC shift registers should be declared as sub chains to the scan insertion tool. The scan insertion tool merges these sub-chains with other scan cells taking care of scan chain length balancing.

Scan Design Insertion

When the Tessent OCC is inserted in a scan design, the Tessent OCC shift registers should be connected into one or more dedicated scan chains by the user, based on the target scan chain length. The Tessent OCC scan enable pin is connected to the design scan enable during scan insertion. The length of the sub chains should be the length of the Tessent OCC shift register. The OCC sub chains are always made part of scan chains considering the EDT mode in contrast to LBIST mode where the OCC shift register is bypassed by a single flop. The sub chains are internally bypassed within the OCC during LBIST.

OCC EDT/LBIST IP Creation

The method you use for Tessent OCC EDT/LBIST creation depends on which flow you are using: pre-synthesis or gate level.

Pre-Synthesis Flow

In pre-synthesis flow, the Tessent OCC is not present in the input skeleton design and the ICL/TCD files for the OCC cannot be read during IP creation. You should instruct the tool to generate LBIST controller compatible with Tessent OCCs. When using this flow, you are responsible for making all connections between the LBIST controller, EDT blocks, NCP index decoder and OCCs. You do this by using the "set_lbist_controller_options -tessent_occ on" command and options.

Gate-Level Flow

During EDT IP creation, you should input into Tessent Shell the Tessent OCC inserted design with OCC shift registers included in scan chains should be read. The OCC scan chains can be either part of compressed or uncompressed chains. Do not add Tessent OCC uncompressed scan chains during IP creation; only add them during pattern generation. You must configure the Tessent OCC correctly to pass the IP creation DRC checks, specifically the shift clock, scan enable, and capture enable signals of the Tessent OCC are properly connected and operated in the incoming test procedures.

During IP creation, you read in the ICL, PDL and Tessent Core Description (TCD) for the OCC. This is required to properly setup the Tessent OCC during IP creation. The TCD description is bound to a netlist instance by treating it as a core instance, similar to how scan pattern retargeting uses TCD. The tool identifies the Tessent OCCs when the tessent_instrument_type ICL attribute is set to "mentor::occ". This attribute value is considered when generating the ICL signature, so it cannot be added to user OCCs. When Tessent OCCs are present in the design, the LBIST controller is modified to correctly interface with the OCC.

The TK/LBIST compatible Tessent OCCs are required to have the following two features: capture trigger using capture enable and static clock control either external or both when using multiple NCPs. See "Static Clock Control" on page 154 and "Capture Trigger" on page 154.

Note

Do not mix Tessent OCCs and custom OCCs (defined using set_clock_controller_pins command) in the same LBIST controller. The tool performs rule checks to validate this requirement.

During EDT IP creation, the Tessent OCC capture enable pins that are not functionally driven are driven by the inverted OCC scan enable. The Tessent OCC capture enable pins that are functionally driven (that is, by inverted scan enable) are multiplexed between existing functional connection and LBIST capture enable.

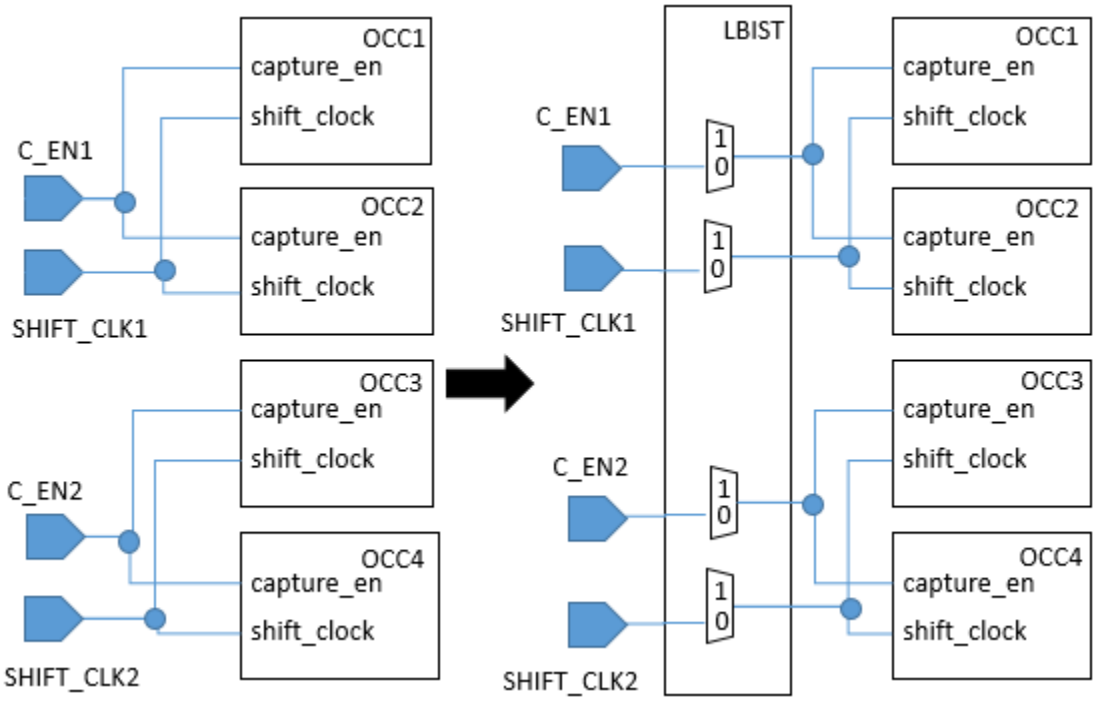
OCC Connections Interception

During hybrid TK/LBIST and OCC insertion with the TSDB flow, when adding edt_clock, shift_capture_clock dft signals as top-level ports, the LogicBIST IP intercepts and multiplexes

the existing connections of the OCCs. The tool attempts to reduce the amount of generated LogicBIST logic used to complete the intercepts.

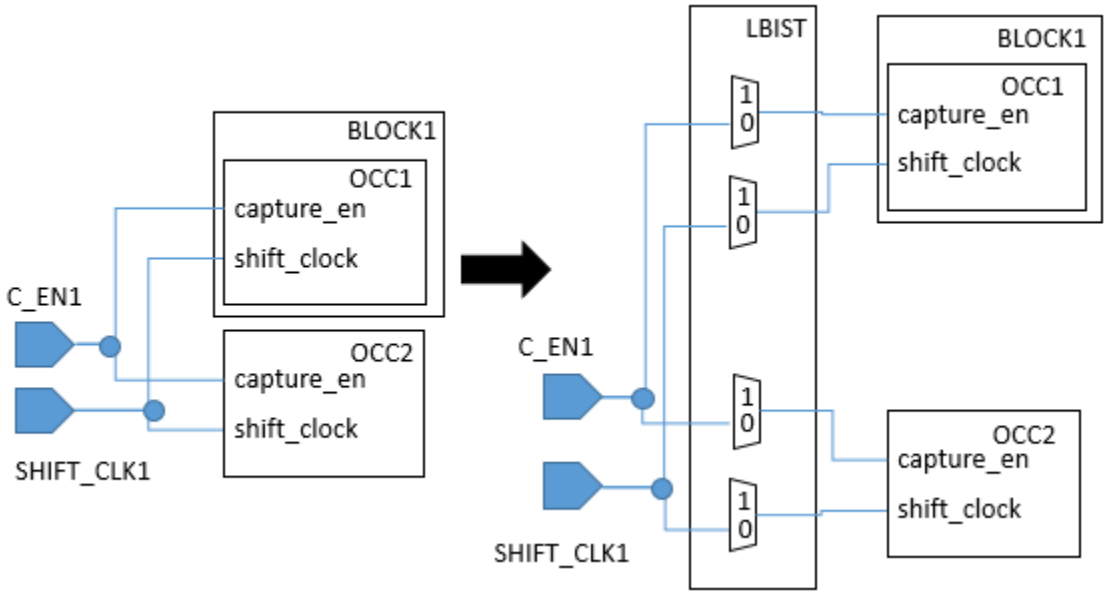
If, for a given signal, multiple OCCs have the same signal source, then one mux is sufficient for intercepting all of them. The tool considers OCCs as having the same sources if their nets fan in from the same net, as observed during LogicBIST validation. As shown in the following figure, instead of inserting a mux for each intercept at the OCCs—eight intercepts—the tool inserts only four muxes. Optimization occurs for the capture enable and shift clock signals.

Figure 9-3. OCC/LogicBIST Connection Intercept With Same Signal Source



The signal sources are considered the same if their nets fan in from the same net, as observed during LogicBIST validation. The following example shows the case when the fanin nets differ for each OCC because one of them resides within a sub-module. The tool treats these OCCs as having different sources. It generates a mux for each intercept at the OCCs, leading to four generated muxes instead of two.

Figure 9-4. OCC/LogicBIST Connection Intercept With Different Signal Sources



Capture Procedures

During IP creation, you specify the total number of NCPs used for LBIST.

This is required to synthesize the NCP index output and NCP activity percentage registers. If the exact number of NCPs is not known during IP creation, an upper bound can be used. During fault simulation, unused NCP indices can be specified as 0%. If the names and activity percentage of the NCPs is specified during IP creation, this is used for the hardware default mode. When not specified, the tool defaults to equal activity for all the NCPs for the hardware default mode.

NCP Index Decoder Synthesis

The NCP index decoder is synthesized is normally done with a third-party synthesis tool.

The number of NCPs generated should be equal to or less than the number specified during hybrid TK/LBIST IP generation.

Fault Simulation with a Tessent OCC

Fault simulation with the Tessent OCCs is similar to a flow that uses custom OCCs. Tessent Shell reads in the complete gate level Verilog netlist with EDT and OCC, ICL/PDL, and TCD files for EDT and OCC instruments.

Tessent OCC core instances can be added during fault simulation.

Tessent Shell automatically adds two internal clocks for each OCC instance:

- A pulse in capture clock at the output of shift register clock gater (cgc_SHIFT_REG_CLK/clkg).
- A programmable capture clock at the output of the programmable clock gater (cgc_CLK_OUT/clkg) at the output of the mux [tessent_persistent_cell_clock_out_mux/Z].

Provide NCPs that refer to these internal clocks. If you have already added internal clocks in the same location as the tool would, then Tessent Shell tool recognizes this and does not add duplicate clocks. The Design Rule Check R18, which validates whether OCC control register is part of scan chains, is bypassed in LBIST mode. Clock control definitions are not generated in the LBIST mode. The output core description TCD file includes the parameters that were specified for the OCC instruments used in this run.

In the LBIST mode dofile, an internal user-PI is added for the LBIST capture enable signal and constrained to 1. For fault simulating chain test patterns, you manually change this constraint to 0.

Considerations When Only Using One NCP

When you are using only one NCP, you must manually create the NCP description because the NcpIndexDecoder, which usually creates the named capture procedure description, cannot be used for a single NCP. The OCC can have either an external or internal static clock control. During fault simulation, specify the name of the NCP as you would when you have more than one NCP.

You can use the create_capture_procedures command to create an NCP description in the tool instead of reading a manually-created description from a file. This user-created NCP should reflect the waveform that you also provide. For external static clock control, the waveform could be constant values provided on the OCC clock_sequence input pins by the netlist. For internal static control, this could be the value loaded into the OCC internal clock_sequence TDR.

For the internal static clock control, load the clock sequence corresponding to the user-created NCP through the ICL network. For Tessent OCCs, you can do this by using the clock_sequence core instance parameter. For the external static clock control, connect the Tessent OCC's clock sequence pins to constant values that generate the required NCP clock waveforms.

Pattern Generation with a Tessent OCC

Pattern generation is performed through pattern specification.

The presence of Tessent OCC is identified through the ICL attribute on the OCC modules and is matched with TCD description for the OCC. When using Tessent OCCs, the pattern

specification processing automatically calls the OCC setup iProc with the parameter values that were used during fault simulation.

Example Tessent OCC TK/LBIST Flow

As part of the Hybrid TK/LBIST flow, you can insert the OCC and NCP Index Decoder during the IP generation step. This example demonstrates an RTL-level flow using the configuration based flow. You can use the same methodology for the gate-level flow.

This example demonstrates a gate-level flow using DFTSpecification for the OCC insertion. The flow uses the TSDB (Tessent Shell Database).

Generating and Inserting the Tessent OCC

The first step to using a Tessent OCC in the TK/LBIST flow is creating the OCC, then subsequently insert the Tessent OCC into your design.

Procedure

1. Invoke Tessent Shell from the shell prompt.

   ```
   % tessent -shell
   ```

2. Set the Tessent Shell context to “dft” and specify a design identifier (rtl2) for the current design.

   ```
   SETUP> set_context dft -rtl -design_identifier rtl2
   ```

3. Point to the TSDB that you used for any previous steps.

   ```
   SETUP> set_tsdb_output_directory ../tsdb_outdir
   ```

4. Read the design and other files from the previous step.

   ```
   SETUP> read_design m8051 -design_id rtl1
   ```

5. Read the cell library. For example:

   ```
   SETUP> read_cell_library atpg.lib
   ```

6. Specify the top-level module of the current design. For example:

   ```
   SETUP> set_current_design physical_block
   ```

7. Set the design level. The physical_block level indicates that the design is a block that is synthesized and laid out as an independent block. For example:

   ```
   SETUP> set_design_level physical_block
   ```

8. Specify any constraints and other specific design settings. For more details on commands related to IP insertion, see “EDT and LogicBIST IP Generation” on page 23.

9. Perform system mode transition.

```
SETUP> check_design_rules
```

10. Read the OCC specific configuration data. For example:

```
ANALYSIS> read_config_data -from_string {
  DftSpecification(m8051, rtl2) {
    reuse_modules_when_possible: on;
    Occ() {
      capture_trigger: capture_en;
      static_clock_control: external;
      Controller(clk) {
        clock_intercept_node: clk;
      }
    }
  }
}
```

11. Read the DftSpecification for the EDT and LogicBIST controllers per the hybrid TK/LBIST insertion flow described in EDT and LogicBIST IP Generation. Additionally, include the NCP details using the NcpIndexDecoder and NCP sub-wrappers under the LogicBIST wrapper. Read in the NCP index decoder specification. For example:

```
SETUP> read_config_data -from_string {
  LogicBist {
    NcpIndexDecoder {
      Ncp(pulse_once) {
        cycle(0) : m8051_gate1_tessent_occ;
      }
      Ncp(pulse_twice) {
        cycle(0) : m8051_gate1_tessent_occ;
        cycle(1) : m8051_gate1_tessent_occ;
      }
    }
  }
}
```

12. Validate and process the content defined in the DftSpecification wrapper.

```
ANALYSIS> process_dft_specification
```

13. Extract ICL to trace the inserted instruments.

```
ANALYSIS> extract_icl
```

Results

The tool creates the Tessent OCC and the NCP index decoder and writes out the relevant output files in the *instruments* sub-directory of the TSDB directory. The output files include TCD files, RTLs, ICL, and PDL for both the OCC and NCP index decoder. Both the OCC and the NCP index decoder are instantiated in the DFT inserted design that is also written out.

Following this process, the sequence of steps is the same as described in "Test Point Analysis and Insertion, Scan Insertion, and X-Bounding" on page 77, "LogicBIST Fault Simulation and

Pattern Creation" on page 89, and "Pattern Generation" on page 101. During fault simulation, the tool reads in the TCD file of the NCP Index decoder.

Tessent OCC Examples

The examples in this section illustrate usage models for using a Tessent OCC in the hybrid TK/LBIST flow.

Clock Domain Analysis for NCP Decoder Generation

This example shows how to use Tessent Shell to perform clock domain analysis and identify NCPs required for LBIST test. It uses a small design that has the following clocks:

- NX1
- NX2
- NX3

The following dofile reports the clock domains and the percentage of faults in each of the domains. Since the netlist is non-scan, the dofile instructs Tessent Shell to treat the netlist as a full-scan design, using the "add scan groups dummy dummy" command. If the design were already scan-inserted, you would instead specify the actual test procedure file and scan chains.

```
set_context pattern -scan
read_verilog nonscan_netlist.v
read_cell_library atpg.lib
set_current_design
add scan groups dummy dummy
add clock 0 NX1
add clock 0 NX2
add clock 0 NX3
set system mode analysis
add_faults -all
report_statistics -clock_domains summary
report_clock_domains -compatible_clocks -details
```

The "Clock Domain Summary" section of report_statistics command's output is shown below:

```
-----------------------------------------------------
Clock Domain Summary    % faults      Test Coverage
                         (total)    (total relevant)
  ------------------   ----------   ----------------
  /NX1                     22.38%              0.00%
  /NX2                     74.70%              0.00%
  /NX3                      0.33%              0.00%
-----------------------------------------------------
```

The output of report_clock_domains command is shown below:

```
//  No. | Clock Name  Domain | Clock Compatibility
// -----+--------------------+--------------------
//    1 | '/NX1' (1)       1 |    .
//    2 | '/NX2' (2)       2 | 437  .
//    3 | '/NX3' (56)      3 |   8 44   .
// --------------------------+--------------------
//                       No. |   1  2  3
//       . = Compatible (non-interacting) clock pair
// <number> = Incompatible (interacting) clock pair
// Compatibility analysis is based on same-edge clock interaction.
```

From the above reports, there are small number of interacting flops (8) between NX1 and NX3. The paths between NX1 and NX3 should be bounded, which can be accomplished by an SDC file that describes all paths between these clock domains as false, as shown below (declared at the mux output):

```
create_clock occ_NX1/clock_out -period 40 -name NX1
create_clock occ_NX2/clock_out -period 40 -name NX2
create_clock occ_NX3/clock_out -period 40 -name NX3
set_false_path -from NX1 -to NX3
set_false_path -from NX3 -to NX1
```

The tool can now treat NX1 and NX3 as compatible clock domains and pulse them together, since all interactions between them are blocked during X-bounding. From the prior clock activity table, we can divide the design into two clock domains: NX2 and NX1_NX3. Consequently, the NX2 and NX1_NX3 domains can be tested for 75% and 25% of the test duration, respectively.

During fault simulation, the output of the report_clock_domains command shows that NX1 and NX3 are indeed compatible after X-bounding. The functional clocks referred earlier are numbered 7-9 in the output below.

```
//No.| Clock Name              Domain | Clock Compatibility
//---+--------------------------------+----------------------------
//  1| '/RST' (1)                   1 | .
//  2| '/refclk' (55)               1 | .   .
//  3| '/shift_clock'(56)           1 | .   .   .
//  4| '/tck' (57)                  1 | .   .   .   .
//  5| '/edt_clock' (64)            1 | .   .   .   .   .
//  6| '/edt_lbist_int_clock' (73)  1 | .   .   .   .   .   .
//  7| '/NX3' (75)                  1 | .   .   .   .   .   .   .
//  8| '/NX2' (76)                  2 | 9   .   .   .   .   .  36   .
//  9| '/NX1' (77)                  1 | .   .   .   .   .   .   . 437  .
// --------------------------------+----------------------------
//                                No. | 1   2   3   4   5   6   7   8  9
//        . = Compatible (non-interacting) clock pair
// <number> = Incompatible (interacting) clock pair
```

A sample DftSpecification for this design is as follows:

```
DftSpecification(m8051,rtl3) {
  LogicBist {
    NcpIndexDecoder {
      Ncp(NX1_NX3_single_pulse) {
        cycle(0): occ_NX1, occ_NX3;
      }
      Ncp(NX2_single_pulse) {
        cycle(0): occ_NX2;
      }
      Ncp(NX1_NX3_double_pulse) {
        cycle(0): occ_NX1, occ_NX3;
        cycle(1): occ_NX1, occ_NX3;
      }
      Ncp(NX2_double_pulse) {
        cycle(0): occ_NX2;
        cycle(1): occ_NX2;
      }
    }
  }
}
```

The NCP following shows the index decoder TCD file created from the preceding DftSpecification:

```
Core(m8051_rtl3_tessent_lbist_ncp_index_decoder) {
  LbistNcpIndexDecoder {
    Interface {
      NcpIndex(ncp_index[3:0]) {
        persistent_pin(0): tessent_persistent_cell_ncp_index_buf_0/y;
        persistent_pin(1): tessent_persistent_cell_ncp_index_buf_1/y;
        persistent_pin(2): tessent_persistent_cell_ncp_index_buf_2/y;
        persistent_pin(3): tessent_persistent_cell_ncp_index_buf_3/y;
      }
      ClockSequence(occ_NX1) {
        persistent_pin :
        tessent_persistent_cell_occ1_clock_sequence_buf_0/y;
      }
      ClockSequence(occ_NX2) {
        persistent_pin:
        tessent_persistent_cell_occ2_clock_sequence_buf_0/y;
      }
      ClockSequence(occ_NX3) {
        persistent_pin :
        tessent_persistent_cell_occ3_clock_sequence_buf_0/y;
      }
    }
    CaptureProcedures {
      Ncp(NX1_NX3_single_pulse) {
        Cycle(0) : occ_NX1, occ_NX3 ;
      }
      Ncp(NX2_single_pulse) {
        Cycle(0) : occ_NX2 ;
      }
      Ncp(NX1_NX3_double_pulse) {
        Cycle(0) : occ_NX1, occ_NX3;
        Cycle(1) : occ_NX1, occ_NX3;
      }
      Ncp(NX2_double_pulse) {
        Cycle(0) : occ_NX2;
        Cycle(1) : occ_NX2;
      }
    }
  }
}
```

Clock Gating When Inserting OCC and Hybrid EDT/LBIST In Different Passes

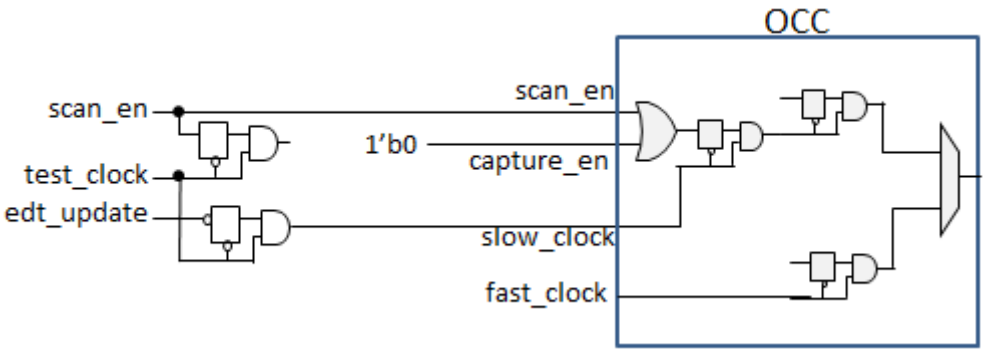
This example illustrates clock-gating with the hybrid TK/LBIST DFT insertion flow with OCC. This usage uses the TSDB flow to insert OCC with the edt_clock and shift_capture_clock DFT signals in the first insertion pass, followed by inserting EDT and LogicBIST in the second insertion pass.

Generate the edt_clock and shift_capture_clock signals by using the add_dft_signals -create_from_other_signals command as shown in the following dofile example.

```
set_context dft -no_rtl -design_identifier dft_signals
set_tsdb_output_directory tsdb_outdir
read_core_descriptions [lsort [glob design/mem/*.lib]]
read_cell_library ../tessent/adk.tcelllib
read_cell_library design/mem/mems.atpglib

read_verilog design/gate/elt1.v
set_current_design elt1
set_design_level physical_block
set_dft_specification_requirements -memory_test off -logic_test on

add_dft_signals scan_en test_clock edt_update \
  -source_node {scan_enable test_clock edt_update}
add_dft_signals edt_clock -create_from_other_signals
add_dft_signals shift_capture_clock -create_from_other_signals
report_dft_signals

add_clocks CLK_F300 -period [expr {1000.0/300.0}]

check_design_rules
set_system_mode analysis
set spec [create_dft_specification -sri_sib_list {occ}]
report_config_data $spec
set_config_value use_rtl_cells on -in_wrapper $spec
read_config_data -in_wrapper $spec -from_string {
  OCC {
    ijtag_host_interface : Sib(occ);
    static_clock_control : external;
    capture_trigger      : capture_en;
    Controller(clk_controller) {
      clock_intercept_node : CLK_F300;
      parent_instance : dft_inst;
    }
  }
}
report_config_data $spec
process_dft_specification
extract_icl
run_synthesis -startup_file \
  ../prerequisites/techlib_adk.tnt/current/synopsys/synopsys_dc.setup
```

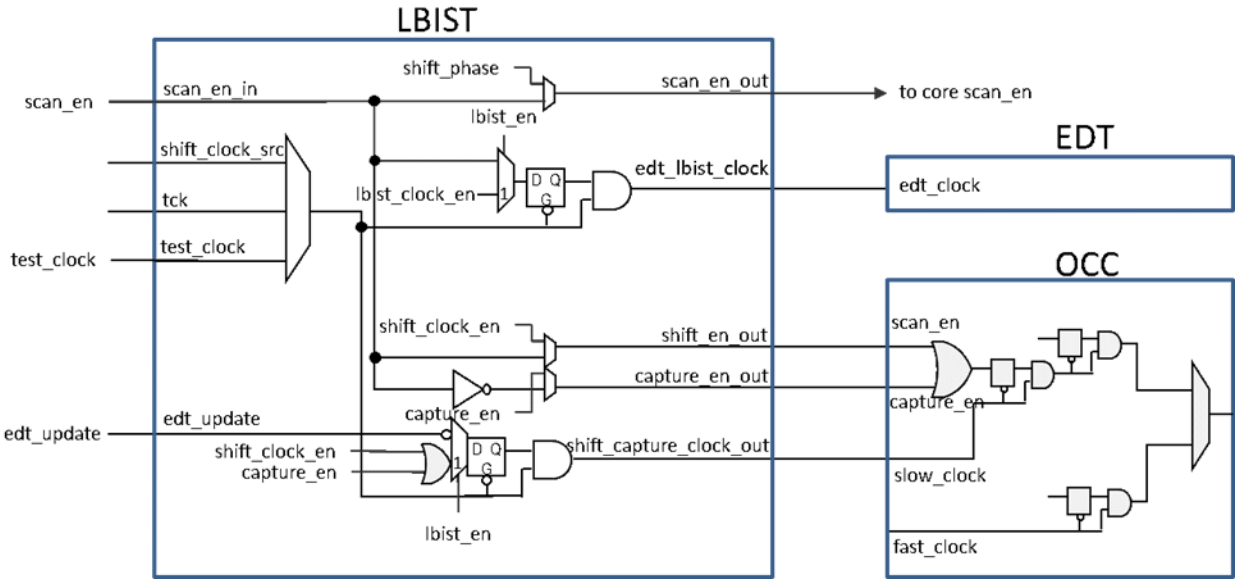
After the first pass, the OCC slow_clock is driven by the shift_capture_clock gater and the edt_clock gater has no fanout.

Figure 9-5. Clock Gating With DFT Signals and OCC in the First Pass



After inserting EDT and LogicBIST in the second hybrid DFT insertion pass, the tool creates the circuit shown below. The edt_clock gater and shift_capture_clock gaters have been removed and their previous connections are now driven by their respective ports on the LogicBIST controller.

Figure 9-6. Clock Gating With EDT and LogicBIST in the Second Pass



Note

This is also the resulting circuit when you first run the TSDB flow to insert the DFT signals and then the dofile flow to insert the LogicBIST controller. In the dofile flow for LogicBIST insertion, the source clock of the DFT signal gaters supply the test_clock to the LogicBIST controller.

Chapter 10
Third-Party OCC for Hybrid TK/LBIST

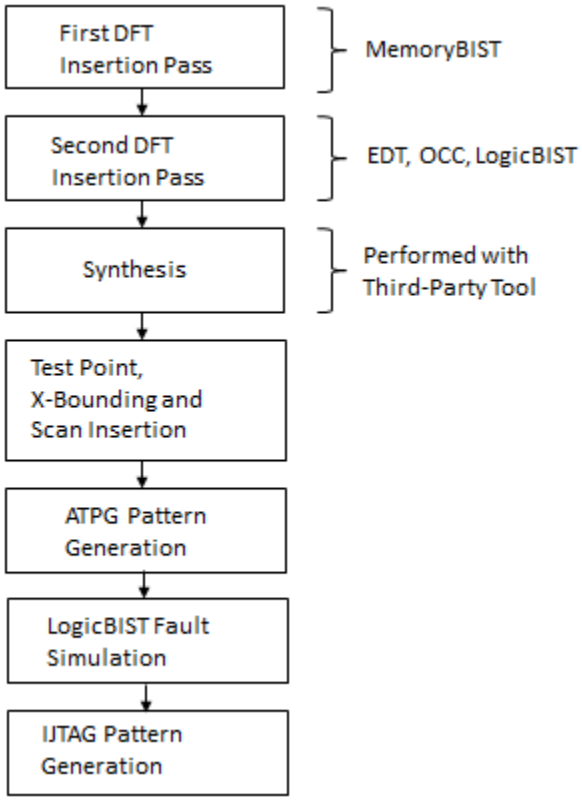
The hybrid TK/LBIST flow supports using third-party OCCs that have already been inserted into the design. To do so, you must first generate Tessent Core Description (TCD) files for each third-party OCC instrument.

Overview of the Third-Party OCC Flow

After generating OCC TCD files and during hybrid IP insertion, read in the third-party OCCs explicitly using the read_core_descriptions command.

As described in "RTL and Scan DFT Insertion Flow With Hybrid TK/LBIST" in the *Tessent Shell User's Manual*, the high-level Tessent Shell flow with hybrid TK/LBIST is:

Figure 10-1. Hybrid TK/LBIST Flow With Tessent Shell



During the second DFT insertion pass, for third-party OCC pins to be properly identified during process_dft_specification, you must load the TCD file and associate it with the OCC module/ instance with the following two commands:

```
read_core_descriptions TCD_file_name ...
add_core_instances -module {OCC_module ...}
```

When using third-party OCCs rather than Tessent OCCs, you must:

- Declare how many NCPs are required. In the DftSpecification, include a LogicBist/ Controller/NcpOptions wrapper with either the count or percentage_of_patterns_per_ncp property; these properties are mutually exclusive. For example:

```
DftSpecification(module_name,id) {
  LogicBist {
    Controller(id) {
      NcpOptions {
        count : int ;                          // default: 1
        percentage_of_patterns_per_ncp : int, ... ;
      }
    }
  }
}
```

- Create the decoding logic. This is the responsibility of the user.
- Hook the decoding logic to the NCP output of the LogicBIST controller. This is the responsibility of the user.
- Ensure that the OCC uses the decoded logic to control the clock sequence when the OCC is active and creates NCPs for various clocking sequences.

After inserting the hybrid IP, proceed with the rest of the flow as normal, including DFT signal declarations with add_dft_signals.

A mix of Tessent OCC and third-party OCCs is not supported.

ThirdPartyOcc TCD File Syntax

Use the ThirdPartyOCC TCD to define third-party OCCs already inserted in the design.

Usage

```
Core(module_name) {
  ThirdPartyOcc {
    Interface {
      ShiftClock(port_name)              {}
      ShiftClockEn(port_name)            {}
      ShiftCaptureClock(port_name)       {}
      ShiftCaptureClockEn(port_name)     {}
      ScanEn(port_name)                  {}
      CaptureEn(port_name)               {}
      LbistEn(port_name)                 {}
      DiagClockEn(port_name)             {}
      ClockOut(port_name)                {}
      InjectTck(port_name)               {}
    }
  }
}
```

Description

Describes the interface for one third-party OCC already inserted in the design.

During hybrid IP insertion, the process_dft_specification command:

- Rule checks that no Tessent OCC wrappers are specified.
- Rule checks that Tessent OCCs were not inserted in a previous insertion pass. You may have Tessent OCCs present, but they cannot be configured for use with LogicBIST.
- Rule checks that the NcpIndexDecoder wrapper is not specified.
- Validates that the ThirdPartyOCC TCD includes all required interface pins and that mutually exclusive pins are not specified. For example, if one OCC uses a shift clock and another uses a shift clock enable, the tool issues an error.
- Traces each intercepted OCC pin—that is, ShiftClock and ScanEn—to find a common source. This helps prevent redundant muxes from being inserted in the LogicBIST controller.
- Disconnects non-intercept OCC pins (except for ClockOut and InjectTck) and re-connects them to the LogicBIST controller source pin.

In Tessent Shell, all TCDs are automatically organized under the Core wrapper, which is unique for a given module name. Depending on the design, the Core wrapper may include descriptions for other core elements in addition to ThirdPartyOCC descriptions. Refer to "Tessent Core Description" in the *Tessent Shell Reference Manual* for more information.

Arguments

- Interface/ScanEn(*port_name*)

  A wrapper that specifies the clock controller scan enable input. The inverted LogicBIST capture enable output from the controller in LogicBIST mode drives this port.

- Interface/CaptureEn(*port_name*)

  A wrapper that specifies the capture enable trigger LogicBIST clock enable input and is typically connected to the LogicBIST capture enable output.

- Interface/ShiftClock(*port_name*)

  A wrapper that specifies the clock controller shift clock input. The LogicBIST shift clock output from the controller in LogicBIST mode drives this pin. This property is mutually exclusive with ShiftClockEn.

- Interface/ShiftClockEn(*port_name*)

  A wrapper that specifies the clock controller shift clock enable input. The LogicBIST shift clock enable output from the controller in LogicBIST mode drives this pin. This property is mutually exclusive with ShiftClock.

- Interface/ShiftCaptureClock(*port_name*)

  A wrapper property that specifies an input on the clock controller that needs to receive a clock that pulses during shift and capture. This behavior differs from the shift clock, which only pulses during the shift mode operation. The LogicBIST shift capture clock output from the controller in LogicBIST mode drives the ShiftCaptureClock pin. If you do not specify this pin, the tool does not create the output port. This property is mutually exclusive with Interface/ShiftCaptureClockEn.

- Interface/ShiftCaptureClockEn(*port_name*)

  A wrapper property that specifies an input on the clock controller that needs to receive a shift and capture clock enable signal. This behavior differs from the shift enable and capture enable, which are high only during the shift or capture mode operation. The LogicBIST shift capture clock enable output from the controller in LogicBIST mode drives the ShiftCaptureClockEn pin of the OCC. If you do not specify this pin, the tool does not create the LBIST output port. This property is mutually exclusive with Interface/ ShiftCaptureClock.

- Interface/LbistEn(*port_name*)

  A wrapper that specifies the LogicBIST enable pins.

- Interface/DiagClockEn(*port_name*)

  A wrapper that specifies the clock controller diagnosis clock enable input. The LogicBIST controller DiagClockEn output drives this pin. If you are using the ShiftClockEn clock controller pin, you need a DiagClockEn connection to perform LogicBIST diagnosis. The DiagClockEn signal is only legal when DftSpecification/LogicBist/Controller/ SingleChainForDiagnosis/present is on.

- Interface/InjectTck(*port_name*)

  A wrapper that specifies the clock controller pin that injects TCK onto the clock. When MemoryBIST is present, the tool connects this port to the tck_select DFT signal, if present.

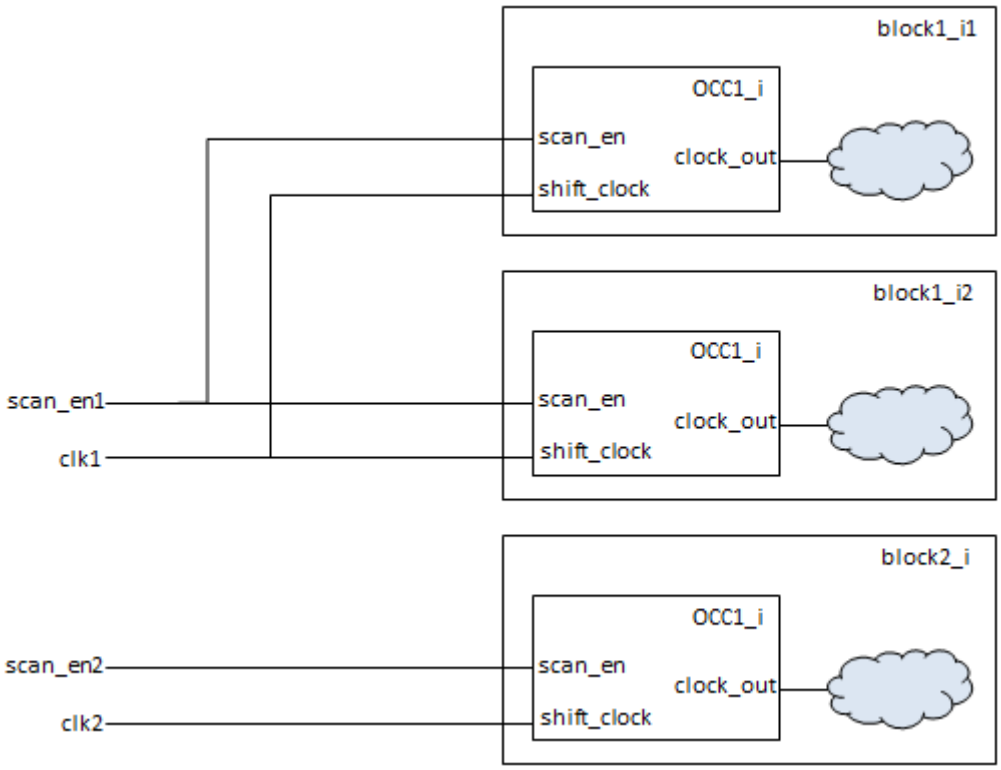
Usage Examples for Third-Party OCC

The default flow uses the LogicBIST controller as the clock generator.

Default Third-Party OCC Flow With Shared Shift Clock Source

The following example shows that there are three instances of one OCC module. The shift clock and scan enable on two of the instances share a common source.

Figure 10-2. Third-Party OCC With Shared Shift Clock Source, Pre-Insertion



The TCD file is:

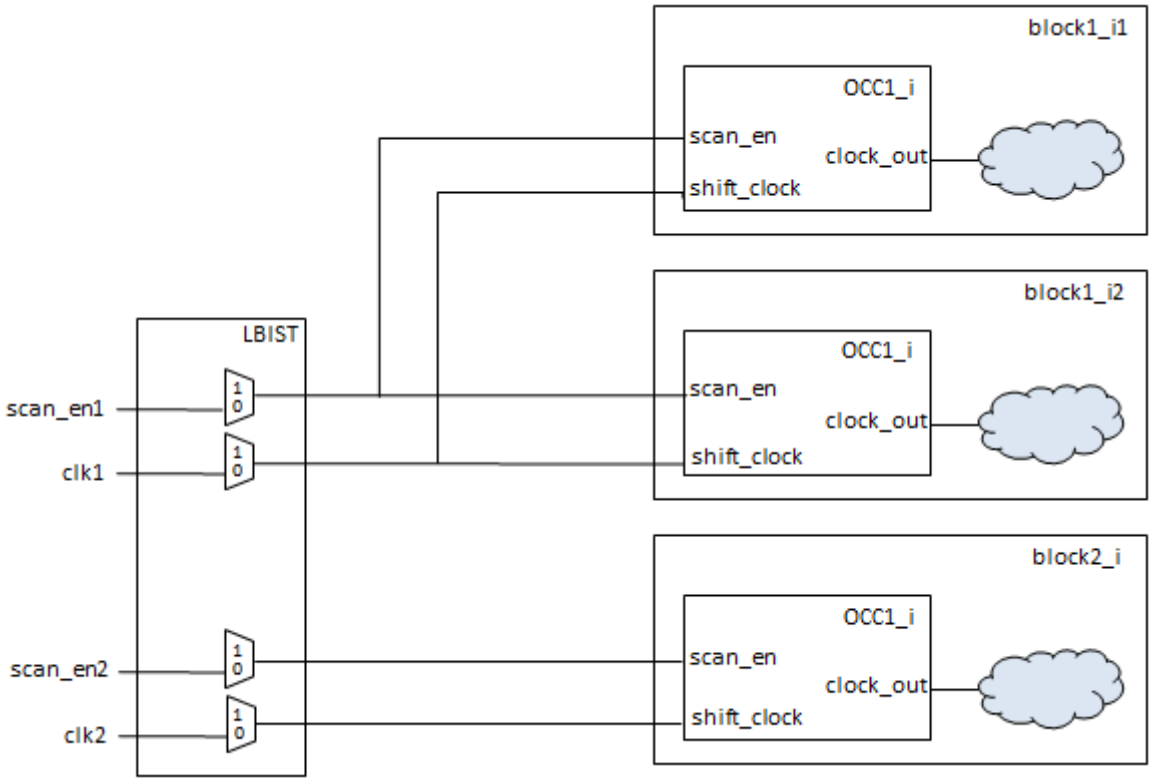
```
Core(OCC1) {
  ThirdPartyOCC {
    Interface {
      ScanEn(scan_en) {}
      ShiftClock(shift_clock) {}
      ClockOut(clock_out) {}
    }
  }
}
```

When running Tessent Shell, the relevant commands are:

```
...
SETUP> read_core_descriptions OCC1.tcd
SETUP> add_core_instances -module OCC1
...
SETUP> set_system_mode analysis
ANALSYIS> create_patterns_specification -sri_sib_list {edt lbist}
...
ANALYSIS> process_dft_specification
```

When Tessent Shell inserts the LogicBIST IP, it sees that block1_i1/OCC1_i and block1_i2/OCC1_i share the same shift clock and scan enable sources, and it only inserts one shift clock and scan enable mux, as shown below.

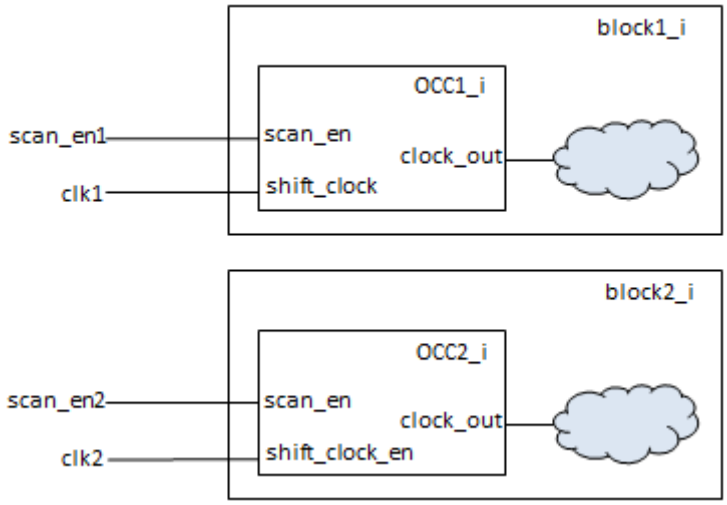
Figure 10-3. Third-Party OCC With Shared Shift Clock Source, Post-Insertion



Default Third-Party OCC Flow With Different Shift Clock Inputs

In the following example, there are two OCC modules and one instance of each. One OCC TCD specifies the shift clock input while the other OCC TCD specifies the shift clock enable input.

Figure 10-4. Third-Party OCC With Different Shift Clock Inputs (Error Condition)



The TCD file is:

```
Core(OCC1) {
  ThirdPartyOCC {
    Interface {
      ScanEn(scan_en) {}
      ShiftClock(shift_clock) {}
      ClockOut(clock_out) {}
    }
  }
}
Core(OCC2) {
  ThirdPartyOCC {
    Interface {
      ScanEn(scan_en) {}
      ShiftClockEn(shift_clock_en) {}
      ClockOut(clock_out) {}
    }
  }
}
```

In this case, the tool issues an error because the TCD files have mutually exclusive ports.

```
...
SETUP> read_core_descriptions OCC1.tcd
SETUP> add_core_instances -module {OCC1 OCC2}
...
ANALYSIS> process_dft_specification
...
//  Error: /DftSpecification(xxx,gate)/LogicBist
//         The 'ShiftClock' and 'ShiftClockEn' ports on third party OCC
modules are mutually exclusive.
//         OCC modules with 'ShiftClock' port:
//           OCC1
//         OCC modules with 'ShiftClockEn' port:
//           OCC2
```

Chapter 11
Observation Scan Technology

This chapter documents Observation Scan Technology (OST) features within the hybrid TK/LBIST flow. The flow supports LogicBIST for in-system testing, which requires high test coverage in a short time span. Using the observation scan features minimizes test times when running LogicBIST in-system by reducing the pattern count needed to achieve a target test coverage.

Note

This chapter uses the terms “Observation Scan Technology” and “observation scan” interchangeably.

Overview

Observation scan includes the following features:

- **The observation scan observe point (OP)**, which the tool inserts during test point insertion. The tool monitors observation scan OPs during every shift cycle and capture cycle. This is different than traditional OPs, which the tool monitors only during capture. The tool treats each shift cycle as a pseudo-random pattern, and adjusts the detection probability and test coverage estimations accordingly.

  For example, given a chain length of 100 with 10,000 patterns, the observation scan OPs observe for one million cycles.

- **Two DFT signals, capture_per_cycle_static_en and capture_per_cycle_dynamic_en.** See “Observation Scan Observe Point Design” on page 179 for details.

- **Test point analysis options, -capture_per_cycle_observe_points and -minimum_shift_length**, which you use together to activate observation scan mode and ensure accurate test coverage.

- **Two BIST DRCs, B5 and B6**, which the tool applies during fault simulation to ensure that the observation scan cells are stitched into their own scan chain and to ensure connectivity between the scan cells and observation scan OPs.

Note

You must perform test point insertion and scan insertion in one session if Observation Scan Technology (OST) testpoints are enabled. If OST test points are not enabled, you can perform these steps in separate sessions.

Licensing

The tool requires an Observation Scan Technology (OST) license to insert observation points in the "dft -test_points -scan" sub-context. The tool checks out the license during analyze_test_points when "set_test_point_analysis_options -capture_per_cycle_observe_points" is set to "on". In "patterns -scan" context, the OST license is required as soon as the tool detects an observation scan during transition to analysis mode. This requirement is in addition to the LogicBist license.

The OST license additionally gives you access to eight child processes. Therefore, one LogicBist license plus one OST license enables you to use one parent plus eight child processes for fault simulation. Every additional OST license gives you access to an additional eight child processes. For example:

- One LogicBist license plus two OST licenses gives you access to one parent and 16 child processes.
- One LogicBist license plus four OST licenses gives you access to one parent and 32 child processes.

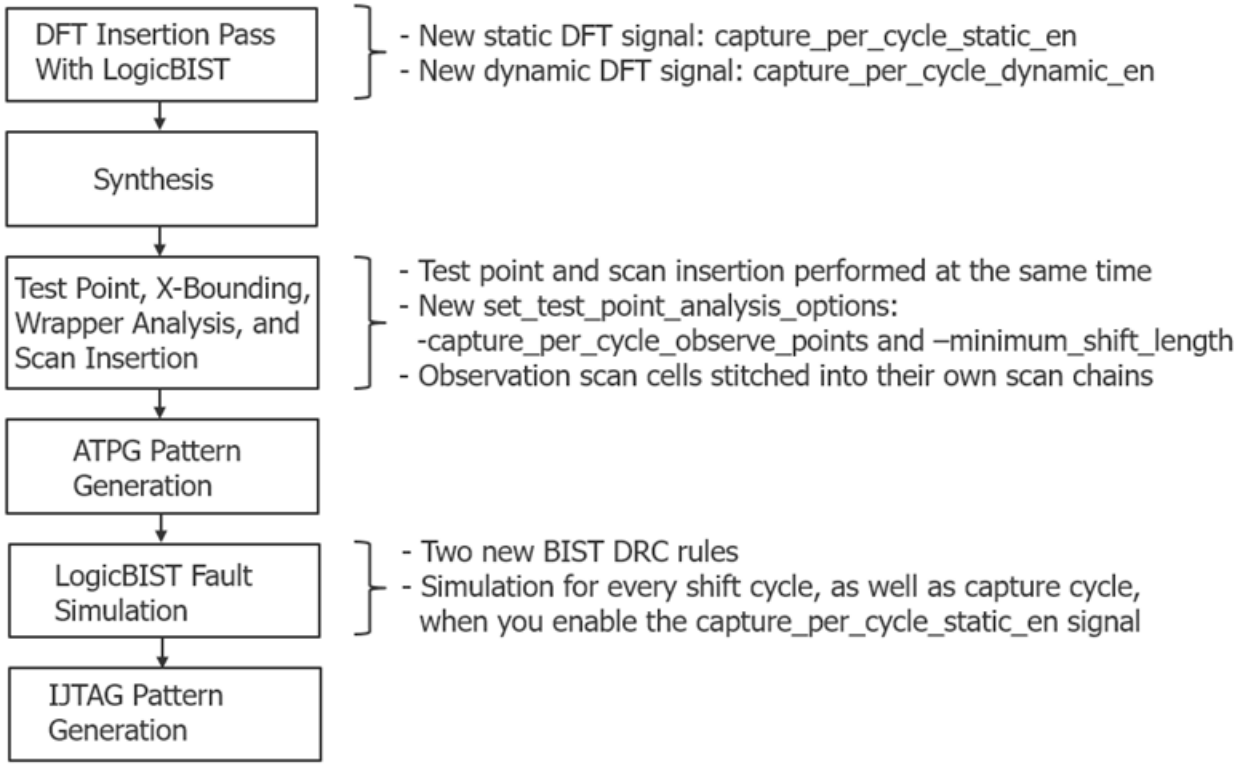
Additional LogicBist or TestKompress licenses do not increase distribution. For observation scan, the only way to increase distribution is by adding more OST licenses.

For additional information about distributed computing in the Tessent environment, see "Multiprocessing for ATPG and Simulation" in the *Tessent Scan and ATPG User's Manual*.

High-Level Flow

The flow outlined in this chapter follows the basic Tessent Shell RTL and scan DFT insertion flow with hybrid TK/LBIST as described in the *Tessent Shell User's Manual*. Figure 11-1 notes the differing aspects of the flow when you are using observation scan; otherwise, the flow remains the same.

Figure 11-1. High-Level DFT Insertion Flow with Observation Scan



For dofile flow considerations, see “Observation Scan Technology Dofile Flow” on page 303.

Limitations

- The tool does not support performing diagnosis with observation scan patterns. To bypass this limitation, rerun fault simulation with the capture_per_cycle_static_en signal disabled. The results then support diagnosis as well.

  The simulation of shift cycle faults with observation scan only supports static fault models. You should turn off observation scan for transition fault types.

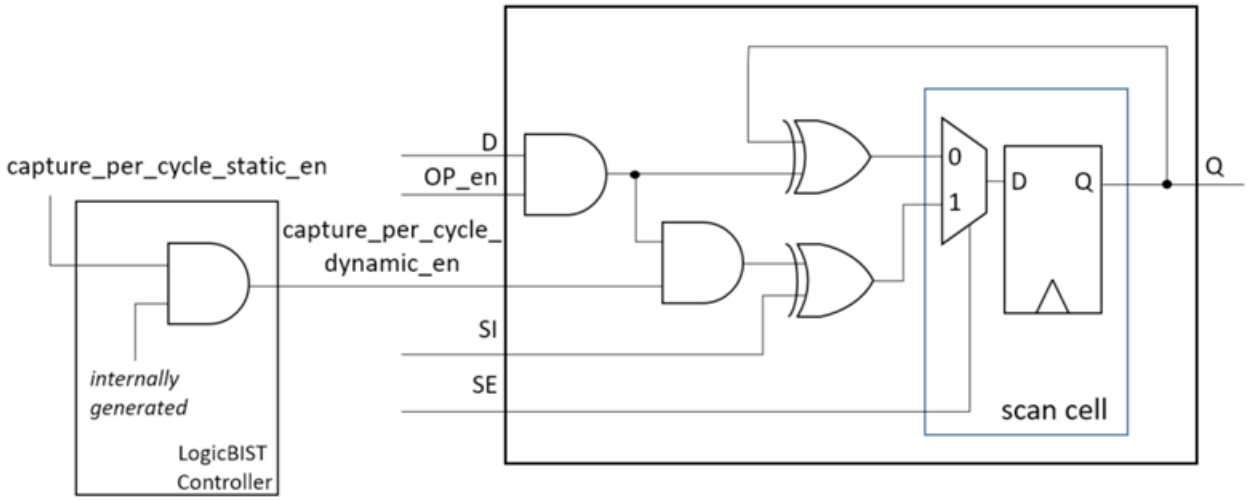
- The tool inserts regular OPs instead of observation scan OPs in the fanout cone of shadow cells. Regular OPs do not capture the initial unknown values of the shadow cells because they do not capture on every cycle. This prevents potential simulation mismatches with parallel patterns.

Observation Scan Observe Point Design

Each observation scan OP consists of several discrete components. The tool implements each OP as a single Verilog module and then instantiates in the netlist at the observe location. This simplifies the task of finding the hardware for a specific test point within the generated netlist.

Figure 11-2 shows the observation scan OP design.

Figure 11-2. Observation Scan Observe Point Design



Observation scan has two mode control signals, OP_en and capture_per_cycle_static_en:

- OP_en — Observe point enable signal that is similar to regular OPs for controlling the basic observation functionality of an observe test point.
- capture_per_cycle_static_en — The enable signal for turning observation scan on and off. When capture_per_cycle_static_en is turned off, observation scan test points activate during capture; that is, they behave as regular observe points. When capture_per_cycle_static_en is turned on, the test points activate during both capture and shift. The capture_per_cycle_static_en signal is ANDed with an internal FSM-generated signal to create a dynamic capture_per_cycle_dynamic_en signal that is connected to the observation points. The tool connects these signals during scan insertion.

  The capture_per_cycle_dynamic_en signal ensures that the observation scan cells are inactive during warm-up patterns and the first regular LogicBIST pattern shift-in. You cannot explicitly specify the signal. However, it is visible through commands that have an access to DFT signals, such as get_dft_signal and report_dft_signals.

Table 11-1 lists the modes of operation with observation scan enabled (OP_en=1 and capture_per_cycle_static_en=1). Refer to Figure 11-2 to see the signal configurations displayed in the table header. The “d”, “s”, and “q” values in the table body represent the current values of the D, SI, and Q pins, respectively.

Table 11-1. Modes of Operation for Observation Scan Cells

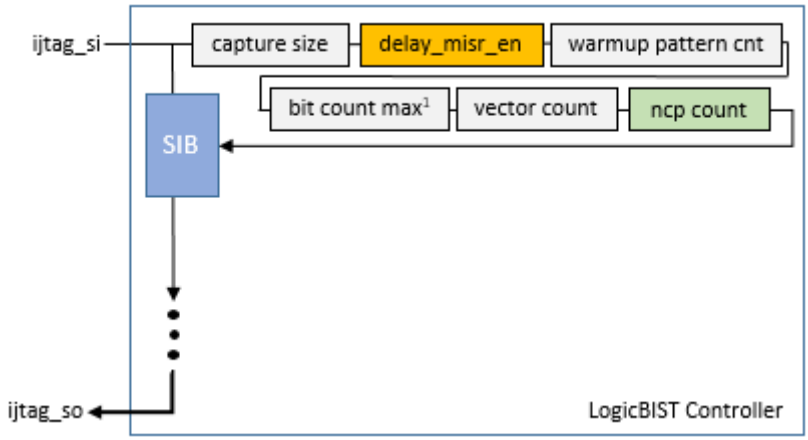
Modes	SE	capture_per_cycle_dynamic_en	D	SI	Q
Shift	1	0	d	s	s
Shift + Capture	1	1	d	s	d ⊕ s
Functional Mode	0	0	d	s	d
Capture	0	1	d	s	d ⊕ q

IJTAG Network in the LogicBIST Controller

When you enable observation scan, the tool inserts an additional TDR, named delay_misr_en, on the LogicBIST controller's IJTAG network. This TDR supports LogicBIST simulations with begin patterns greater than 0.

During observation scan, when you run LogicBIST simulation with patterns greater than 0, the tool applies an extra starting pattern with capture_per_cycle_static_en held low. This enables the unknown (X) values to be flushed from the observation scan chains. The extra pattern means that the tool must delay accumulating the scan chain data in the MISRs by one pattern. The delay_misr_en TDR enables this delay.

Figure 11-3. IJTAG Network in the LogicBIST Controller



DFT Insertion

The DFT insertion process for LogicBIST includes specifying a new static signal, capture_per_cycle_static_en. This DFT signal and the existing observe_test_point_en DFT signal control the observation scan test points.

Specify the capture_per_cycle_static_en signal with the add_dft_signals command during the DFT insertion pass in which you are generating the hybrid TK/LBIST IP. For example:

```
# Load the design
# Add DFT signals
.
.
.
# Required DFT signals for hybrid TK/LBIST
add_dft_signals control_test_point_en observe_test_point_en x_bounding_en

# Required DFT signal for observation scan
add_dft_signals capture_per_cycle_static_en

add_dft_signals int_ltest ext_ltest_en

set_dft_specification_requirements -logic_test on
.
.
# Create DFT specification
set spec [create_dft_specification -sri_sib_list {occ edt lbist} ]

read_config_data logic_instruments.dftspec -in_wrapper $spec -replace

process_dft_specification
.
.
```

For details about this process, refer to "Second DFT Insertion Pass: Inserting Top-Level EDT and OCC" in the *Tessent Shell User's Manual*.

The capture_per_cycle_static_en signal is a static DFT signal with defaults similar to other static DFT signals. You can specify it with either a source node or create it with a TDR:

```
add_dft_signals capture_per_cycle_static_en
     { -source_node pin_port_spec [-make_ijtag_port] | -create_with_tdr }
```

Within the DFT specification, the default for Controller/capture_per_cycle_en is "auto." This resolves to "on" when a DFT signal called capture_per_cycle_static_en is present, and it resolves to "off" otherwise.

```
DftSpecification(module_name,id) {
  LogicBist {
    Controller(id) {
      capture_per_cycle_en   : on | off | auto ;  // default: auto
      Connections {
        capture_per_cycle_en :
          OptionalDftSignal(capture_per_cycle_static_en) ;
      }
    }
  }
}
```

You can also use the explicit “on” or “off” values to enable or disable capture per cycle. Tessent Shell issues an error message when this property is “off” and the capture_per_cycle_static_en DFT signal exists.

The Connections/capture_per_cycle_en property connects the signal to the input capture_per_cycle_static_en pin in the LogicBIST controller. Scan insertion connects the corresponding LogicBIST controller output pin to the observation scan cells.

Timing Constraints

Tessent Shell generates timing constraints for the hybrid IP during IP generation in Synopsys Design Constraints (SDC) format. The SDC file contains timing constraints and exceptions for all modes of operation of the IP.

The *Tessent Shell User’s Manual* describes the timing constraints and exceptions for logic test instruments in “LOGICTEST Instruments.” For observation scan, the following applies:

- The capture_per_cycle_dynamic_en signal from the LogicBIST controller is a multi-cycle path exception in the following procs, similar to prpg_en and misr_en:
  - tessent_set_ltest_modal_lbist_shift
  - tessent_set_ltest_non_modal
- The capture_per_cycle_static_en TDR is a false path exception in the following procs, similar to x_bounding_en implemented inside the LogicBIST controller (valid for the dofile flow only):
  - tessent_set_ltest_modal_controller_chain
  - tessent_set_ltest_non_modal

Test Point and Scan Insertion

Observation scan includes functionality that enables you to activate the “dft -test_points” and “dft -scan” sub-contexts at the same time. You can use all of the commands from either sub-context, which means you only need to specify the insert_test_logic command once to insert both test points and scan chains. For any actions that differ between the two sub-contexts, such as S-rule handling, the merged scenario uses the requirements for scan insertion, which are usually stricter than those for test point insertion.

The merged dft -test_points -scan context supports wrapper analysis, X-bounding, test point analysis and insertion, scan chain analysis and insertion, and so on. Ensure that you specify the following commands in the following order:

1. Test point identification and analysis: analyze_test_points
2. X-bounding: analyze_xbounding

3. Wrapper analysis (as needed): analyze_wrapper_cells
4. Scan insertion: analyze_scan_chains

Before scan insertion, the tool automatically stitches the observation scan cells into their own scan chains.

Note

Using a third-party tool for test point insertion is limited to inserting test points and stitching observation scan observe points into dedicated scan chains. For more information, see "How to Create a Test Points and OST Scan Insertion Script for DC or Genus" in the *Tessent Scan and ATPG User's Manual* and the "set_insert_test_logic_options -generate_third_party_script" command description in the *Tessent Shell Reference Manual*.

Dofile Example

The following dofile example shows a simple test point insertion and scan insertion session:

- Line 1: Set the context to dft -test_points -scan for both test point insertion and scan insertion.
- Lines 11-12: Use the "set_test_point_analysis_options -capture_per_cycle_observe_points" command to activate observation scan mode.

  In addition, for accurate test coverage estimation, Tessent Shell requires the shift length of the observation scan chains during test point analysis. Specify the "set_test_point_analysis_options -minimum_shift_length" option, setting the shift length to the anticipated scan chain length of the design. This option is required to initiate the test point insertion algorithm.

  Note

  You cannot generate both regular observe points and observation scan observe points in the same run.

- Line 13 (optional): You can use the set_test_point_insertion_options command to specify the name of the observation scan enable pin rather than use the default name "capture_per_cycle_static_en".
- Line 31: The insert_test_logic command specified once for both test points and scan chains.

```
1  set_context dft -test_points -scan -no_rtl
2
3  set_tsdb_output_directory tsdb_outdir
4
5  read_verilog piccpu_gate.v
6  read_cell_library ../tessent/adk.tcelllib ../data/picdram.atpglib
7  read_cell_library ../libs/mgc_cp.lib
8  read_design piccpu -design_id rtl -no_hdl
9  set_current_design
```

```
10
11 set_test_point_analysis_options -capture_per_cycle_observe_points on
12 set_test_point_analysis_options -minimum_shift_length 50
13 # set_test_point_insertion_options -capture_per_cycle_en obs_scan_en
14 set_test_point_type lbist_test_coverage
15
16 set_system_mode analysis
17
18 set_test_point_analysis -pattern_count_target 10 \
19 -test_coverage_target 99.9 -total_number 10
20
21 analyze_test_points
22
23 analyze_xbounding
24
25 add_scan_mode short_chains -edt [get_instance -of_module *_edt_lbist_c0]
26
27 analyze_scan_chains
28
29 write_test_point_dofile -replace -output_file tpDofile.do
30
31 insert_test_logic
```

LogicBIST Fault Simulation

Observation scan chains capture and accumulate test responses at every capture and every shift cycle, in contrast to standard scan chains. Every clock cycle can be thought of as a pattern. Stimuli provided during capture cycles are called parent patterns, while stimuli provided during shift cycles constitute intermediate patterns.

Intermediate pattern stimuli consist of the parent pattern test responses being shifted out and the subsequent parent pattern load values being shifted in. While the tool shifts the data through the regular scan chains, the observation scan cells capture and accumulate the circuit's responses every clock cycle. This translates to a faster fault coverage gradient, but it also means that the number of fault simulations increases by the number of regular patterns multiplied by the shift length.

Note

The number of simulations increases by a factor of the shift length, which means increased simulation times and memory requirements. Capture occurs for every shift, and hence there is a direct dependency on the shift length. In order to reduce simulation time, you should use distributed processing. Turning off multithreading (set_multiprocess_options –multithreading off) typically yields the best simulation time reduction.

Simulation Options for Observation Scan

When fault-simulating BIST patterns, Tessent Shell stores and writes out the per-cycle snapshots of the observation scan chains into the PatDB. Because the data volume may be significant, this only happens for the first 256 patterns, by default. During simulation, when the

tool reaches the limit, it no longer stores all per-cycle data. This can impact the scan cell monitoring debugging feature when observation scan is enabled; scan cell monitoring stops monitoring observation scans when it reaches the limit of per-cycle snapshots stored in the PatDB.

To change the default maximum number of patterns for which the per-cycle data is stored (and thus can be monitored for debugging purposes), specify the following command in your fault simulation dofile:

```
set_simulation_options -obs_scan_per_cycle_data_limit integer
```

The specified integer is the maximum number of patterns that can be written to the PatDB. You can write out less per-cycle data into the PatDB by specifying the write_tsdb_data -max_per_cycle_pattern switch.

For observation_scan, the tool can write out parallel testbenches that contain a certain number of last shift cycles (using "write_patterns -parameter_list {SIM_POST_SHIFT integer}") if the corresponding shift cycle data have been stored during fault simulation. If the parallel patterns are within the pattern range specified by "set_simulation_options -obs_scan_per_cycle_data_limit," then for every pattern, the tool stores all of the per-cycle data during fault simulation, and you can write out any legal number of last shift cycles for the parallel testbenches after fault simulation.

If the parallel patterns are beyond the specified pattern range for storing all per-cycle data, or there are no patterns containing per-cycle data ("set_simulation_options -obs_scan_per_cycle_data_limit 0"), by default the tool stores only the data for one shift cycle. In this case, if you want to simulate parallel patterns with more shift cycles, use the following command to specify the total number of shift cycles to be stored during the fault simulation:

```
set_simulation_options -obs_scan_last_shift_cycles integer
```

For *integer*, specify up to 10% of the total shift length. The tool issues a warning and re-sets the integer to 10% of the total shift length if you specify a value greater than this threshold.

BIST DRC Rules for Observation Scan

The following BIST DRC rules are specific to observation scan:

- B5 — Validates that the observation scan cells are connected in their own scan chain distinct from other scan cells.
- B6 — Validates the connectivity of the observation scan OPs to the observation scan cells to ensure that the scan cells can capture data correctly for every shift cycle.

For information, see the B5 and B6 rule descriptions in the *Tessent Shell Reference Manual*.

Pattern Mismatch Debugging Based on Scan Cell Monitoring

You can debug serial patterns with observation scan enabled.

Once the data is stored in the PatDB, you can use it to detect the mismatches as soon as an unexpected value reaches the observation scan cell. To enable debugging, set the SimulationOptions/logic_bist_debug property in the PatternsSpecification wrapper to monitor_scan_cells. For details, refer to "Debug Based On Scan Cell Monitoring" on page 115.

In the following example, a stuck-at 0 fault injected at the KEY_SCHEDULE_0/sub_67/U36/Y pin forces a mismatch in pattern 1. The fault effect is observed in the observation scan chain during shift.

The first lines of the simulation transcript for this pattern set look as follows. Without enabling simulation debug, the only failure you would see is the final MISR signature that is scanned out and compared at the end of the pattern.

```
# 100ns: Pattern_set serial_load_initial_settings
# 400ns: Pattern_set serial_load
# 400ns:  Setting up controller xtea_tk_lbist_ip_tessent_lbist_i
# 400ns:     Number of patterns  : 8 (8 + 0 warm-up patterns)
# 400ns:     Pattern Length      : 17
# 400ns:     Shift Clk Select    : 0b00
# 400ns:     Capture Phase Width : 0x7 Shift Clock Cycles
# 400ns:     PRPG Seed           : 0x0a4f
# 400ns:     MISR Seed           : 0x000000
# 27500ns:  Starting controller xtea_tk_lbist_ip_tessent_lbist_i in Normal mode, patterns 0
to 7
# 29900ns:     Checking that the controller xtea_tk_lbist_ip_tessent_lbist_i DONE signal is
NO at the beginning of the test
# 35591ns: Mismatch at pin
KEY_SCHEDULE_0.sub_67.ts_1_osp_465smodp1_i.ts_1_logic_0fsffp1_i.Q, Simulated 1, Expected 0
# 35591ns: Corresponding scan cell for pattern 1 at shift cycle 12: KEY_SCHEDULE_0/sub_67/
ts_1_osp_465smodp1_i/ts_1_logic_0fsffp1_i/Q, Simulated 1, Expected 0
# 35601ns: Mismatch at pin
KEY_SCHEDULE_0.sub_67.ts_1_osp_467smodp1_i.ts_1_logic_0fsffp1_i.Q, Simulated 0, Expected 1
# 35601ns: Corresponding scan cell for pattern 1 at shift cycle 13: KEY_SCHEDULE_0/sub_67/
ts_1_osp_467smodp1_i/ts_1_logic_0fsffp1_i/Q, Simulated 0, Expected 1
...
```

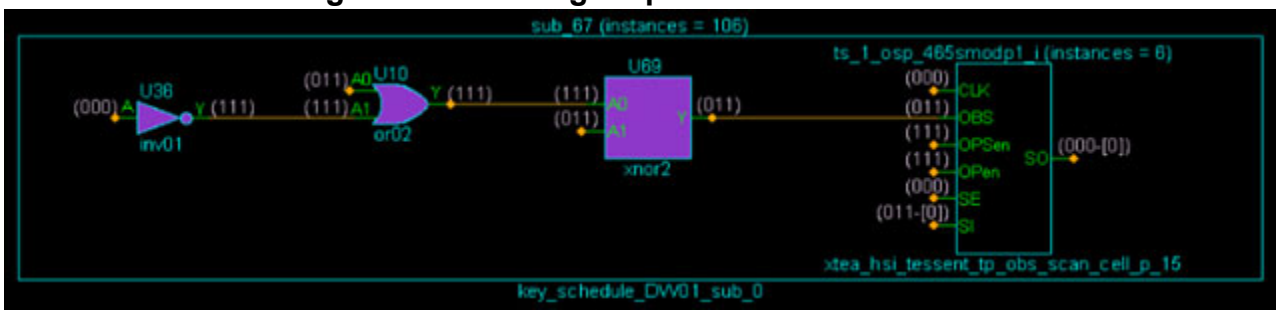
With monitor_scan_cells enabled, you can see that the first mismatch was observed in pattern 1 at shift cycle 12 at the scan cell KEY_SCHEDULE_0/sub_67/ts_1_osp_465smodp1_i/ ts_1_logic_0fsffp1_i/Q.

To find the cause of this mismatch, compare the simulation waveform against LogicBIST fault simulation. First, use the add_schematic_objects command to view the failing flop in the schematic viewer, and then specify the following command to display the relevant data:

```
set_gate_report pattern_index 1 -obs_scan_unload_shift_cycle 12
```

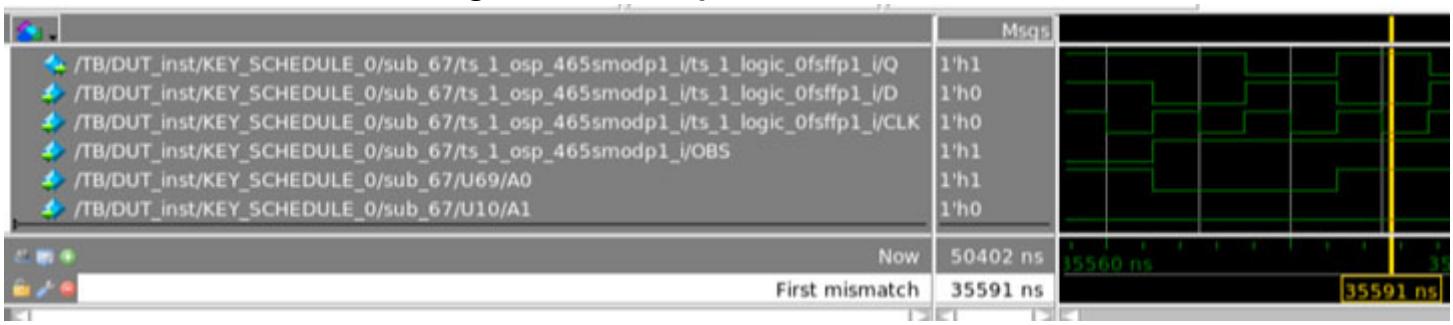
A mismatch reported in shift cycle 0 corresponds to the values at the end of capture phase; display these values with the “set_gate_report pattern_index *n*” command.

Figure 11-4. Failing Flop in Schematic Viewer



You can now compare the fault simulation data with the testbench simulation. The value captured in the observation scan cell is 0 in the fault simulation and 1 in the testbench simulation waveform. Starting at the OBS input of the observation scan cell, you can trace the difference in the simulations to the fault injection site.

Figure 11-5. Compare Simulation Data



In this example, the failure is easy to identify with minimal tracing. In some cases, you may be required to trace backwards through many gates to find the point where fault simulation and Verilog simulation diverged.

Pattern Mismatch Debugging for Parallel Patterns

You can use parallel patterns to validate observation scan simulation on shift cycles, also known as pattern mismatch debugging for parallel patterns.

For observation scan fault simulation you can report the simulation values for a given shift cycle using the “set_gate_report pattern_index -obs_scan_unload_shift_cycle” command. You can then compare the results to the actual Verilog simulation values (or scan cell monitoring results) for debugging serial pattern simulation mismatches. For more information, see “Pattern Mismatch Debugging Based on Scan Cell Monitoring” on page 187.

The tool also enables generating the parallel testbench with last shift cycles for observation scan using the "set_simulation_options -obs_scan_last_shift_cycles integer" and "write_patterns -parallel -parameter_list {SIM_POST_SHIFT integer}" commands. (See "Simulation Options for Observation Scan" on page 185). If there are simulation mismatches on these parallel patterns with shift cycles, you can compare the Verilog simulation results with the corresponding gate report simulation values ("set_gate_report pattern_index -obs_scan_unload_shift_cycle") for tracing the cause of the simulation mismatches.

For observation scan parallel patterns that simulate shift cycles, the load values of observation scan chains for the given last shift cycle are obtained from the corresponding scan cells' content stored during fault simulation. The tool then performs Verilog simulation on the following shift and capture cycles after loading the values.

Because each shift cycle simulation depends on the simulation results of the previous cycle, the parallel testbench might not contain enough cycles for tracing the cause of the simulation mismatches. The tool does not enable generating a parallel testbench with too many shift cycles: only 10% of the total shift length for parallel patterns. For information, see the description of "set_simulation_options -obs_scan_last_shift_cycles."

In this case, either increase the last shift cycles for the parallel patterns (within the maximum legal range) to see if you can trace to the cause to the mismatches. Or use serial patterns with more complete simulation results. See "Parallel Versus Serial Patterns" in the *Tessent Scan and ATPG User's Manual*.

Although the tool does not enable generating the parallel testbench with complete shift cycles, generating parallel patterns with some last shift cycles is still a good idea because parallel patterns simulate more quickly than serial patterns. Simulating parallel patterns with a number of shift cycles provides a way to quickly check if a certain number of the shift cycles are simulated as expected.

Chapter 12
Independent Hybrid TK/LBIST Insertion Flow

The insertion of the LBIST controller and EDT controllers has been decoupled to allow you to insert LBIST-ready EDT separately from the LogicBIST. This chapter describes the features that support this enhanced flow. Hybrid EDT/LBIST controller and hybrid EDT controller references are used interchangeably.

Independent Insertion Flow Overview

The independent insertion flow enables you to insert an LBIST-ready EDT controller independently from the LBIST controller.

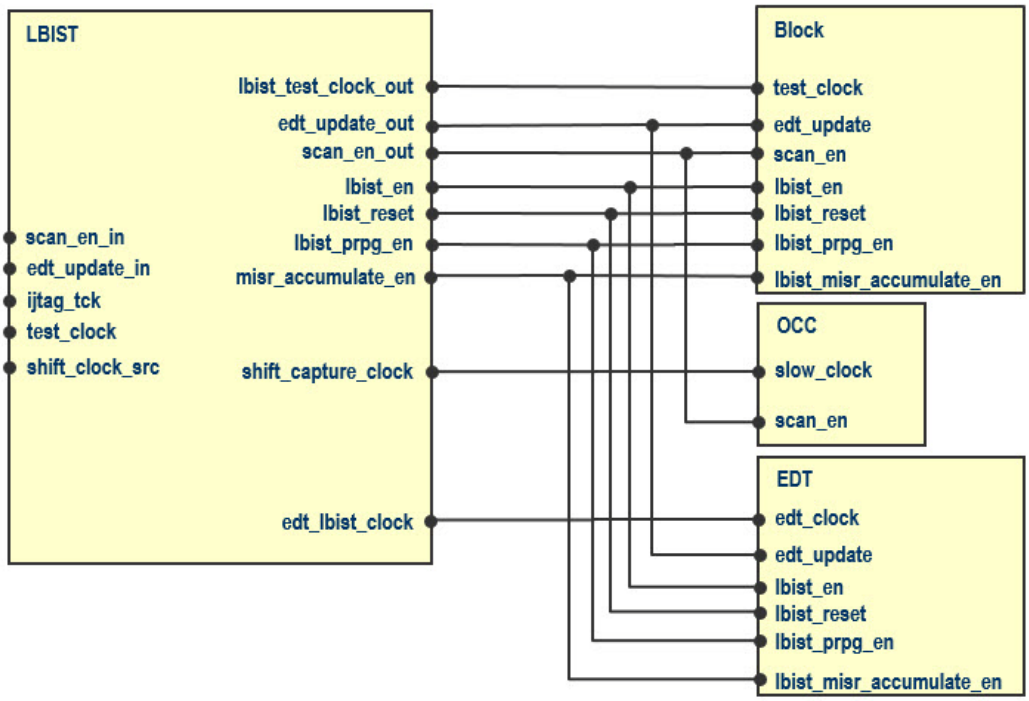
The following enhancements support the independent insertion flow:

- The LBIST-ready EDT controllers can now be added to child design levels making them easier to reuse as sub-blocks in other designs.

- EDT controllers are now directly on the IJTAG network instead of being on a sub-network of the LBIST controller's IJTAG network.
- Simplified single chain mode using the existing IJTAG network chains instead of having a separate IJTAG interface for LBIST.
- Sub-blocks now reuse edt_clock/shift_capture_clock gaters so that OCC and EDT receive the expected clock pulses.
- New DFT signals provide control signals from the LBIST controller to the sub-blocks. These signals enable communication between the two passes when inserting LBIST-ready blocks and the LBIST controller.
- LBIST load/unload waveforms are now fully programmable during pattern retargeting.
- The LBIST controller now allows OCCs with a shift_en capture trigger. The value "auto" for the property DftSpecification/OCC/capture_trigger now resolves to capture_en when you specify an LpctType3 wrapper in the same insertion pass; otherwise, it resolves to shift_en.

Figure 12-1 illustrates the resulting block diagram after independent insertion. The EDT controller has been inserted in Block, which is in a lower design level, and the LBIST controller and another hybrid EDT controller have been inserted in the current physical block.

Figure 12-1. Independent Insertion Flow Block Diagram



Block contains an LBIST-ready EDT controller at a different design level, and the LBIST controller supplies the following new dynamic DFT control signals: *lbist_reset*, *lbist_prpg_en*, and *lbist_misr_accumulate_en*. The LBIST controller port *lbist_test_clock_out* drives the *test_clock* port on Block, and the LBIST controller port *edt_update_out* drives the *edt_update* port on Block.

Figure 12-2 shows the contents of Block. Typically, when the LBIST controller is inserted in the same insertion pass as the EDT controller, the *shift_capture_clock* and *edt_clock* gaters are moved within the LBIST controller. However, within Block, these gaters remain because there is no LBIST controller to move them into. The LBIST controller reuses these gaters to ensure that OCC and EDT get the correct clock waveforms.

Figure 12-2. Block Contents

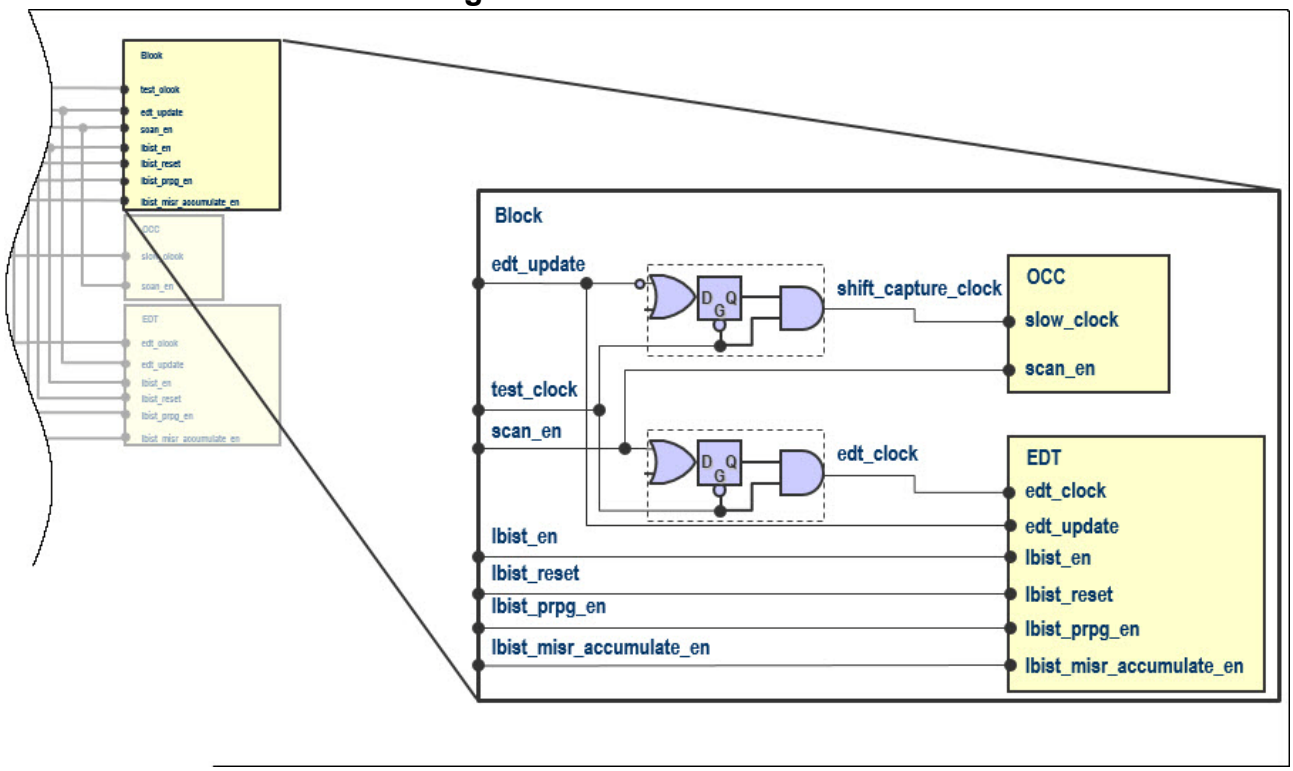
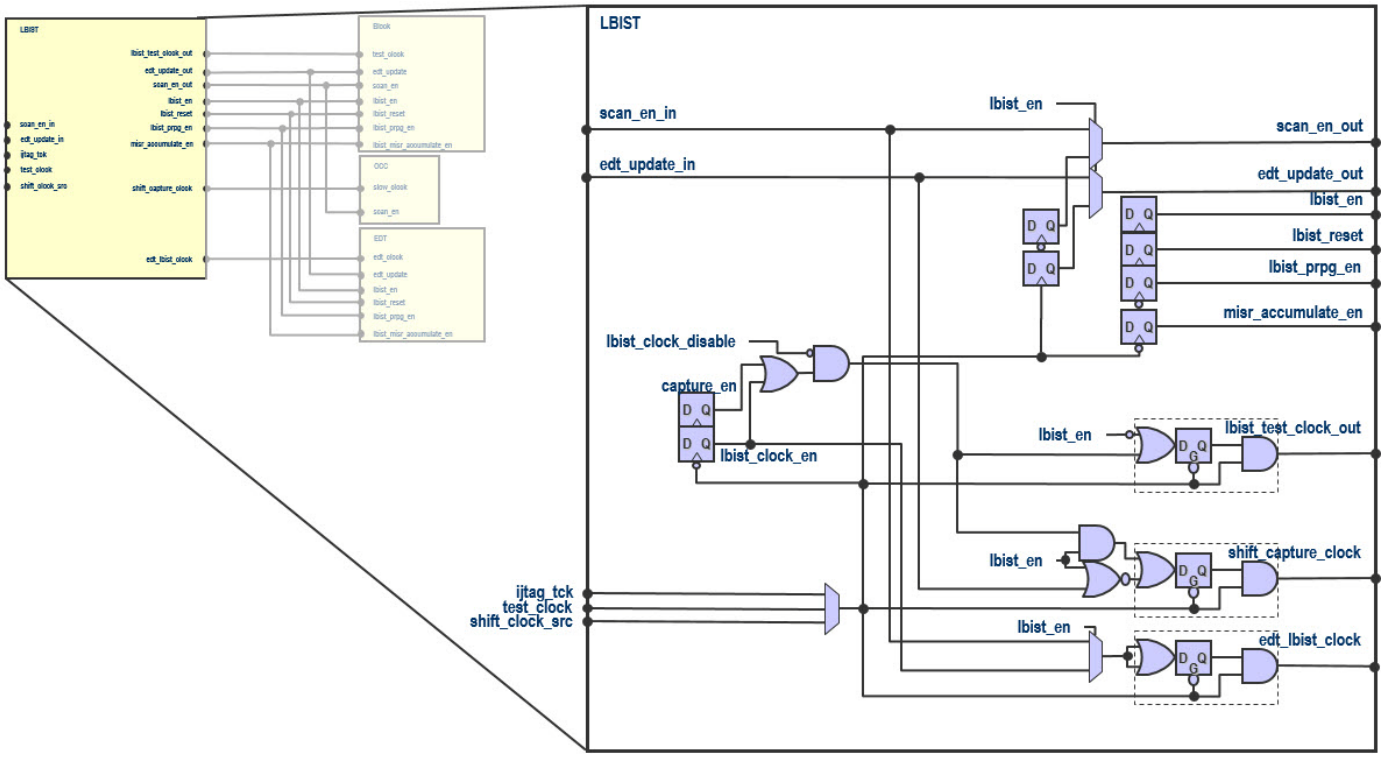


Figure 12-3 shows the contents of the LBIST controller. The *lbist_test_clock_out* port has its own clock gater. During LBIST mode, the LBIST controller pulses the clock during *shift*, *capture*, and when the *edt_lbist_clock* needs a single pulse after capture. The clock gaters in Block use the *test_clock* (from *lbist_test_clock_out*), *edt_update*, and *scan_en* signals. This ensures that all the desired pulses reach the OCC and EDT.

Figure 12-3. LBIST Controller Contents



The single chain mode for the TK/LBIST flow is implemented using the IJTAG network that is created when inserting the EDT and LogicBIST controllers. The IJTAG network of the EDT uses a SIB that allows access to all scan cells associated with the particular EDT for the single chain mode. The IJTAG network for EDT controllers is merged with the child-level IJTAG network for other instruments such as MBIST.

In the independent insertion flow, the single chain mode for LBIST pattern diagnosis is simplified so that the Single Chain Mode Logic module is not needed to concatenate all of the EDT controllers short chains into a single chain. With this flow, each EDT controller contains a SIB that provides access to its short chains through the IJTAG network. The DftSpecification for the EDT single_bypass_chain property determines whether or not the EDT uses the single chain mode. Setting the single_bypass_chain property to on indicates that a single bypass chain is present and generates new single chain mode logic in the EDT. All independently generated EDT controllers should have the single_bypass_chain property enabled in order to use the single chain mode for LBIST pattern diagnosis.

Tessent EDT and LogicBIST IP Generation

As part of the hybrid IP generation step, you generate the shared Tessent EDT and LogicBIST RTL. The tool generates one LogicBIST controller for all EDT and LogicBIST blocks in a core.

You can also configure the low-power scheme to control the switching activity during “shift” to reduce power consumption.

There is no TAP controller at the core level. The tool integrates the access mechanism in the IJTAG network at the core level. This step of the flow creates new core-level pins corresponding to the Segment Insertion Bit (SIB) control signals, tck, and LBIST scan I/O. The core-level Verilog patterns operate these pins directly. These pins connect to the TAP controller at the top level of the design. See "Top-Level ICL Network Integration" on page 125 for more information.

As part of IP generation, the tool writes the following files to the TSDB:

- **ICL file —** Consists of the ICL module description for the LBIST controller, the NCP index decoder, and all EDT and LogicBIST blocks that the controller tests.
- **PDL file —** Contains iProcs at the core level that use the ICL modules.

During IP generation, the generated ICL file describes only the LogicBIST, NCP index decoder, and EDT modules. The extracted ICL file includes the core-level pin names and connectivity found from the core-level design netlist. The tool uses the extracted ICL file during top-level pattern generation. See "ICL Extraction and Pattern Retargeting" on page 131 for more information. You can write Verilog patterns in this step and simulate them to verify the test operation at the core level.

For complete information, see "EDT and LogicBIST IP Generation" on page 23. During integration with the top level, the tool adds new top-level test pins or uses existing top-level test pins controlled internally by the EDT and LogicBIST IP.

Logic Synthesis

You must synthesize all of the EDT and LogicBIST blocks and the common LogicBIST controller. Synthesis is fully automated. In the gate-level flow, you can use the run_synthesis command to synthesize the controllers and the test logic in the TSDB and integrate them into the gate-level design. When the run_synthesis command completes successfully, it creates a concatenated netlist of the design that contains the synthesized test logic and modified design modules and places them in the *dft_inserted_designs* directory of the TSDB.

In the RTL-level flow, you can use the run_synthesis command to synthesize the test logic inserted by the tool, but the netlists are not concatenated.

EDT and LogicBIST IP Generation Overview (Independent Insertion Flow)

EDT and LogicBIST IP generation stores several files in the TSDB. The dofiles are used for LogicBIST fault simulation and pattern creation, while the ICL, PDL, and IP netlist files are used as inputs for pattern generation.

IJTAG Network in EDT/LogicBIST IP (Independent Insertion Flow)

SIBs are a mechanism to provide flexible access to data registers they control. For the Tessent version of a SIB, the data registers are accessible via the IJTAG network when the SIB controlling a data register is set to 1, and the data register is bypassed when the SIB is set to 0.

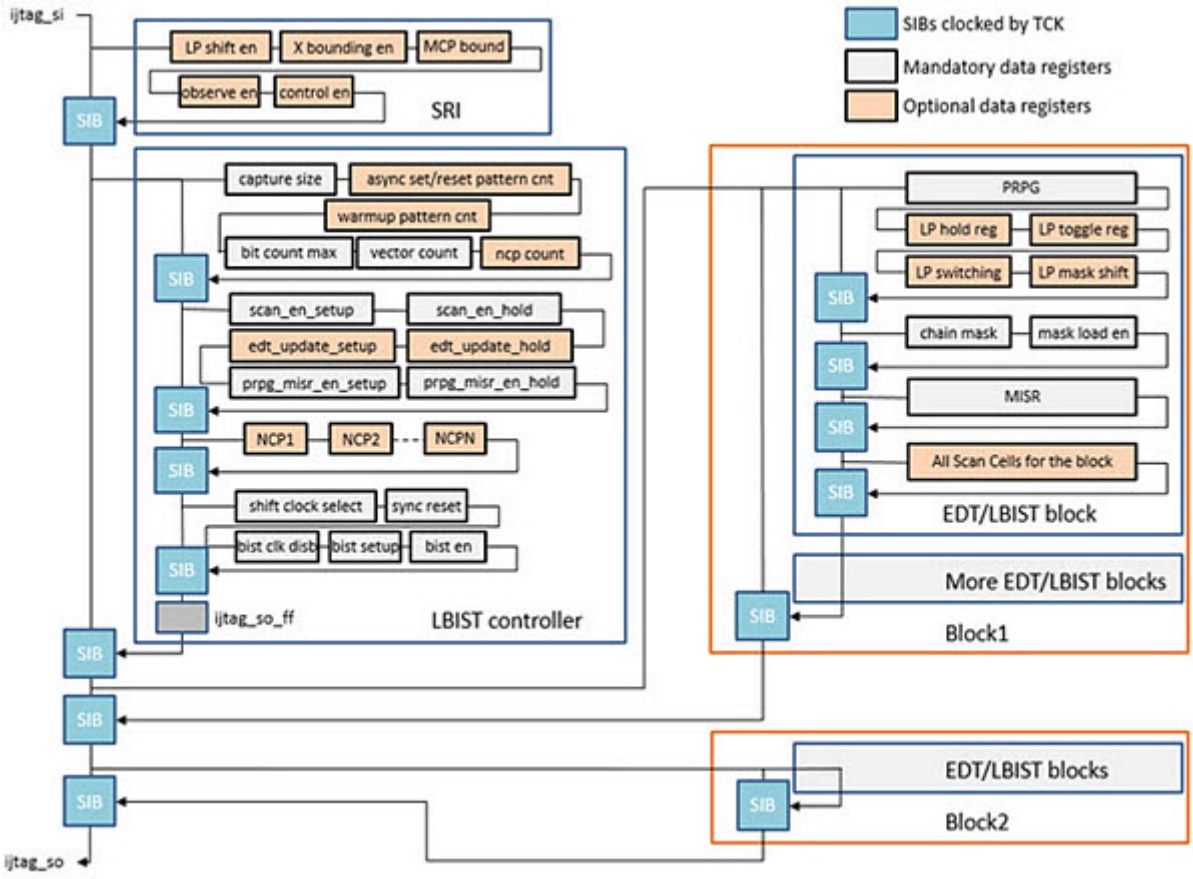
SIBs are clocked by TCK. To provide flexibility in accessing specified registers in the IJTAG network at different times, the SIBs are arranged in a hierarchical tree structure. The SIB enable signals for the child SIBs are controlled by their parents. This is how all the SIBs inside the LogicBIST controller, EDT/LogicBIST controllers are configured.

Each EDT/LogicBIST controller contains three SIBs. They provide access to the PRPG, EDT chain mask register, and MISR, respectively. The EDT SIBs are clocked by tck, and the data registers they control are clocked by edt_clock. The tool adds a lockup cell to avoid clock skew between these two clock domains.

For each specified NCP, an 8-bit register is created and inserted on the ICL network. These registers are loaded at runtime so that the proportion of patterns applied for each NCP is programmable. When an integer percentage is provided during IP creation, the NCP register values are reset to the specified values if ijtag reset is asserted. Otherwise, these registers are reset to equal percentages across all NCPs. For more information, see “Generating the EDT and LogicBIST IP” on page 64.

The following figure shows the IJTAG network in a hybrid EDT/LBIST-inserted design. In this case, the EDT controllers can exist in the lower level block. Each EDT controller adds their scan chains to the IJTAG network using a SIB.

Figure 12-4. SIBs Insertion and Integration of Cores for the Independent Insertion Flow



LBIST-Related Clock Signals for the Independent Insertion Flow

The independent insertion flow requires LBIST-related clock signal routing.

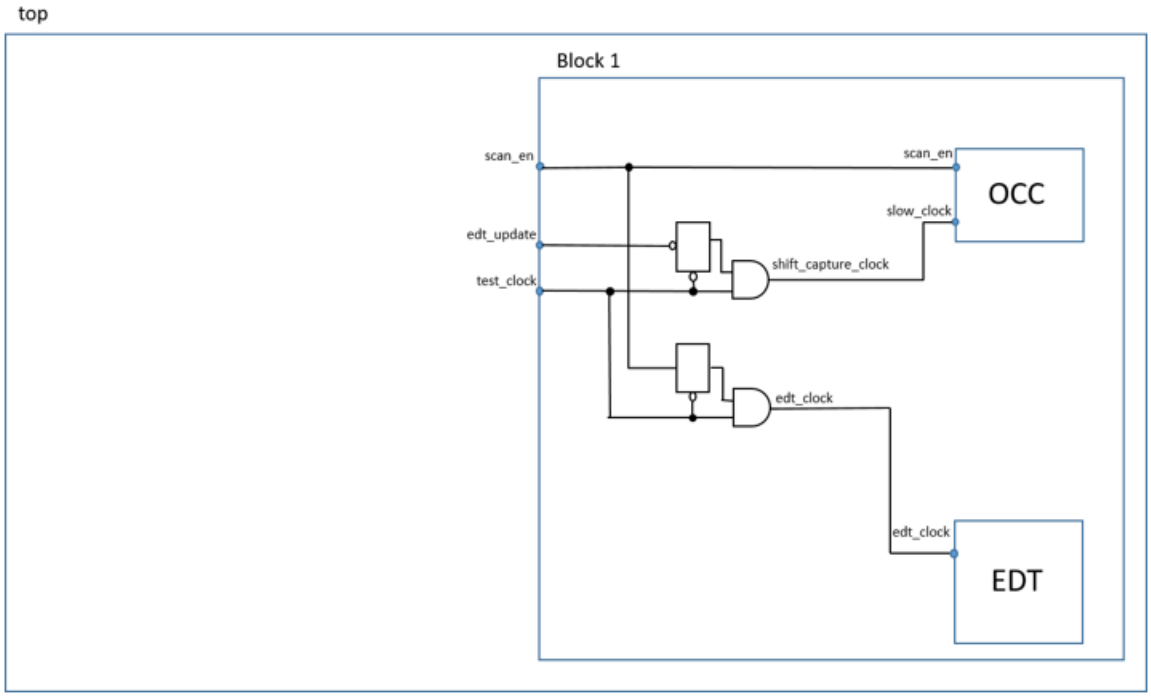
The TK/LBIST clock signal routing consists of two steps.

1. Prepare the LBIST-ready block.
2. Insert the LBIST controller.

LBIST-Ready Block Preparation

Figure 12-5 shows the circuit logic for the shift_capture_clock and edt_clock signals. The clock gaters for these signals are created by the add_dft_signal command with the create_from_other_signals option. OCCs with shift_en trigger are used automatically to reduce the number of required connections.

Figure 12-5. LBIST-Ready Block Before Insertion



LBIST Controller Insertion

Figure 12-6 shows the design after the LBIST controller is inserted. Here the OCC uses shift_en as the capture trigger.

Figure 12-6. LBIST-Ready Block After Insertion Using shift_en

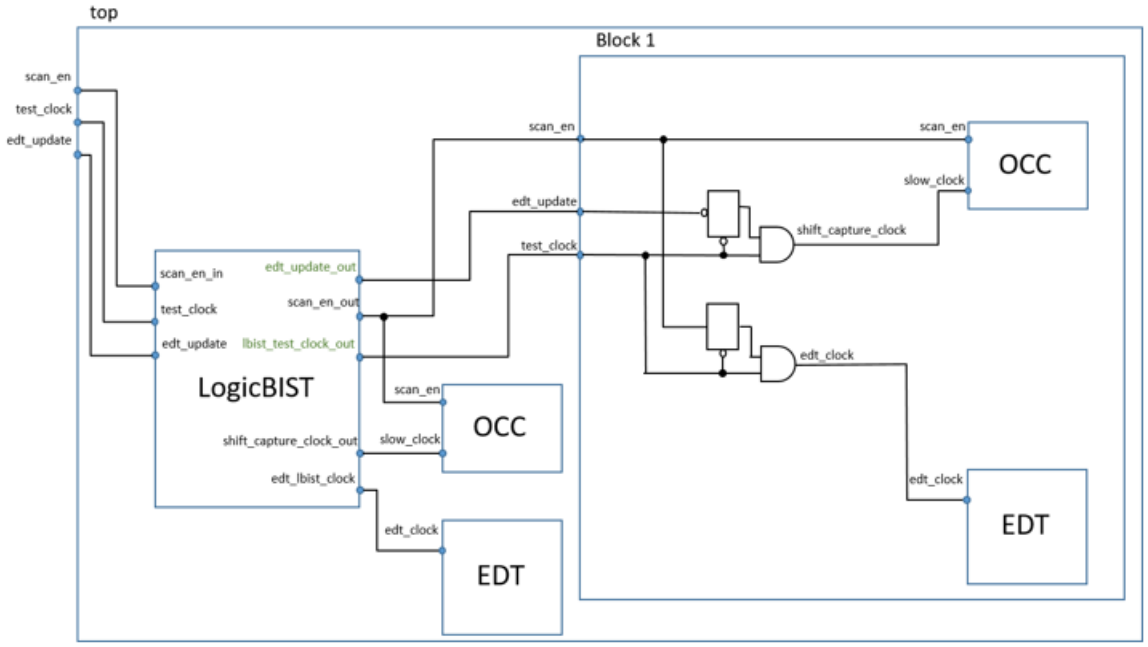
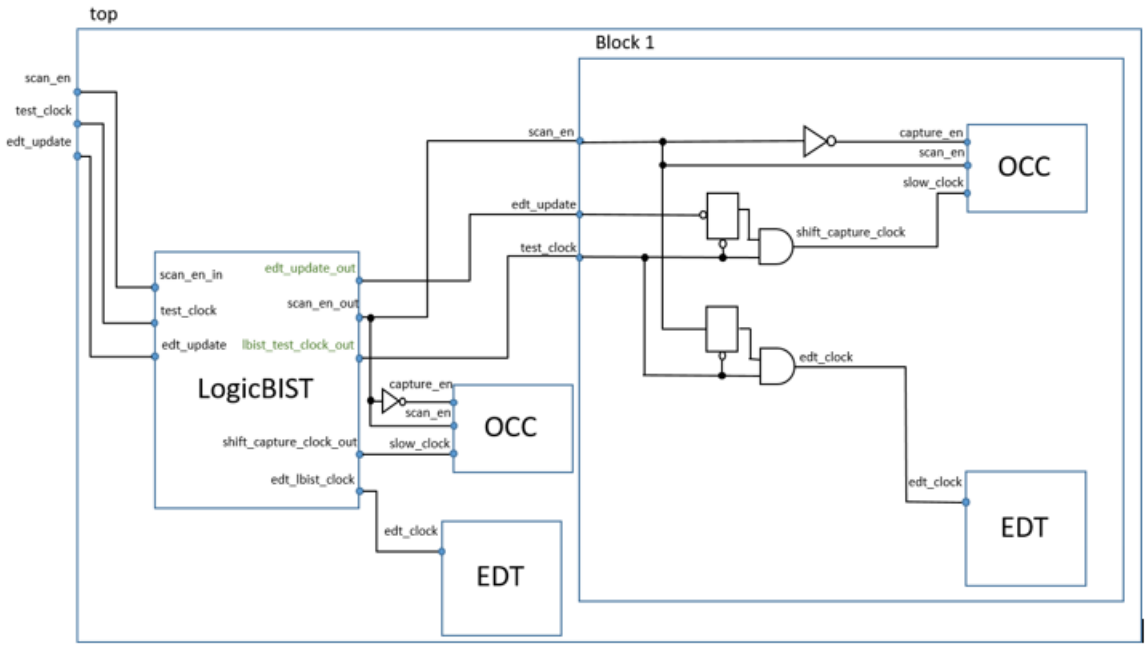


Figure 12-7 shows the design after the LBIST controller is inserted with OCC, where the OCC uses capture_en as the trigger to generate the programmable capture pulses. Use only when you have a third-party OCC that uses capture_en, or if the design uses Low Pin Count Test controllers.

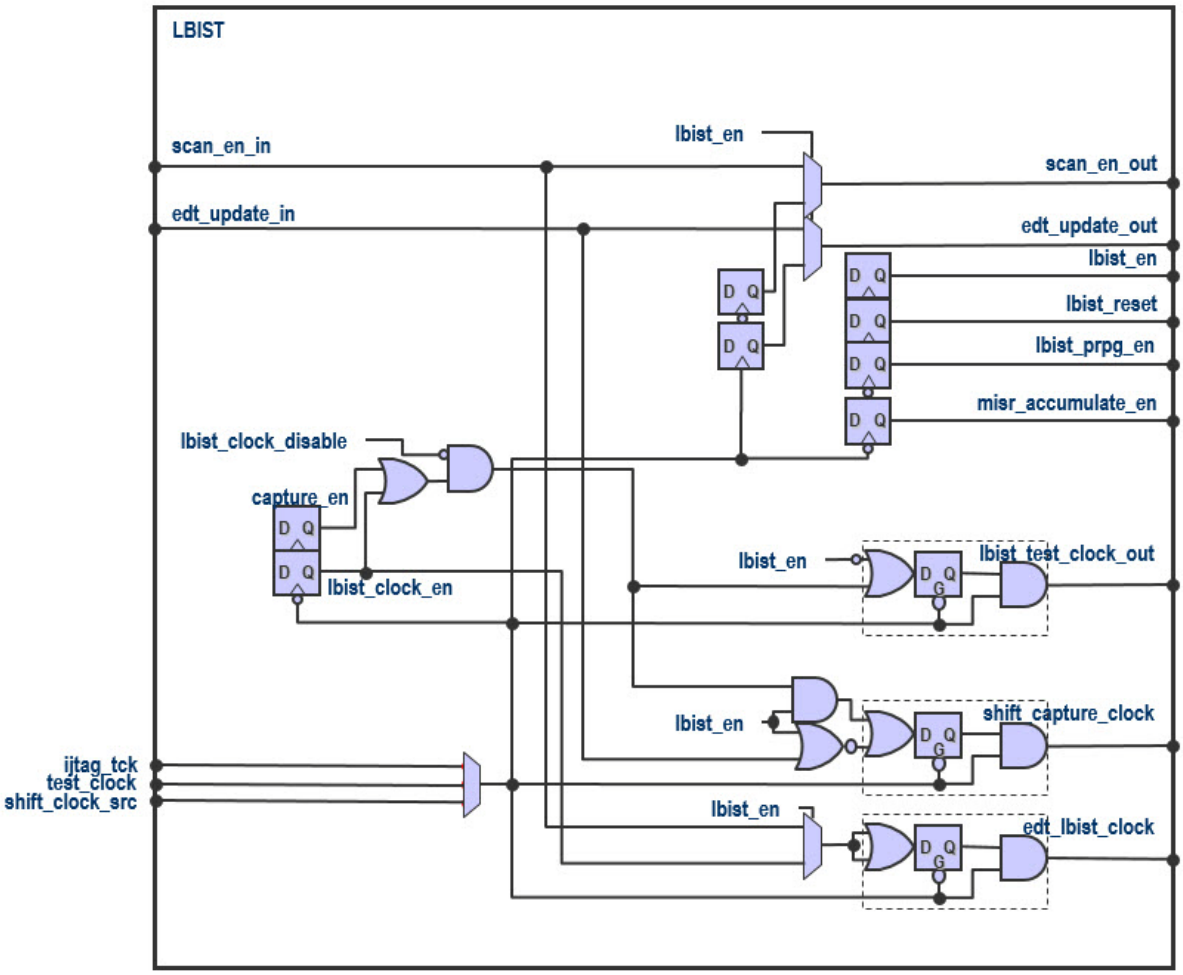
Figure 12-7. LBIST-Ready Block After Insertion Using capture_en



LBIST Controller Clock Gating Signals

In order for the LogicBist controller to work with the LBIST-ready EDT blocks, it must provide lbist_test_clock_output. Figure 12-8 shows how the clock-gating signals are generated inside the LBIST controller.

Figure 12-8. LBIST Controller Clock-Gating Signals



LBIST Load/Unload Timing

The load/unload timing during the LogicBIST operation is fully programmable in the independent insertion flow.

The set_load_unload_timing_options command enables you to modify aspects of the timing diagram such as the amount of setup and hold margin provided between the shift_capture_clock and the controls (scan_en, lbist_prpg_en, misr_accumulate_en, edt_clock, and edt_update).

The required amount of setup and hold timing for at-speed signals is difficult to predict until you have implemented the clock trees and high-fanout distribution buffering schemes during the layout implementation step. After synthesis, those are still ideal nets, making it impossible to get a precise timing relationship between the clock and the controls. To account for this, use the set_load_unload_timing_options command to program the timing for the LogicBIST controller.

See set_load_unload_timing_options in the *Tessent Shell Reference Manual* for details on control signal timing in the independent insertion flow.

When the independent insertion flow is used:

- The *scan_en* transitions on the positive edge of the shift clock.
- *lbist_prpg_en* and *misr_accumulate_en* transition on the negative edge of the clock.

You can add extra setup and hold cycles for the scan_en, edt_update, lbist_prpg_en, and misr_accumulate_en signals. See the HardwareDefaults properties in the SetLoadUnloadTimingOptions wrapper, and the set_load_unload_timing_options command. The scan_en, edt_update, prpg_en, and misr_en signals each get new TDRs that are placed behind an SIB so that the number of extra setup/hold cycles can be loaded at runtime. The prpg_en and misr_en signals share the same TDR because they have the same timing characteristics. Use the SetLoadUnloadTimingOptions wrapper to control the maximum number of cycles for the scan_en, setup_en, hold_en, edt_update, prpg_en, and misr_en signals.

During the LBIST controller operation, the tool can apply a total of max cycles before and after the capture clocks are pulsed to help with the timing closure of the test logic. By default, the lowest possible number of dead cycles are used. There is one dead cycle during shift pause, and two in capture pause. This gives a setup martin of 0 extra cycles and a scan_en hold margin of one cycle. These are hard-coded values.

Timing Constraints (SDC)

In the TK/LBIST independent insertion flow, child physical blocks may include LBIST-ready EDT controller blocks. In this flow, the block-level SDC should constrain these controllers for both EDT and LBIST modes.

SDC File Contents

Features in certain SDC procedures support LBIST-ready blocks.

LBIST-Ready Blocks

You can create LBIST-ready blocks with edt_clock and shift_capture_clock provided by primary inputs, or generated internally from the test_clock signal. Depending on the clocking scheme and whether the block contains OCC, there are several valid LBIST-ready block configurations.

Figure 12-9. LBIST-Ready Block With PI Clocking Scheme and No OCC

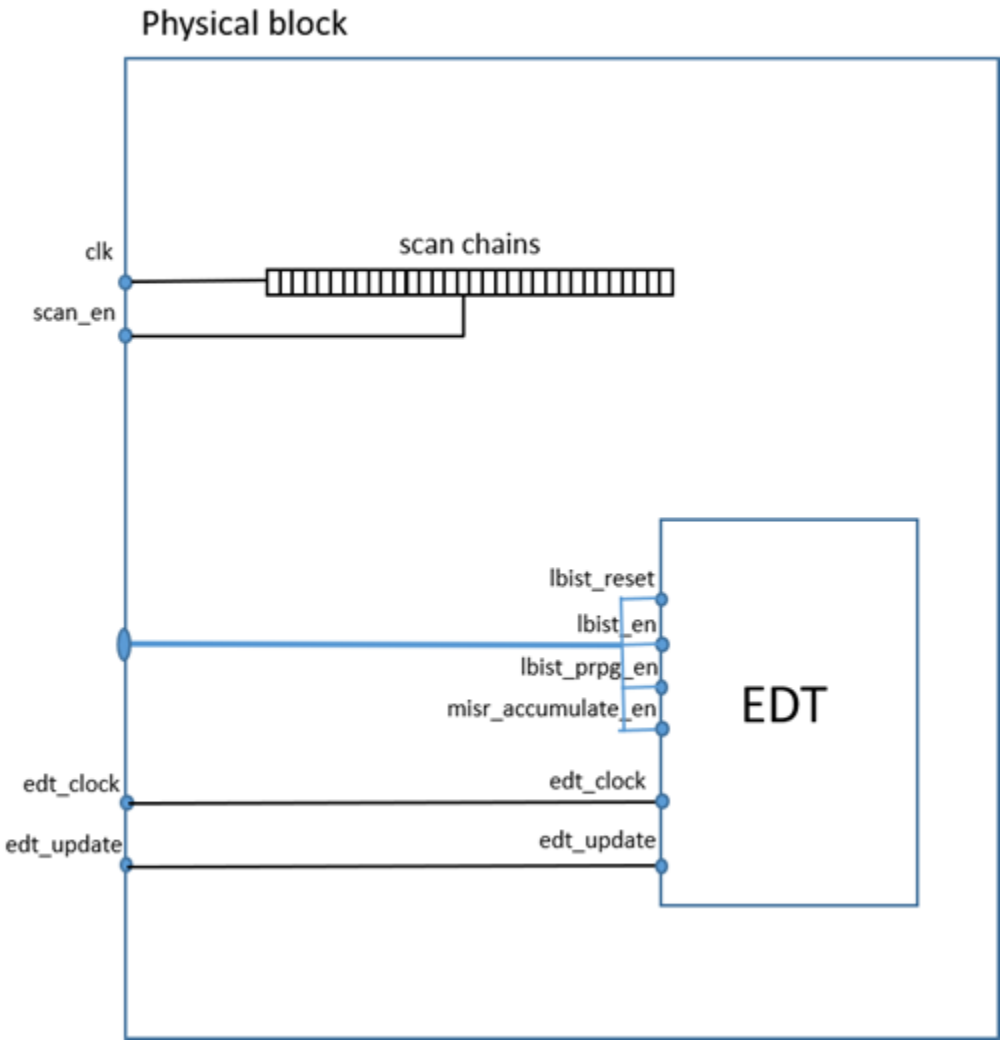


Figure 12-10. LBIST-Ready Block With PI Clocking Scheme and OCC

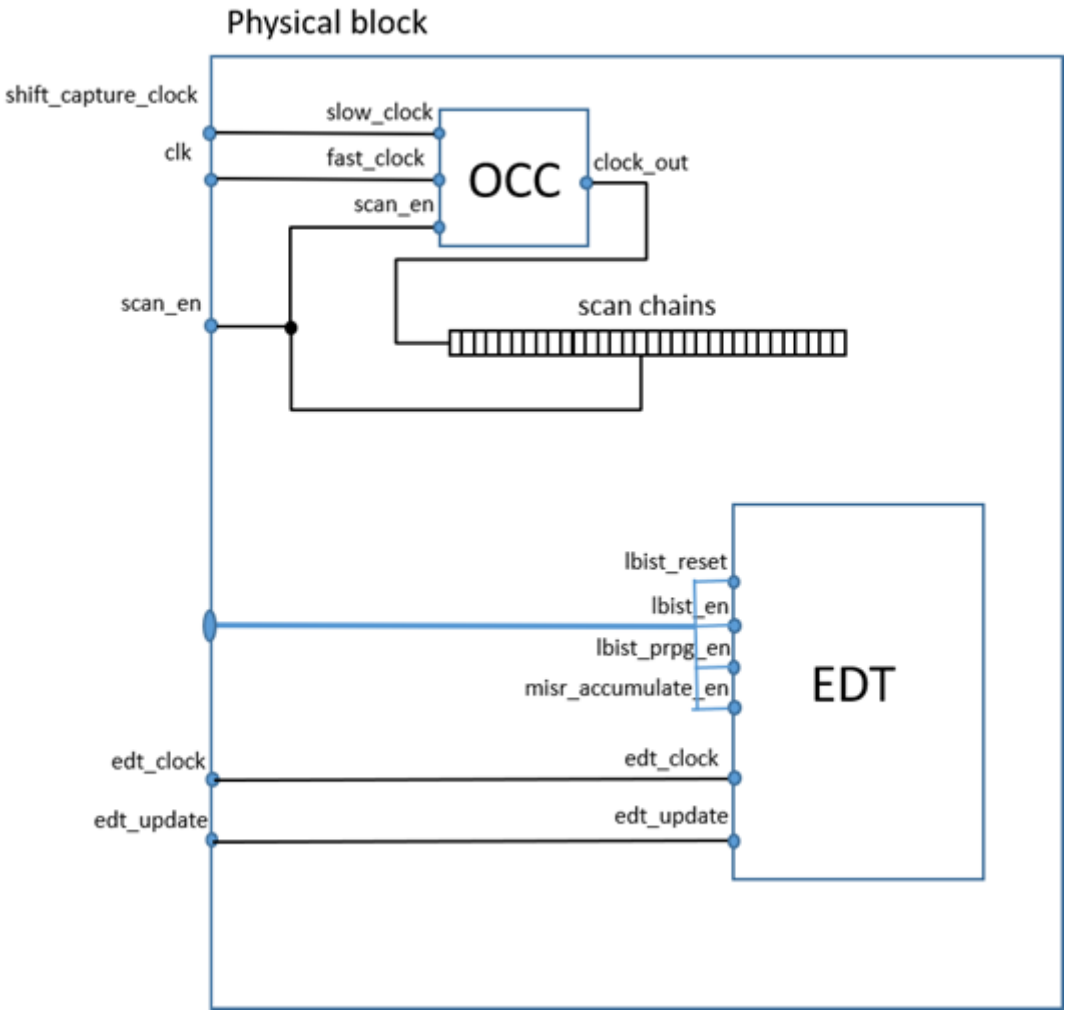
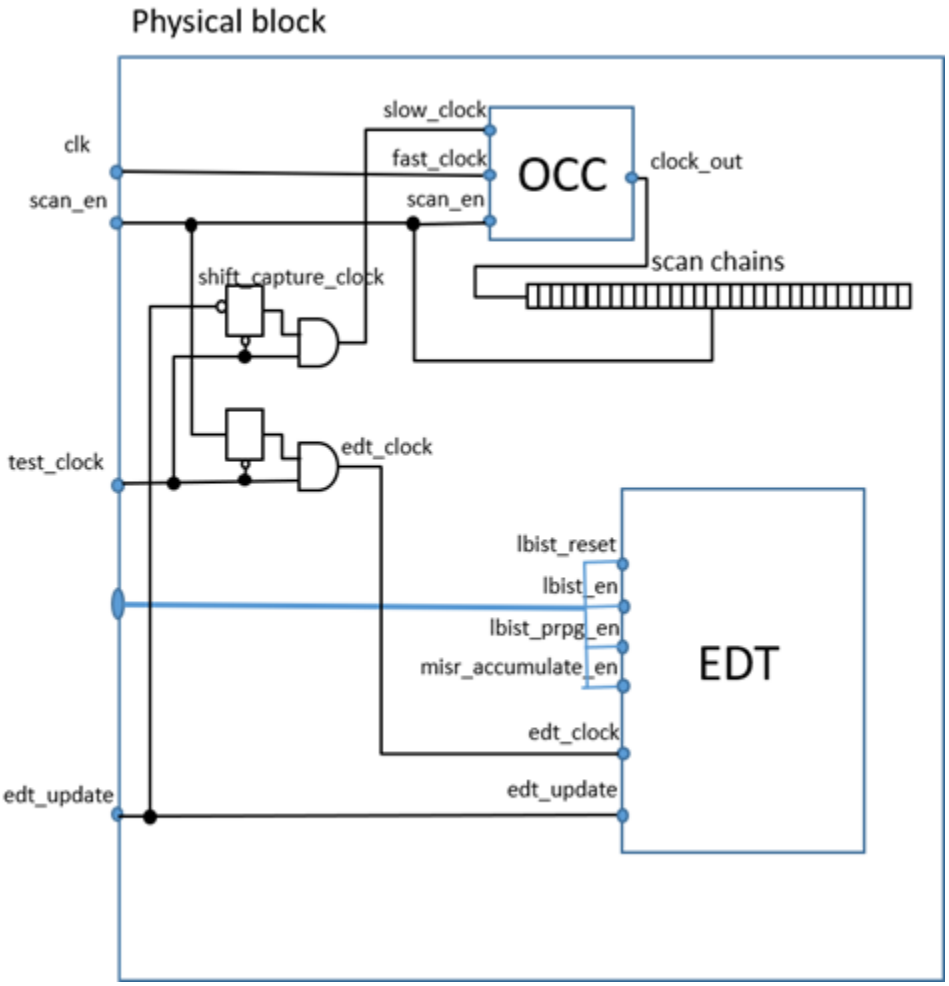


Figure 12-11. LBIST-Ready Block With test_clock Clocking Scheme and OCC



Hierarchical STA

You can create LBIST-ready physical blocks in the independent insertion flow with the LBIST controller inserted in a parent block. This requires a set of procedures that can retarget SDC constraints for the hybrid EDT logic in the individual child physical blocks.

All physical blocks use the same set of timing parameters (for example, the number of scan_en, edt_update, prpg_en, and misr_en extra LBIST setup and hold cycles) specified for the current design by the set_load_unload_timing_options command.

If your flow requires child LBIST-ready physical block instance netlists, replace your call to the LBIST modal procedures with their "*_with_sub_PBs" equivalents (See "SDC Procedures for Hierarchical STA With Independent Insertion Flow" on page 216) to enable the blocks associated with the LBIST controller. To use these procedures, you must load the full or partial (graybox) netlists of the sub-physical LBIST-ready blocks.

The SDC extraction infrastructure at the parent level with the LBIST controller provides an SDC procedure to prepare all sub-physical blocks and any nested physical blocks for LBIST-related static timing analysis.

See Figure 12-12 and Figure 12-13 for an example of LBIST-ready physical blocks controlled by the LBIST controller in the parent block.

Figure 12-12. LBIST-Ready Physical Block

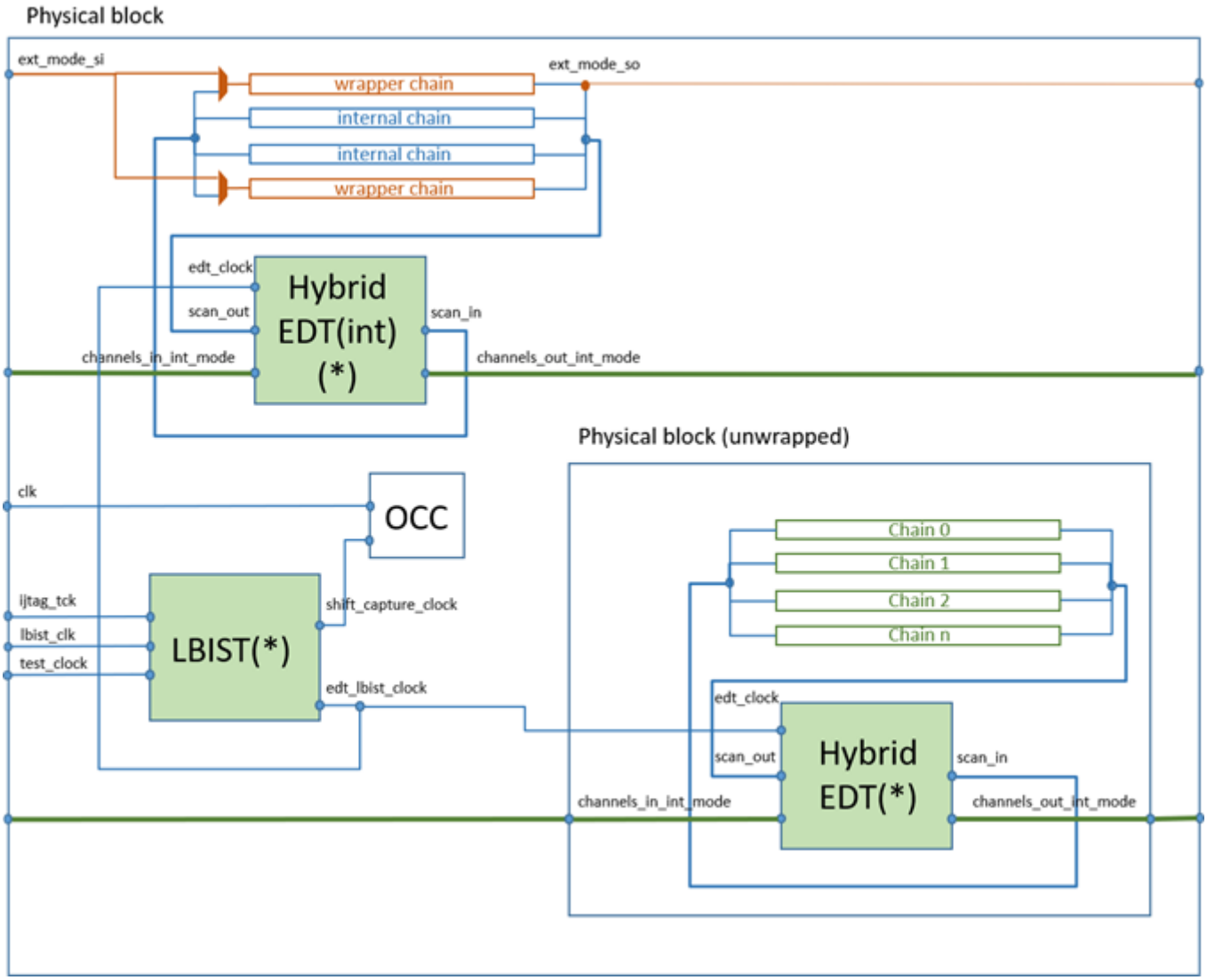
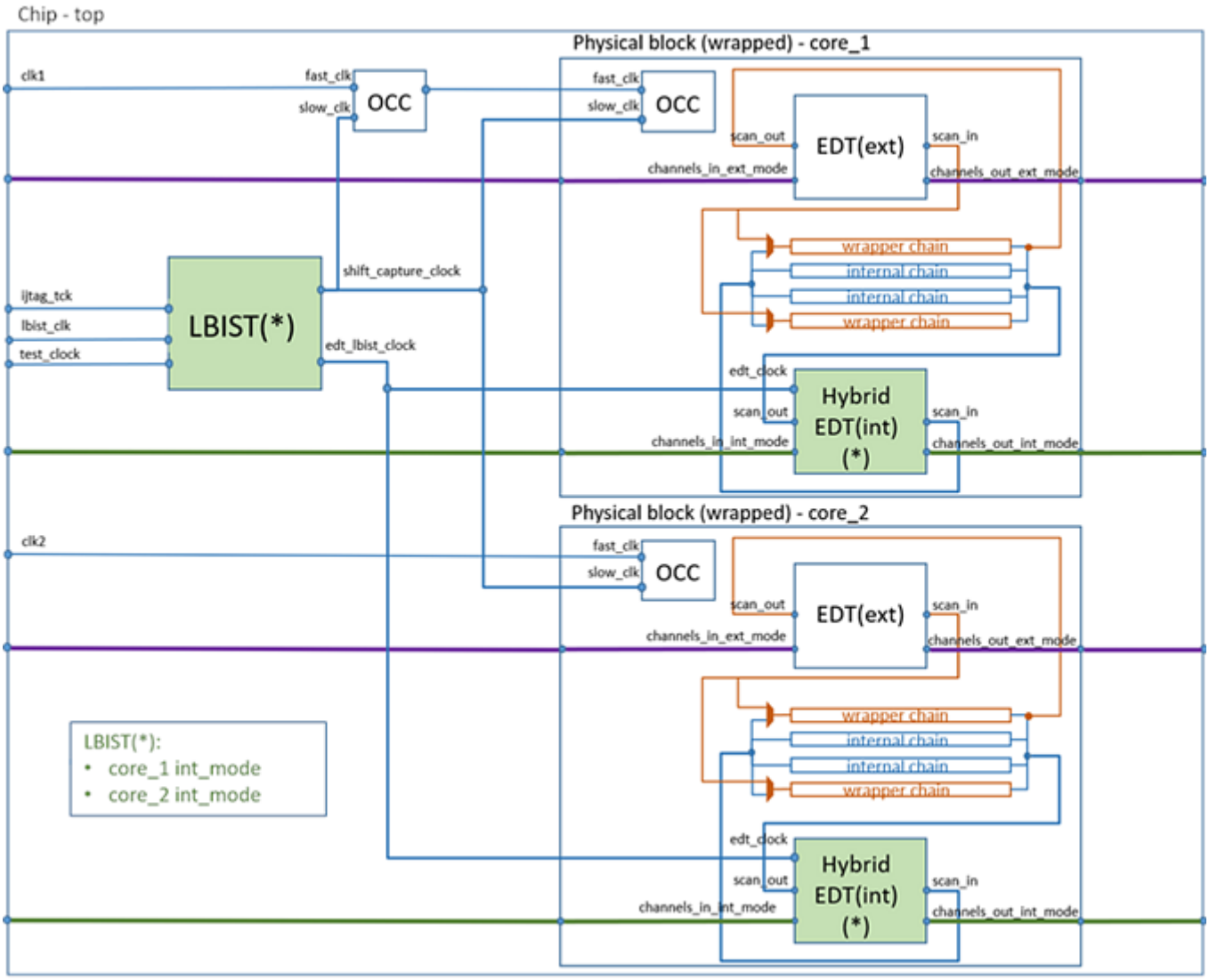
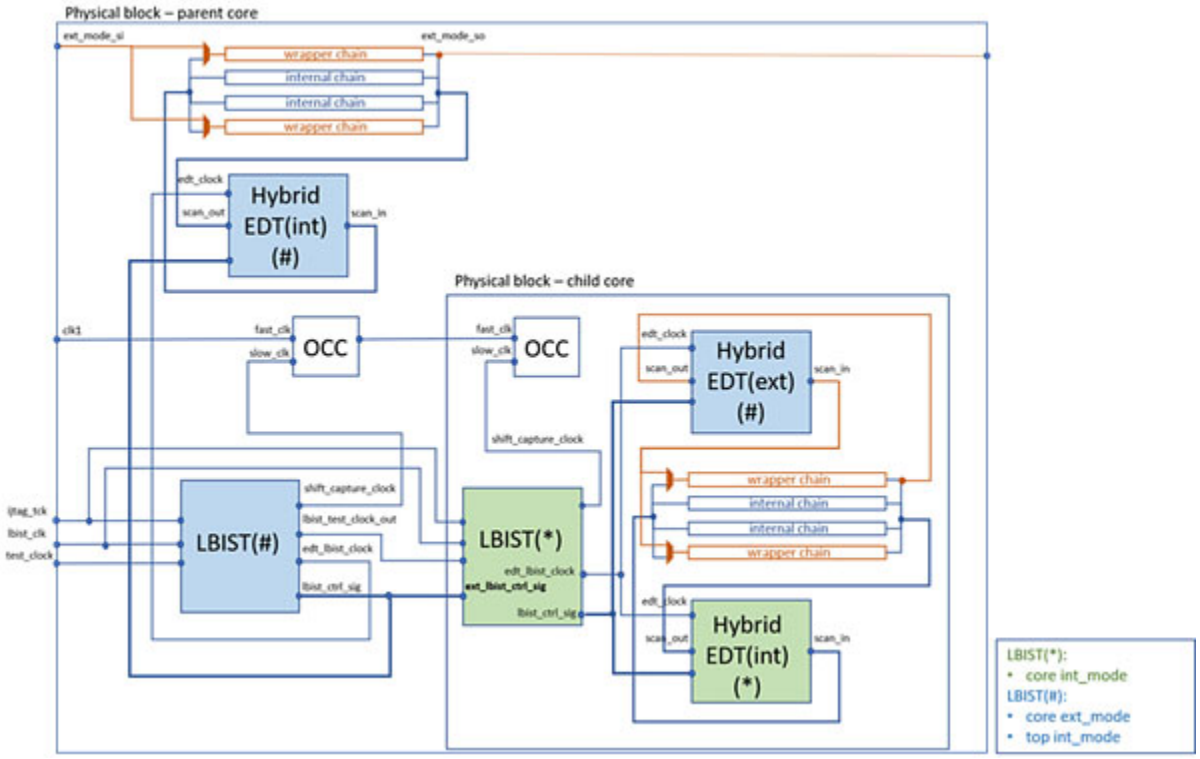


Figure 12-13. LBIST Controller in Parent Block



Hierarchical STA support is also provided for designs where the child physical block has a LBIST controller and a separate hybrid EDT for its wrapper chains, and the parent level LBIST controller is associated with the ext_mode hybrid EDT from core as shown in Figure 12-14. In this case, the call to the LBIST modal procedures are replaced with their "*_with_sub_PBs" equivalents. See "SDC Procedures for Hierarchical STA With Independent Insertion Flow" on page 216for details.

Figure 12-14. Hybrid EDT for External Mode Controlled by Parent-Level LBIST



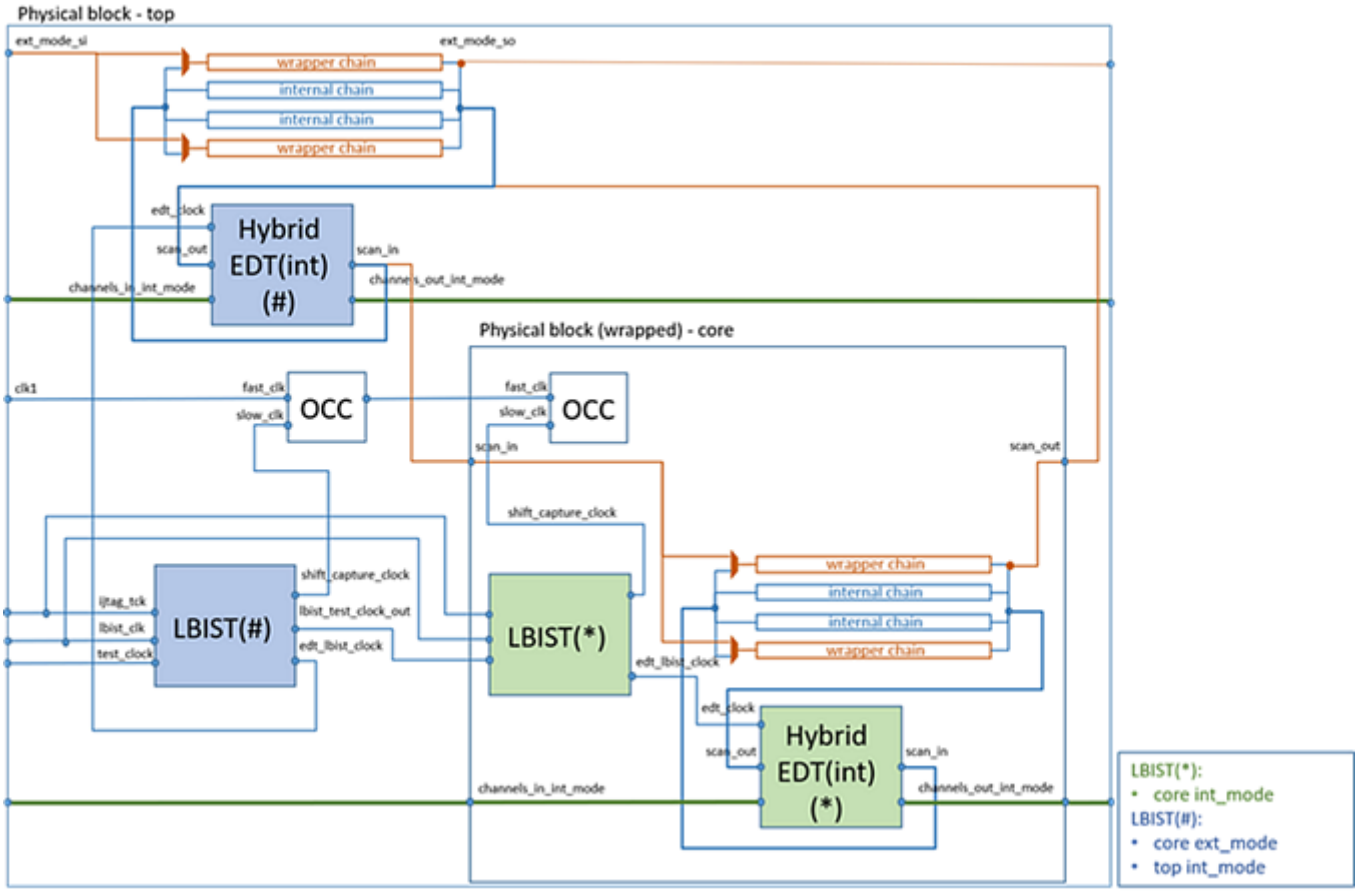
STA For Legacy Hierarchical TK/LBIST Flow

The independent insertion flow supports the legacy flow for concurrent hierarchical insertion of TK/LBIST. No changes for SDC are required.

In the TK/LBIST concurrent insertion flow for hierarchical designs, there is one LBIST controller per physical region, and the LBIST controller is defined with hybrid EDTs in the same pass. The LBIST controller inserted into the sub-physical block controls associated hybrid EDTs and performs LBIST for int_mode of the core. The LBIST for ext_mode of the sub-physical block is handled by another LBIST controller inserted at the parent level. The wrapper chains of the sub-physical block are part of the int_mode of the parent level design and are driven by hybrid EDT from the parent level.

This flow is also supported in the independent insertion flow. The hierarchical STA of LBIST-related modes does not require any special procedures. You can perform STA independently for the sub-physical block and for the parent level by calling the procedure tessent_set_ltest_lower_pbs_external_mode.

Figure 12-15. Illustration of the Legacy Hierarchical TK/LBIST Flow



Extended SDC Procedures

Certain existing SDC procedures are extended for the independent TK/LBIST insertion flow.

- tessent_set_ltest_non_modal
- tessent_set_ltest_modal_shift
- tessent_set_ltest_modal_edt_fast_capture
- tessent_set_ltest_modal_edt_slow_capture
- tessent_set_ltest_create_clocks
- tessent_set_ijtag_non_modal

See "LOGICTEST Instruments" in the *Tessent Shell User's Manual* for complete information on these procedures.

tessent_set_ltest_non_modal

This procedure provides SDC timing constraints to add to combined functional DFT non-modal timing scripts for one-pass synthesis or layout.

Note

Before running this procedure for an LBIST-ready block, you must specify the lbist_shift_clock parameters by setting the <*tessent_lbist_shift_clock_name*> and <*tessent_lbist_shift_clock_period*> variables. These variables define the name and period of the clock that the procedure adds on the appropriate ports of the LBIST-ready block.

When scan cells are clocked by the functional clock input as shown in Figure 12-9 on page 204, the procedure defines lbist_shift_clock on the top-level ports you specify with the <*tessent_lbist_clock_source_list*> global variable.

The independent insertion flow adds the following constraints related to LBIST mode:

- Propagate both fast lbist_shift_clock and slow test_clock to the design cells, and block interactions between them.
- Add false paths from the EDT mask registers to all clock domains except TCK.
- When CCM is implemented with the EDT clock as the CCM clock, add false paths from:
  - the EDT SIBs.
  - single chain mode logic SIBs.
- Disable clock gating checks for MUXes that inject TCK.
- Disable TCK propagation through the edt_lbist clock path of the inject_tck mux.
- Add multi-cycle paths (MCPs) from the scan enable port to the design scan cells using the fast lbist_shift_clock. The tool determines the number of cycles with the set_load_unload_timing_options procedure. The tool adds this MCP in addition to the scan_en MCP for ATPG that uses the same port but the slow test_clock instead.
- Add MCPs from ports that provide the prpg_en, misr_en, LogicBIST async reset, and edt_update signals to the hybrid EDT blocks.

tessent_set_ltest_modal_shift

This procedure sets the circuit in scan shift mode. The independent insertion flow adds the following constraints related to LBIST mode:

- Add case analysis to set lbist_en to inactive.
- Add false paths from the EDT mask registers.

- Set case analysis to disable the EDT controller chain mode.
- Set case analysis to disable LBIST single chain mode concatenation.

tessent_set_ltest_modal_edt_fast_capture

This procedure sets the circuit in EDT fast capture mode. The independent insertion flow adds the following constraints related to LBIST mode:

- Add case analysis to set lbist_en to inactive.
- Set case analysis to disable the EDT controller chain mode.
- Set case analysis to disable LBIST single chain mode concatenation.

tessent_set_ltest_modal_edt_slow_capture

This procedure sets the circuit in EDT slow capture mode. The independent insertion flow adds the following constraints related to LBIST mode:

- Add case analysis to set lbist_en to inactive.
- Add false paths from the EDT mask registers.
- Set case analysis to disable the EDT controller chain mode.
- Set case analysis to disable LBIST single chain mode concatenation.
- Add false paths from ports providing to signals (lbist_misr_accumulate_en and lbist_reset) that control MISR.

tessent_set_ltest_create_clocks

This procedure creates the slow-speed test clocks used during scan mode. The independent insertion flow adds the following constraint related to LBIST mode:

- Create a slow test clock on the ports that are specified with the tessent_lbist_clock_source_list global variable. This functionality is needed when the scan cells are clocked by functional clocks as shown in Figure 12-9 on page 204.

tessent_set_ijtag_non_modal

This procedure creates the clock for TCK and configures input and output delays for ports created at the sub_block and physical_block design levels. The independent insertion flow adds the following constraints related to LBIST mode:

- Defines the ijtag_tck clock on the appropriate ports of the lbist-ready block (for example, test_clock, shift_capture_clock, and edt_clock).

- Defines the ports to be clocked by ijtag_tck when the scan cells are clocked by a functional clock input. These ports are identified with the *<tessent_lbist_clock_source_list>* global variable.

SDC Procedure Generation for Hybrid EDTs

The tool generates the following SDC procedures in the flow when hybrid EDTs are present. These procedures are also generated when the block contains hybrid EDTs without LBIST controllers for STA purposes.

- tessent_set_ltest_modal_lbist_shift
- tessent_set_ltest_modal_lbist_capture
- tessent_set_ltest_modal_lbist_setup
- tessent_set_ltest_modal_lbist_single_chain
- tessent_set_ltest_modal_lbist_controller_chain
- set_load_unload_timing_options

Note

Before running the tessent_set_ltest_modal_shift and tessent_set_ltest_modal_capture procedures for LBIST-ready blocks in STA using the fast lbist_shift_clock, you must specify the lbist_shift_clock parameters by setting the tessent_lbist_shift_clock_name and tessent_lbist_shift_clock_period variables. These variables define the name and period of the clock that the procedure adds on the appropriate ports of the LBIST-ready block.

tessent_set_ltest_modal_lbist_shift

This procedure sets the circuit in LBIST shift mode. You can configure the timing analysis for this mode to run with either shift_clock_src (by default) or test_clock.

To run this procedure with timing analysis for shift_clock_src, set the tessent_lbist_shift_clock_name and tessent_lbist_shift_clock_period variables before invoking the procedure. These variables define the name and period of the clock that will be automatically added on the appropriate ports of the LBIST-ready block.

When the scan cells are clocked by functional clock inputs (see Figure 12-9 on page 204), the SDC procedure must define the clock signals (slow test_clock or fast lbist_shift_clock) on the top level ports that you specify with the tessent_lbist_clock_source_list global variable.

The procedure does the following:

- Adds multi-cycle path exceptions on dynamic signals from the ports related to LBIST signals to design scan cells and hybrid EDT blocks. These include scan enable, prpg_en, misr_en, LogicBIST async reset, and edt_update.

- Adds case analysis to set lbist_en to active.
- Adds false paths from the EDT mask registers.
- Sets case analysis to disable EDT controller chain mode.
- Sets case analysis to disable LBIST single chain mode concatenation.
- Adds false paths from all static DFT signals defined on ports, except lbist_en.

tessent_set_ltest_modal_lbist_capture

This procedure sets the circuit in lbist capture mode. Configure the timing analysis for this mode as described for tessent_set_ltest_modal_lbist_shift. The procedure does the following:

- Sets case analysis to constrain scan enable to off.
- Adds multi-cycle path exceptions on dynamic signals from the ports related to LBIST signals to design scan cells and hybrid EDT blocks. These include prpg_en, misr_en, LogicBIST async reset, and edt_update.
- Adds case analysis to set lbist_en to active.
- Adds false paths from the EDT mask registers.
- Sets case analysis to disable EDT controller chain mode.
- Sets case analysis to disable LBIST single chain mode concatenation.
- Adds false paths from all static DFT signals defined on ports, except lbist_en.

tessent_set_ltest_modal_lbist_setup

This mode propagates TCK through the IJTAG network in the hybrid EDT blocks to time the LogicBIST test_setup paths that initialize registers such as PRPG and edt_chain_mask, as well as test_end paths that read the MISR signature.

This mode does not propagate TCK to the design scan cells. It is only intended to check the IJTAG network paths.

The procedure does the following:

- Adds multi-cycle path exceptions on dynamic signals from the ports related to LBIST signals to design scan cells and hybrid EDT blocks. These include prpg_en, misr_en, LogicBIST async reset, and edt_update.
- Adds case analysis to set lbist_en to active.
- Sets case analysis to disable the EDT controller chain mode.
- Sets case analysis to disable LBIST single chain mode concatenation.

- Sets case analysis to propagate TCK through the inject_tck MUX.
- Adds false paths from all static DFT signals defined on ports, except lbist_en.

tessent_set_ltest_modal_lbist_single_chain

This mode is available when single chain mode logic that enables LogicBIST diagnosis is enabled during IP generation. This mode propagates TCK through the IJTAG network paths and design scan cells, including the concatenation of the internal scan chains for single-chain shifting. The LogicBIST scan enable is constrained to 1 to time the shift paths in the design only with the TCK signal.

The procedure does the following:

- Sets case analysis to constrain the LogicBist mode scan enable signal to on.
- Adds multi-cycle path exceptions on dynamic signals from the ports related to LBIST signals to design scan cells and hybrid EDT blocks. These include prpg_en, misr_en, LogicBIST async reset, and edt_update.
- Adds case analysis to set lbist_en to active.
- Adds false paths from the EDT mask registers.
- Sets case analysis to disable the EDT controller chain mode.
- Sets case analysis to enable LBIST single chain mode concatenation.
- Sets case analysis to propagate edt_lbist clock through the inject_tck MUX.
- Adds false paths from all static DFT signals defined on ports, except lbist_en.

tessent_set_ltest_modal_lbist_controller_chain

This mode is available when the controller chain mode (CCM) logic is enabled during IP generation. The clock for this mode is either TCK, test_clock, or edt_clock, depending on the clocking scheme specified during IP creation. This mode tests only the LogicBIST and hybrid EDT controller blocks. Clocks are not propagated to design scan cells.

The procedure does the following:

- Sets input and output pin delays from the top-level ports to EDT blocks.
- Blocks propagation to scan cells by disabling timing for ports that provide the source of shift_capture_clock. When the scan cells are clocked by functional clock inputs as shown in Figure 12-9 on page 204, the procedure disables timing on the top level ports specified by the tessent_lbist_clock_source_list variable.
- Sets case analysis to enable the EDT controller chain mode.
- Sets case analysis to disable LBIST single chain mode concatenation.

- Sets case analysis to propagate TCK through the inject_tck MUX.
- Adds false paths from all static DFT signals defined on ports.

set_load_unload_timing_options

This procedure sets global Tcl timing variable values used in LBIST constraints. These include extra setup and hold clock cycles for scan_en, edt_update, prpg_en, and misr_en.

The LBIST-related modal STA procedures generated for designs where not all hybrid EDTs are associated with the LBIST controller from the current design level set different constraints for EDTs. For the unassociated EDTs, there are applied constraints setting them into the inactive state. This rule applies for all modal lbist procedures except for <*ltest_prefix*>_modal_lbist_controller_chain, which includes the same constraints for all available hybrid EDTs because this mode is not controlled by the LBIST controller but rather by a tester.

SDC Procedures for Hierarchical STA With Independent Insertion Flow

If the design contains LBIST-ready or extest LBIST-ready physical blocks, the tool generates the following SDC procedures for the flow.

These procedures are chip-level versions of those described in "SDC Procedure Generation for Hybrid EDTs" on page 213. For example, tessent_set_ltest_modal_lbist_shift_with_sub_PBs is the hierarchical STA version of tessent_set_ltest_modal_lbist_shift.

- tessent_set_ltest_modal_lbist_shift_with_sub_PBs
- tessent_set_ltest_modal_lbist_capture_with_sub_PBs
- tessent_set_ltest_modal_lbist_setup_with_sub_PBs
- tessent_set_ltest_modal_lbist_single_chain_with_sub_PBs
- tessent_set_ltest_modal_lbist_controller_chain_with_sub_PBs

In addition to these, the tool provides tessent_set_ltest_lower_pbs_logicbist_internal_mode or tessent_set_ltest_lower_pbs_logicbist_external_mode (depending on existing child cores) to prepare the sub-physical blocks and any nested physical blocks for LBIST-related static timing analysis. This procedure performs the following tasks:

- Sets all wrapped sub-physical blocks to internal mode, if the corresponding DFT signals are present:
  - For tessent_set_ltest_lower_pbs_logicbist_internal_mode:
    - int_ltest_en = "1"

    - int_mode = “1”
    - ext_ltest_en = “0”
    - ext_mode = “0”
  - For tessent_set_ltest_lower_pbs_logicbist_external_mode:
    - int_ltest_en = “0”
    - int_mode = “0”
    - ext_ltest_en = “1”
    - ext_mode = “1”
    - ext_lbist_en = “1”
- Disables the clock gating checks on the clock multiplexers for cascaded OCCs from sub-physical blocks (OCCs for which the fast_clock input is connected to the parent-level OCC’s output). Also, timing is disabled on the slow_clock paths of these OCCs because they are inactive during LBIST.

Example: STA of the lbist_shift and lbist_capture Modes

```
# load netlist of lbist-ready PBs and top design
read_verilog <sub-PB_netlist>
read_verilog <top_netlist>
current_design <top_design_name>
link_design

set tessent_lbist_shift_clock_src(lbist_inst0) lbist_clock
source <top_sdc_file>
# lbist_shift with lbist_clock
create_clock lbist_clock -period 10 -name \
  $tessent_lbist_shift_clock_src(lbist_inst0)
tessent_set_default_variables
tessent_set_ltest_lower_pbs_logicbist_internal_mode lbist_shift
tessent_constrain_<top>_mentor_ltest_modal_lbist_shift_with_sub_PBs
update_timing

# lbist_capture with lbist_clock
reset_design
set tessent_lbist_shift_clock_src(lbist_inst0) lbist_clock
create_clock lbist_clock -period 10 -name \
  $tessent_lbist_shift_clock_src(lbist_inst0)
tessent_set_default_variables
tessent_set_ltest_lower_pbs_logicbist_internal_mode lbist_capture
tessent_constrain_<top>_mentor_ltest_modal_lbist_capture_with_sub_PB
s
update_timing
```

Example: STA of the lbist_shift and lbist_capture Modes for Designs Containing Cores With Hybrid EDTs for ext_mode

```
# load netlist of lbist-ready PBs and top design
read_verilog <sub-PB_netlist>
read_verilog <top_netlist>
current_design <top_design_name>
link_design

set tessent_lbist_shift_clock_src(lbist_inst0) lbist_clock
source <top_sdc_file>
# lbist_shift with lbist_clock
create_clock lbist_clock -period 10 -name \
  $tessent_lbist_shift_clock_src(lbist_inst0)
tessent_set_default_variables
tessent_set_ltest_lower_pbs_logicbist_external_mode lbist_shift
tessent_constrain_<top>_mentor_ltest_modal_lbist_shift_with_sub_PBs
update_timing

# lbist_capture with lbist_clock
reset_design
set tessent_lbist_shift_clock_src(lbist_inst0) lbist_clock
create_clock lbist_clock -period 10 -name \
  $tessent_lbist_shift_clock_src(lbist_inst0)
tessent_set_default_variables
tessent_set_ltest_lower_pbs_logicbist_external_mode lbist_capture
tessent_constrain_<top>_mentor_ltest_modal_lbist_capture_with_sub_PBs
update_timing
```

Generating EDT and LogicBIST IP for Independent Insertion

You can generate the LogicBIST-ready EDT for independent insertion in a child block with the OCC in the parent physical block or with the OCC in the child block with the EDT.

Generating LogicBIST-Ready EDT Child Blocks Without OCC

Typically, LBIST-ready EDT blocks are inserted in child blocks. In a subsequent pass, the LBIST controller and OCC are added in the parent physical block.

Independently Inserting the LogicBIST-Ready EDT in Child Blocks

You can insert the LogicBIST-ready EDT child blocks without OCC independently from the LogicBIST controller.

Prerequisites

- RTL or gate-level netlist for the child block. In a typical two-pass DFT insertion flow, this netlist is the output of the MemoryBIST inserted in pass one.
- The OCC is not present in the child block.
- The LBIST NCP index decoder is specified at the parent level.

Procedure

1. Set context to be RTL DFT insertion:

   ```
   set_context dft -rtl -design_id rtl2
   ```

2. Load the cell library and the design from the first insertion pass:

   ```
   read_cell_library tessent.lib
   read_design Block1 -design_id rtl1
   ```

3. Set the current design and the design level:

   ```
   set_current_design Block1
   set_design_level physical_block
   ```

4. Add the standard DFT signal using -source_node option:

   ```
   add_dft_signals  scan_en edt_update  –source_node {scan_en edt_update}
   add_dft_signals edt_clock -source_node edt_clock
   ```

5. Run DRC and go from setup mode to analysis mode:

   ```
   check_design_rules
   ```

6. Create the DftSpecification for the EDT child block:

   ```
   set spec [create_dft_specification -sri_sib_list {edt}]
   ```

7. Specify the LBIST-ready EDT for the child block:

```
read_config_data -in_wrapper $spec -from_string {
   EDT {
      ijtag_host_interface : Sib(edt);
      LogicBistOptions {
         present: on;
         ShiftCycles {
            max                 : 80;
            hardware_default  : 60;
         }
         WarmupPatternCount {
            max                 : 255;
         }
         // seed declaration
         prpg_reference_seed    : 'h1234;
      }
      Controller(c1) {
         scan_chain_count : 100;
         input_channel_count : 2;
         output_channel_count : 2;
         longest_chain_range : 2, 80;
         LogicBistOptions {
            misr_input_ratio :1;
            prpg_seed : 'h4321;
         }
      }
   }
}
```

8. Create the IP and insert the hardware described with the DftSpecification into the design, and update the ICL description with extract ICL:

```
process_dft_specification
extract_icl
```

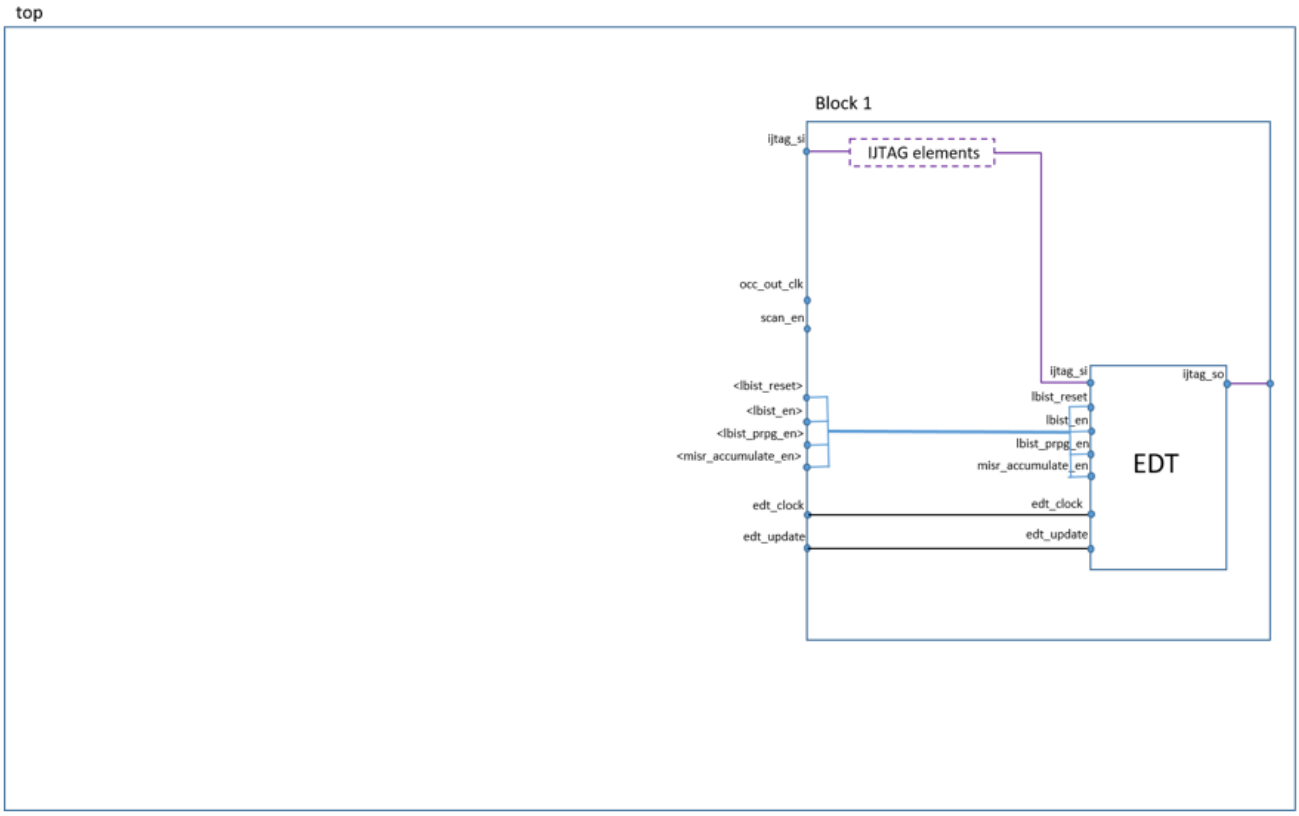
9. Exit the tool.

Results

A design netlist with the instantiation of the EDT logic and a separate RTL file. Figure 12-16 shows the LBIST-ready child block that is created in this example. This child block contains TK/LBIST EDT and all necessary ports to enable control by an LBIST controller. The names of the child block ports were specified by explicitly adding LBIST-related DFT signals with the source_node option.

You are now ready to create the parent-level LogicBIST controller.

Figure 12-16. LBIST-Ready EDT Child Block



Generating the LogicBIST Controller With Parent Level EDT

Once you have created your LogicBIST-ready EDT child blocks, you can then independently insert and connect the LogicBIST controller with the LBIST-ready blocks and the EDT at the parent level. The parent-level netlist will contain the instantiation of the previously generated child blocks with LBIST-ready EDT.

Prerequisites

- MBIST inserted netlist from pass one.
- Independently generated ICL and netlist files, such as Block1.icl and Block1.v.

Procedure

1. Load the design and the child block:

```
set_context dft -rtl -design_id rtl2
read_design parentBlock -design_id rtl1
read_design Block1 -design_id rtl2
```

2. Set the current design:

```
set_current_design parentBlock
set_design_level physical_block
```

3. Define the top-level clocks:

```
add_clocks 0 clk1 -period 10ns
add_clocks lbist_clock -period 10ns
```

4. Define DFT signals at the parent level:

```
add_dft_signals ltest_en int_ltest_en ext_ltest_en int_mode ext_mode
add_dft_signals scan_en test_clock edt_update -source_node \
      { scan_en test_clock edt_update}
add_dft_signals edt_clock shift_capture_clock -create_from_other_signals

add_dft_signals observe_test_point_en control_test_point_en -create_with_tdr
add_dft_signals x_bounding_en -create_with_tdr
add_dft_signals mcp_bounding_en -create_with_tdr
add_dft_signals async_set_reset_static_disable
add_dft_signals async_set_reset_dynamic_disable -create_from_other_signals
```

5. Run DRC:

```
check_design_rules
```

6. Create the DftSpecification for parent level the OCC, LBIST, and EDT:

```
set spec [create_dft_specification -sri_sib_list {occ lbist edt}]
```

7. Insert the LBIST controller together with EDT in the parent-level block:

```
read_config_data -in_wrapper $spec -from_string {
    Occ {
        ijtag_host_interface : Sib(occ);
        static_clock_control: external;
        Controller(clk1) {
            clock_intercept_node : clk1;
        }
    }
     EDT {
        ijtag_host_interface : Sib(edt);
        Controller(c3) {
            scan_chain_count : 80;
            input_channel_count : 2;
            output_channel_count : 2;
            longest_chain_range : 2, 80;
            LogicBistOptions {
                present : on;
                misr_input_ratio :1;
            }
        }
    }
    LogicBist {
        ijtag_host_interface : Sib(lbist);
        Controller(ctrl_lbist) {
            burn_in : on;
            Connections {
                shift_clock_src : lbist_clock;
            }
            ShiftCycles { max : 80;}
            CaptureCycles {max : 3;}
            PatternCount {max : 10000;}
            WarmupPatternCount {max : 255;}

            //Block1 is module name and block1 is instance name in the parent
            DesignInstance(block1) {}
        }
        NcpIndexDecoder {
            ncp(first) {
                cycle(0): Occ(clk1);
                cycle(1): Occ(clk1);
            }
            ncp(second) {
                cycle(0): Occ(clk1);
            }
        }
    }
}
```

8. Create the IP and insert the hardware described with the DftSpecification into the design, and update the ICL:

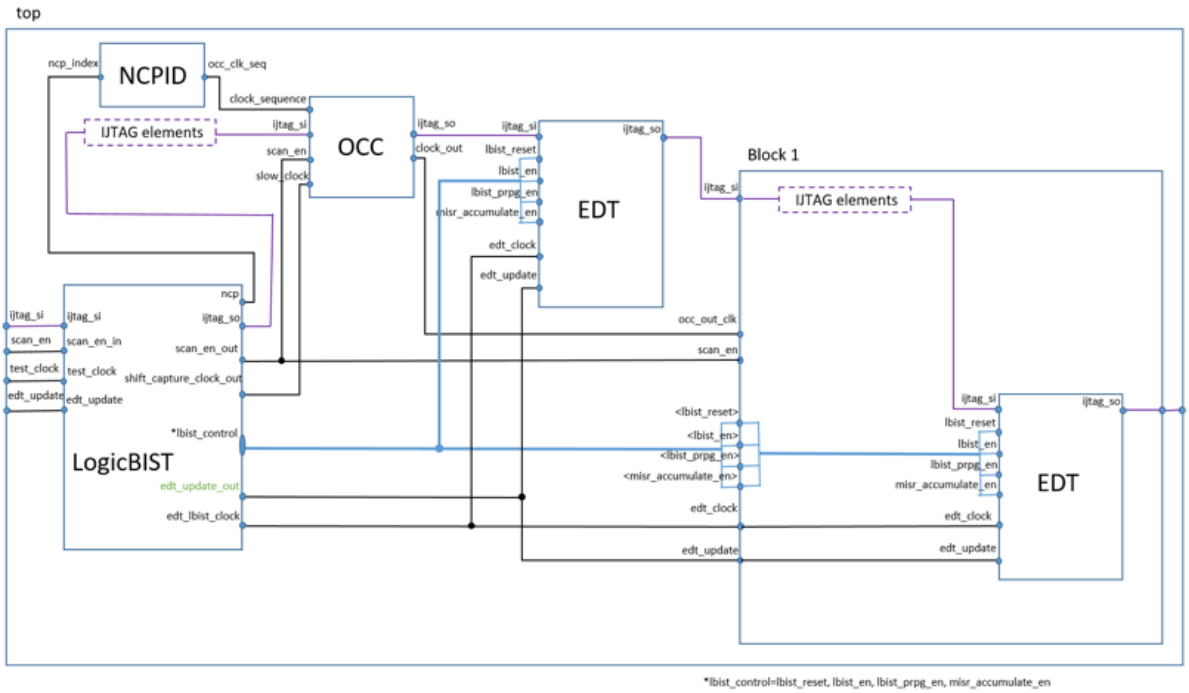
```
process_dft_specification
extract_icl
```

9. Exit the tool.

Results

You have generated the LogicBIST IP for the design at the parent level. Figure 12-17 shows the design after the LBIST controller insertion in the second pass. Note in this example, OCC is only at the parent level.

Figure 12-17. LBIST Controller Inserted After Second Pass



Generating LogicBIST-Ready EDT Child Blocks With OCC

You can insert the LBIST-ready EDT with a separate OCC in the child blocks. You can then insert and connect the LogicBIST controller with the LBIST-ready blocks, OCC, and the EDT specified together with the LogicBIST controller.

Independently Inserting the LogicBIST-Ready EDT in Child Blocks With OCC

You can insert the LBIST-ready EDT with a separate OCC in the child blocks.

Prerequisites

- RTL or gate-level netlist for the child block. In a typical two-pass DFT insertion flow, this netlist is the output of the MemoryBIST inserted in pass 1.

Procedure

1. Load the design for the second insertion pass of the current design level using RTL:

   ```
   set_context dft -rtl -design_id rtl2
   ```

2. Load the cell library and the design from the first insertion pass:

   ```
   read_cell_library tessent.lib
   read_design Block1 -design_id rtl1
   ```

3. Set the current design and the design level:

   ```
   set_current_design Block1
   set_design_level sub_block
   ```

4. Add the standard DFT signals using -source_node option:

   ```
   add_dft_signals  scan_en test_clock edt_update  –source_node \
      {scan_en test_clock edt_update}
   ```

5. Add the DFT signals needed to create the clock gaters that work with the LBIST controller, create shift_capture_clock and edt_clock from other signals:

   ```
   add_dft_signals edt_clock -create_from_other_signals
   add_dft_signals shift_capture_clock -create_from_other_signals
   ```

6. Run DRC and go from setup mode to analysis mode:

   ```
   check_design_rules
   ```

7. Create the DftSpecification for the EDT child block with the OCC:

```
set spec [create_dft_specification -sri_sib_list {edt occ}]
```

8. Specify the LBIST-ready EDT and OCC for the child block:

```
read_config_data -in_wrapper $spec -from_string {
    Occ {
        ijtag_host_interface : Sib(occ);
        static_clock_control: external;
        Controller(clk) {
            clock_intercept_node : clk;
            leaf_instance_name : occ_in_core;
            Connections {
                clock_sequence : clk_seq[%d];
            }
        }
    }


    EDT {
        ijtag_host_interface : Sib(edt);
        LogicBistOptions {
            present: on;
            ShiftCycles {
                max                 : 80;
                hardware_default  : 60;
            }
            WarmupPatternCount {
                max                 : 255;
            }
            // seed declaration
            prpg_reference_seed    : 'h1234;
        }
        Controller(c1) {
            scan_chain_count : 80;
            input_channel_count : 2;
            output_channel_count : 2;
            longest_chain_range : 2, 80;
        }
    }
}
```

9. Create the IP and insert the hardware described with the DftSpecification into the design, and update the ICL:

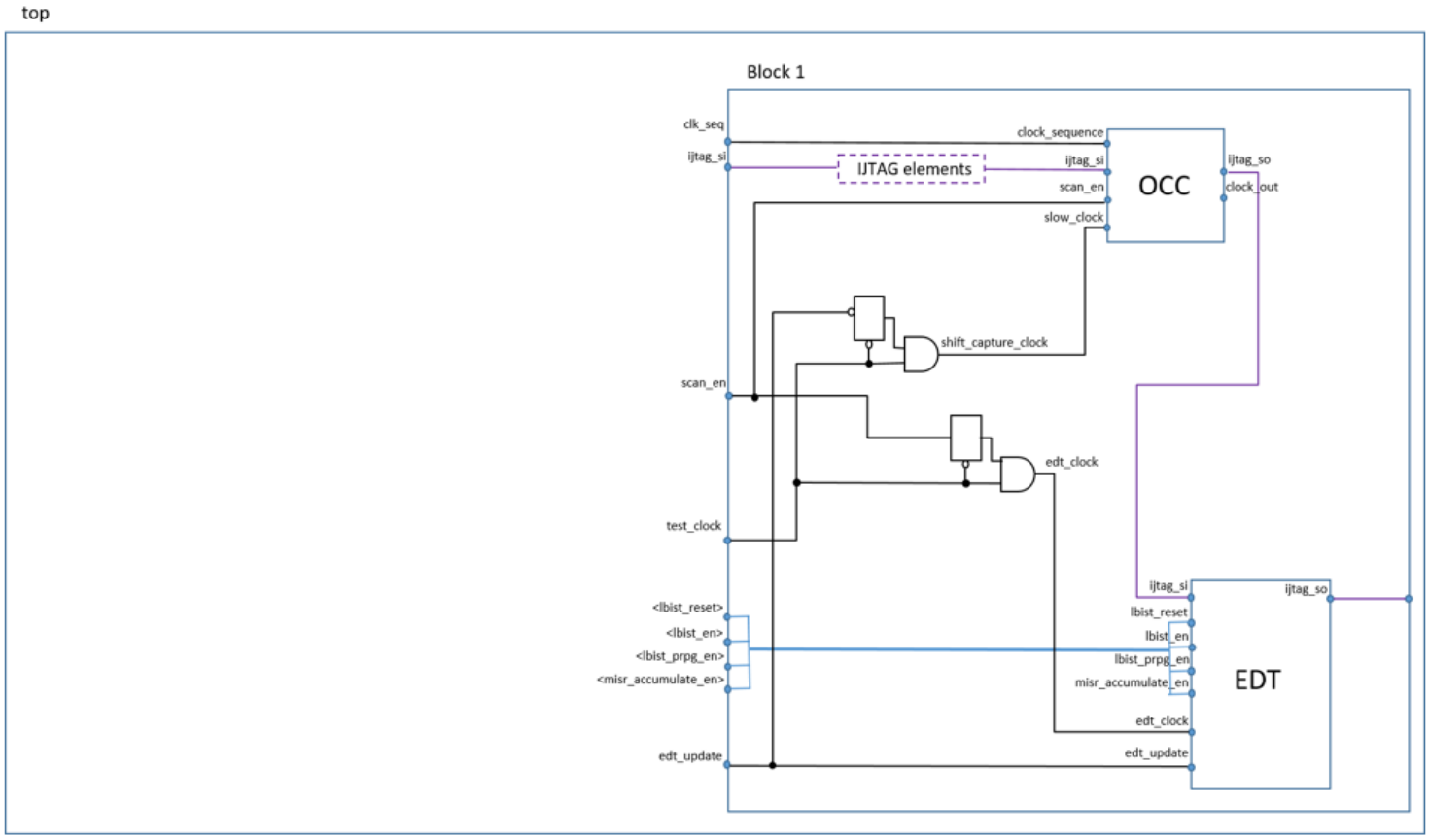
```
process_dft_specification
extract_icl
```

10. Exit the tool.

Results

You have generated a design netlist with the instantiation of the EDT logic and a separate RTL file. Figure 12-18 shows an example of the LBIST-ready child block with a separate OCC that is created with LBIST-ready EDT and OCC IPs. You are now ready to create the parent-level LogicBIST controller.

Figure 12-18. LBIST-Ready EDT Child Block With OCC



Generating the LogicBIST Controller at the Parent Level With EDT and OCC

Once you have created your LogicBIST-ready EDT child blocks with OCC, you can then insert and connect the LogicBIST controller with the LBIST-ready blocks and the EDT at the parent level. The input parent netlist will contain the instantiation of the previously instantiated EDT child blocks.

Prerequisites

- MBIST inserted netlist from pass one.
- Independently generated ICL and netlist files, such as *Block1.icl* and *Block1.v*.

Procedure

1. Load the design and the child block:

```
set_context dft -rtl -design_id rtl2
read_design parentBlock -design_id rtl1
read_design Block1 -design_id rtl2 -view interface
```

2. Set the current design and design level:

```
set_current_design parentBlock
set_design_level physical_block
```

3. Define the parent-level clocks:

```
add_clocks 0 clk1 -period 10ns
add_clocks lbist_clock -period 10ns
```

4. Define the DFT signals at the parent level:

```
add_dft_signals ltest_en int_ltest_en ext_ltest_en int_mode ext_mode
add_dft_signals scan_en test_clock edt_update -source_node \
      { scan_en test_clock edt_update}
add_dft_signals edt_clock shift_capture_clock -create_from_other_signals

add_dft_signals observe_test_point_en control_test_point_en -create_with_tdr
add_dft_signals x_bounding_en -create_with_tdr
add_dft_signals mcp_bounding_en -create_with_tdr
add_dft_signals async_set_reset_static_disable
add_dft_signals async_set_reset_dynamic_disable -create_from_other_signals
```

5. Run DRC:

```
check_design_rules
```

6. Create the DftSpecification for the OCC, LBIST, and EDT:

```
set spec [create_dft_specification -sri_sib_list {occ lbist edt}]
```

7. Insert the LBIST controller together with EDT and OCC in the parent-level block:

```
read_config_data -in_wrapper $spec -from_string {
    Occ {
        ijtag_host_interface : Sib(occ);
        static_clock_control: external;
        Controller(clk1) {
            clock_intercept_node : clk1;
        }
    }
    EDT {
        ijtag_host_interface : Sib(edt);
        Controller(c3) {
            scan_chain_count : 80;
            input_channel_count : 2;
            output_channel_count : 2;
            longest_chain_range : 2, 80;
            LogicBistOptions {
                present : on;
                misr_input_ratio :1;
            }
        }
    }
    LogicBist {
        ijtag_host_interface : Sib(lbist);
        Controller(ctrl_lbist) {
            burn_in : on;
            Connections {
                shift_clock_src : lbist_clock;
            }
            ShiftCycles { max : 80;}
            CaptureCycles {max : 3;}
            PatternCount {max : 10000;}
            WarmupPatternCount {max : 255;}

            #Block1 is module name and block1 is instance name in the parent
            DesignInstance(block1) {}
        }
        NcpIndexDecoder {
            ncp(first) {
                cycle(0): Occ(clk1);
                cycle(1): Occ(clk1);
            }
            ncp(second) {
                cycle(0): Occ(clk1);
                cycle(1): block1/occ_in_core;
            }
        }
    }
}
```

8. Create the IP and insert the hardware described with the DftSpecification into the design, and update the ICL:

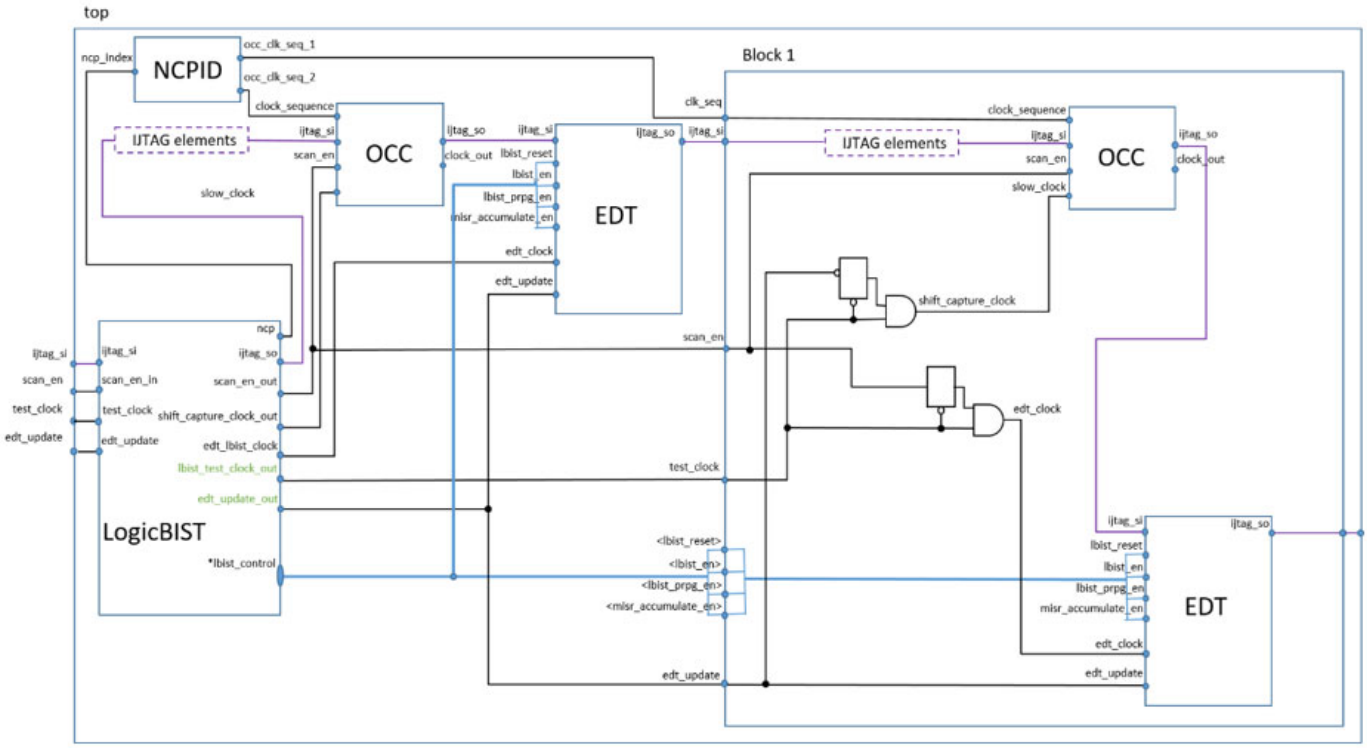
```
process_dft_specification
extract_icl
```

9. Exit the tool.

Results

You have generated the LogicBIST IP for the design at the parent level. Figure 12-19 shows the design after the LBIST controller has been inserted and connected with the LBIST-ready blocks and the EDT specified with the LogicBIST controller in the second pass.

Figure 12-19. LBIST Controller With OCC Inserted After Second Pass



Generating LogicBIST-Ready Grandchild Blocks with OCC

You can insert the LBIST-ready EDT with a separate OCC deep in the design hierarchy, for example, in grandchild blocks.

After inserting the EDT, you insert the LBIST controller. You then connect the controller with LBIST ready blocks (OCC and EDT).

Independently Inserting the LogicBIST-Ready EDT in Grandchild Blocks

You can insert the LBIST-ready EDT with a separate OCC in a grandchild block.

Prerequisites

- An RTL or gate-level netlist for the grandchild block. In a typical two-pass DFT insertion flow, this netlist is the output of the MemoryBIST inserted in pass one.

Procedure

1. Load the design for the second insertion pass of the current design level using RTL. For example:

   ```
   set_context dft -rtl -design_id rtl2
   ```

2. Load the cell library and the design from the first insertion pass:

   ```
   read_cell_library tessent.lib
   read_design Block1 -design_id rtl1
   ```

3. Set the current design and the design level:

   ```
   set_current_design Block1
   set_design_level sub_block
   ```

4. Add the standard DFT signals using the -source_node option:

   ```
   add_dft_signals  scan_en test_clock edt_update  –source_node \
      {scan_en test_clock edt_update}
   ```

5. Add the DFT signals needed to create the clock gaters that work with the LBIST controller, creating shift_capture_clock and edt_clock from other signals:

   ```
   add_dft_signals edt_clock -create_from_other_signals
   add_dft_signals shift_capture_clock -create_from_other_signals
   ```

6. Run DRC and go from setup mode to analysis mode:

```
check_design_rules
```

7. Create the DftSpecification for the EDT grandchild block with the OCC:

```
set spec [create_dft_specification -sri_sib_list {edt occ}]
```

8. Specify the LBIST-ready EDT and OCC for the grandchild block:

```
read_config_data -in_wrapper $spec -from_string {
    Occ {
        ijtag_host_interface : Sib(occ);
        static_clock_control: external;
        Controller(clk) {
            clock_intercept_node : clk;
            leaf_instance_name : occ_in_core;
            Connections {
                clock_sequence : clk_seq[%d];
            }
        }
    }


    EDT {
        ijtag_host_interface : Sib(edt);
        LogicBistOptions {
            present: on;
            ShiftCycles {
                max                 : 80;
                hardware_default  : 60;
            }
            WarmupPatternCount {
                max                 : 255;
            }
            // seed declaration
            prpg_reference_seed : 'h1234;
        }
        Controller(c1) {
            scan_chain_count       : 80;
            input_channel_count   : 2;
            output_channel_count : 2;
            longest_chain_range    : 2, 80;
        }
    }
}
```

9. Create the IP, insert the hardware described with the DftSpecification into the design, and update the ICL:

```
process_dft_specification
extract_icl
```

10. Exit the tool.

Results

You have generated a design netlist with the instantiation of the EDT logic and a separate RTL file.

Instrumenting the Child Block

You must ensure that connections for the clock_sequence bus are propagated to the boundary of the intermediate block (that is, the parent of the grandchild block). The DFT signals infrastructure handles the LogicBIST-related DFT signal.

Prerequisites

- An MBIST-inserted netlist from pass one.
- Independently generated ICL and netlist files, such as *Block1.icl* and *Block1.v*.

Procedure

1. Load the design and the child block:

   ```
   set_context dft -rtl -design_id rtl2
   read_design ChildBlock1 -design_id rtl1
   read_design GrandChildBlock1 -design_id rtl2
   ```

2. Set the current design and design level:

   ```
   set_current_design ChildBlock1
   set_design_level sub_block
   ```

3. Define the clocks at the parent level:

   ```
   add_clocks 0 clk1 -period 10ns
   add_clocks lbist_clock -period 10ns
   ```

4. Define the DFT signals at the parent level:

   ```
   add_dft_signals ltest_en int_ltest_en ext_ltest_en int_mode ext_mode
   add_dft_signals scan_en test_clock edt_update -source_node \
   { scan_en test_clock edt_update}
   ```

5. Run DRC:

   ```
   check_design_rules
   ```

6. Create the DftSpecification:

   ```
   set spec [create_dft_specification]
   ```

7. Propagate the clock_sequence connections using the DftSpecification:

```
read_config_data -in_wrapper $spec -replace -from_string {
 IjtagNetwork {
  DataInPorts {
   port_naming : clk_seq1[2:0] ;
   connection(2:0) : grand_child_block1/clk_seq[2:0]
  {
... // other IJTAG settings
 }
}
```

8. Create the IP and insert the hardware described with the DftSpecification into the design, and update the ICL:

```
process_dft_specification
extract_icl
```

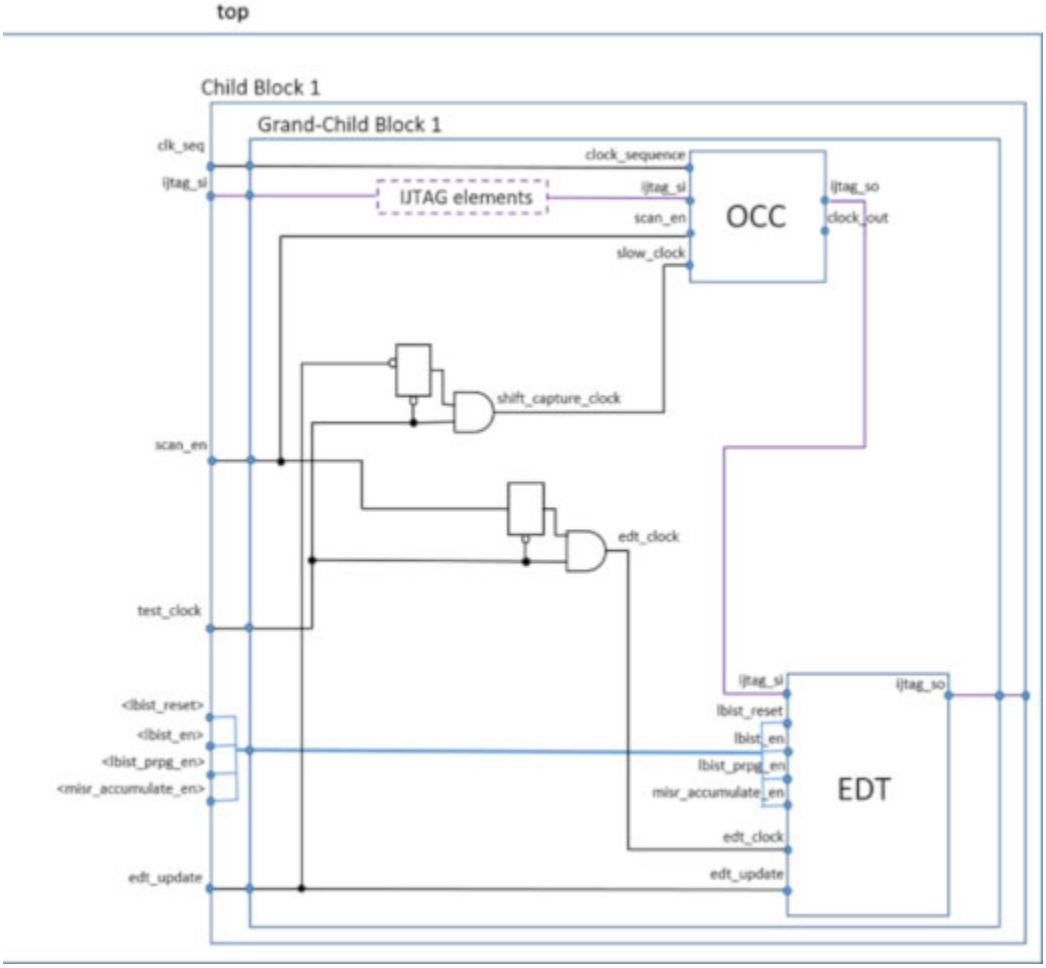
9. Exit the tool.

Results

You have propagated the clock_sequence and DFT signals to the intermediate block. Repeat this for every intermediate level.

Figure 12-20 shows an example of the LBIST-ready grandchild block with a separate OCC that is created with LBIST-ready EDT and OCC IPs, and the signal connections to the intermediate block. You are now ready to create the grandparent-level LogicBIST controller.

Figure 12-20. LBIST-Ready Grandchild Block with OCC



Generating the LogicBist Controller at the Grandparent Level With EDT and OCC

After creating the LogicBIST-ready EDT grandchild blocks with OCC and propagating the DFT signals to the intermediate level, you can then insert and connect the LogicBIST controller with the LBIST-ready blocks and the EDT at the grandparent level. The input grandparent netlist contains the previously instantiated EDT intermediate and grandchild blocks.

Prerequisites

- An MBIST-inserted netlist from pass one.
- Independently generated ICL and netlist files, such as *Block1.icl* and *Block1.v*.

Procedure

1. Load the design, the child block, and the grandchild block:

```
set_context dft -rtl -design_id rtl2
read_design parentblock -design_id rtl1
read_design GrandChildBlock1 -design_id rtl2 -view interface
read_design ChildBlock1 -design_id rtl2 -view interface
```

2. Set the current design and design level:

```
set_current_design parentblock
set_design_level physical_level
```

3. Define the clocks at the parent level:

```
add_clocks 0 clk1 -period 10ns
add_clocks lbist_clock -period 10ns
```

4. Define the DFT signals at the parent level:

```
add_dft_signals ltest_en int_ltest_en ext_ltest_en int_mode ext_mode
add_dft_signals scan_en test_clock edt_update -source_node \
{ scan_en test_clock edt_update}
 add_dft_signals edt_clock shift_capture_clock -create_from_other_signals
```

5. Run DRC:

```
check_design_rules
```

6. Create the DftSpecification for the OCC, LBIST, and EDT:

```
set spec [create_dft_specification -sri_sib_list {occ lbist edt}]
```

7. Insert the LBIST controller together with the EDT and OCC in the parent-level block:

```
Occ {
 ijtag_host_interface : Sib(occ);
 static_clock_control: external;
 Controller(clk1) {
    clock_intercept_node : clk1;
  }
 }
 EDT {
  ijtag_host_interface : Sib(edt);
  Controller(c3) {
     scan_chain_count : 80;
     input_channel_count : 2;
     output_channel_count : 2;
     longest_chain_range : 2, 80;
        LogicBistOptions {
          present : on;
          misr_input_ratio :1;
         }
    }
  }
  LogicBist {
    ijtag_host_interface : Sib(lbist);
    Controller(ctrl_lbist) {
       burn_in : on;
       Connections {
          shift_clock_src : lbist_clock;
   }
   ShiftCycles { max : 80;}
   CaptureCycles {max : 3;}
   PatternCount {max : 10000;}
   WarmupPatternCount {max : 255;}
   #ChildBlock1 is module name and child_block1 is instance name in the parent
   DesignInstance(child_block1) {}
  }
  NcpIndexDecoder {
     ncp(first) {
        cycle(0): Occ(clk1);
        cycle(1): Occ(clk1);
     }
     ncp(second) {
        cycle(0): Occ(clk1);
        cycle(1): child_block1/grand_child_block1/occ_in_core;
      }
    }
   }
}
```

8. Create the IP, insert the hardware described with the DftSpecification into the design, and update the ICL:

```
process_dft_specification
extract_icl
```

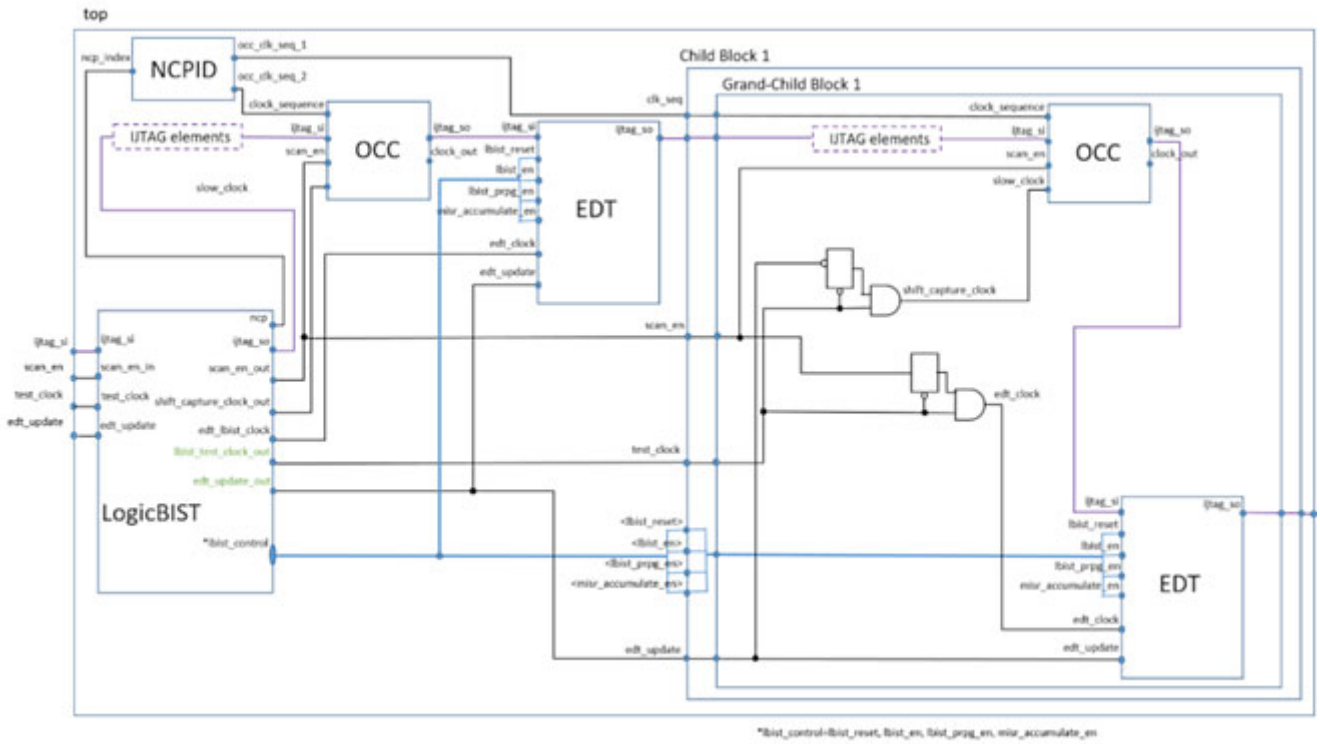
9. Exit the tool.

Results

You have generated a netlist for all three levels of the design with the instantiation of the EDT logic and a separate RTL file.

Figure 12-21 shows an example of the LBIST-ready grandparent, child (intermediate), and grandchild blocks with separate OCCs and LBIST-ready EDTs.

Figure 12-21. LBIST-Ready Grandparent, Intermediate, and Grandchild Block With OCC



SSN and Hybrid TK/LBIST Insertion Flow

The Streaming Scan Network (SSN) effectively distributes scan data for ATPG pattern by hosting EDT controllers with ScanHost nodes. This section describes how to prepare and insert a ScanHost that handles the LBIST-ready EDT controllers.

Independent Insertion With SSN Flow Overview

The SSN node, ScanHost (SSH), can be inserted between the LBIST-ready EDT controller and LogicBIST controller. The scan control signals generated by the SSH are intercepted by bypass multiplexers, which allow them to be driven by the LBIST controller. These signals include scan_en, edt_update, and sometimes edt_clock, shift_capture_clock, and test_clock.

Figure 12-23 shows the resulting block diagram after independent insertion with SSN has completed. The LBIST-ready SSH and EDT controller have been inserted in child Block. The LBIST controller, SSH, and another hybrid EDT controller have been inserted in the current physical block.

Figure 12-22. Independent Insertion With SSH

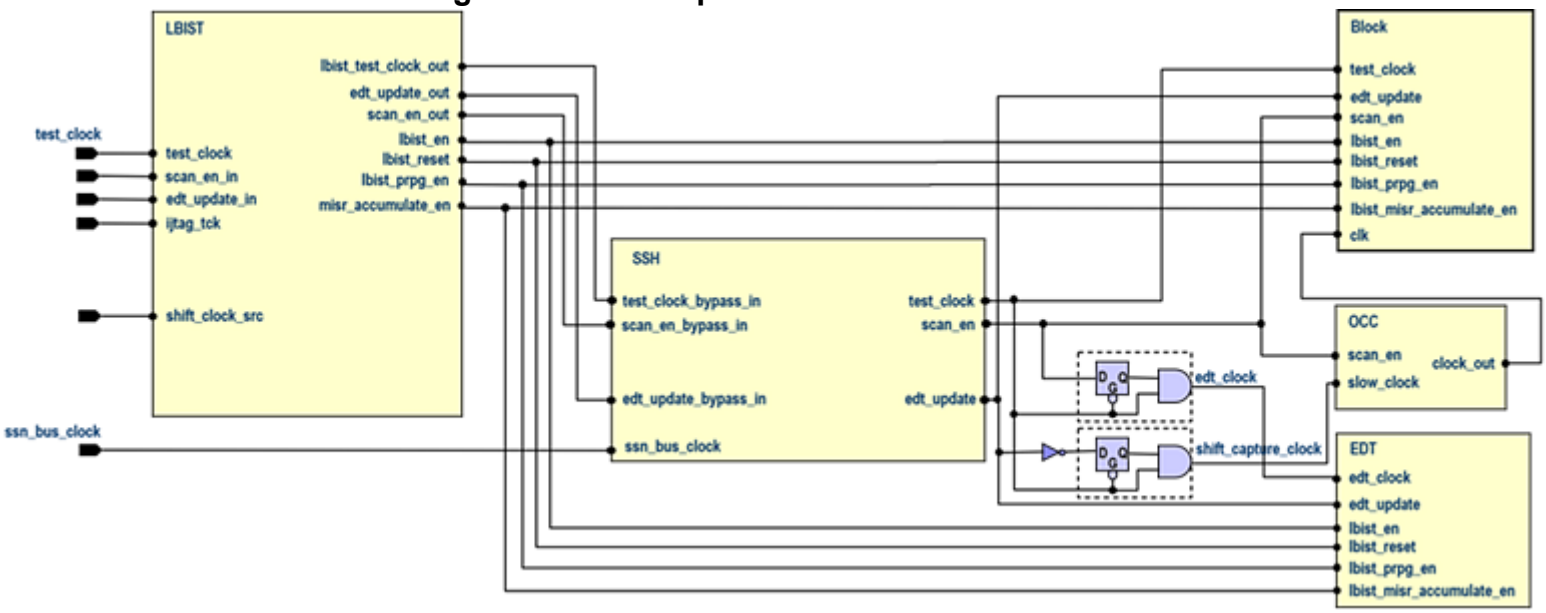
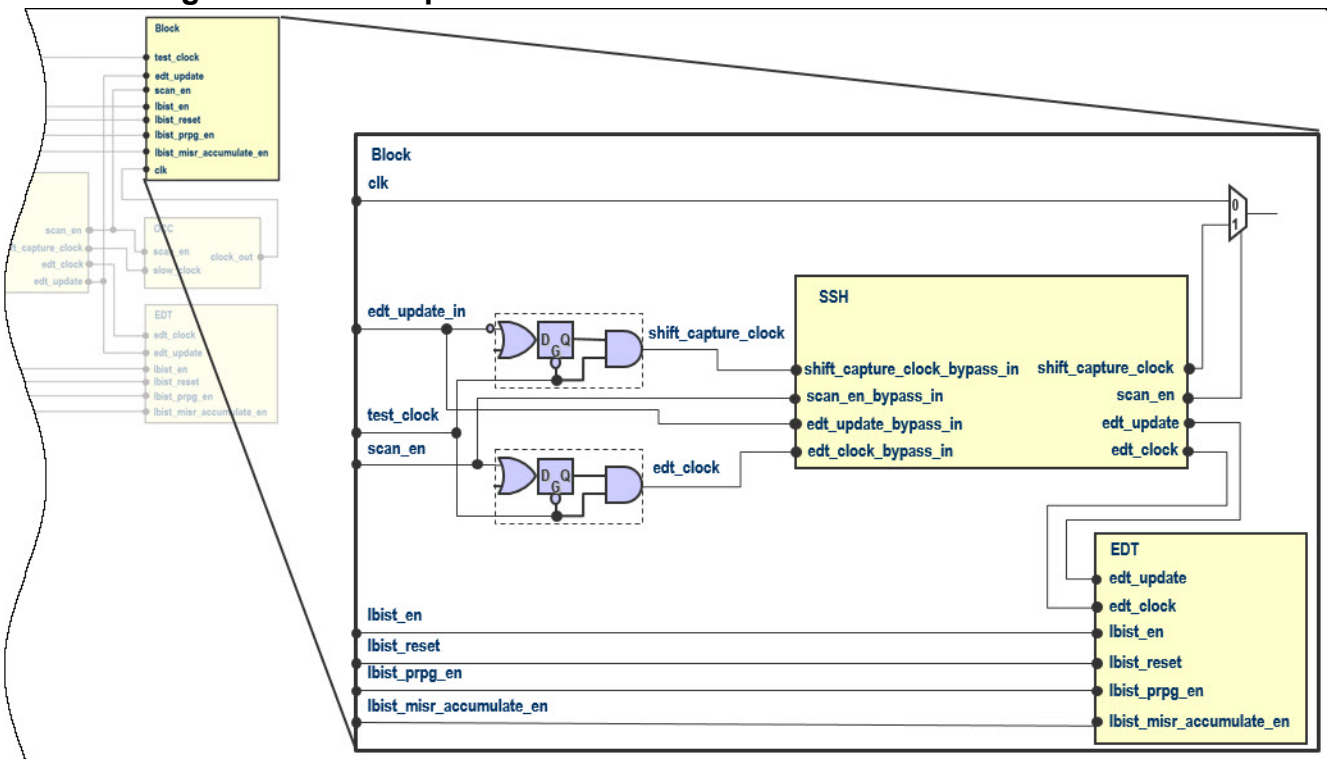


Figure 12-23 shows the contents of the child Block. Typically there is no OCC in an unwrapped Block. Use the DftSpecification/SSN/clock_sources_with_no_local_occs property to insert a clock multiplexer on the clock source. Specifying this property ensures the shift clock generated in the local SSH reach its scan cells. This also guarantees that the capture clock pulses generated in the external OCC reach the scan cells. See the SSN wrapper description in the *Tessent Shell Reference Manual* for more information.

Figure 12-23. Independent Insertion With SSH Child Block Contents

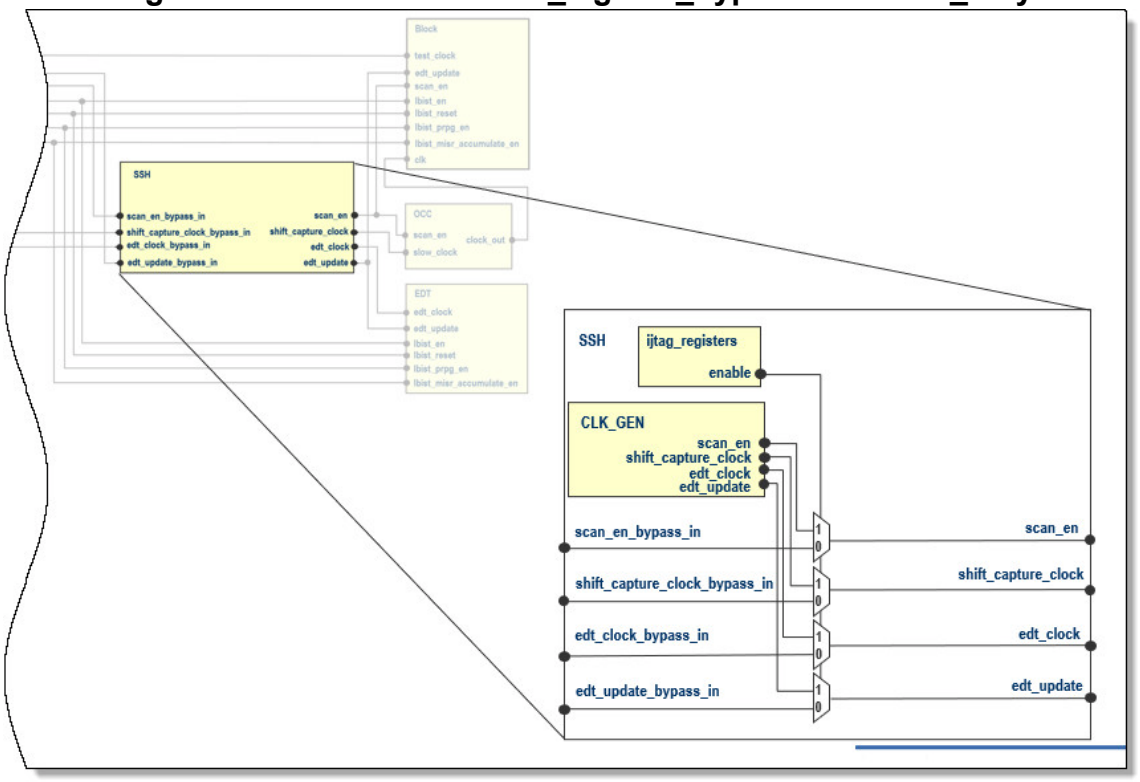


Generating SSN ScanHost IP for Independent Insertion

To prepare the SSN ScanHost (SSH) for integration the scan_signals_bypass property should be set to either controls_only or on. By default this property's auto setting resolves to controls_only when an LBIST-ready EDT or LogicBIST wrapper is present in the same DftSpecification.

Figure 12-24 shows the resulting SSH with scan_signals_bypass set to controls_only.

Figure 12-24. SSH With scan_signals_bypass: controls_only



Independently Inserting the LogicBIST-Ready EDT and SSH in a Child Block

The LogicBIST-ready EDT with SSN can be inserted independently from the LogicBIST controller.

Prerequisites

- RTL gate-level netlist for the child block
- Output generated by the MemoryBIST insertion pass

Procedure

1. Load the design for the second insertion pass of the current design level using RTL:

   ```
   set_context dft -rtl -design_id rtl2
   ```

2. Load the cell library and the design from the first insertion pass:

   ```
   read_cell_library tessent.lib
   read_design Block -design_id rtl1
   ```

3. Insert the EDT in the child block called Block:

   ```
   set_current_design Block
   set_design_level sub_block
   ```

4. Add the standard DFT signal using -source_node option:

   ```
   add_dft_signals  scan_en test_clock edt_update  –source_node {scan_en test_clock edt_update}
   ```

5. Add the DFT signals needed to create the clock gaters that work with the external LBIST controller by creating shift_capture_clock and edt_clock from other signals:

   ```
   add_dft_signals  edt_clock  –create_from_other_signals
   add_dft_signals  shift_capture_clock  –create_from_other_signals
   ```

6. Run DRC and go from setup mode to analysis mode:

   ```
   check_design_rules
   ```

7. Create the DftSpecification for the EDT child block:

   ```
   set spec [create_dft_specification -sri_sib_list {edt ssn}]
   ```

8. Specify the LBIST-ready EDT for the child block:

```
read_config_data -in_wrapper $spec -from_string {
   SSN {
      clock_sources_with_no_local_occs : clk;
      Datapath(1) {
         ijtag_host_interface : Sib(ssn);
         output_bus_width : 48;
         ScanHost(1) {
            ChainGroup (edt) {
            }
         }
      }
   }
   EDT {
      ijtag_host_interface : Sib(edt);
      LogicBistOptions {
         present: on;
         ShiftCycles {
            max                  : 80;
            hardware_default  : 60;
         }
      }
      Controller(c1) {
         scan_chain_count : 80;
         input_channel_count : 2;
         output_channel_count : 2;
         longest_chain_range : 2, 80;
      }
   }
}
```

9. Create the IP and insert the hardware described with the DftSpecification into the design, and update the IJTAG:

```
process_dft_specification
extract_icl
```

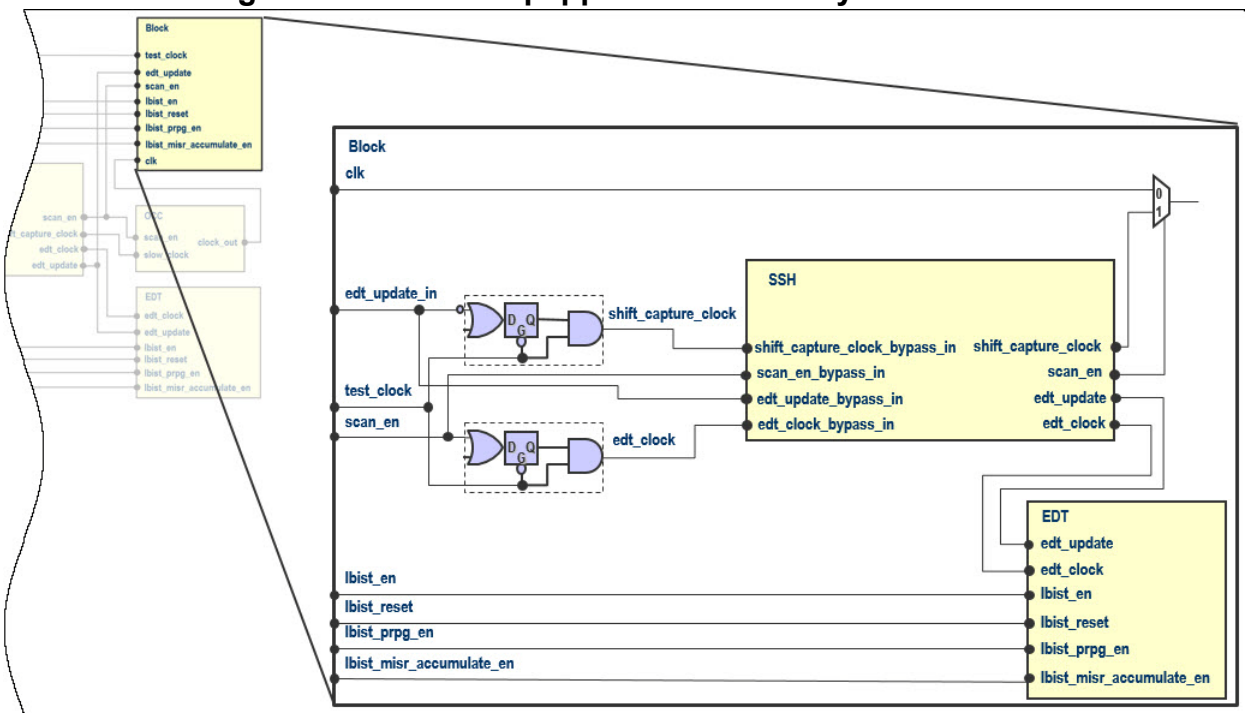
10. Exit the tool.

```
exit
```

Results

You have generated a design netlist with the instantiation of the EDT and SSH logic and a separate RTL file. Figure 12-25 shows the SSN-equipped LBIST-ready child block that is created in this example. This child block contains TK/LBIST EDT, SSN ScanHost, and all of the necessary ports with respective nets to enable control by the LBIST controller and SSN ATPG pattern generation. You are now ready to create the parent-level LogicBIST controller.

Figure 12-25. SSN-Equipped LBIST -Ready Child Block



Generating the LogicBIST, EDT, OCC, and SSH in the Parent Level

Once you have created your LogicBIST-ready child blocks with SSH, you can then independently insert and connect the LogicBIST controller with the LogicBIST-ready blocks and the EDT and SSH at the parent level.

Prerequisites

- MBIST inserted netlist from pass one.
- Scan setup dofile and scan setup test proc created by the scan insertion step.
- Independently generated ICL and netlist files, such as *Block.icl* and *Block.v*.

Procedure

1. Load the design and the child block:

   ```
   set_context dft -rtl -design_id rtl2
   read_design parentBlock -design_id rtl1
   read_design Block -design_id rtl2
   ```

2. Set the current design:

   ```
   set_current_design parentBlock
   set_design_level physical_block
   ```

3. Define the top-level clocks:

```
add_clocks 0 clk -period 10ns
add_clocks lbist_clock -period 10ns
```

4. Define the DFT signals at the top level:

```
add_dft_signals scan_en test_clock edt_update -source_node \
                { scan_en test_clock edt_update}
add_dft_signals edt_clock shift_capture_clock -create_from_other_signals
add_dft_signals ltest_en int_ltest_en ext_ltest_en int_mode ext_mode
add_dft_signals observe_test_point_en control_test_point_en
add_dft_signals x_bounding_en
add_dft_signals mcp_bounding_en
add_dft_signals async_set_reset_static_disable
add_dft_signals async_set_reset_dynamic_disable -create_from_other_signals
```

5. Run DRC:

```
check_design_rules
```

6. Create the DftSpecification for parent level the OCC, LBIST, SSN, and EDT:

```
set spec [create_dft_specification -sri_sib_list {occ lbist edt ssn}]
```

7. Insert the LBIST controller together with EDT and SSH in the parent-level block:

```
read_config_data -in_wrapper $spec -from_string {
    SSN {
        Datapath(1) {
            ijtag_host_interface : Sib(ssn);
            output_bus_width : 48;
            ScanHost(1) {
                ChainGroup (edt) {
                }
            }
            // Block is the module name and block1 is the instance name in the parent
            DesignInstance(block1) {
            }
        }
    }
    Occ {
        ijtag_host_interface : Sib(occ);
        static_clock_control: external;
        Controller(clk1) {
            clock_intercept_node : clk1;
        }
    }
     EDT {
        ijtag_host_interface : Sib(edt);
        Controller(c3) {
            scan_chain_count : 80;
            input_channel_count : 2;
            output_channel_count : 2;
            longest_chain_range : 2, 80;
            LogicBistOptions {
                present : on;
            }
        }
    }
    LogicBist {
        ijtag_host_interface : Sib(lbist);
        Controller(ctrl_lbist) {
            burn_in : on;
            Connections {
                shift_clock_src : lbist_clock;
            }
            ShiftCycles { max : 80;}
            CaptureCycles {max : 3;}
            PatternCount {max : 10000;}
            WarmupPatternCount {max : 255;}
            //Block is module name and block1 is instance name in the parent
            DesignInstance(block1) {}
        }
        NcpIndexDecoder {
            ncp(first) {
                cycle(0): Occ(clk1);
                cycle(1): Occ(clk1);
            }
            ncp(second) {
                cycle(0): Occ(clk1);
            }
        }
```

```
    }
}
```

8. Create the IP and insert the hardware described with the DftSpecification into the design, and extract ICL:

```
process_dft_specification
extract_icl
```

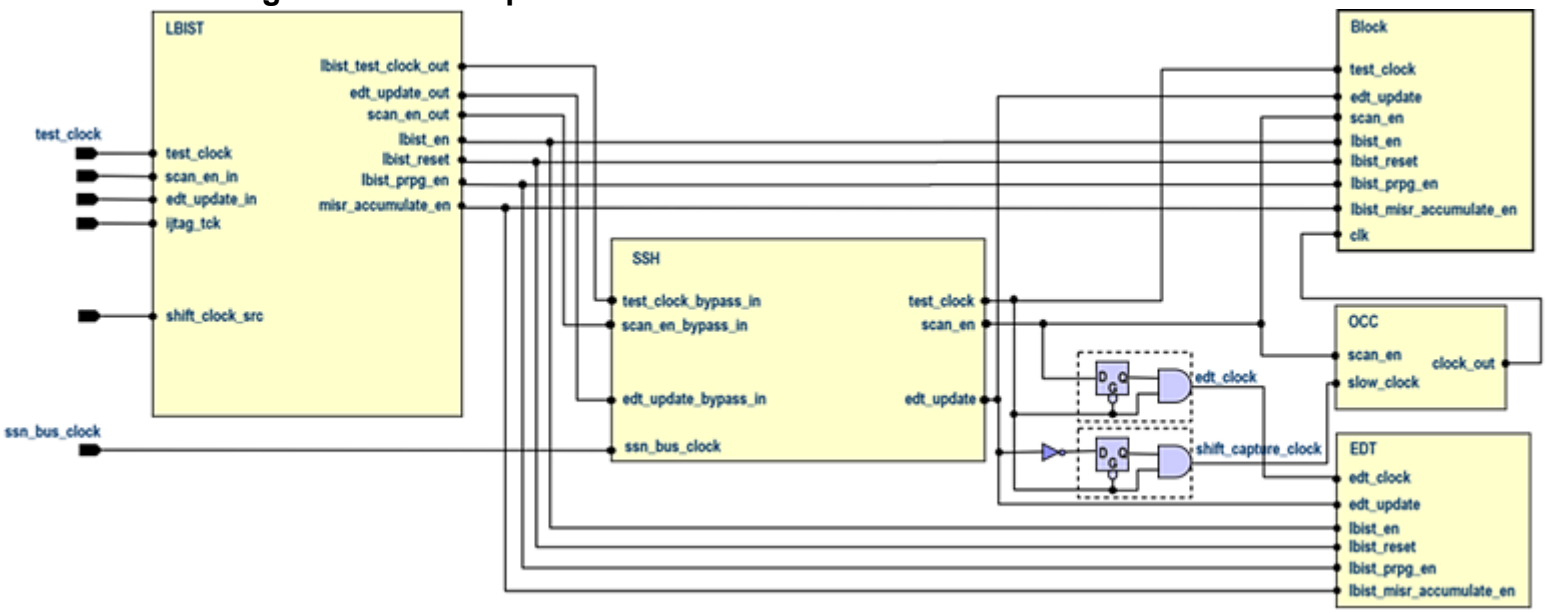
9. Exit the tool.

```
exit
```

Results

You have generated the LogicBIST IP for the design at the parent level. Figure 12-26shows the DFT IP inserted and connected. For simplicity, the SSN bus is not shown. Note in this example, OCC is only at the parent level.

Figure 12-26. Independent Insertion With SSN and OCC at Parent Level



Using ssn_bus_clock as test_clock Bypass

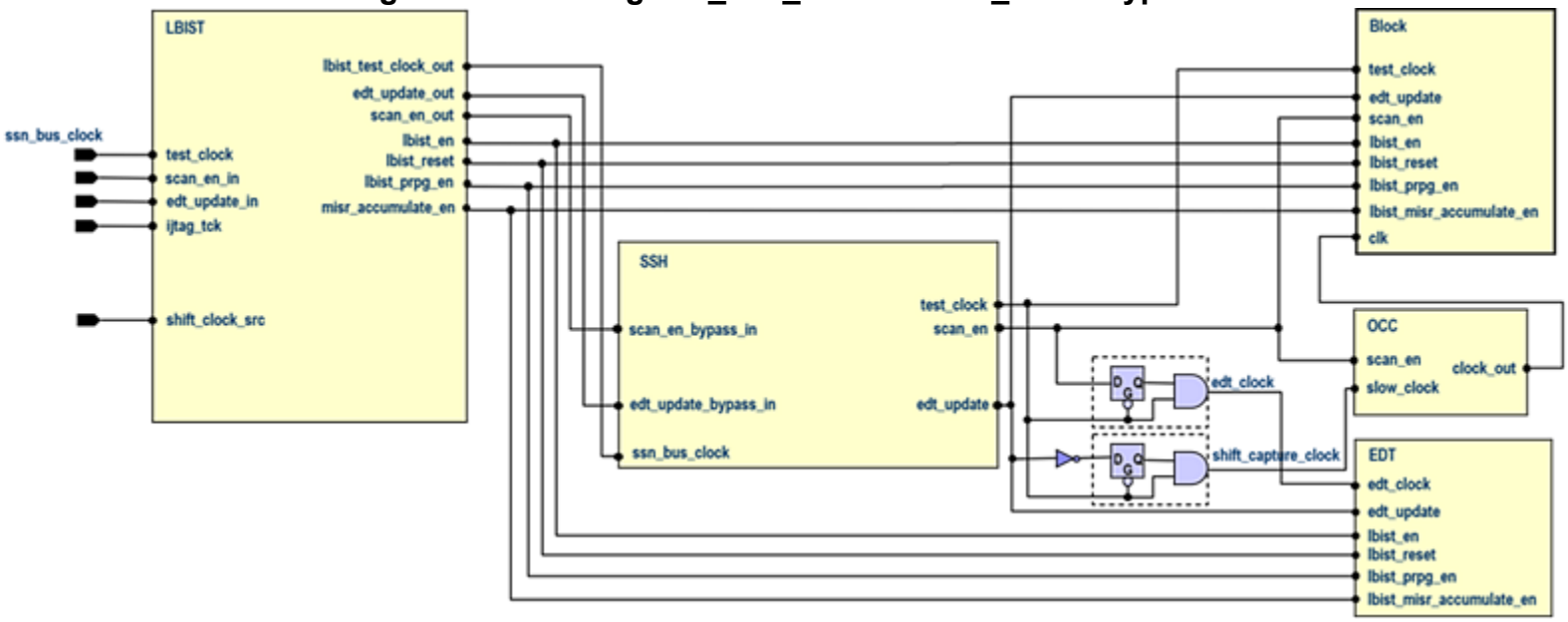
You can integrate the LogicBIST controller with SSH using ssn_bus_clock as test_clock bypass.

If you want to use ssn_bus_clock as a test_clock bypass, the LogicBIST controller must intercept ssn_bus_clock and use it as test_clock for LogicBist purposes as well. To generate the correct connections and a LogicBIST controller that supports this configuration, use the procedure from "Generating the LogicBIST, EDT, OCC, and SSH in the Parent Level" on page 245 with ssn_bus_clock specified to be the source of the test_clock DFT signal and the SSH property set to on:

```
add_dft_signals test_clock -source_node {ssn_bus_clock}
```

```
ScanHost(1) {
  use_ssn_bus_clock_as_test_clock_bypass : on;
  use_clock_shaper_cell : on;
}
```

Figure 12-27. Using ssn_bus_clock as test_clock Bypass



Top-Level LBIST and External Test Mode in Child Cores

The tool can use child-level physical block wrapper chains for LBIST in the parent physical region.

Child-Level OCC Inactive During External Test

You can provide the functional clock from a primary input or a PLL at the parent level to child physical blocks that do not have embedded PLLs.

See Figure 12-28 for an example of core-level physical blocks without embedded PLLs.

Figure 12-28. Top-Level Functional Clock

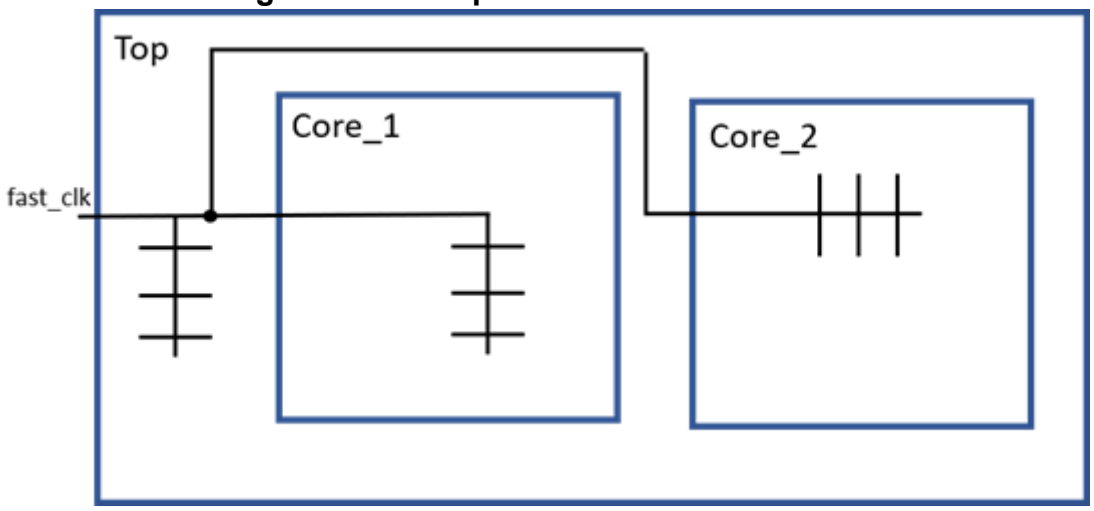
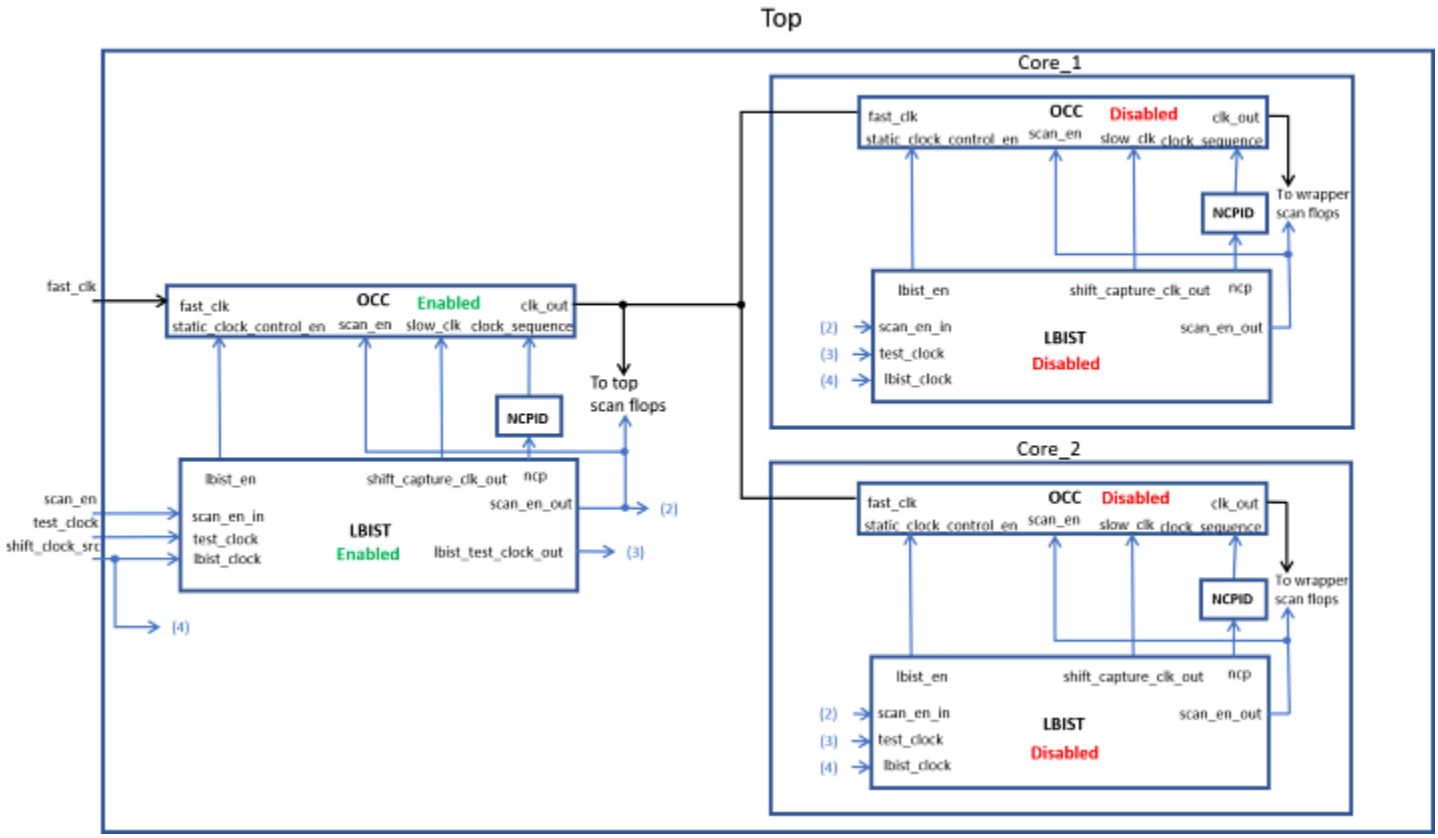


Figure 12-29 shows such a design after IP insertion. The core-level OCCs are disabled, and the wrapper chains are driven by the top-level OCC. The core-level OCCs are transparent, meaning that the top-level LBIST can check the external mode of the child cores using the top-level OCC.

Figure 12-29. Core-Level Chains Driven by Top-Level OCC

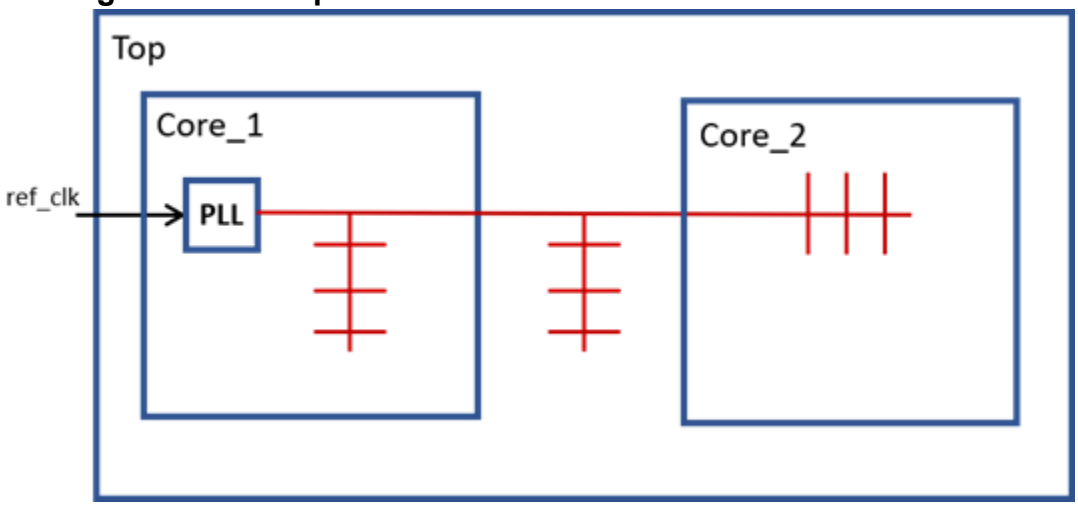


Child-Level OCC Active During External Test

You can provide a reference clock from a primary input at the parent level to a child physical block and generate the functional clock within the child block. This functional clock is provided to other child blocks as well as back to the parent level.

See Figure 12-30 for an example of core-level physical blocks with a PLL-generated functional clock.

Figure 12-30. Top-Level Reference Clock and Core-Level PLL



The embedded PLL in the Core_1 module generates the clock for the design. To generate the LBIST and NCPID instruments in the Core_1 module and reuse the internal OCC driving the wrapper chains for internal-mode LBIST, add the ext_lbist_en DFT signals:

```
add_dft_signals ext_lbist_en -create_with_tdr
```

The tool uses this command to generate the appropriate hardware for the LBIST controller and the NCP index decoder.

Next, create the LogicBist DftSpecification for Core_1:

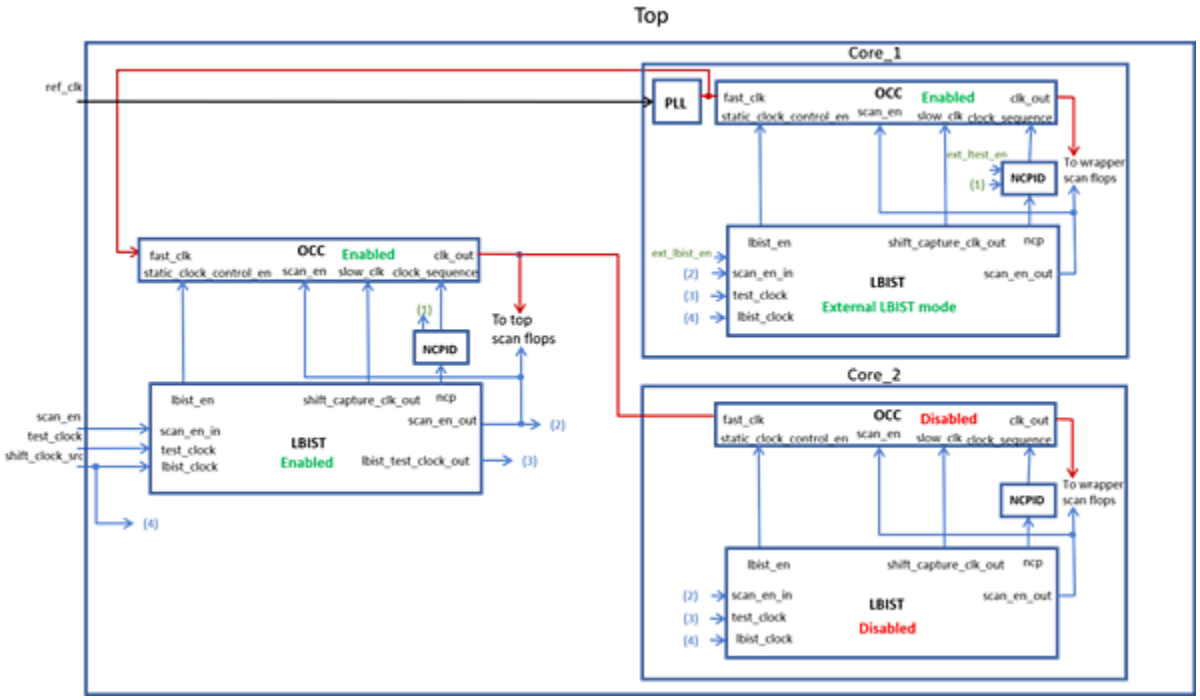
```
OCC {
  ijtag_host_interface : Sib(occ);
  static_clock_control : external;
  shift_only_mode : on;
  Controller(clk_controller) {
    clock_intercept_node : PLL/out;
  }
}

LogicBist {
  ijtag_host_interface : Sib(lbist) ;
  Controller(lbist_1) {
    // extest_lbist : auto; -> resolves to 'on' due to DftSignal
    // ext_lbist_en
    ShiftCycles { max : 100 ; }
    CaptureCycles { max : 10 ; }
    PatternCount { max : 16384; }
    WarmupPatternCount { max: 31; }
    Connections {
      tck              : ijtag_tck ; // default: tck
      shift_clock_src  : lbist_clock;
      ext_lbist_en : DftSignalOrTiedLow(ext_lbist_en); // -> default
    }
    Interface {
      ext_lbist_en : my_ext_lbist_en;
    }
  }
  NcpIndexDecoder {
    // extest_lbist : auto; -> resolves to 'on' due to DftSignal
    // ext_lbist_en
    Connections {
      ext_ltest_en : DftSignalOrTiedLow(ext_ltest_en) ;
      ExtestClockSequence {
        Occ(clk_controller) : ext_occ%d_clock_sequence[%d] ;
              // default option which ensures that the corresponding
              // port will be created at the boundary of the Core_1
        }
      }
      Interface {
        ext_clock_sequence : my_ext_occ%d_clock_sequence ;
      }
      Ncp(pulse_x1a) {
        cycle(0) : Occ(clk_controller);
      }
      Ncp(pulse_x2) {
        cycle(0) : Occ(clk_controller);
        cycle(1) : Occ(clk_controller);
      }
    }
  }
}
```

The tool creates the instrumentation of Core_2 without this functionality. Add the Core_1 OCC at the parent level to keep it active during top-level LBIST and for use in the DftSpecification of the NCP Index Controller:

```
add_core_instances -instances \
  Core_1/core_1_dft_tessent_occ_clk_controller_inst

LogicBist {
  ijtag_host_interface : Sib(lbist) ;
  Controller(lbist_1) {
  ...
  }
  NcpIndexDecoder {
    Ncp(pulse_x1a) {
      cycle(0) : Occ(clk_controller);
      cycle(1) : Core_1/core_1_dft_tessent_occ_clk_controller_inst;
    }
    Ncp(pulse_x2) {
      cycle(0) : Occ(clk_controller), \
          Core_1/core_1_dft_tessent_occ_clk_controller_inst;
      cycle(1) : Occ(clk_controller), \
          Core_1/core_1_dft_tessent_occ_clk_controller_inst;
    }
  }
}
OCC {
  ijtag_host_interface : Sib(occ);
  static_clock_control : external;
  Controller(clk_controller) {
    clock_intercept_node : Core_1/clk_out;
  }
}
```

The following figure contains the completed instrumentation of the design.

Figure 12-31. Core-Level Chains Driven by Core-Level OCC



If Core_1 has more than one embedded OCC, but it does not drive a wrapper, you can keep it inactive during top-level LBIST and active in core-level LBIST by not propagating the ext_clock_sequence port to the block's boundary. Do this by skipping that entry in the ExtestClockSequence wrapper. The tool optimizes the corresponding hardware of the NCPID so that the ext_clock_sequence path is not created for this OCC.

In this example, the second OCC is a mini-OCC from Sib(sti) and is named core_1_mbist_tessent_sib_sti_inst:

```
NcpIndexDecoder {
  Connections {
    ExtestClockSequence {
      Occ(clk_controller) : ext_occ%d_clock_sequence[%d] ;
      // Do not specify the entry for this OCC
      // (optimal hardware is generated)
    }
  }
  Ncp(pulse_x1a) {
    cycle(0) : Occ(clk_controller);
  }
  Ncp(pulse_x1b) {
    cycle(0) : core_1_mbist_tessent_sib_sti_inst;
  }
  Ncp(pulse_x2) {
    cycle(0) : Occ(clk_controller);
    cycle(1) : Occ(clk_controller);
  }
}
```

Child-Level Hybrid EDT For Wrapper Chains Active During External Test

You can control embedded hybrid EDT blocks with parent-level LBIST controllers or with LBIST controllers in both the core and at the parent level. The choice between these two scenarios depends on the implementation of the scan chain architecture.

If the wrapper chains are connected to the EDT that handles the internal mode of the core, the separate hybrid EDT for the external mode can only be controlled by the parent-level LBIST as shown in Figure 12-32.

Figure 12-32. Hybrid EDT for the External Mode Controlled by Parent-Level LBIST

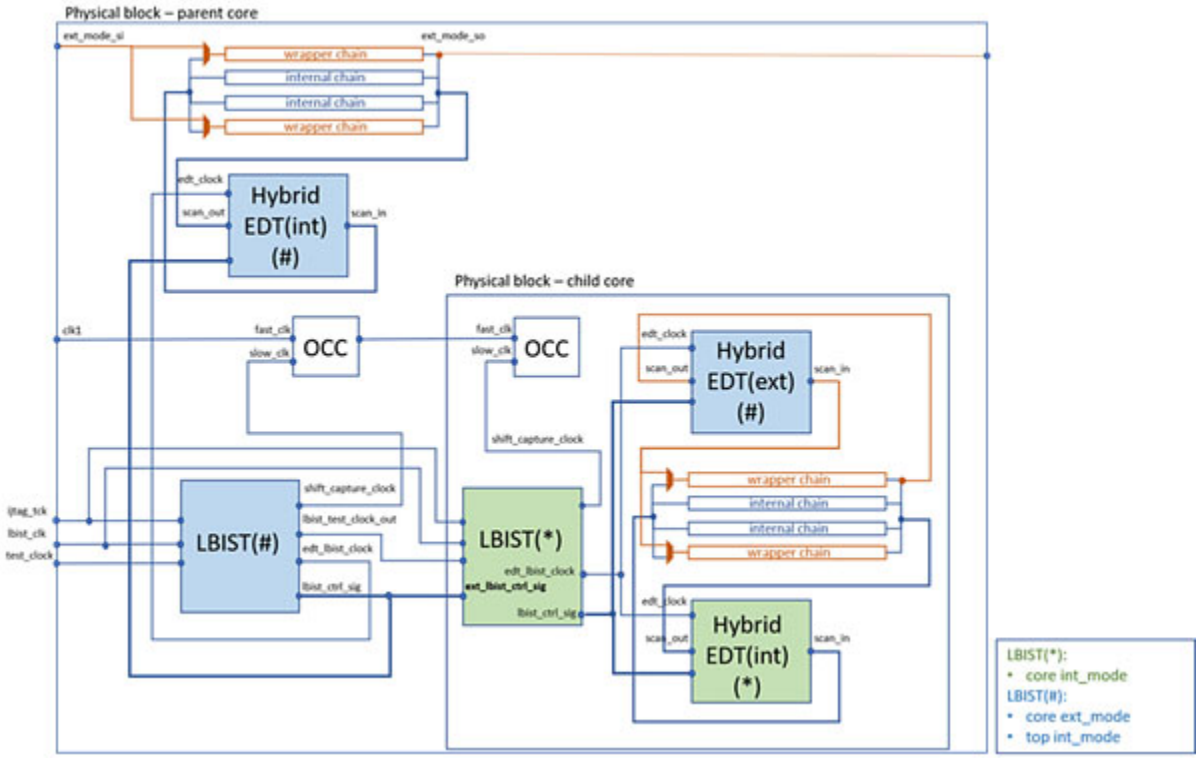
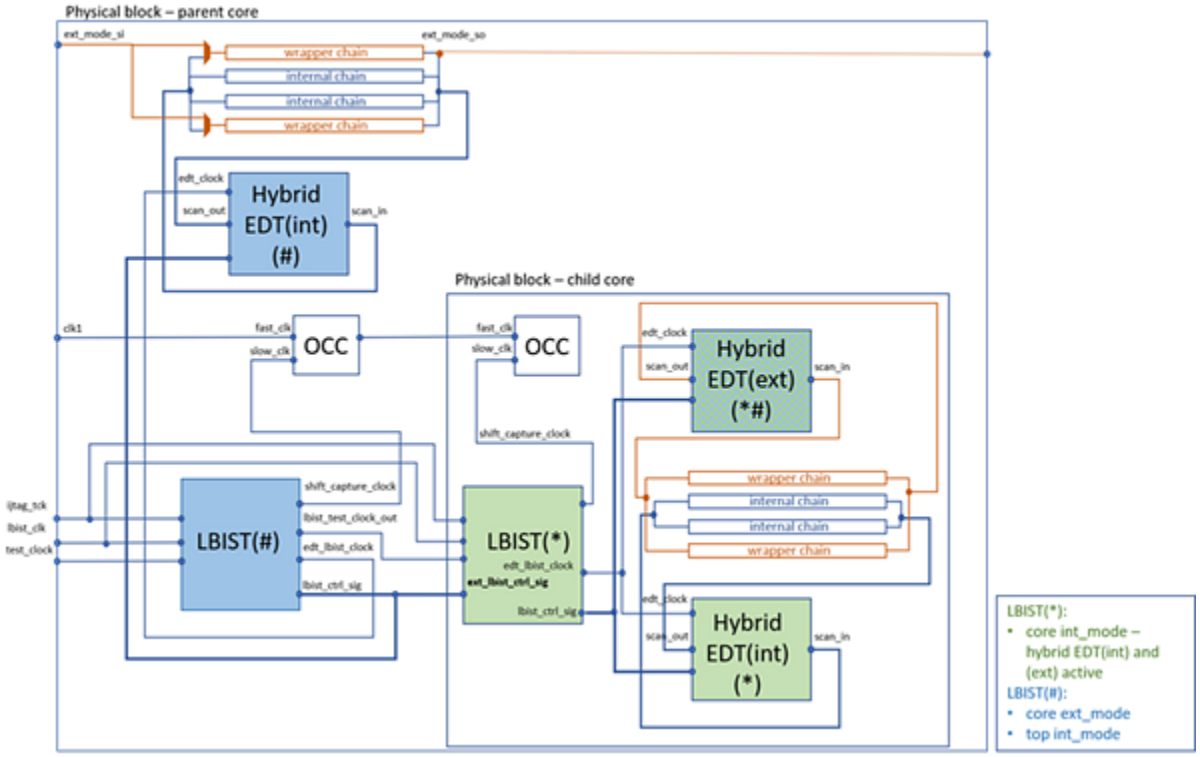


Figure 12-33 shows that the wrapper chains of the core are connected only with the EDT that handles external mode, and therefore the EDT must be controlled by LBIST controllers from both the parent and the child core. The required DFT signals in each case are passed through the child-level LBIST to the child-level hybrid EDT.

Figure 12-33. Hybrid EDT for Wrapper Chains Shared Between Core-Level and Parent-Level LBIST

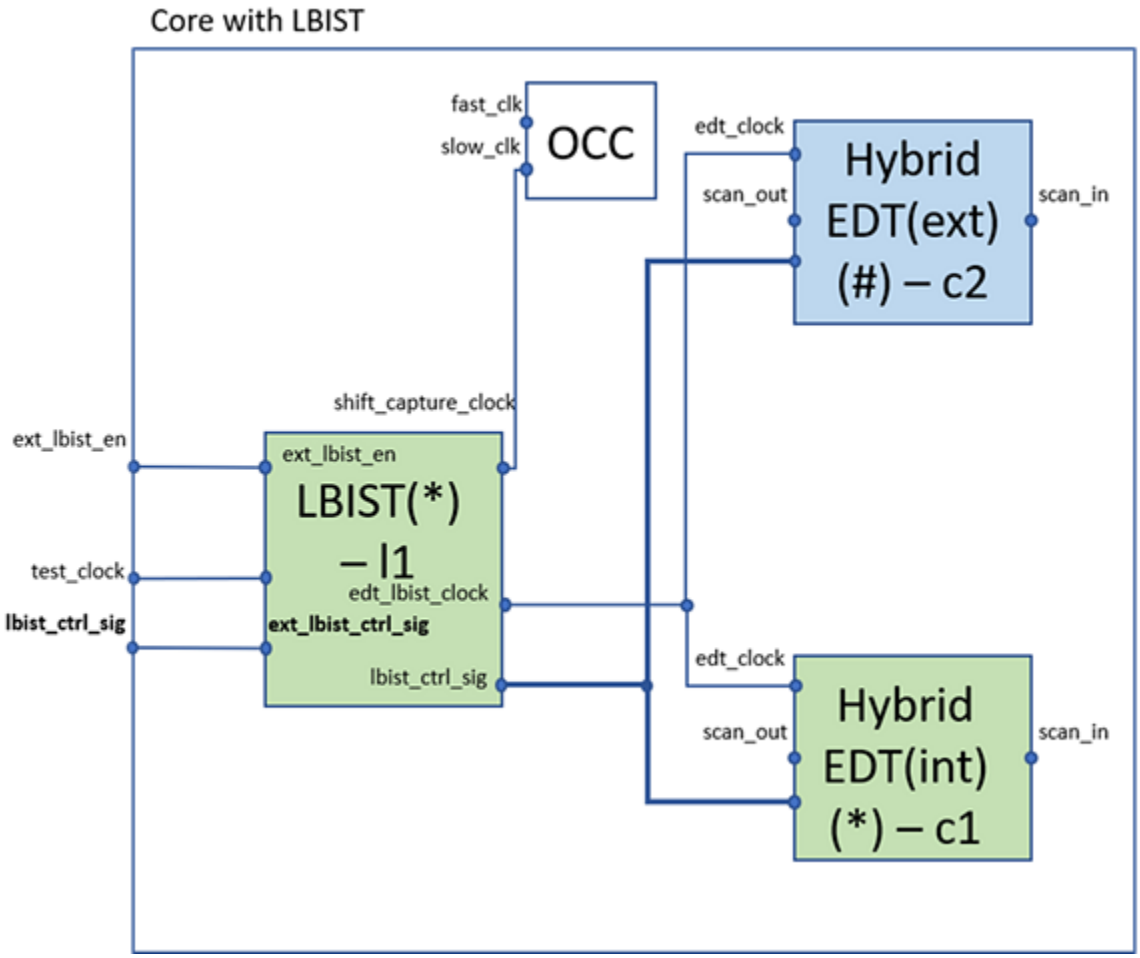


Example

You can create the DFT structure shown in Figure 12-34 in a single insertion pass by specifying the associated_edt and ext_mode_edt_present properties for the LogicBist instrument indicating which EDT should be controlled by the core-level LBIST and letting the tool auto-infer the ext_lbist_en DFT signal:

```
Edt {
  Controller(c1) {
  }
  Controller(c2) {
  }
}
LogicBist {
  extest_lbist : on; // This option set to on will infer
                     // ext_lbist_en DFT signal on port
  ext_mode_edt_present : on;
  Controller(l1) {
      associated_edt : Edt(c1) ;
  }
}
```

Figure 12-34. Single-Pass EDT



Alternately, if you use the EDT controllers' mode_enable properties to indicate the scan modes for which they operate, you can leave the LBISTs' associated_edt and ext_mode_edt_present properties unspecified:

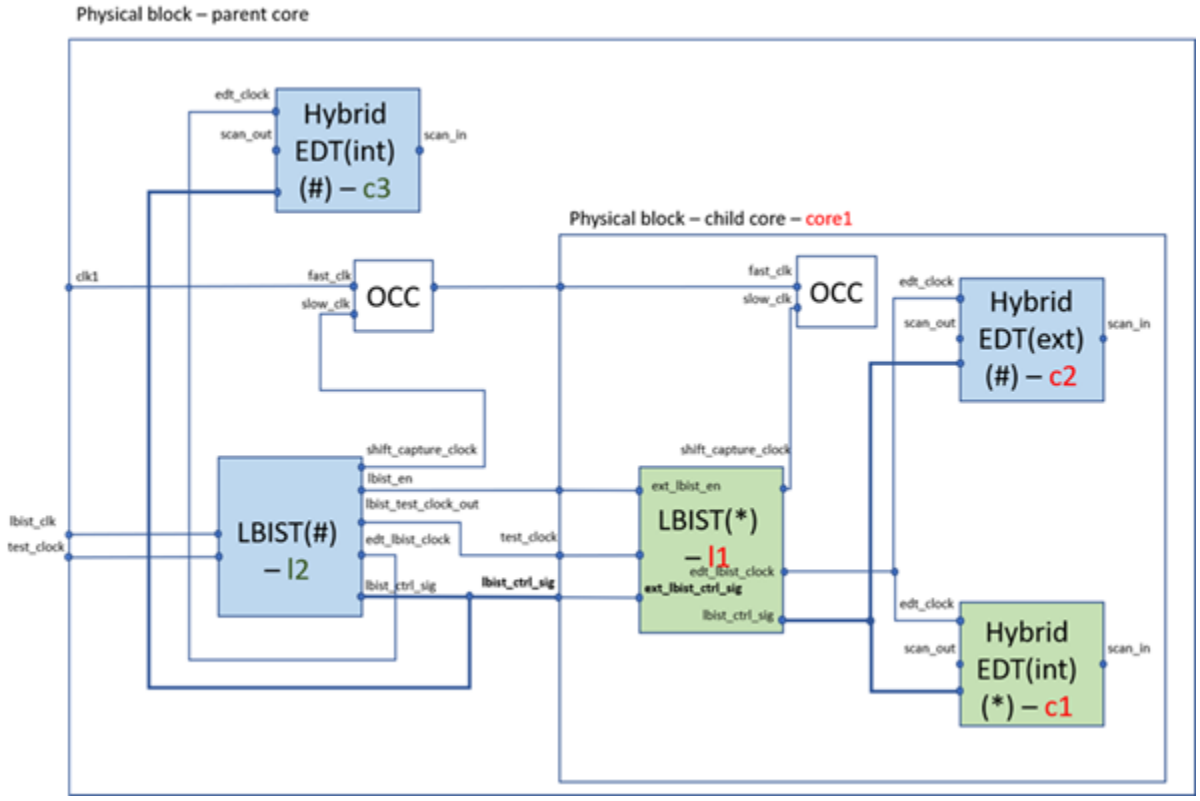
```
Edt {
  Controller(c1) {
    Connections {
      mode_enables : DftSignal(int_mode);
    }
  }
  Controller(c2) {
    Connections {
      mode_enables : DftSignal(ext_mode);
    }
  }
}
LogicBist {
  extest_lbist : on;
  // ext_mode_edt_present : on; // can be unspecified because mode_enables
                                // indicates EDT mode
  Controller(l1) {
      // associated_edt : Edt(c1); // can be unspecified because
                                   // mode_enables indicates EDT mode
  }
}
```

Next, at the parent design level you can reuse the embedded hybrid EDT to drive the core's wrapper chains for parent-level LBIST. Ensure that all EDTs from the core are properly associated with the parent-level LBIST controller. If no EDTs in the design (both in the child core and at the current design level) have modes assigned by mode_enables property, use the following DftSpecification:

```
Edt {
  Controller(c3) {
  }
}
LogicBist {
  Controller(l2) {
    DesignInstance(core1) {
      associated_edt : c2;
    }
  }
}
```

The EDT controller c3 is automatically associated with LBIST l2 (shown in Figure 12-35) because by default, all of the hybrid EDTs from the same insertion pass are automatically assumed to be controlled by the LBIST controller.

Figure 12-35. EDT and LBIST Association



Note

It is important to ensure that the core instance of EDT c2 is loaded in the LBIST fault simulation script.

Limitations of the Independent Insertion Flow

The integration of LBIST and SSN has certain limitations you must take into account.

- The use of the ThirdPartyOCC TCD is not fully supported in the SSN-ATPG mode of operation. Carefully examine SSN capabilities in terms of clock and enable signal generation so that both LBIST and SSN-ATPG modes of operation work properly. Currently, SSN can generate ShiftCaptureClock and ScanEn signals; in spite of that, the ShiftClock, ShiftClockEn, and CaptureEn signals for ThirdPartyOCC are automatically sourced by respectively created ports of the LBIST controller, without being intercepted by an SSN ScanHost.
- Inserting a LogicBIST controller in the presence of SSN when the edt_clock and shift_capture_clock signals are specified with the -source_nodes option is not supported.
- Inserting a LogicBIST controller without specifying test_clock and scan_en DFT signals specified with the -source_nodes option is not supported.

- When implementing the Controller Chain Mode (CCM) of LogicBIST and Hybrid EDT controllers, you should consider the impact of SSH. A CCM chain at a given level should be an uncompressed chain and not driven from local SSH. When generating CCM patterns, you should disable SSH by setting set_ssn_options off.

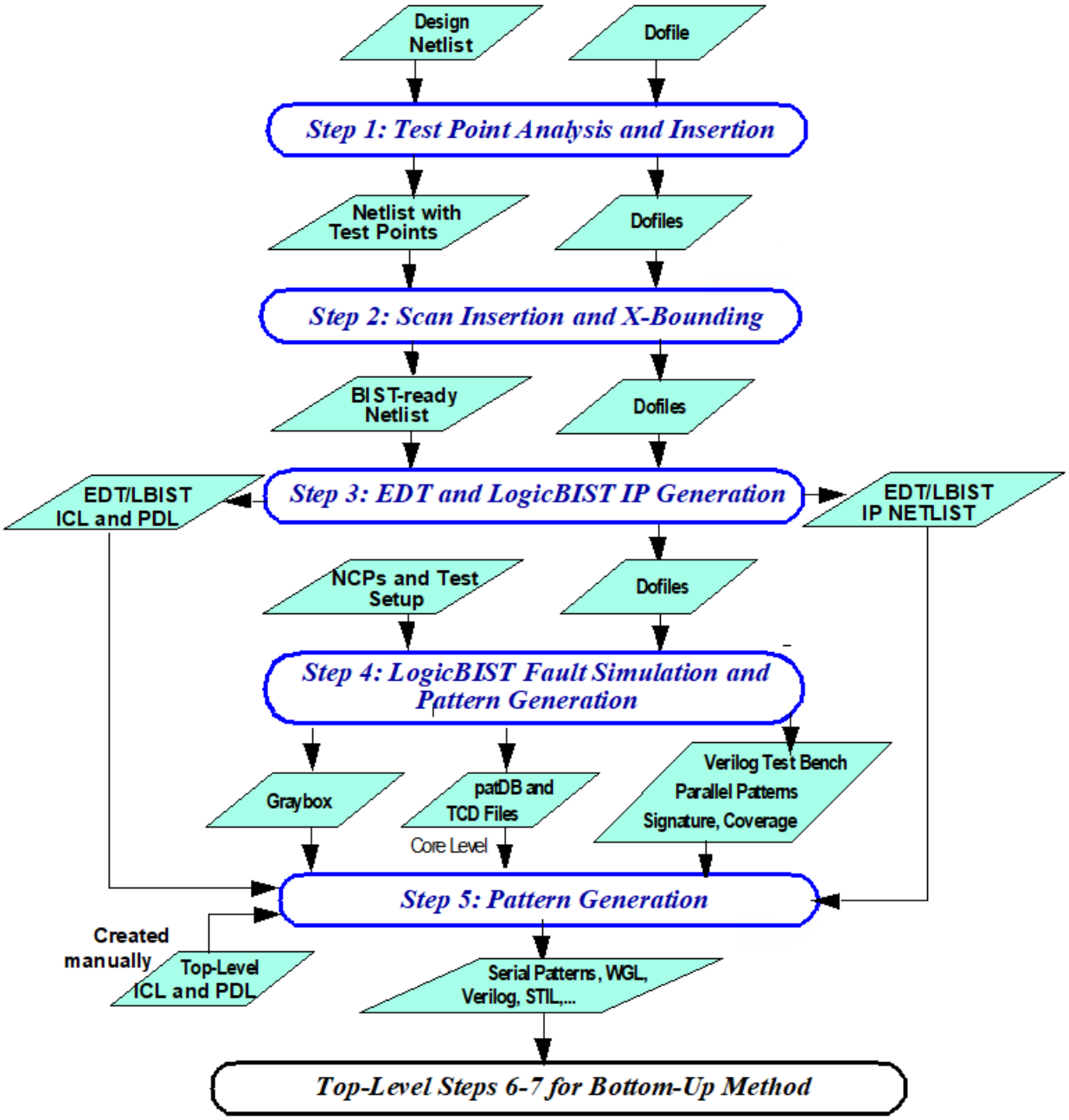
Appendix A
The Dofile Flow

The hybrid TK/LBIST flow supports a dofile flow that does not use the TSDB. In the dofile flow, the tool does not write output files to the TSDB. The five steps used for the TSDB bottom up method remain the same, as do the two additional steps used for the top-down method.

The following figure shows the steps you perform on each of your cores with the dofile flow. Refer to "Hybrid TK/LBIST Implementation" for details. For the dofile flow, the differences are:

- In Step 3, the tool does not store the output EDT, LogicBIST ICL and PDL files, or EDT/LogicBIST IP netlist in the TSDB. These files are direct inputs to pattern generation (Step 5).
- In Step 4, the output graybox, patternDB file, TCD file, testbenches, patterns, and so on are direct inputs to pattern generation (Step 5). The tool does not store these in the TSDB.
- During Step 3, logic synthesis is not fully automated. To synthesize the EDT/Tessent LogicBIST blocks and the common LogicBIST controller, you must use the synthesis script that the tool generates. You use the synthesized gate-level netlist output from your synthesis tool (for example, the output of the logic synthesis step with Synopsys Design Compiler®) as the input to the next step of the flow.

Figure A-1. The Dofile Flow



EDT and LogicBIST IP Generation Command Summary

Tessent Shell provides a numbers of commands for performing EDT and LogicBIST IP generation.

Table A-1. EDT and LogicBIST IP Generation Commands

Command	Description
read_cell_library	Loads one or more cell libraries into the tool.
read_verilog	Reads one or more Verilog files into the specified or default logical library.
report_clock_controller_pins	Reports the clock controller pins.
report_lbist_configuration	Reports the global LogicBIST controller configuration parameters.
report_lbist_pins	Reports the pins specified using the set_lbist_pins command.
set_clock_controller_pins	Specifies the connection information for the clock controller pins.
set_context	Specifies the current usage context of Tessent Shell. You must set the context before you can enter any other commands in Tessent Shell.
set_current_design	Specifies the top level of the design for all subsequent commands until reset by another execution of this command.
set_edt_options	Sets options for EDT IP creation.
set_lbist_controller_options	Specifies global options to configure the LogicBIST controller.
set_lbist_instances	Specifies the instance in which the LogicBIST controller or single chain mode logic is placed.

Table A-1. EDT and LogicBIST IP Generation Commands (cont.)

Command	Description
set_lbist_pins	Specifies the connection information for LogicBIST controller pins.
set_lbist_power_controller_options	Specifies creating the low-power shift controller for LogicBIST.
set_system_mode	Specifies the operational state you want the tool to enter.
set_tsdb_output_directory	Enables changing the TSDB directory name where the generated files and modified netlists are stored.
write_design	Writes out the modified netlist.
write_edt_files	Creates the files that implement EDT logic.

Generating the EDT and LogicBIST IP (Dofile Flow)

When generating the hybrid EDT and LogicBIST IP, you read in a TCD and bind it to a core instance in the design.

You can define dual compression configurations for the hybrid IP. See "Dual Compression Configurations for the Hybrid IP" on page 66 for an example.

Prerequisites

- You need the scan insertion dofile and BIST-ready netlist you created during Test Point Analysis and Insertion, Scan Insertion, and X-Bounding.

Procedure

1. From a shell, invoke Tessent Shell:

   ```
   % tessent -shell
   ```

   After invocation, the tool is in unspecified setup mode. You must set the context before you use the IP generation commands.

2. Set the tool context to EDT/LogicBIST generation using the set_context command as follows:

   SETUP> set_context dft -edt -logic_bist -no_rtl

3. Set the TSDB location if necessary. For example:

   SETUP> set_tsdb_output_directory top_level.tsdb

4. Load the LogicBIST-ready design netlist using the read_verilog command. For example:

   SETUP> read_verilog my_modified_design.v

5. Load one or more cell libraries into the tool using the read_cell_library command:

   SETUP> read_cell_library atpg.lib

6. Set the top design using the set_current_design command. For example:

   SETUP> set_current_design top_module

7. Extract the ICL using the extract_icl command. For example:

   SETUP> extract_icl

8. Read in the *design_name*_tessent_occ.tcd file from the TSDB directory using the read_core_descriptions command. For example:

   SETUP> read_core_descriptions ./top_level.tsdb/instruments/
   my_design_occ.instrument/my_design_tessent_occ.tcd

9. Bind the core description in memory with the specified core instance in the design using the add_core_instances command. For example:

   SETUP> add_core_instances -instance block_inst1/my_occ

   The add_core_instances command also provides a method of specifying the core instance parameters. See "Core Instance Parameters" in the *Tessent TestKompress User's Manual* for more information.

10. Read in the dofile you generated with Tessent Shell that contains the scan insertion and X-bounding information using the dofile command. See "Test Point Analysis and Insertion, Scan Insertion, and X-Bounding" on page 77 for more information. For example:

    SETUP> dofile my_setup.dofile

11. Set additional EDT/LogicBIST IP generation options depending on your design objective using the following commands:

    - set_clock_controller_pins
    - set_edt_options
    - set_lbist_controller_options
    - set_lbist_instances
    - set_lbist_pins
    - set_lbist_power_controller_options
    - set_lpct_controller

See also "Low-Power Shift" on page 35, "Controller Chain Mode" on page 48, and "Low Pin Count Test Controller" on page 325.

12. Change the tool's system mode to analysis using the set_system_mode command as follows:

    ```
    SETUP> set_system_mode analysis
    ```

    During the transition from setup to analysis mode, the tool performs design rule checking.

13. You can optionally report the clock controller pins, global LogicBIST controller configuration parameters, or specified pins results using the following commands:

    - report_clock_controller_pins
    - report_lbist_configuration
    - report_lbist_pins

14. Use the write_edt_files command with the -tsdb option to create EDT/LBIST files and insert the generated hardware into the design. For example, the following command generates timing constraints and writes out the EDT/LBIST files into the instruments directory within the TSDB:

    ```
    ANALYSIS> write_edt_files -tsdb
    ```

    The tool transitions to insertion mode after inserting the EDT/LBIST logic into the design and writes the modified netlist to the dft_inserted_designs directory within the TSDB.

    For the dofile flow, specify the write_edt_files command as follows:

    ```
    ANALYSIS> write_edt_files my_edt_logic -timing_constraints
    ```

Results

The following files are generated in during this step:

Table A-2. Output Files, EDT and LogicBIST IP Generation, TSDB Flow

File	Contents
<prefix>.synthesis_dictionary	A Tcl dictionary-formatted file that is used by the run_synthesis command. This command processes the dictionary to create a synthesis script compatible with the chosen synthesis tool.
<prefix>.tcd	File containing the IP core description.
<prefix>.icl	ICL file describing hybrid EDT/LBIST logic for EDT and LBIST modes — to be used during EDT pattern generation and BIST fault simulation.

Table A-2. Output Files, EDT and LogicBIST IP Generation, TSDB Flow (cont.)

File	Contents
<prefix>.pdl	PDL file describing test_setup initialization of hybrid EDT/LBIST logic for EDT pattern generation and BIST fault simulation.
<prefix>.v	RTL for the hybrid TK/LBIST IP.

Table A-3. Output Files, EDT and LogicBIST IP Generation, Dofile Flow

File	Contents
<prefix>_bypass_shift_sdc.tcl	EDT bypass shift mode timing constraints file.
<prefix>_dc_script.scr	Synopsys script for synthesizing EDT and BIST logic.
<prefix>_<design_name>_edt.tcd	File containing the EDT IP core description.
<prefix>_<design_name>_lbist.tcd	File containing the LBIST IP core description.
<prefix>_edt_fast_capture_sdc.tcl	Capture mode timing constraints file for EDT or EDT-bypass mode.
<prefix>_edt_shift_sdc.tcl	EDT shift mode timing constraints file.
<prefix>_edt_top_rtl.v	Gate-level netlist that instantiates the EDT and BIST logic.
<prefix>_ltest.icl	ICL file describing hybrid EDT/LBIST logic for EDT and LBIST modes — to be used during EDT pattern generation and BIST fault simulation.
<prefix>_ltest.pdl	PDL file describing test_setup initialization of hybrid EDT/LBIST logic for EDT pattern generation and BIST fault simulation.
<prefix>_lbist.v	Per-block MISRs and top-level BIST controller.
<prefix>_lbist_sdc.tcl	File containing all LogicBIST modes including LBIST setup, shift, capture, and single chain mode.

Examples

The following example shows a single EDT/LogicBIST block. The design has X-bounding, control, and observe points controlled by the same design pin named *lbist_en*. This design does not use low-power hybrid IP.

```
// Set context for generating hybrid EDT/LogicBIST IP
set_context dft -edt -logic_bist

// Read design and DFT library
```

```
read_verilog my_core_scan_xbound.v
read_cell_library atpg.lib
set_current_design

// Define clocks and pin constraints
add_clocks 0 occ/occNX1/U7/Z -internal -pin_name NX1   //Internally generated capture clock
add_clocks 0 occ/occNX2/U7/Z -internal -pin_name NX2  // Internally generated capture clock
add_clocks 0 shift_clock
add_clocks 0 refclk -pulse_always
add_input_constraint shift_clock -c0
add_input_constraint scan_en -c0

// Scan chains and test procedure file from scan insertion
add_scan_groups grp1 scan_setup.testproc
for {set i 1} {$i <= 16} {incr i} {
  add_scan_chains chain$i grp1 scan_in$i scan_out$i
}
// Configure EDT logic
set_edt_options -location internal

// Configure LogicBIST controller
set_edt_options -lbist_misr_input_ratio 4
set_lbist_controller -max_shift 400 -max_capture 3 -max_pattern 10000 \
  -capture {clk_once 100}

// Specify LogicBIST pins
set_lbist_pins clock refclk
set_lbist_pins scan_en scan_en
set_dft_enable_options -type xbounding_en -pin_name lbist_en
set_dft_enable_options -type control_point_en -pin_name lbist_en
set_dft_enable_options -type observe_point_en -pin_name lbist_en

// Specify clock controller pins
set_clock_controller_pins shift_clock occ/shift_clock
set_clock_controller_pins scan_en occ/scan_en
```

```
set system mode atpg

// Report user settings
report_edt_configuration
report_lbist_configuration
report_lbist_pins
report_clock_controller_pins

// Generate EDT files
write edt files -tsdb -replace

// For the dofile flow, do not use the -tsdb option. For example:
// write_edt_files created -replace
```

This is a modular IP generation example. The scan chains are defined at internal pins at the block boundary. The design has a TAP controller to which the LogicBIST controller is connected during IP generation. Separate control registers are synthesized in the LogicBIST controller for independently controlling X-bounding, control and observe points. There are multiple clock controllers in the design. The design has multiple NCPs that are used in LogicBIST test in the proportion specified.

```
// Set context for generating hybrid EDT/LogicBIST IP
set_context dft -edt -logic_bist

// Read BIST-ready block designs and TOP level
read_verilog TOP.v my_core_scan_xbound.v piccpu_scan_xbound.v
read_cell_library atpg.lib
set_current_design TOP

// Define clocks and RAM control signals
add_scan_groups grp1 TOP_scan_setup.testproc
add_clocks 0 NX1 NX2 clk ramclk
add_read_control 0 ramclk
add_write_control 0 ramclk
add clock 0 XCLK -pulse_always
add pin constraint scan_en c0
add pin constraint RST c0

// Configure LogicBIST controller
```

```
set_lbist_controller_options -max_shift 400 -max_capture 2 -max_pattern 256000
set_lbist_controller_options -capture {pulseC1P 60 pulseC2P 35 pulseR 5}
set_lbist_controller_options -burn-in on # enable burn-in mode
set_edt_options -location internal
set_lbist_instances -controller_location /dftblk

// Define LogicBIST pins
set_lbist_pins tck tck
set_lbist_pins clock XCLK
set_lbist_pins scan_en scan_en
set_dft_enable_options -type xbounding_en -pin_name lbist_en
set_dft_enable_options -type mcp_bounding_en -pin_name mcp_bound_en
set_dft_enable_options -type control_point_en -pin_name tp_ctrl_en
set_dft_enable_options -type observe_point_en -pin_name tp_obs_en

// Define TAP pins for connecting to LogicBIST controller
set_lbist_pins setup_shift_scan_in tdi
set_lbist_pins setup_shift_scan_out { tdo jtag/scanCfgReg_so }
set_lbist_pins shift_dr { - jtag/shift_dr }
set_lbist_pins capture_dr { - jtag/capture_dr }
set_lbist_pins update_dr { - jtag/update_dr }
set_lbist_pins test_logic_reset { - jtag/tlr }
set_lbist_pins tap_instruction_decode { - jtag/scanCfgReg_en }

// Define clock controller pins
set_clock_controller_pins lbist_en {cc_clk/lbist_en cc_NX1/lbist_en cc_NX2/lbist_en}
set_clock_controller_pins shift_clock_en {cc_clk/shift_clock_en cc_NX1/shift_clock_en
cc_NX2/shift_clock_en}
set_clock_controller_pins scan_en {cc_clk/scan_en cc_NX1/scan_en cc_NX2/scan_en}
set_clock_controller_pins capture_procedure_index {NCPdecoder/i[1] NCPdecoder/i[0]}

// Define EDT block my_core_A of "my_core" design
add_edt_block my_core_A
for {set i 1} {$i <= 16} {incr i} {
  add_scan_chain -internal my_core_A_chain$i grp1 my_core_A/scan_in$i my_core_A/
  scan_out$i
```

```
}
set_edt_power_controller shift enabled -min_switching 15
set_lbist_power_controller shift enabled -min_switching 12
set_edt_instance -block_location /dftblk

// Define EDT block my_core_B of "my_core" design
add_edt_block my_core_B
for {set i 1} {$i <= 16} {incr i} {
  add_scan_chain -internal my_core_B_chain$i grp1 my_core_B/scan_in$i my_core_B/
  scan_out$i
}
set_edt_power_controller shift enabled -min_switching 15
set_lbist_power_controller shift enabled -min_switching 12
set_edt_instance -block_location /dftblk

// Define EDT block piccpu of design "piccpu"
add_edt_block piccpu
for {set i 1} {$i <= 8} {incr i} {
  add_scan_chain -internal piccpu_chain$i grp1 piccpu/edt_si$i piccpu/edt_so$i
}
set_edt_power_controller shift enabled -min_switching 25
set_lbist_power_controller shift enabled -min_switching 25
set_edt_instance -block_location /dftblk
report_lbist_configuration
report_lbist_pins
report_clock_controller_pins
report_edt_configuration -all_blocks
write_edt_files -tsdb -replace

// For the dofile flow, do not use the -tsdb option. For example:
// write_edt_files created -replace -timing_constraints
```

Performing Scan Insertion and X-Bounding

Before inserting the scan and X-bounding logic, you characterize the scan signals with the set_scan_signals command and X-bounding with the set_xbounding_options command.

Prerequisites

- Design netlist with test points and dofile that are output by Tessent Shell during Test Point Analysis and Insertion, Scan Insertion, and X-Bounding.

Procedure

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

   ```
   % tessent -shell
   ```

   After invocation, the tool is in unspecified setup mode. You must set the context before you can use the X-bounding and scan insertion commands.

2. Set the tool context to scan insertion using the set_context command as follows:

   SETUP> set_context dft -scan

3. Load the non-scan gate-level Verilog netlist containing the test points using the read_verilog command.

   SETUP> read_verilog my_modified_netlist.v

4. Load one or more cell libraries into the tool using the read_cell_library command.

   SETUP> read_cell_library atpg.lib

5. Using the dofile command, load the Tessent Shell tool-produced dofile you generated during test point analysis and insertion. For example:

   SETUP> dofile lbist_scan_setup.dofile

   This dofile loads the netlist and the Tessent cell library. It also includes the necessary commands to set up the circuit for DRC.

6. If required in your design flow, load the SDC file using the read_sdc command.

   SETUP> read_sdc my_sdc

   The SDC file should describe false and multicycle paths that should be blocked during LogicBIST.

7. Set the scan insertion options using the set_scan_signals command.

   SETUP> set_scan_signals -ten t_enable

8. Set the X-bounding options with the set_xbounding_options command. For example:

   SETUP > set_xbounding_options -xbounding_enable my_enable1

   This command specifies the name of the top-level pin that enables the X-bounding signals.

9. Change the tool's system mode to analysis using the set_system_mode command.

   SETUP> set_system_mode analysis

During the transition from setup to analysis mode, the tool performs design rule checking and inserts the scan chains.

10. Perform the X-bounding analysis using the analyze_xbounding command.

    ANALYSIS> analyze_xbounding

    The tool reports a summary of how many bounding muxes were added.

11. Optionally report the results of the X-bounding analysis using the report_xbounding command.

    ANALYSIS> report_xbounding

12. Perform scan insertion and X-bounding logic insertion using the insert_test_logic command.

    ANALYSIS> insert_test_logic

    This command modifies the internal representation of the netlist to include the X-bounding muxes, as well as performing scan cell replacement and stitching. This command triggers a transition from analysis to insertion mode.

13. Write out the modified design using the write_design command. For example:

    INSERTION> write_design -output design_with_scan.v

14. Write out the test procedure file and dofile using the write_atpg_setup command as follows:

    INSERTION> write_atpg_setup my_scan_setup

    You need this file for subsequent steps in the process.

Results

In this example, the tool produces the following two files that are to be used as input for the next step of the flow:

- *my_scan_setup.dofile*
- *my_scan_setup.testproc*

Now you are ready to perform "EDT and LogicBIST IP Generation."

Example Dofiles for Core-Level Simulation

In the dofile flow, the tool uses the write_core_description and write_patterns commands.

The following example shows this flow.

```
set_context patterns -scan
read_verilog alu_edt_top_gate.v
```

```
read_cell_library atpg.lib
set_current_design
dofile alu_lbist.dofile
set_lbist_controller_options -programmable_ncp_list {clkseq1 clkseq2 clkseq3 clkseq4}
set_lbist_controller_opt -capture_procedures {clkseq1 40 clkseq2 40 clkseq3 10 clkseq4 10}
set_system_mode analysis
add_faults -all
set_random_patterns 100
simulate_patterns -source bist
write_core_description alu.tcd -replace
write_patterns alu.patdb -patdb -replace
```

Pattern Generation for the Dofile Flow

Generating IJTAG patterns with the dofile flow requires a pattern retargeting dofile template. This dofile is a template with Tcl-style comments that you must modify with the information specific to your design before using the dofile in Tessent Shell. Generating CCM patterns requires the created_ccm.dofile that was generated during IP creation.

For IJTAG pattern generation, you can use the following pattern retargeting dofile template.

```
###########################################################################
#Set the following variables before sourcing this dofile:
#  Required variables:
#        set edt_lbist_icl_file_list   <filenames>
#        set edt_lbist_pdl_file_list   <filenames>
#        set edt_lbist_tcd_file        <filenames>
#        set edt_lbist_patdb_file_list <filenames>
#        set edt_lbist_reference_clock <clockname>
#
#  Optional variables:
#        set edt_lbist_tester_clock_period        <period>
#        set edt_lbist_user_iproc                 <procname>
#        set edt_lbist_write_hw_default_patterns  <0|1: default=0>
#        set edt_lbist_write_diag_patterns        <0|1: default=0>
###########################################################################
#
#Set context to ijtag pattern retargeting
set_context pattern -ijtag

#
#Read library and design
read_cell_library ../data/atpg.lib
read_verilog created_edt_top_gate.v
#
#Read fault simulation data
read_config_data  $edt_lbist_tcd_file

#
#Read ICL files
read_icl $edt_lbist_icl_file_list
#
#Set top level module for ijtag retargeting
set_current_design m8051

#
#Define top-level clocks
if {[info exists edt_lbist_tester_clock_period]} {
     add_clocks $edt_lbist_reference_clock -period
$edt_lbist_tester_clock_period ns -free_running
```

```
} else {
     add_clocks $edt_lbist_reference_clock -free_running
}
add_clocks 1 tck

#
#Define pin constraints
add_input_constraints RST -c0
add_input_constraints edt_reset -c0

#
#Change to analysis mode to generate patterns
set_system_mode analysis
if {[get_context -extraction]} {
     write_icl -o m8051.icl -replace
}
foreach patdb_file $edt_lbist_patdb_file_list {
     open_patdb $patdb_file
}
report_open_patdb

#
#Read PDL files
foreach pdlfile $edt_lbist_pdl_file_list {
     source $pdlfile
}

#
#Write regular LBIST patterns
set begin_pattern 0
set end_pattern  31
set warmup_pattern_count 0
open_pattern_set lbist_normal
if {[info exists edt_lbist_user_iproc]} {
     iCall $edt_lbist_user_iproc
}
iCall run_lbist_normal lbist_clock $begin_pattern $end_pattern lbist
$warmup_pattern_count
close_pattern_set
write_patterns m8051_lbist_normal_${begin_pattern}_${end_pattern}.v
-pattern_set lbist_normal -verilog -replace

#
#Write hardware default LBIST patterns
if {[info exists edt_lbist_write_hw_default_patterns]&&
$edt_lbist_write_hw_default_patterns} {
     open_pattern_set lbist_hw_default
     if {[info exists edt_lbist_user_iproc]} {
         iCall $edt_lbist_user_iproc
     }
     iCall run_lbist_hw_default lbist
     close_pattern_set
     write_patterns m8051_lbist_hw_default.v -pattern_set
lbist_hw_default -verilog -replace
}

#
#Write diagnostic LBIST patterns
```

```
set diag_begin_pattern 0
set diag_end_pattern   1
set diag_warmup_pattern_count 0
if {[info exists edt_lbist_write_diag_patterns] &&
$edt_lbist_write_diag_patterns} {
     open_pattern_set lbist_diag
     if {[info exists edt_lbist_user_iproc]} {
        iCall $edt_lbist_user_iproc
     }
     iCall scan_unload_register lbist_clock $diag_begin_pattern
$diag_end_pattern lbist $diag_warmup_pattern_count
     close_pattern_set
     write_patterns
m8051_lbist_diag_${diag_begin_pattern}_${diag_end_pattern}.v -pattern_set
lbist_diag -verilog -replace
}

exit
```


Performing Pattern Generation for the Dofile Flow

The procedure for generating chip-level serial patterns uses a pattern retargeting dofile that you have created for your design. The write_patterns command specified in the dofile writes out the PatternDB files.

Prerequisites

- Modified design netlist.
- PDL data file, *my_edt_logic*_ltest.pdl, produced by the write_edt_files command.
- ICL data file, *my_edt_logic*_ltest.icl, produced by the write_edt_files command.
- Top-level ICL describing how the signals at the interface of the LogicBIST controller are connected to chip-level pins.
- A PDL that describes the test setup at the chip level if there is any. For example, if there is a TAP controller at the top level, then the tool requires an ICL and, optionally, PDL for the TAP controller.

Procedure

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

   ```
   % tessent -shell
   ```

2. Set the tool context to IJTAG mode using the set_context command as follows:

   SETUP> set_context patterns -ijtag

3. Execute the modified pattern retargeting dofile. For example:

```
SETUP> dofile ./test_retarget.dofile
```

Results

Upon completion, Tessent Shell outputs the testbench/vectors for the entire pattern set, range specific vectors, or Hardware default mode as specified in the dofile.

- If you are using the Considerations for Top-Down Implementation, you finished all necessary steps in this flow.
- If you are using the Hybrid TK/LBIST Implementation, you are ready to perform the Top-Level ICL Network Integration.

Examples

Example 1

In the following example, the core-level ICL IJTAG dofile uses the TCD and PatternDB files to generate a Verilog testbench for core-level pattern verification.

```
set_context patterns -ijtag
read_verilog alu_edt_top_gate.v
read_cell_library atpg.lib
read_icl alu_ltest.icl
set_current_design
set_system_mode analysis
source alu_ltest.pdl
// save the extracted ICL; used at the top level
write_icl -o alu.icl -replace
open_pattern_set alu_core_patt
iCall run_lbist_normal
close_pattern_set
write_patterns alu_core_patt.v -verilog -replace
```

Example 2

When you perform an iCall to the same controller within the same pattern, you receive an error because some iReadVar variables are not uniquified across the pattern set. The following example shows two iCalls that are applied to the same LBIST controller in the same pattern set and the resulting error.

```
open_pattern_set lbist_normal
   iCall run_lbist_normal lbist_clock 6 31 lbist
   iCall run_lbist_normal lbist_clock 6 31 chain_test
Error: iReadVar variable 'bist_done_start[0]' already exists in ICL
instance 'piccpu_lbist_i'.
Error: The iCall of iProc 'run_lbist_normal', which is associated with the
ICL module 'piccpu', failed.
```

To prevent this type of error, use the teststep_name argument in the iProc. This argument adds a unique string for each iRead. For example:

```
open_pattern_set lbist_normal
   iCall run_lbist_normal lbist_clock 6 31 lbist      0 1 lbist_mode
   iCall run_lbist_normal lbist_clock 6 31 chain_test 0 1 chain_test_mode
close_pattern_set
write_patterns piccpu_lbist_normal_6_31.v -pattern_set lbist_normal
   -verilog -replace
exit
```

Performing Pattern Generation for CCM in the Dofile Flow

When generating CCM patterns using the dofile flow, you must use the CCM dofile, create_ccm.dofile, and the CCM .testproc file, created_ccm.testproc, that are generated during IP creation.

For details, refer to "Controller Chain Mode" on page 48.

Prerequisites

- Modified design netlist.
- ICL description of the current design.
- A PDL that describes the test setup at the chip level if there is any. For example, if there is a TAP controller at the top level, then the tool requires an ICL and, optionally, PDL for the TAP controller.

Procedure

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

   ```
   % tessent -shell
   ```

2. Set the tool context to ATPG pattern generation:

   ```
   SETUP> set_context pattern -scan
   ```

3. Read in the design and libraries. For example:

   ```
   SETUP> read_verilog created_edt_top_gate.v
   SETUP> read_cell_library ../data/atpg.lib
   ```

4. Source the created_ccm.dofile generated during IP creation:

   ```
   SETUP> dofile created_ccm.dofile
   ```

   The created_ccm.dofile calls the CCM .testproc file, created_ccm.testproc, that was also generated during IP creation. For details, refer to the examples that follow.

5. Change the system mode to analysis:

   ```
   SETUP> set_system_mode analysis
   ```

6. Add the CCM faults.

   ```
   ANALYSIS> add_ccm_faults
   ```

   The add_ccm_faults() TCL proc is contained within the created_ccm.dofile. This proc targets the faults to the hybrid IP.

7. Create and save the CCM patterns. For example:

   ```
   ANALYSIS> create_patterns
   ANALYSIS> write_patterns ccm_patt.v -verilog -replace -serial
   ```

Examples

The follow example generates CCM patterns.

```
set_context pattern -scan

read_verilog created_edt_top_gate.v
read_cell_lib ../data/atpg.lib

dofile created_ccm.dofile

set_system_mode analysis

add_ccm_faults

create_patterns
write_patterns ccm_patt.v -verilog -replace -serial
```

Example created_ccm.dofile

The following example shows a snippet of the CCM dofile, created_ccm.dofile, that was generated during IP creation. When a TAP is present, the tool places the TAP controller in the run-test-idle state to ensure that the IJTAG control signals are in a known state. IJTAG control signals that are top-level pins are constrained to their inactive values, as shown below.

```
add_scan_groups grp1 created_ccm.testproc
add_scan_chains ccm_chain grp1 ccm_scan_in ccm_scan_out
add_clocks 0 tck

add_input_constraints ijtag_reset c0
add_input_constraints ijtag_sel c0
add_input_constraints ijtag_ce c0
add_input_constraints ijtag_se c0
add_input_constraints ijtag_ue c0
add_input_constraints ccm_en c1

proc add_ccm_faults {} {          // Adding faults only on the hybrid logic
   add_faults {lbist_instance}
   add_faults {edt_instances}
   add_faults {single_chain_mode_logic_instance}
   }
```

Example CCM .testproc file

The generated CCM .testproc file, created_ccm.testproc, is identical to the LBIST and EDT .testproc files except for minor modifications for use with CCM. Most notably, CCM .testproc files do not include internal signals (internal_shift_clock, internal_scan_en, internal_capture_en) and capture procedures. See the annotations in the following example.

```
set time scale 1.000000 ns ;
set strobe_window time 100 ;

  timeplate gen_tp1 =
    force_pi 0 ;
    measure_po 100 ;
    pulse clk_ 200 100;       //No internal_shift_clock
    pulse ramclk 200 100;
    pulse refclk 200 100;
    pulse reset 200 100;
    pulse shift_clock 200 100;
    pulse tck 200 100;       //Controller chain clock set to default tck
    period 400 ;
  end;

  always =
      pulse refclk ;
  end;
```

```
procedure test_setup =      //Capture procedures are not necessary
  timeplate gen_tp1 ;
   // cycle 1 starts at time 0
  cycle =
     force edt_clock 0 ;
     force refclk 0 ;
     force reset 0 ;
     force scan_enable 0 ;
     force shift_clock 0 ;
     force test_logic_reset 1 ;
     pulse refclk ;
  end ;
   // cycle 2 starts at time 400
   cycle =
     force test_logic_reset 0 ;
     pulse refclk ;
  end;
end;

procedure shift =
  scan_group grp1 ;
  timeplate gen_tp1 ;
  // cycle 1 starts at time 0
  cycle =
     force_sci ;
     force tck 0 ;//Force tck to 0 rather than internal_shift_clock to 1
     measure_sco ;
     pulse refclk
     pulse shift_clock ;
     pulse tck ;    //Pulsing tck rather than internal_shift_clock
  end;
end;

procedure load_unload =
  scan_group grp1 ;
  timeplate gen_tp1 ;
  // cycle 1 starts at time 0
  cycle =
       force clk_ 0 ;
       force edt_clock 0 ;   //no internal_capture_en, internal_scan_en,
       force ramclk 0 ;      //and internal_shift_clock
       force refclk 0 ;
       force reset 0 ;
       force scan_enable 1 ;/
       force shift_clock 0 ;
       force tck 0 ;
       force test_logic_reset 0 ;
       pulse refclk ;
  end ;
  apply shift 16;
end;
```

Pattern Mismatch Debugging in the Dofile Flow

You can use a dofile debugging flow to verify clocks prior to running serial pattern simulation.

Debug Based on MISR Signature Divergence (Dofile Flow)

You can verify the BIST registers and clocks at the same time, which in turn enables you to identify the patterns corresponding to mismatches (such as MISR) as the mismatches occur.

Restrictions and Limitations

- Once the TCD file is created as described below, you cannot alter the core hierarchy (such as by ungrouping the LogicBIST controller). Altering the core hierarchy causes the list of monitor points in the TCD file to become out of sync.

Prerequisites

- You have performed the hybrid TK/LBIST flow through the pattern generation step.

Procedure

1. Verify the clocks.

   Modify the default LogicBIST retargeting dofile template that the tool creates during fault simulation (with the write_lbist_register_data command).The following example shows the default LogicBIST retargeting dofile template. To verify the clocks, uncomment and set the edt_lbist_setup_and_clock_verification_patterns variable to 1.

```
#####################################################################
#Set the following variables before sourcing this dofile:
#  Required variables:
#       set edt_lbist_icl_file_list   <filenames>
#       set edt_lbist_pdl_file_list   <filenames>
#       set edt_lbist_tcd_file        <filenames>
#       set edt_lbist_patdb_file_list <filenames>
#       set edt_lbist_reference_clock <clockname>
#
#  Optional variables:
#       set edt_lbist_tester_clock_period        <period>
#       set edt_lbist_user_iproc                 <procname>
#       set edt_lbist_write_hw_default_patterns  <0|1: default=0>
#       set edt_lbist_write_diag_patterns        <0|1: default=0>
#       set edt_lbist_setup_and_clock_verification_patterns <0|1: \
          default=0>
#####################################################################

...
#
#Write setup and clock verification LBIST patterns
if {[info exists edt_lbist_setup_and_clock_verification_patterns]
&& $edt_lbist_setup_and_clock_verification_patterns} {
     set one_pattern_per_ncp 1
     open_pattern_set lbist_setup_and_clock_verification
     if {[info exists edt_lbist_user_iproc]} {
         iCall $edt_lbist_user_iproc
     }
     iCall cpu_gates_tessent_occ_NX1_inst.setup fast_capture_mode \
 on capture_window_size 2 static_clock_control external
     iCall cpu_gates_tessent_occ_NX2_inst.setup fast_capture_mode \
 on capture_window_size 2 static_clock_control external
     iCall cpu_gates_tessent_occ_NX3_inst.setup fast_capture_mode \
 on capture_window_size 2 static_clock_control external
     iCall lbist_setup_and_clock_verification lbist_clock lbist \
       $one_pattern_per_ncp
     close_pattern_set
     write_patterns cpu_lbist_setup_and_clock_verification.v \
 -pattern_set lbist_setup_and_clock_verification -verilog -replace
}

exit
```

This is the syntax for the lbist_setup_and_clock_verification iProc:

iProc lbist_setup_and_clock_verification { {clock_select lbist_clock} {mode_name lbist} {one_pattern_per_ncp 0} {teststep_name ""} } {…}

As shown in the example iCall, the generated template sets the one_pattern_per_ncp variable to 1 by default. This means that if you have four NCPs, the tool runs four tests, one for each NCP.

The default pattern run is 256. To exercise the full pattern range, in the generated template, change the line “set one_pattern_per_ncp 1” from 1 to 0.

2. If clock verification fails, investigate and fix possible causes, such as:

   a. The On-Chip Clock Controller (OCC) is not set up correctly. Ensure that the arguments are specified correctly if you are using an iCall to setup the OCC.

   b. Ensure that you have correctly set any primary input pin asserts that are needed for the clocks to operate.

   c. You may find that while debugging the clock verification failures the capture window is too small. That is, capture clock pulses occur outside the window where "capture_en=1." If this is the case, return to LogicBIST IP generation and specify the following command to increase the capture window:

      ```
      set_lbist_controller_options -max_capture_cycles #
      ```

3. Run Verilog simulation with the LogicBIST debugging feature enabled as shown below, and identify any failing LogicBIST patterns.

   The resulting transcript includes mismatch statements such as those shown in bold following. The statements tell you at which pattern the MISR signature started to diverge from the expected value.

   Note

   To display the passing data, specify "+show_passing_regs" when you start the simulator.

```
...
Setting up controller TLB_coreB_I1.coreB_lbist_i
     Number of patterns     : 5 (5 + 0 warm-up patterns)
     Pattern Length         : 40
     Shift Clk Select       : 0b01
     Capture Phase Width    : 0x3 Shift Clock Cycles
     PRPG Seed              : 0x66241da0
     MISR Seed              : 0x000000
Starting controller TLB_coreB_I1.coreB_lbist_i in Normal mode,
patterns 0 to 3
     Checking that the controller TLB_coreB_I1.coreB_lbist_i DONE
signal is NO at the beginning of the test
   Mismatch at pattern 2 for TLB.coreB_I1.coreB_edt_lbist_i.misr:
Expected = 83ab37 Actual = 7854d0
   Mismatch at pattern 3 for TLB.coreB_I1.coreB_edt_lbist_i.misr:
Expected = 9b96e3 Actual = 5e161a
Test Complete for controller TLB_coreB_I1.coreB_lbist_i
     Checking that signal DONE is YES for controller
TLB_coreB_I1.coreB_lbist_i
Checking results of controller TLB_coreB_I1.coreB_lbist_i
     Expected Signature for controller TLB_coreB_I1.coreB_lbist_i :
0x9b96e3
Turning off LogicBist controller TLB_coreB_I1.coreB_lbist_i
...
```

Enable the debugging feature by specifying sim_monitor with the run_lbist_normal iProc. This is the syntax:

iProc run_lbist_normal { {clock_select lbist_clock} {begin_pattern 0} {end_pattern 31} {mode_name lbist} {warmup_pattern_count 0} {misr_compares 1} {sim_monitor off} {teststep_name ""} } {…}

For example:

```
set sim_monitor 1
open_pattern_set lbist
   iCall run_lbist_normal lbist_clock 980 999 lbist 0 0 $sim_monitor
close_pattern_set
```

4. Re-run the simulation so that you can identify the failing flop associated with the particular pattern where the MISR started to diverge.

   Use the scan_unload_register iProc, as shown in the following sample dofile template. Set the edt_lbist_write_diag_patterns variable to 1, and adjust the diag_begin_pattern and diag_end_pattern settings accordingly. For example, if the MISR signature failed at pattern 2, you would set the begin and end patterns to 2.

```
#####################################################################
#Set the following variables before sourcing this dofile:
#  Required variables:
#        set edt_lbist_icl_file_list   <filenames>
#        set edt_lbist_pdl_file_list   <filenames>
#        set edt_lbist_tcd_file        <filenames>
#        set edt_lbist_patdb_file_list <filenames>
#        set edt_lbist_reference_clock <clockname>
#
#  Optional variables:
#        set edt_lbist_tester_clock_period        <period>
#        set edt_lbist_user_iproc                 <procname>
#     set edt_lbist_write_hw_default_patterns <0|1: default=0>#
set edt_lbist_write_diag_patterns        <0|1: default=0>
#        set edt_lbist_setup_and_clock_verification_patterns <0|1:
#          default=0>
#####################################################################
...
#
#Write diagnostic LBIST patterns
set diag_begin_pattern 2
set diag_end_pattern 2
set diag_warmup_pattern_cnt 0
if {[info exists edt_lbist_write_diag_patterns] &&
$edt_lbist_write_diag_patterns} {
     open_pattern_set lbist_diag
     if {[info exists edt_lbist_user_iproc]} {
          iCall $edt_lbist_user_iproc
     }
     iCall cpu_gates_tessent_occ_NX1_inst.setup fast_capture_mode
on capture_window_size 2 static_clock_control external
     iCall cpu_gates_tessent_occ_NX2_inst.setup fast_capture_mode
on capture_window_size 2 static_clock_control external
     iCall cpu_gates_tessent_occ_NX3_inst.setup fast_capture_mode
on capture_window_size 2 static_clock_control external
     iCall scan_unload_register lbist_clock $diag_begin_pattern
$diag_end_pattern lbist $diag_warmup_pattern_cnt
     close_pattern_set
     write_patternscpu_lbist_diag_${diag_begin_pattern}_
${diag_end_pattern}.v -pattern_set lbist_diag -verilog -replace
}
```

Results

The following transcript example shows a mismatch at an lbist_scan_out pin.

```
   ...
   300ns:  piccpu MISR Seed           : 0x000000
 49300ns:  Starting controller TLB_coreB_I1.coreB_edt_lbist_i in Normal
mode, patterns 0 to 0
 51000ns:  Checking that the controller TLB_coreB_I1.coreB_edt_lbist_i
DONE signal is NO at the beginning of the test
 62800ns:  Test Complete for controller TLB_coreB_I1.coreB_edt_lbist_i
 69000ns:  Scanning out capture results of vector 0 for controller
TLB_coreB_I1.coreB_edt_lbist_i
180024ns: Mismatch at pin          0 name            lbist_scan_out,
Simulated x, Expected 0
180100ns: Corresponding ICL register:
TLB_coreB_I1.coreB_edt_single_chain_mode_logic_i.TLB_coreB_I1.coreB_edt_i
nternal_scan_registers_i.coreB_A_chain1[18]
180100ns: Corresponding design object:  coreB_A/u11/PRB_reg/DFF1
181700ns:  Turning off LogicBist controller TLB_coreB_I1.coreB_edt_lbist_i
```

Debug Based on Scan Cell Monitoring (Dofile Flow)

You can have the tool monitor the scan chains and return information about scan cells associated with unexpected unload values.

Restrictions and Limitations

- Once the TCD file is created as described below, you cannot alter the core hierarchy (such as by ungrouping the LogicBIST controller). Altering the core hierarchy causes the list of monitor points in the TCD file to become out of sync.

Prerequisites

- You have performed the hybrid TK/LBIST flow through the pattern generation step.

Procedure

1. Verify the clocks as described in step 1 of “Debug Based on MISR Signature Divergence (Dofile Flow)” on page 285.
2. If clock verification fails, investigate and fix possible causes as described in step 2 of “Debug Based on Scan Cell Monitoring (Dofile Flow)” on page 290.
3. Run Verilog simulation with the monitor_scan_cells LogicBIST debugging feature enabled as shown below. When specified, the tool monitors the scan chain output pins, detects when an unexpected value is unloaded, and reports which shift cycle and scan cell failed.

   Specify sim_monitor with the run_lbist_normal iProc and include the monitor_scan_cell argument. For example:

   ```
   set sim_monitor 1
   open_pattern_set lbist
     iCall run_lbist_normal lbist_clock 980 999 lbist 0 0 \
       monitor_scan_cell $sim_monitor
   close_pattern_set
   ```

Results

When a mismatch occurs the tool first reports the scan chain output pin where the mismatch was observed, and then maps the mismatch to a pattern, shift cycle, and scan cell. For both messages it reports the simulated and expected values. If there is inversion between the scan cell and the scan out, the simulated/expected values on these two lines is different. If the failing scan cell is within a sub-chain of a hard module, then the message only reports the scan cell and not the pin of the scan cell that failed.

The following transcript example shows mismatches when the wrong values are observed on scan chain cells.

```
#ns: Pattern_set serial_load
#ns: Setting up controller xtea_tk_lbist_ip_tessent_lbist_i
#ns: Number of patterns : 3 (3 + 0 warm-up patterns)
#ns: Pattern Length : 2 #ns: Shift Clk Select : 0b00
#ns: Capture Phase Width : 0x20 Shift Clock Cycles
#ns: PRPG Seed : 0x3e0a
#ns: MISR Seed : 0x000000
#ns: Starting controller xtea_tk_lbist_ip_tessent_lbist_i in Normal mode,
patterns 0 to 2
#ns: Checking that the controller xtea_tk_lbist_ip_tessent_lbist_i DONE
signal is NO at the beginning of the test

#ns: Mismatch at pin xtea_tk_lbist_ip_tessent_edt_lbist_c0_inst/
tessent_persistent_cell_edt_scan_out_0_buf/Y, Simulated x, Expected 1
#ns: Corresponding scan cell for pattern 0 at shift cycle 0:
hard_mod2_inst1/OUT_R_reg_0_, Simulated x, Expected 0

#ns: Mismatch at pin xtea_tk_lbist_ip_dft_tessent_edt_lbist_c0_inst/
tessent_persistent_cell_edt_scan_out_0_buf/Y, Simulated x, Expected 0
#ns: Corresponding scan cell for pattern 0 at shift cycle 11:
hard_mod2_inst1/IN2_R_reg_3_, Simulated x, Expected 1

#ns: Mismatch at pin xtea_tk_lbist_ip_tessent_edt_lbist_c1_inst/
tessent_persistent_cell_edt_scan_out_2_buf/Y, Simulated x, Expected 1
#: Corresponding scan cell for pattern 1 at shift cycle 0: B_R_reg_2_/Q,
Simulated x, Expected 1
```

Tessent OCC for Hybrid TK/LBIST in the Dofile Flow

The Tessent On-Chip Clock Controller (OCC) can be used in the Hybrid TK/LBIST flow. Tessent Shell can generate and insert the Tessent OCCs with programmable capture clock sequences for use with Hybrid TK/LBIST applications.

During LBIST mode, the actual clock sequence is parallel loaded into the Tessent OCC. You can configure the Tessent OCC so that the values can be loaded through OCC module input ports or through a TDR inside the OCC. When an LBIST test uses only one NCP at a time, this value can be loaded through the TDR or be available as constants at the module inputs. If the LBIST test uses multiple NCPs, then the tool generates the parallel load clock sequence for the currently-active NCP, the index for which is provided by the LBIST controller.

When Tessent OCC is generated with internal IJTAG control (that is, you have specified the Occ/ijtag_host_interface property), the static signals for controlling the OCC for LBIST mode are included within the OCC. Additionally, when static_clock_control is either internal or both, a TDR is included for generating the LBIST capture clock sequence. However, you can use this internal TDR only when LBIST test uses only one active NCP.

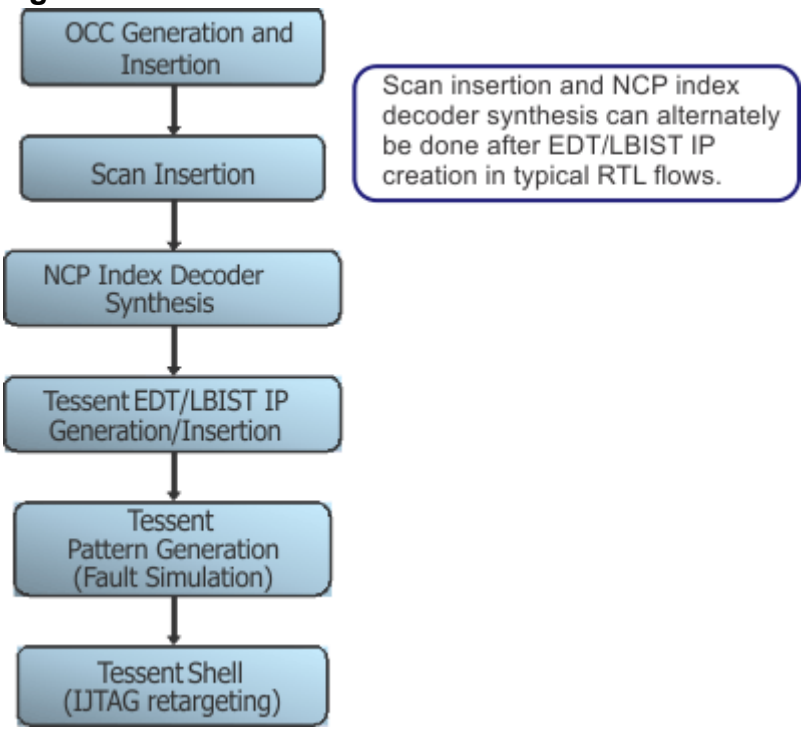
For additional Tessent OCC-specific information, see "Tessent On-Chip Clock Controller" in the *Tessent Scan and ATPG User's Manual*.

Tessent OCC TK/LBIST (Dofile Flow)

The Tessent OCC TK/LBIST flow varies from the standard TK/LBIST flow. Use the information in this section as a guide to configure and insert the Tessent OCC, and interface the Tessent OCC with the hybrid TK/LBIST flow controller.

In general, you use the standard TK/LBIST Hybrid TK/LBIST Implementation with the modifications specific to the TK/LBIST flow as illustrated in Figure A-2. See also Example Tessent OCC TK/LBIST Flow for a step-by-step illustration of the Tessent OCC TK/LBIST flow.

Figure A-2. Modified TK/LBIST Flow for Tessent OCC



Tessent OCC for TK/LBIST Flow Configuration (Dofile Flow)

The clock controller can be used in two modes for LBIST: using a static sequence loaded through an ICL network for all patterns in a session; or, using a set of sequences that are cycled through every 256 patterns.

A Tessent OCC configured with *capture_en* as the capture trigger is used for hybrid TK/LBIST flow The slow clock input of the Tessent OCC is connected to TK/LBIST reference input clock in LBIST mode. The scan enable input of the Tessent OCC is connected to LBIST shift enable output in LBIST mode. The capture enable input of the Tessent OCC is connected to the LBIST capture enable output in LBIST mode. The tool adds muxes inside the LBIST controller to choose between LBIST and ATPG mode signals.

TCD for the Clock Controller

The tool writes out a Tessent Core Description file after you execute the process_dft_specification command when processing the clock controller wrapper. This file is written out in the instrument directory for the clock controller with *.tcd_occ* filename extension. This file is for input into the Tessent Scan insertion tool.

NCP Index Decoder (Dofile Flow)

The NCP index decoder is a simple combinational logic block that decodes the NCP index output of the Hybrid TK/LBIST controller into clock sequences to be generated by the Tessent OCCs across all capture procedures.

The LBIST controller generates an NCP index that cycles through the NCPs based on the active percentage for each NCP. This NCP index is decoded to provide the actual clock sequence that is parallel loaded to the OCC.

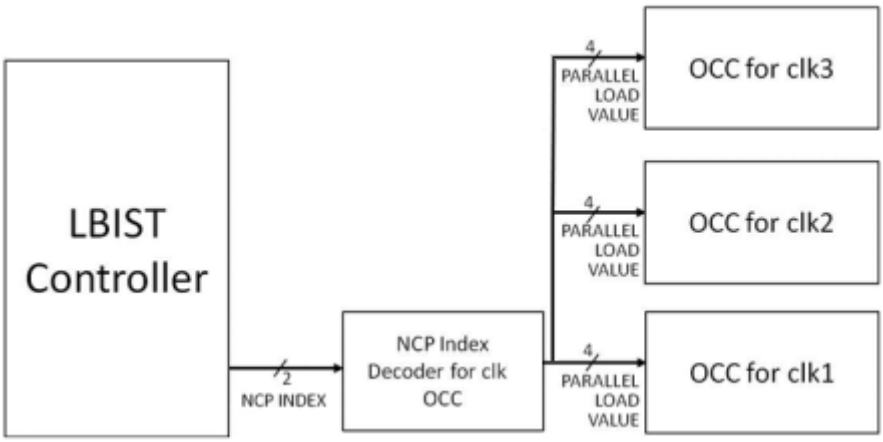
You must specify the clocking combinations to be used during TK/LBIST test. The tool synthesizes the NCP index decoder and generates named capture procedures based on this description.

You can use the NCP index decoder with only a single clock domain. The NCP index decoder is based on the number of unique clocking waveforms, not on the number of clocks. For example, with a single clock you can generate two NCPs (a single pulse and double pulse).

To reduce the test time and achieve high coverage, it is possible to activate multiple clock domains at the same time. This is a trade-off between test time and hardware cost: the cost comes from adding bounding logic for paths crossing clock domains. You may need bounding for both stuck-at and transition patterns. You lose coverage in all blocked paths, but you can control the blocking with the McpBoundingEn dft_signal. It is possible to disable blocking at runtime, but the NCPs can only pulse compatible clocks.

One NCP index decoder is synthesized for each LogicBIST controller and can be used for controlling all the OCCs involved in LogicBIST. In Figure A-3, there are three OCCs configured for four cycles each. There are two input binary values to the NCP index decoder (indicating a maximum of four NCPs), which is decoded as a single control signal per OCC per cycle that reflects the required clocking waveform.

Figure A-3. NCP Index Decoder Connections



The tool generates only one index decoder for all OCCs. The NCP index decoder is instantiated by default at the top-level, or as controlled with the parent_instance property of the NcpIndexDecoder specification.

NCP Index Decoder Creation

Specify the NCP index decoder with the DftSpecification/LogicBist/NcpIndexDecoder wrapper. For a complete description and usage, see NcpIndexDecoder in the *Tessent Shell Reference Manual*.

If you are using only one NCP, you cannot use the NcpIndexDecoder wrapper because it is supported only for external static clock controls and two or more NCPs. Refer to "Considerations When Only Using One NCP" on page 159 for fault simulation considerations.

When the NcpIndexDecoder is generated in the same run as the LogicBist IP, the NCP count is automatically inferred from the number of Ncp() wrappers in the NcpIndexDecoder wrapper. When NcpIndexDecoder is generated in a different run, you must specify the LogicBist/Controller/NcpOptions/count property = 1.

Note

NCP index decoder generation requires an elaborated design and Tessent OCC instances you have added with the add_core_instances command.

Examples

In the following example, assume the design has two top-level Tessent OCC instances named *m8051_gate_tessent_occ_clk1_inst* and *m8051_gate_tessent_occ_clk2_inst* of the same Tessent OCC module *m8051_gate_tessent_occ*.

```
LogicBist
  NcpIndexDecoder {
    Connections {
      NcpIndex: m8051_lbist_i/ncp;
    }
  Ncp(stuck) {
    cycle(0): m8051_gate_tessent_occ;
      //Specified using module name, refers to both the OCC instances.
  }
  Ncp(clk1_double_pulse) {
    cycle(0): m8051_gate_tessent_occ_clk1_inst;
    cycle(1): m8051_gate_tessent_occ_clk1_inst;
  }
  Ncp(clk2_double_pulse) {
    cycle(0): m8051_gate_tessent_occ_clk2_inst;
     //Note - cycle(1) is omitted, so no clock activity
    cycle(2): m8051_gate_tessent_occ_clk2_inst;
  }
}
```

Assuming the above specification is in a file named *ncp.dft_spec*, the following Tessent Shell dofile synthesizes and inserts the NCP index decoder.

```
set_context dft -no_rtl
read_verilog \
  tsdb_outdir/dft_inserted_designs/m8051_gates.dft_inserted_design/
m8051.vg
read_verilog tsdb_outdir/instruments/m8051_gates_occ.instrument/
m8051_gate_tessent_occ.vg
read_cell_library atpg.lib
set_current_design
read_core_description \
      tsdb_outdir/instruments/m8051_gates_occ.instrument/
      m8051_gate_tessent_occ.tcd
add_core_instances -module m8051_gates_tessent_occ
report_core_instances
read_config_data ncp.dft_spec
set_design_level physical_block
process_dft_specification
```

OCC Generation and Insertion (Dofile Flow)

You create an OCC-specific DftSpecification to interface the hybrid controller to the OCC. Then you use Tessent Shell to process this specification.

You define the OCC DftSpecification using DftSpecification configuration syntax. A separate OCC DftSpecification with controller wrappers should be added for each different clock that

needs to be programmable during capture. Additionally, OCCs may be added for asynchronous reset signals declared as a clock.

Tessent Shell creates Verilog RTL, ICL, PDL, TCD files, and TCD scan describing the Tessent OCC instrument, as well as a Verilog netlist that instantiates the OCC in the user design. Many generation and insertion options are available in the Dft Specification to control this process. The ICL and TCD outputs are used in later steps like EDT/LBIST IP creation to describe the configuration of the generated OCCs as well identifying the port functions. The Tessent OCC RTL should be synthesized to a gate level design before it is used for downstream steps that require a gate level netlist.

The Tessent OCC can be inserted in a design either at RTL or gate level. When inserted at RTL level or before EDT IP, the Tessent OCC shift registers can either be merged with design scan cells or stitched up into dedicated Tessent OCC scan chains. When using stitched into dedicated chains, these can be either compressed or uncompressed. When OCC is inserted after EDT IP, the OCC shift registers have to be stitched into dedicated OCC scan chains and handled as uncompressed chains driven directly by the tester.

Static Clock Control

By default, the Tessent OCC generated by Tessent Shell provides programmability only through the clock control shift register suitable for ATPG. To use the OCC on a LBIST design, you should add static clock control. Static clock control, which refers to clock sequence not decided by ATPG, can be one of the following options:

- **Internal —** The Tessent OCC is statically programmable using an internal TDR for both LBIST and ATPG modes. When using this option, the LBIST test can use only one NCP at a time. When multiple NCPs are to be used, it needs to be done in multiple pattern sets.
- **External —** The Tessent OCC is statically programmable through OCC module ports for the LBIST mode. This enables use of multiple NCPs for LBIST test in a single pattern set. An NCP index decoder is synthesized to provide the clock sequence for the different NCPs based on the ncp_index output from the LBIST controller. The Tessent OCC external clock control module port is available only for the LBIST mode and unavailable for ATPG.
- **Both —** This combines both the internal and external options described above. ATPG can use the TDR for static clock control. LBIST can use either the TDR or the OCC module ports.

Capture Trigger

To use the Tessent OCC for TK/LBIST operation, you should set the capture trigger to capture enable. In this case, scan enable is replaced by the LBIST capture enable signal as the trigger. To enable either fast capture or slow capture to be used during LBIST, the slow clock signal is

connected to the free running LBIST controller input clock so that it pulses on all capture cycles.

The capture enable signal should to be tied to constant-0 or connected to inverted scan enable during OCC insertion.

Connection to OCC External Clock Ports

By default, when configuring the Tessent OCC with static_clock_control as either “external” or “both’, the tool ties the OCC external clock control module ports to constant-0. These ports are not described in the OCC ICL and are not referenced in the OCC setup iProc. In an LBIST application with multiple NCPs, these OCC pins would be eventually connected to the NCP index decoder.

Scan Insertion (Dofile Flow)

During Tessent OCC insertion, the clock control shift register IO of the Tessent OCCs are left unconnected. Integrating these Tessent OCC shift register sub-chains into the design is performed outside of the Dft Specification. Scan insertion can be performed using third-party tools.

You can insert the Tessent OCC into either a non-scan design or a scan design.

Non-Scan Design Scan Insertion

When the Tessent OCC is inserted in a non-scan design, the OCC shift registers should be declared as sub chains to the scan insertion tool. The scan insertion tool merges these sub-chains with other scan cells taking care of scan chain length balancing.

Scan Design Insertion

When the Tessent OCC is inserted in a scan design, the Tessent OCC shift registers should be connected into one or more dedicated scan chains by the user, based on the target scan chain length. The Tessent OCC scan enable pin is connected to the design scan enable during scan insertion. The length of the sub chains should be the length of the Tessent OCC shift register. The OCC sub chains are always made part of scan chains considering the EDT mode in contrast to LBIST mode where the OCC shift register is bypassed by a single flop. The sub chains are internally bypassed within the OCC during LBIST.

OCC EDT/LBIST IP Creation (Dofile Flow)

The method you use for Tessent OCC EDT/LBIST creation depends on which flow you are using: pre-synthesis or gate level.

Pre-Synthesis Flow

In pre-synthesis flow, the Tessent OCC is not present in the input skeleton design and the ICL/TCD files for the OCC cannot be read during IP creation. You should instruct the tool to generate LBIST controller compatible with Tessent OCCs. When using this flow, you are responsible for making all connections between the LBIST controller, EDT blocks, NCP index decoder and OCCs. You do this by using the "set_lbist_controller_options -tessent_occ on" command and options.

Gate-Level Flow

During EDT IP creation, you should input into Tessent Shell the Tessent OCC inserted design with OCC shift registers included in scan chains should be read. The OCC scan chains can be either part of compressed or uncompressed chains. Do not add Tessent OCC uncompressed scan chains during IP creation; only add them during pattern generation. You must configure the Tessent OCC correctly to pass the IP creation DRC checks, specifically the shift clock, scan enable, and capture enable signals of the Tessent OCC are properly connected and operated in the incoming test procedures.

During IP creation, you read in the ICL, PDL and Tessent Core Description (TCD) for the OCC. This is required to properly setup the Tessent OCC during IP creation. The TCD description is bound to a netlist instance by treating it as a core instance, similar to how scan pattern retargeting uses TCD. The tool identifies the Tessent OCCs when the tessent_instrument_type ICL attribute is set to "mentor::occ". This attribute value is considered when generating the ICL signature, so it cannot be added to user OCCs. When Tessent OCCs are present in the design, the LBIST controller is modified to correctly interface with the OCC.

The TK/LBIST compatible Tessent OCCs are required to have the following two features: capture trigger using capture enable and static clock control either external or both when using multiple NCPs. See "Static Clock Control" on page 154 and "Capture Trigger" on page 154.

Note

Do not mix Tessent OCCs and custom OCCs (defined using set_clock_controller_pins command) in the same LBIST controller. The tool performs rule checks to validate this requirement.

During EDT IP creation, the Tessent OCC capture enable pins that are not functionally driven are driven by the inverted OCC scan enable. The Tessent OCC capture enable pins that are functionally driven (that is, by inverted scan enable) are multiplexed between existing functional connection and LBIST capture enable.

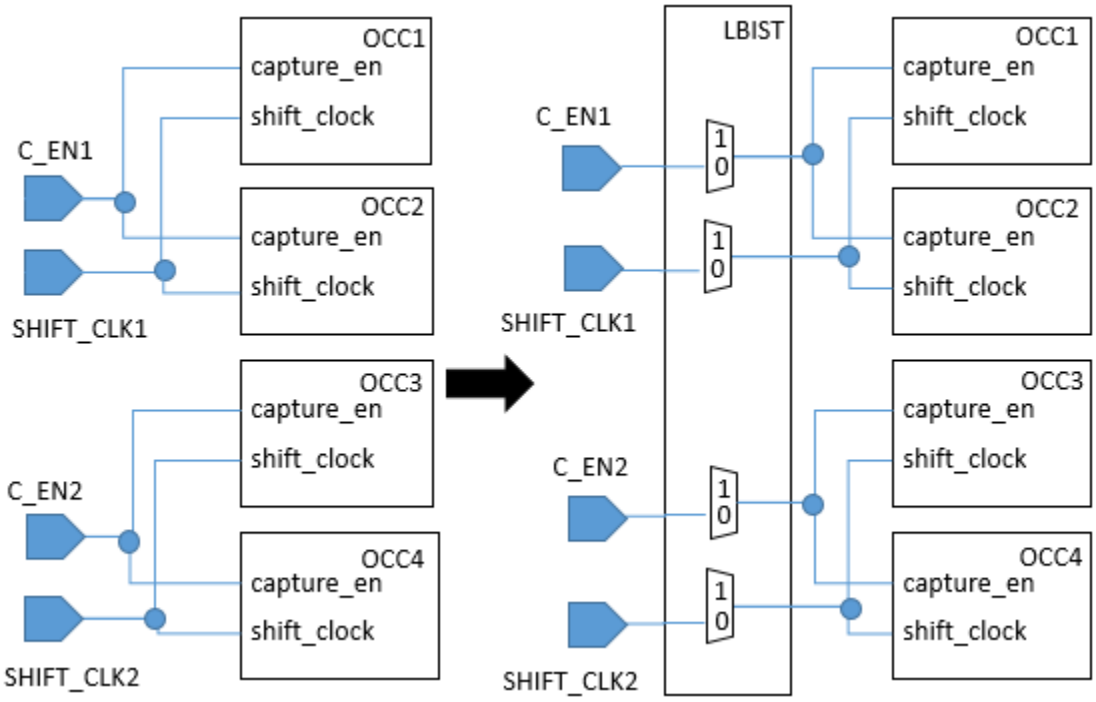
OCC Connections Interception

During hybrid TK/LBIST and OCC insertion with the TSDB flow, when adding edt_clock, shift_capture_clock dft signals as top-level ports, the LogicBIST IP intercepts and multiplexes

the existing connections of the OCCs. The tool attempts to reduce the amount of generated LogicBIST logic used to complete the intercepts.

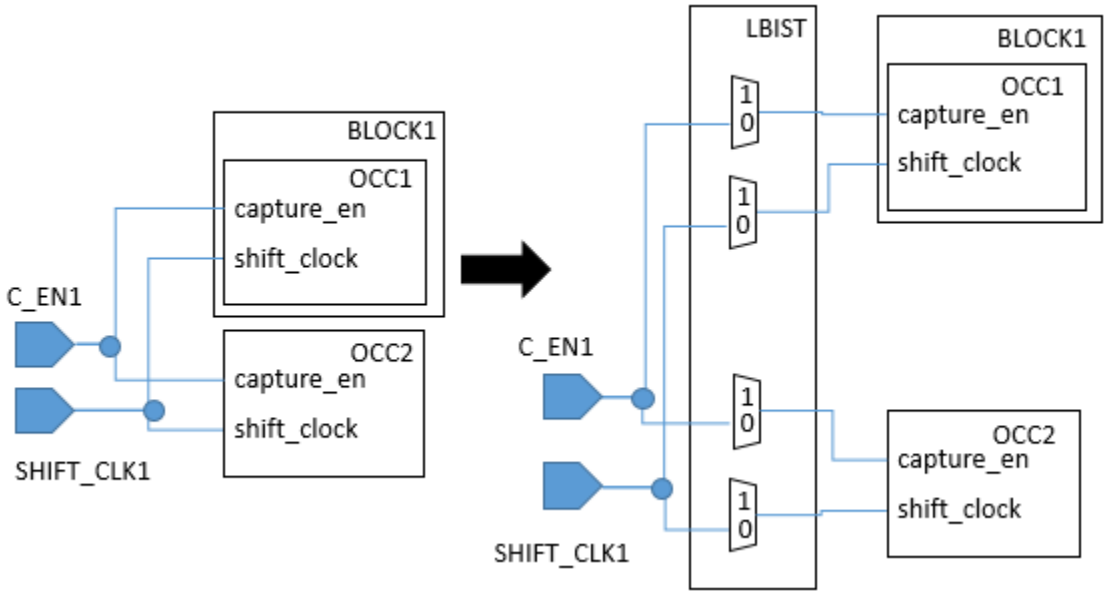
If, for a given signal, multiple OCCs have the same signal source, then one mux is sufficient for intercepting all of them. The tool considers OCCs as having the same sources if their nets fan in from the same net, as observed during LogicBIST validation. As shown in the following figure, instead of inserting a mux for each intercept at the OCCs—eight intercepts—the tool inserts only four muxes. Optimization occurs for the capture enable and shift clock signals.

Figure A-4. OCC/LogicBIST Connection Intercept With Same Signal Source



The signal sources are considered the same if their nets fan in from the same net, as observed during LogicBIST validation. The following example shows the case when the fanin nets differ for each OCC because one of them resides within a sub-module. The tool treats these OCCs as having different sources. It generates a mux for each intercept at the OCCs, leading to four generated muxes instead of two.

Figure A-5. OCC/LogicBIST Connection Intercept With Different Signal Sources



Capture Procedures

During IP creation, you specify the total number of NCPs used for LBIST.

This is required to synthesize the NCP index output and NCP activity percentage registers. If the exact number of NCPs is not known during IP creation, an upper bound can be used. During fault simulation, unused NCP indices can be specified as 0%. If the names and activity percentage of the NCPs is specified during IP creation, this is used for the hardware default mode. When not specified, the tool defaults to equal activity for all the NCPs for the hardware default mode.

NCP Index Decoder Synthesis (Dofile Flow)

The NCP index decoder is synthesized is normally done with a third-party synthesis tool.

The number of NCPs generated should be equal to or less than the number specified during hybrid TK/LBIST IP generation.

Fault Simulation with a Tessent OCC (Dofile Flow)

Fault simulation with the Tessent OCCs is similar to a flow that uses custom OCCs. Tessent Shell reads in the complete gate level Verilog netlist with EDT and OCC, ICL/PDL, and TCD files for EDT and OCC instruments.

Tessent OCC core instances can be added during fault simulation.

Tessent Shell automatically adds two internal clocks for each OCC instance:

- A pulse in capture clock at the output of shift register clock gater (cgc_SHIFT_REG_CLK/clkg).
- A programmable capture clock at the output of the programmable clock gater (cgc_CLK_OUT/clkg) at the output of the mux [tessent_persistent_cell_clock_out_mux/Z].

Provide NCPs that refer to these internal clocks. If you have already added internal clocks in the same location as the tool would, then Tessent Shell tool recognizes this and does not add duplicate clocks. The Design Rule Check R18, which validates whether OCC control register is part of scan chains, is bypassed in LBIST mode. Clock control definitions are not generated in the LBIST mode. The output core description TCD file includes the parameters that were specified for the OCC instruments used in this run.

In the LBIST mode dofile, an internal user-PI is added for the LBIST capture enable signal and constrained to 1. For fault simulating chain test patterns, you manually change this constraint to 0.

Considerations When Only Using One NCP

When you are using only one NCP, you must manually create the NCP description because the NcpIndexDecoder, which usually creates the named capture procedure description, cannot be used for a single NCP. The OCC can have either an external or internal static clock control. During fault simulation, specify the name of the NCP as you would when you have more than one NCP.

You can use the create_capture_procedures command to create an NCP description in the tool instead of reading a manually-created description from a file. This user-created NCP should reflect the waveform that you also provide. For external static clock control, the waveform could be constant values provided on the OCC clock_sequence input pins by the netlist. For internal static control, this could be the value loaded into the OCC internal clock_sequence TDR.

For the internal static clock control, load the clock sequence corresponding to the user-created NCP through the ICL network. For Tessent OCCs, you can do this by using the clock_sequence core instance parameter. For the external static clock control, connect the Tessent OCC's clock sequence pins to constant values that generate the required NCP clock waveforms.

Pattern Generation with a Tessent OCC (Dofile Flow)

Pattern generation is performed through pattern specification.

The presence of Tessent OCC is identified through the ICL attribute on the OCC modules and is matched with TCD description for the OCC. When using Tessent OCCs, the pattern

specification processing automatically calls the OCC setup iProc with the parameter values that were used during fault simulation.

Observation Scan Technology Dofile Flow

There are observation scan considerations when you are using the legacy hybrid TK/LBIST dofile flow.

Test Point and Scan Insertion

Perform test point insertion as usual, with the addition of the following option to trigger observation scan logic insertion:

```
set_test_point_analysis_options -capture_per_cycle_observe_points on
```

By default, the tool creates a pin named “capture_per_cycle_static_en” if you haven’t previously named the enable signal for observation scan OPs using the add_dft_signals command. Optionally, you can assign your own name to the pin with the following command:

```
set_test_point_insertion_options -capture_per_cycle_en path_name
```

Specify these commands prior to analyze_test_points. Perform X-bounding and wrapper analysis as usual. The number of observe points is controlled the same way as without observation scan by using the -observe_points_number or -total_number options.

In the same session, perform scan insertion as usual. You must perform test point insertion and scan insertion together.

Note

When using the skeleton hybrid TK/LBIST dofile flow, in which you insert IP prior to performing test point and scan insertion, you must specify the “set_test_point_insertion_options -capture_per_cycle_en” command to identify the observation scan output pin on the LogicBIST controller. For example: lbist_i/capture_per_cycle_dynamic_en.

IP Generation

The set_dft_enable_options command supports the pin type capture_per_cycle_static_en for observation scan. This pin is required for observation scan.

```
set_dft_enable_options -type capture_per_cycle_static_en
    -pin_name name [ -intercept {on | off} ]
```

For -pin_name, specify the name you defined with the set_test_point_insertion_options -capture_per_cycle_static_en option.

In addition, the command supports the -intercept switch, which is only available when the pin type is capture_per_cycle_static_en.

Fault Simulation

In the generated fault simulation dofile, fault simulation for observation scan is indicated by an “on” value for capture_per_cycle_static_en. For example:

```
set_dft_enable_options -type capture_per_cycle_static_en \
-pin_name design_id_tessent_lbist_inst/capture_per_cycle_static_en
-value on
```

If you want to disable capture per cycle, set -value to off.

Pattern Generation

Perform pattern generation as usual.

If you need to perform pattern simulation mismatch debugging on observation scan cells, you can do so by enabling scan chain output monitoring. Set the iProc sim_monitor argument and specify monitor_scan_cells with the iCall. For example:

```
set sim_monitor 1
open_pattern_set lbist
   iCall run_lbist_normal lbist_clock 980 999 lbist 0 0 monitor_scan_cell  \
$sim_monitor
close_pattern_set
```

Example Tessent OCC TK/LBIST Flow (Dofile Flow)

In the OCC flow, you insert Tessent OCC and scan chains before generating the hybrid IP. Unlike the non-OCC hybrid flow, you also need to insert the LogicBIST NCP index decoder before generating your patterns and performing fault simulation.

This example demonstrates a gate-level flow using DFTSpecification for the OCC insertion. The flow uses the TSDB (Tessent Shell Database).

Generating and Inserting the Tessent OCC (Dofile Flow)

The first step to using a Tessent OCC in the TK/LBIST flow is creating the OCC, then subsequently insert the Tessent OCC into your design.

Procedure

1. Invoke Tessent Shell from the shell prompt.

   ```
   % tessent -shell
   ```

2. Set the Tessent Shell context to “dft” and specify a design identifier (gate1) for the current design.

   ```
   SETUP> set_context dft -no_rtl -design_identifier gate1
   ```

3. Read in the design netlist. For example:

   ```
   SETUP> read_verilog m8051_nonscan.v
   ```

4. Read the cell library. For example:

   ```
   SETUP> read_cell_library atpg.lib
   ```

5. Specify the top-level module of the current design. For example:

   ```
   SETUP> set_current_design
   ```

6. Set the design level. The physical_block level indicates the design is a block that is synthesized and laid out as an independent block. For example:

   ```
   SETUP> set_design_level physical_block
   ```

7. Read the OCC specific configuration data. For example:.

```
ANALYSIS> read_config_data-from_string {
  DftSpecification(m8051, gate1) {
    reuse_modules_when_possible: on;
    Occ {
      capture_trigger: capture_en;
      static_clock_control: external;
      Controller(clk) {
        clock_intercept_node: clk;
      }
    }
  }
}
```

8. Validate and process the content defined in the DftSpecification wrapper.

```
ANALYSIS> process_dft_specification
```

9. Synthesize the generated RTL.

```
ANALYSIS> run_synthesis
```

10. Exit the tool.

```
ANALYSIS> exit
```

Results

After OCC insertion, the created hardware is synthesized and replaces the inserted RTL modules with gate-level modules in the Tessent Shell Data Base (TSDB).

Inserting the Scan Chains (Dofile Flow)

During scan insertion, the top level shift clock reaches all the design scan cells, hence the entire design is treated as a single clock domain design.

Procedure

1. Invoke Tessent Shell from the shell prompt.

```
% tessent -shell
```

2. Set the Tessent Shell context to ‘dft’ and the design identifier, in this case to gate2.

```
SETUP> set_context dft -scan -design_identifier gate2
```

3. Reload the design created in the OCC insertion. For example:

```
SETUP> read_design m8051 -design_identifier gate1
```

4. Read the cell library. For example:

```
SETUP> read_cell_library atpg.lib
```

5. Set the current design. For example:

```
SETUP> set_current_design
```

6. Add signal constraints. For example:

```
SETUP> add_input_constraint RST -c0
```

7. The OCCs are automatically marked as hard macros.
8. Go to analysis mode.

```
SETUP> set_system_mode analysis
```

9. Perform X-bounding analysis to identify memory elements that might capture an unknown during Logic BIST. For example:

```
ANALYSIS> analyze_xbounding
```

10. Specify the scan chain mode. In this case the scan chain mode is edt and the chain count is 16. For example:

```
ANALYSIS>  add_scan_mode edt -chain_count 16
```

11. Run scan chain analysis to distribute the scan elements into new chains. For example:

```
ANALYSIS> analyze_scan_chains
```

12. Insert the test structures in to the netlist and stitch up the scan chains. For example:

```
ANALYSIS> insert_test_logic
```

13. Write the test procedure file and dofile that describe the chains created during scan insertion. For example

```
ANALYSIS> write_atpg_setup -replace
```

14. Exit the tool.

```
SETUP> exit
```

Generating the Hybrid TK/LBIST IP (Dofile Flow)

To generate the hybrid TK/LBIST IP, you read in the scan-inserted design with the Tessent OCC chains integrated with design scan cells. The Tessent OCC is thus part of compressed chains driven by the EDT decompressor.

Procedure

1. Invoke Tessent Shell from the shell prompt.

```
% tessent -shell
```

2. Set the Tessent Shell context to ‘dft -edt -logic_bist’. Specify the design identifier for, in this case it is gate3. For example:

```
SETUP> set_context dft -edt -logic_bist -design_identifier gate3
```

3. Reload the design created during scan insertion.For example:

```
SETUP> read_design m8051 -design_identifier gate2
```

4. Read the cell library. For example:

```
SETUP> read_cell_library atpg.lib
```

5. Set the current design. For example:

```
SETUP> set_current_design
```

6. Add the core instances. For example:

```
SETUP> add_core_instances –module m8051_gate1_tessent_occ
```

7. Setup up scan and constrain the inputs as required. The dofile and procedure were created by the write_atpg_setup command. For example:

```
SETUP> dofile scan_setup.dofile
SETUP> tessent_scan_setup edt
```

8. Setup the EDT IP. For example:

```
SETUP> set_edt_options -location internal -channels 1
```

9. Setup the LBIST controller. For example:

```
SETUP> set_lbist_controller_options -max_shift 100 -max_capture 3 \
  -max_pattern 100000
SETUP> set_lbist_controller_options -capture_procedures 2
SETUP> set_lbist_pins clock REFCLK
SETUP>set_lbist_pins scan_en scan_en
SETUP>set_lbist_pins xbounding_en lbist_en
SETUP>set_clock_controller_pins capture_procedure_index \
  -no_connection
```

10. Go to analysis mode.

```
SETUP> set_system_mode analysis
```

11. Write the IP into the Tessent Shell Data Base (TSDB). For example:

```
ANALYSIS> write_edt_files -tsdb -replace
```

12. Run synthesis. For example:

```
ANALYSIS> run_synthesis
```

13. Exit the tool.

```
SETUP> exit
```

Synthesizing and Inserting the LBIST NCP Index Decoder (Dofile Flow)

You can use Tessent Shell to read the synthesized gate-level netlist and identify the clock usage in the design.

The "report_clock_domains –compatible_clocks –details" command provides information that can be used to design effective NCP sequences for LogicBIST.

Procedure

1. Invoke Tessent Shell from the shell prompt.

   ```
   % tessent -shell
   ```

2. Set the Tessent Shell context to 'dft'.

   ```
   SETUP> set_context dft -no_rtl -design_identifier gate4
   ```

3. Read in the design. For example:

   ```
   SETUP> read_design m8051 -design_identifier gate4
   ```

4. Read the cell library. For example:

   ```
   SETUP> read_cell_library atpg.lib
   ```

5. Set the current design. For example:

   ```
   SETUP> set_current_design
   ```

6. Set the design level. For example:

   ```
   SETUP> set_design_level physical_block
   ```

7. Add the OCC instances. For example:

   ```
   SETUP>  add_core_instances -module m8051_gate1_tessent_occ
   ```

8. Read in the NCP index decoder specification. For example:

```
SETUP> read_config_data -from_string {
  DftSpecification(m8051, gate4) {
    LogicBist {
      NcpIndexDecoder {
        Ncp(pulse_once) {
          cycle(0): m8051_gate1_tessent_occ;
        }
        Ncp(pulse_twice) {
          cycle(0): m8051_gate1_tessent_occ;
          cycle(1): m8051_gate1_tessent_occ;
        }
      }
    }
  }
}
```

9. Process the NCP index decoder specification. For example:

```
SETUP> process_dft_specification
```

10. Setup and run synthesis. For example:

```
SETUP> run_synthesis
```

11. Exit the tool.

```
SETUP> exit
```

Generating the EDT Patterns (Dofile Flow)

To switch between capture modes, you only need to change the OCC setup iProc parameters.

Procedure

1. Invoke Tessent Shell from the shell prompt.

```
% tessent -shell
```

2. Set the Tessent Shell context to ‘patterns -scan’.

```
SETUP> set_context patterns -scan -design_identifier gate4
```

3. Reload the EDT-inserted design. For example:

```
SETUP> read_design m8051
```

4. Read the cell library. For example:

```
SETUP> read_cell_library atpg.lib
```

5. Set the current design. For example:

```
SETUP> set_current_design
```

6. Extract the ICL. For example:

```
SETUP> extract_icl
```

7. Add signal constraints. For example:

```
SETUP> add_input_constraint RST -c0
```

8. Add the core instances including the EDT core. For example:

```
SETUP> add_core_instances -module m8051_gate1_tessent_occ \
       -param {fast_capture_mode 1}
SETUP> add_core_instances -module m8051_gate3_tessent_edt_lbist
```

9. Set the fault type. For example:

```
SETUP> set_fault_type transition
```

10. Set other parameters as required by your design style. For example:

```
SETUP> set_output_masks on
SETUP> add_input_constraints -all -hold
SETUP> set_pattern_type -sequential 2
```

11. Change the system mode to analysis to run DRCs.

```
SETUP> set_system_mode analysis
```

12. Read in the procedure file and setup the capture options. For example:

```
ANALYSIS> read_procfile external_capture_options \
           external_capture.testproc
ANALYSIS> set_external_capture_options -capture_procedure \
          ext_fast_cap_proc
```

13. Create and write out the patterns. For example:

```
ANALYSIS> create_patterns
ANALYSIS> write_patterns edt_patt_fast_capture.v -verilog \
          -serial -replace
```

14. Exit the tool.

```
ANALYSIS> exit
```

Performing the LBIST Fault Simulation (Dofile Flow)

The final phase of the hybrid flow for OCC is to use Tessent Shell to perform the LBIST fault simulation.

Procedure

1. Invoke Tessent Shell from the shell prompt.

```
% tessent -shell
```

2. Set the Tessent Shell context to ‘patterns -scan’.

```
SETUP> set_context patterns -scan -design_identifier gate4
```

3. Reload the design. For example:

```
SETUP> read_design m8051
```

4. Read cell library to enable creation of some instances such as muxes. For example:

```
SETUP> read_cell_library ../data/atpg.lib
```

5. Set the current design. For example:

```
SETUP> set_current_design
```

6. Extract the ICL. For example:

```
SETUP> extract_icl
```

7. Add signal constraints. For example:

```
SETUP> add_input_constraints  RST -c0
```

8. Add the core instances. For example:

```
SETUP> add_core_instances -module m8051_gate1_tessent_occ
SETUP> add_core_instances -module m8051_gate3_tessent_edt_lbist
SETUP> add_core_instances -module m8051_gate3_tessent_lbist
```

9. Set the LBIST controller options. For example:

```
SETUP>dofile tsdb_outdir/instruments/    \
m8051_gate4_lbist_ncp_index_decoder.instrument/    \
m8051_gate4_tessent_lbist_ncp_index_decoder.dofile
SETUP> set_lbist_controller_options -capture_procedure \
        {clkseq1 40 clkseq2 40 clkseq3 10 clkseq4 10}
```

10. Change the system mode to analysis to run DRCs.

```
SETUP> set_system_mode analysis
```

11. Read in the NCP index decoder test procedure file for the TSDB instruments directory using the read_procfile command. For example:

```
read_procfile tsdb_outdir/instruments/    \
m8051_gate4_lbist_ncp_index_decoder.instrument/    \
m8051_gate4_tessent_lbist_ncp_index_decoder.testproc
```

12. Set capture options. For example:

```
ANALYSIS> set_external_capture_options -fixed 4
```

13. Add faults. For example:

```
ANALYSIS> add_faults
```

14. Specify the pattern count and add the faults. For example:

```
ANALYSIS> set_random_patterns 100
```

15. Perform the LBIST simulation.

```
ANALYSIS> simulate_patterns -source bist -store_patterns all
```

16. Exit the tool.

```
ANALYSIS> exit
```

Tessent OCC Dofile Examples

The examples in this section illustrate usage models for using a Tessent OCC in the hybrid TK/LBIST flow.

Clock Domain Analysis for NCP Decoder Generation

This example shows how to use Tessent Shell to perform clock domain analysis and identify NCPs required for LBIST test. It uses a small design that has the following clocks:

- NX1
- NX2
- NX3

The following dofile reports the clock domains and the percentage of faults in each of the domains. Since the netlist is non-scan, the dofile instructs Tessent Shell to treat the netlist as a full-scan design, using the “add scan groups dummy dummy” command. If the design were already scan-inserted, you would instead specify the actual test procedure file and scan chains.

```
set_context pattern -scan
read_verilog nonscan_netlist.v
read_cell_library atpg.lib
set_current_design
add scan groups dummy dummy
add clock 0 NX1
add clock 0 NX2
add clock 0 NX3
set system mode analysis
add_faults -all
report_statistics -clock_domains summary
report_clock_domains -compatible_clocks -details
```

The “Clock Domain Summary” section of report_statistics command’s output is shown below:

```
-------------------------------------------------
Clock Domain Summary    % faults      Test Coverage
                        (total)     (total relevant)
  ------------------   ----------   ----------------
  /NX1                    22.38%              0.00%
  /NX2                    74.70%              0.00%
  /NX3                     0.33%              0.00%
-------------------------------------------------
```

The output of report_clock_domains command is shown below:

```
//  No. | Clock Name  Domain | Clock Compatibility
// -----+--------------------+--------------------
//    1 | '/NX1' (1)       1 |    .
//    2 | '/NX2' (2)       2 | 437  .
//    3 | '/NX3' (56)      3 |   8 44   .
// --------------------------+--------------------
//                        No. |   1  2  3
//       . = Compatible (non-interacting) clock pair
// <number> = Incompatible (interacting) clock pair
// Compatibility analysis is based on same-edge clock interaction.
```

From the above reports, there are small number of interacting flops (8) between NX1 and NX3. The paths between NX1 and NX3 should be bounded, which can be accomplished by an SDC file that describes all paths between these clock domains as false, as shown below (declared at the mux output):

```
create_clock occ_NX1/clock_out -period 40 -name NX1
create_clock occ_NX2/clock_out -period 40 -name NX2
create_clock occ_NX3/clock_out -period 40 -name NX3
set_false_path -from NX1 -to NX3
set_false_path -from NX3 -to NX1
```

The tool can now treat NX1 and NX3 as compatible clock domains and pulse them together, since all interactions between them are blocked during X-bounding. From the prior clock activity table, we can divide the design into two clock domains: NX2 and NX1_NX3. Consequently, the NX2 and NX1_NX3 domains can be tested for 75% and 25% of the test duration, respectively.

During fault simulation, the output of the report_clock_domains command shows that NX1 and NX3 are indeed compatible after X-bounding. The functional clocks referred earlier are numbered 7-9 in the output below.

```
//No.| Clock Name                Domain | Clock Compatibility
//---+--------------------------------+----------------------------
//  1| '/RST' (1)                   1 | .
//  2| '/refclk' (55)               1 | .   .
//  3| '/shift_clock'(56)           1 | .   .   .
//  4| '/tck' (57)                  1 | .   .   .   .
//  5| '/edt_clock' (64)            1 | .   .   .   .   .
//  6| '/edt_lbist_int_clock' (73)  1 | .   .   .   .   .   .
//  7| '/NX3' (75)                  1 | .   .   .   .   .   .   .
//  8| '/NX2' (76)                  2 | 9   .   .   .   .   .  36   .
//  9| '/NX1' (77)                  1 | .   .   .   .   .   .   . 437  .
// ---------------------------------+----------------------------
//                                No. | 1   2   3   4   5   6   7   8  9
//        . = Compatible (non-interacting) clock pair
// <number> = Incompatible (interacting) clock pair
```

A sample DftSpecification for this design is as follows:

```
LbistNcpIndexDecoder {
  LbistNcpIndex: m8051_lbist_i/ncp;
  Ncp(NX1_NX3_single_pulse) {
    cycle(0): occ_NX1, occ_NX3;
  }
  Ncp(NX2_single_pulse) {
    cycle(0): occ_NX2;
  }
  Ncp(NX1_NX3_double_pulse) {
    cycle(0): occ_NX1, occ_NX3;
    cycle(1): occ_NX1, occ_NX3;
  }
  Ncp(NX2_double_pulse) {
    cycle(0): occ_NX2;
    cycle(1): occ_NX2;
  }
}
```

The following NCPs are generated for the above DftSpecification:

```
procedure capture NX1_NX3_single_pulse =
  timeplate gen_tp1;
  cycle =
    force_pi;
    force occ_NX2_clock_out 0;
    force shift_clock 0;
    pulse occ_NX1_clock_out;
    pulse occ_NX3_clock_out;
  end;
end;
```

```
procedure capture NX2_single_pulse =
  timeplate gen_tp1;
  cycle =
    force_pi;
    force occ_NX1_clock_out 0;
    force occ_NX3_clock_out 0;
    force shift_clock 0;
    pulse occ_NX2_clock_out;
  end;
end;

procedure capture NX1_NX3_double_pulse =
  timeplate gen_tp1;
  cycle =
    force_pi;
    force occ_NX2_clock_out 0;
    force shift_clock 0;
    pulse occ_NX1_clock_out;
    pulse occ_NX3_clock_out;
  end;
  cycle =
    pulse occ_NX1_clock_out;
    pulse occ_NX3_clock_out;
  end;
end;

procedure capture NX2_double_pulse =
  timeplate gen_tp1;
  cycle =
    force_pi;
    force occ_NX1_clock_out 0;
    force occ_NX3_clock_out 0;
    force shift_clock 0;
    pulse occ_NX2_clock_out;
  end;
  cycle =
    pulse_occ_NX2_clock_out;
  end;
end;
```

Clock Gating When Inserting OCC and Hybrid EDT/LBIST In Different Passes

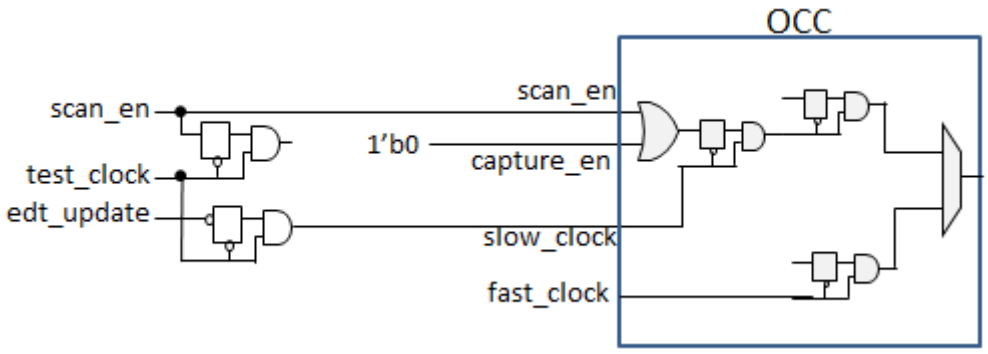
This example illustrates clock-gating with the hybrid TK/LBIST DFT insertion flow with OCC. This usage uses the TSDB flow to insert OCC with the edt_clock and shift_capture_clock DFT signals in the first insertion pass, followed by inserting EDT and LogicBIST in the second insertion pass.

Generate the edt_clock and shift_capture_clock signals by using the add_dft_signals -create_from_other_signals command as shown in the following dofile example.

```
set_context dft -no_rtl -design_identifier dft_signals
set_tsdb_output_directory tsdb_outdir
read_core_descriptions [lsort [glob design/mem/*.lib]]
read_cell_library ../tessent/adk.tcelllib
read_cell_library design/mem/mems.atpglib

read_verilog design/gate/elt1.v
set_current_design elt1
set_design_level physical_block
set_dft_specification_requirements -memory_test off -logic_test on

add_dft_signals scan_en test_clock edt_update \
      -source_node {scan_enable test_clock edt_update}
add_dft_signals edt_clock -create_from_other_signals
add_dft_signals shift_capture_clock -create_from_other_signals
report_dft_signals

add_clocks CLK_F300 -period [expr {1000.0/300.0}]

check_design_rules
set_system_mode analysis
set spec [create_dft_specification -sri_sib_list {occ}]
report_config_data $spec
set_config_value use_rtl_cells on -in_wrapper $spec
read_config_data -in_wrapper $spec -from_string {
    OCC {
        ijtag_host_interface : Sib(occ);
        static_clock_control : external;
        capture_trigger      : capture_en;
        Controller(clk_controller) {
            clock_intercept_node : CLK_F300;
            parent_instance : dft_inst;
        }
    }
}
report_config_data $spec
process_dft_specification
extract_icl
run_synthesis -startup_file ../prerequisites/techlib_adk.tnt/current/synopsys/
synopsys_dc.setup
```

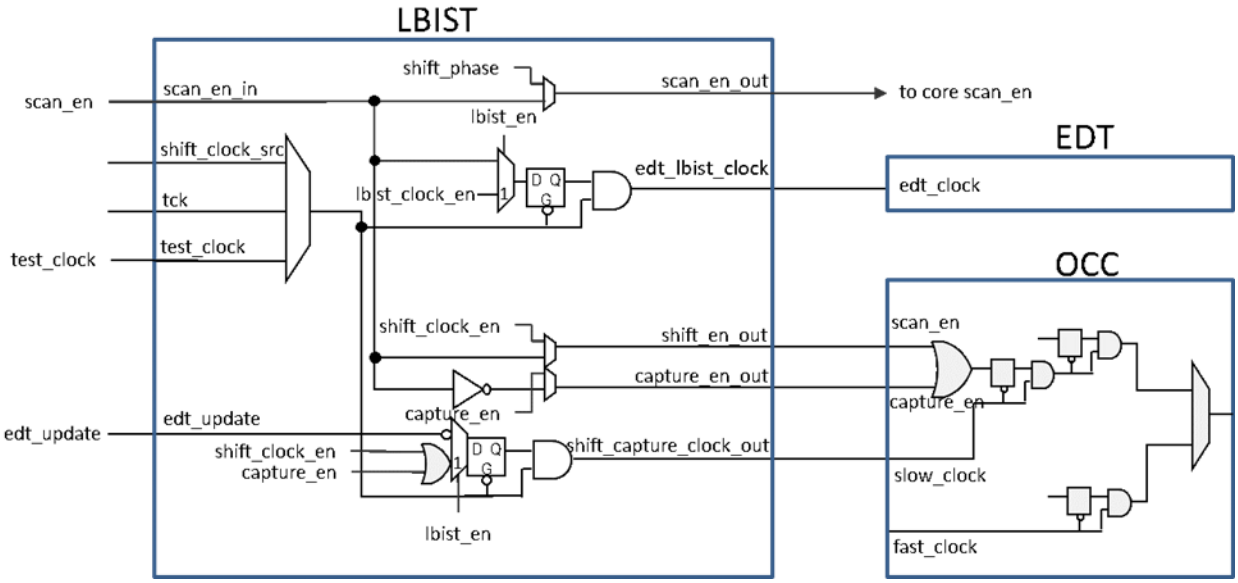
After the first pass, the OCC slow_clock is driven by the shift_capture_clock gater and the edt_clock gater has no fanout.

Figure A-6. Clock Gating With DFT Signals and OCC in the First Pass



After inserting EDT and LogicBIST in the second hybrid DFT insertion pass, the tool creates the circuit shown below. The edt_clock gater and shift_capture_clock gaters have been removed and their previous connections are now driven by their respective ports on the LogicBIST controller.

Figure A-7. Clock Gating With EDT and LogicBIST in the Second Pass



Note

This is also the resulting circuit when you first run the TSDB flow to insert the DFT signals and then the dofile flow to insert the LogicBIST controller. In the dofile flow for LogicBIST insertion, the source clock of the DFT signal gaters supply the test_clock to the LogicBIST controller.

File Examples for the Dofile Flow

This section provides various dofile file examples for the Hybrid TK/LBIST flow.

Synthesis Script Example

The following example excerpt shows the tool-produced script for synthesizing the EDT/LogicBIST logic.

```
#*************************************************************************
#  Synopsys Design Compiler synthesis script for created_edt.v
#  Tessent TestKompress version: 2013.1-snapshot_2013.01.29_06.01
#  Date:                         Tue Jan 29 11:21:40 2013
#*************************************************************************

# Bus naming style for Verilog
set bus_naming_style {%s[%d]}
set hdlin_ff_always_async_set_reset true

# Read input design files
read_file -f verilog created_edt.v
read_file -f verilog created_lbist.v


# Synthesize EDT IP
current_design my_core_edt

# Check design for inconsistencies
check_design

# Timing specification
create_clock -period 10 -waveform {0 5} edt_clock

# Avoid clock buffering during synthesis. However, remember
# to perform clock tree synthesis later for edt_clock
set_clock_transition 0.0 edt_clock
set_dont_touch_network edt_clock

# Avoid reset signal buffering during synthesis. However, remember
# to perform reset tree synthesis later for edt_reset
set_drive 0 edt_reset
set_max_fanout 1000 edt_reset

# Avoid assign statements in the synthesized netlist.
set_fix_multiple_port_nets -feedthroughs -outputs -buffer_constants

# Compile design
uniquify
compile -map_effort medium

# Report design results for EDT IP
report_area > created_dc_script_report.out
report_constraint -all_violators -verbose >> created_dc_script_report.out
report_timing -path full -delay max >> created_dc_script_report.out
report_reference >> created_dc_script_report.out

write -f verilog -hierarchy -o created_my_core_edt_gate.v

# Synthesize single chain mode logic
current_design my_core_single_chain_mode_logic
...
```

Timing Script Example

The following example excerpt shows a tool-produced timing script.

```
#*************************************************************************
#  Timing constraints for EDT/LBIST logic in module my_core during LBIST
#  mode
#  Tessent TestKompress version: 2013.1-snapshot_2013.01.29_06.01
#  Date:                         Tue Jan 29 11:21:41 2013
#*************************************************************************

# The following variables can be set to customize this script before
# sourcing this file:
#   lbist_clock_latency_min        [default=0]
#   lbist_clock_latency_max        [default=0]
#   lbist_clock_uncertainty_setup  [default=0]
#   lbist_clock_uncertainty_hold   [default=0]
#
#   tck_clock_latency_min          [default=0]
#   tck_clock_latency_max          [default=0]
#   tck_clock_uncertainty_setup    [default=0]
#   tck_clock_uncertainty_hold     [default=0]
#
#   lbist_pins_input_delay         [default=0]
#   lbist_pins_output_delay        [default=0]
#
#   reg_suffix                     [default=_reg]
#
#   reg_output                     [default=Q]
#

# Create lbist clock
proc create_lbist_clock {} {
   create_clock -period 40 -waveform {20 30} refclk

   global lbist_clock_latency_min lbist_clock_latency_max
lbist_clock_uncertainty_setup lbist_clock_uncertainty_hold
   set cmin [expr {[info exists lbist_clock_latency_min] ?
$lbist_clock_latency_min : 0}]
   set cmax [expr {[info exists lbist_clock_latency_max] ?
$lbist_clock_latency_max : 0}]
   set csetup [expr {[info exists lbist_clock_uncertainty_setup] ?
$lbist_clock_uncertainty_setup : 0}]
   set chold [expr {[info exists lbist_clock_uncertainty_hold] ?
$lbist_clock_uncertainty_hold : 0}]

   set all_lbist_clocks [list refclk]
   set_clock_latency -min $cmin $all_lbist_clocks
   set_clock_latency -max $cmax $all_lbist_clocks
   set_clock_uncertainty -setup $csetup $all_lbist_clocks
   set_clock_uncertainty -hold $chold $all_lbist_clocks
}
...
```

ICL Example

The following example shows an ICL output written by the tool.

```
//*************************************************************************
//  ICL script for module my_core
//
//  Tessent TestKompress version: 2013.1-snapshot_2013.01.29_06.01
//  Tue Jan 29 06:11:31 GMT 2013
//  Date:                         01/29/13 11:21:40
//*************************************************************************


Module my_core_lbist { // {{{
    TCKPort             tck;
    ClockPort           edt_clock;
    ClockPort           shift_clock_src;
    ToClockPort         lbist_clock      { Source shiftClkSelected_MUX;}
    ScanInPort          from_edt_scan_out;
    ScanInPort          lbist_scan_in;
    ScanOutPort         lbist_scan_out   { Source lbist_scan_out_ff; }
    DataOutPort         lbist_en         { Source bist_en; }
    ResetPort           sib_reset;
    SelectPort          sib_en;
    CaptureEnPort       sib_capture_en;
    ShiftEnPort         sib_shift_en;
    UpdateEnPort        sib_update_en;
    ToSelectPort  edt_sib_en { Source my_core_lbist_edt_sib_i.to_enable; }

    ScanInterface host {
       Port lbist_scan_in;
       Port lbist_scan_out;
    }
    ScanInterface client {
       Port from_edt_scan_out;
       Port lbist_scan_out;
       Port edt_sib_en;
    }

   Alias lbist_ctrl_sib      = my_core_lbist_sib_bist_registers_i.sib;
   Alias lbist_ctrl_signals_sib = my_core_lbist_sib_control_registers_i.sib;
   Alias bist_done           = bist_en { RefEnum YesNo;}

    Instance my_core_lbist_edt_sib_i Of my_core_lbist_sib {
        InputPort reset         = sib_reset;
        InputPort enable        = sib_en;
        InputPort scan_in       = lbist_scan_in;
        InputPort capture_en    = sib_capture_en;
        InputPort shift_en      = sib_shift_en;
        InputPort update_en     = sib_update_en;
        InputPort tck           = tck;
        InputPort from_scan_out = from_edt_scan_out;
    }

    //
    // Bist registers
    //
    ScanRegister capture_phase_size[1:0] {
        ScanInSource    my_core_lbist_edt_sib_i.scan_out;
    }
    ScanRegister shift_clock_select[1:0] {
        ScanInSource    capture_phase_size[0];
```

```
            ResetValue      2'b01;
        }
        ScanRegister warmup_pattern_count[8:0] {
            ScanInSource    shift_clock_select[0];
        }

    ...
```

Appendix B
Low Pin Count Test Controller

The hybrid flow supports low pin count test (LPCT) type-1 and type-2 controllers but not type-3 LPCT controllers.

Low Pin Count Test Controller Overview

The LPCT controller internally generates the EDT clock, and, when operating in LBIST mode, the tool modifies the controller to generate a free-running clock to enable using EDT clock for LBIST test. The LBIST controller's shift_clock_src and tck clock inputs are not impacted by using LPCT.

For hybrid IP generation and insertion, you specify the scan enable and TAP controller pins using the set_lbist_pins command. Similarly, for LPCT controller generation and insertion, you specify the scan enable and TAP controller pins when using type-2 LPCT using the set_lpct_pins command, as follows:

set_lpct_pins scan_en

When using LPCT with hybrid IP, the scan_enable signal using the set_lbist_pins and set_lpct_pins are handled as follows:

- When only LPCT scan enable is specified, the value is also used for the LBIST controller.
- When only LBIST scan enable signals are specified, the first specified scan enable is used for the LPCT controller.
- When neither LPCT nor LBIST scan enable signals are specified, the tool issues an error message.
- For type-2 LPCT, the tool ignores the scan enable signals for the LBIST controller and instead uses the scan enable generated by the LPCT controller.

Similarly, the TAP pins for the LBIST and LPCT controllers are handled as follows:

- When either LBIST or LPCT pins only are specified for tck, active-high test_logic_reset, shift_dr, capture_dr and update_dr signals, they are also used for the other controller.

- When both LPCT and LBIST pins are specified, they are both used for connecting to the respective test logic IP.
- When neither LPCT nor LBIST pins are specified, default pin names are used for sharing between both controllers.
- The LBIST tap_instruction_decode signal is not shared with LPCT test_mode signal because these are controlled by separate TAP instructions or internal test logic.

The hybrid flow supports Controller Chain Mode (CCM) with both type-1 and type-2 LPCT controllers. You can use CCM with either edt_clock or tck with type-1 LPCT. The type-2 LPCT only supports the edt_clock option. This is because tck is used to operate the TAP controller to generate the scan enable, which disturbs the controller logic scan cells due to these extra tck pulses in pre-shift and post-shift cycles.

When using IJTAG with TAP controller, you must preserve the state of the IJTAG network using the following command:

```
set_ijtag_retargeting_options -test_setup_network_end_state keep
```

This requirement applies to both the hybrid IP and EDT-only type-2 LPCT controller. Since the network state is kept instead of reset, the BIST controller setup registers should be explicitly changed to the Idle state to enable EDT pattern generation. Refer to the following example for a pattern generation dofile.

Type-2 LPCT Controller Example

The following DftSpecification generates a TAP controller, excluding boundary scan cells for the primary IO pins. Three user TAP instructions are synthesized.

- An LPCT instruction to generate the LPCT test_mode signal and provide access to EDT channels using TDI/TDO.
- An LBIST instruction to provide the IJTAG scan interface for the LBIST controller.
- A STATIC instruction that generates a TDR for controlling OCC and EDT static ports.

```
DftSpecification(m8051, tap)  {
   use_rtl_cells: on;
   IjtagNetwork {
   HostScanInterface(dft) {
      Interface {
         tck: tck;
         tdo_en_polarity: active_high;
      }
      Tap(dft) {
         HostIjtag(lpct) {
         }
         HostIjtag(lbist) {
         }
         HostIjtag(static) {
           Tdr(static) {
             Interface {
               reset_polarity: active_high;
             }
             DataOutPorts {
               port_naming: test_mode, fast_capture_mode,
                 capture_cycle_width[1:0], edt_bypass;
             }
           }
         }
      }
   }
 }
}

DftSpecification(m8051, occ) {
   use_rtl_cells: on;
   reuse_modules_when_possible: on;
   OCC {
      DefaultConnections {
        slow_clock: tck;
        StaticExternalControls {
          test_mode: m8051_tap_tessent_tdr_static_inst/test_mode;
          fast_capture_mode: m8051_tap_tessent_tdr_static_inst/
fast_capture_mode;
          capture_cycle_width:
             m8051_tap_tessent_tdr_static_inst/capture_cycle_width;
        }
      }
      capture_trigger: capture_en;
      static_clock_control: external;
      Controller(NX1) {
        clock_intercept_node: NX1g/Z;
      }
      Controller(NX2) {
        clock_intercept_node: NX2g/Z;
      }
   }
}
```

To configure the hybrid IP with a type-2 LPCT controller given the DftSpecification shown above, you would specify:

```
#
#EDT settings
set_edt_options -location internal -channel 1
set_edt_pins input 1 tdi
set_edt_pins output 1 tdo \
             m8051_tap_tessent_tap_dft_inst/host_lpct_from_so
set_edt_pins bypass - m8051_tap_tessent_tdr_static_inst/edt_bypass

#
#Type-2 LPCT settings
set_lpct_controller -generate_scan_enable on -tap_controller_interface on
set_lpct_pins clock tck
set_lpct_pins output_scan_en scan_en
set_lpct_pins reset - reset_inverter_dft/y
set_lpct_pins capture_dr - m8051_tap_tessent_tap_dft_inst/capture_dr_en
set_lpct_pins shift_dr - m8051_tap_tessent_tap_dft_inst/shift_dr_en
set_lpct_pins update_dr -  m8051_tap_tessent_tap_dft_inst/update_dr_en
set_lpct_pins atpg_enable - \
              m8051_tap_tessent_tap_dft_inst/host_lpct_to_sel

#
#Tk/Lbist hybrid settings
set_lbist_controller_options -max_shift 100 -max_capture 7 \
                             -max_pattern 100000 -capture_procedure 3
set_lbist_pins clock {- pll/pll_clock_0}
set_lbist_pins scan_en scan_en
set_lbist_pins tck tck
set_lbist_pins test_logic_reset {- reset_inverter_dft/y}
set_lbist_pins tap_instruction_decode {-
               m8051_tap_tessent_tap_dft_inst/host_lbist_to_sel}
set_lbist_pins shift_dr {- m8051_tap_tessent_tap_dft_inst/shift_dr_en}
set_lbist_pins update_dr {- m8051_tap_tessent_tap_dft_inst/
update_dr_en}set_lbist_pins capture_dr {-
               m8051_tap_tessent_tap_dft_inst/capture_dr_en}
set_lbist_pins setup_shift_scan_in tdi
set_lbist_pins setup_shift_scan_out {tdo
               m8051_tap_tessent_tap_dft_inst/host_lbist_from_so}
set_dft_enable_options -type xbounding -pin_name xbnd_en
set_clock_controller_pins capture_procedure_index -no_connection
```

The mapping flow for EDT pattern generation setup using TCD is shown below.

```
read_core_descriptions {created_m8051_edt.tcd created_m8051_lpct.tcd}
add_core_instances -module {*_tessent_occ *_edt *_lpct}

add_input_constraint RST -c0
add_input_constraint tms -c0
add_input_constraint trst -c1
add_clocks 0 tck -pulse_always       ;#reference clock to pll
add_clocks 0 pll/pll_clock_0 -pin_name fast_clock -pulse_always
set_procfile_name ../data/scan_setup.testproc

#
#Load LPCT instruction to enable EDT channels access to tdi/tdo
iProcsForModule m8051
iProc load_lpct_instruction {} {
   iWrite m8051_lbist_i.bist_setup Idle
   iWrite m8051_tap_tessent_tap_dft_inst.instruction HOSTIJTAG_LPCT
   iApply
}
set_test_setup_icall "load_lpct_instruction" -append
set_ijtag_retargeting_options -compare_constant_capture_values off \
   -test_setup_network_end_state keep
```

Appendix C
EDT Pattern Generation for the Hybrid IP

The EDT technology within the hybrid IP functions as described in the *Tessent TestKompress Users Manual*. In addition, EDT mode contains functionality specific to the hybrid IP.

EDT Mode Initialization with IJTAG

IJTAG designs contain hardware components such as TDRs and SIBs, and an ICL network to provide access to IJTAG compatible instruments. The design should have either a TAP controller or top-level pins to operate the IJTAG circuitry. ICL describes the instruments and access network, and PDL describes the procedures for operating the instruments. The hybrid IP always includes IJTAG hardware and generates ICL and PDL (iProcs) for hybrid LBIST and EDT instruments.

Using iProcs is required for LBIST mode. Different LBIST pattern sets such as runtime programmable, hardware default, and diagnostics use different LBIST mode iProcs.

Using iProcs is optional for EDT mode with one exception. When EDT reset is not synthesized, the tool must use the EDT setup iProc to generate the initialization sequence for the chain masking register. In this case, the tool automatically provides the values for the EDT static control signals based on the iProc parameters. When iProcs are not used, the test procedures should provide the values required for EDT static control signals.

You can initialize the chain masking register either by adding an EDT reset signal to the hardware or by using IJTAG for seeding values into this register. By default, the tool initializes the chain masking register by using IJTAG. This reduces the need for extra hardware.

You can change the default behavior by synthesizing the EDT reset signal. Specify the following command during IP creation:

```
set_edt_options -reset asynchronous
```

The EDT Setup iProc

During the EDT and LogicBIST generation step, you use the write_edt_files command to generate several EDT files needed for subsequent flow steps, such as test pattern generation.

Among these files, the tool creates a PDL file that is linked to a generated ICL module. The PDL file includes one EDT iProc named “setup” in addition to several LogicBIST iProcs for LBIST mode.

The tool-generated EDT setup iProc contains parameters for EDT static signals (EDT bypass, EDT single-chain bypass, EDT low power and EDT dual configuration), the EDT reset signal, and serial-load initializing of chain mask registers. The parameters for the EDT static signals and EDT reset are present in the iProc only when the corresponding hardware is synthesized. You use the static EDT control signals as you would for non-hybrid EDT.

The default values for the iProc parameters are:

- edt_reset = on
- edt_bypass = off
- edt_single_bypass_chain = off
- edt_configuration = low compression configuration
- edt_low_power_shift_en = on when the EDT low-power hardware is synthesized and enabled during IP creation and off when it is disabled

Refer to “EDT IP Setup for IJTAG Integration” for more information.

Usage Examples

To add parameters to the setup iProc, specify iCalls with the set_test_setup_icall command in the EDT pattern generation dofile.

For the following usage examples, suppose you have a tool-generated setup iProc as shown in Example C-1:

Figure C-1. Example of Tool-Generated Setup iProc

```
iProcsForModule piccpu_edt
iProc setup {args} {
   if {[expr [llength $args]%2 != 0]} {
     display_message -error "Odd number of arguments. Expecting parameter
and value pairs."
      return -code error
   }

   set edt_reset 1
   set edt_configuration low_compression_cfg
   set edt_low_power_shift_en 1
   set edt_bypass 0
   set edt_single_bypass_chain 0
   set edt_chain_mask 1111111111111111
   set iwrite_chain_mask 0
   foreach {param value} $args {
      set param [string tolower $param]
      if {$param == "edt_reset"} {
         set edt_reset $value
      } elseif {$param == "edt_configuration"} {
         set edt_configuration $value
      } elseif {$param == "edt_low_power_shift_en"} {
         set edt_low_power_shift_en $value
      } elseif {$param == "edt_bypass"} {
         set edt_bypass $value
      } elseif {$param == "edt_single_bypass_chain"} {
         set edt_single_bypass_chain $value
      } elseif {$param == "tessent_chain_masking"} {
         foreach chain $value {
            if {$chain < 1 || $chain > 16} {
               display_message -error "Invalid chain index '$chain'. Must
be 1 to 16."
               return -code error
            }
            set pos [expr {[string length $edt_chain_mask]-$chain}]
          set edt_chain_mask [string replace $edt_chain_mask $pos $pos 0]
         }
         set iwrite_chain_mask 1
         continue
      } else {
         display_message -error "Invalid parameter '$param'. Valid
parameters are 'edt_reset', 'edt_configuration',
'edt_low_power_shift_en', 'edt_bypass', 'edt_single_bypass_chain' and
'tessent_chain_masking'."
         return -code error
      }
      if {$param == "edt_configuration"} {
         if {$value == "low_compression_cfg" || $value ==
"high_compression_cfg" || [string is boolean -strict $value]} {
            set edt_configuration $value
         } else {
            display_message -error "Invalid EDT configuration value
```

```
'$value'. Valid values are boolean, 'low_compression_cfg' and
'high_compression_cfg'."
            return -code error
         }
      } elseif {![string is boolean -strict $value]} {
         display_message -error "Invalid non-boolean value '$value' for
parameter '$param'."
         return -code error
      }
   }

   if {$edt_reset} {
      iWrite edt_reset on
      iApply
      iWrite edt_reset off
      iApply
   }
   if {![string is boolean -strict $edt_configuration]} {
      iWrite edt_configuration $edt_configuration
   } elseif {$edt_configuration} {
      iWrite edt_configuration high_compression_cfg
   } else {
      iWrite edt_configuration low_compression_cfg
   }
   if {$edt_low_power_shift_en} {
      iWrite edt_low_power_shift_en on
   } else {
      iWrite edt_low_power_shift_en off
   }
   if {$edt_bypass} {
      iWrite edt_bypass on
   } else {
      iWrite edt_bypass off
   }
   if {$edt_single_bypass_chain} {
      iWrite edt_single_bypass_chain on
   } else {
      iWrite edt_single_bypass_chain off
   }
   if {$iwrite_chain_mask} {
      iWrite bist_chain_mask 0b$edt_chain_mask
      iWrite bist_chain_mask_load_en 0b0
   }
   iApply
}
```

Example 1: EDT Reset

Suppose the EDT reset signal is asserted as part of test_setup—for example, through power-on-reset—and you do not want to pulse edt_reset in the EDT setup iProc. Specify the following command:

set_test_setup_icall {piccpu_edt_i.setup edt_reset off}

Example 2: Static Chain Masking

To support static chain masking, the tool uses the tessent_chain_masking parameter. The term "internally" distinguishes this masking, which is performed by the IP, from masking performed outside the IP by user-added chain masking.

The value of this parameter is a Tcl list of chain indexes, counted from 1 for the first chain connected to the specified EDT block.

The tool handles internally masked chains as follows depending on whether the edt_reset parameter is present in the setup iProc:

- Without edt_reset, always initialize the chain mask register.
- With edt_reset, initialize the chain mask register only when the parameter is explicitly specified. You can use an empty list ({}) to load the chain mask register to all 1's (unmasked).

Suppose the design has edt_reset and was pulsed in test_setup, but you want to serial load the mask chain register for EDT mode later in the setup iProc. Specify the following command:

```
set_test_setup_icall {piccpu_edt_i.setup edt_reset off tessent_chain_masking {}}
```

In the following example, assume that chains "chain1" and "chain2" are broken and need to be output masked for EDT. Also assume they are the first 2 scan chains in the design. Chain indices 1 and 2 corresponds to chains "chain1" and "chain2."

```
add_chain_mask chain1 chain2 -unload 0

set_test_setup_icall {piccpu_edt_i.setup tessent_chain_masking {1 2}}
```

Example 3: EDT Static Control Signals

In the following example, the first iCall generates patterns with EDT low power disabled and with a high compression configuration. The second iCall generates uncompressed EDT bypass patterns.

```
set_test_setup_icall {piccpu_edt_i.edt_low_power_shift_en off edt_configuration
high_compression_cfg}

set_test_setup_icall {piccpu_edt_i.edt_bypass on}
```

Appendix D
Interface Pins

The interface pins on the various hybrid IP hardware components enable you to control various aspects of the hybrid IP.

LogicBIST Controller Pins

The following table lists the pin names on the LogicBIST controller.

Table D-1. LogicBIST Controller Pins

Pin Name	Type	Description
capture_dr: ijtag_se	Input	capture_dr output pin from the TAP controller.
clock: shift_clock_src	Input	LogicBIST clock pins.
edt_lbist_clock	Output	Drive the edt_clock port on the EDT block(s) and supply the lbist_clock during LBIST operation.
edt_update_in	Input	Used by the shift_capture_clock gater duplicated inside the LBIST controller.
edt_update_out	Output	edt_update enable pin.
control_point_en: control_point_en	Output	Control point enable pins.
lbist_test_clock_out	Output	Output of shift_capture_clock and edt_lbist_clock.
mcp_bounding_en: mcp_bounding_en	Output	MCP bounding enable pins.
observe_point_en: observe_point_en	Output	Observe point enable pins.
scan_en: scan_en_in	Input	Scan enable pin and, optionally, the internal pin corresponding to the pad input.

Table D-1. LogicBIST Controller Pins (cont.)

Pin Name	Type	Description
scan_en: scan_en_out	Output	Scan enable pin and, optionally, the internal pin corresponding to the pad output.
setup_shift_scan_in: ijtag_si	Input	Seeding register input connection pin.
setup_shift_scan_out: ijtag_so	Output	Seeding register output connection pin.
shift_dr: ijtag_se	Input	shift_dr output from the TAP controller.
tap_instruction_decode: ijtag_sel	Input	Instruction decoder output connection.
tck: ijtag_tck	Input	JTAG TCK input.
test_en: test_en_in	Input	Test enable pins.
test_en: test_en_out	Output	Test enable pins.
test_logic_reset: ijtag_reset	Input	test_logic_reset output connection from the TAP controller.
update_dr: ijtag_ue	Input	update_dr output from the TAP controller.
x_bounding_en: x_bounding_en	Output	X-bounding enable pins.

Clock Controller Pins

The following table lists the pin names on the clock controller for connecting to your clock controller in the design.

Table D-2. Clock Controller Pins

Pin Name	Type	Description
capture_procedure_index: ncp	Output	NCP signal that identifies the NCP used for the current pattern.
capture_phase	Output	Indicates Capture BIST controller FSM is in capture state.
lbist_en: lbist_en	Output	LogicBIST enable pins.
scan_en: shift_phase	Output	Indicates BIST controller FSM is in shift state.
shift_clock_en: shift_clock_en	Output	Clock controller shift clock enable pins.
shift_clock: shift_clock_in	Input	Clock controller shift clock pins.
shift_clock: shift_clock_out	Output	Clock controller shift clock pins.
scan_en: shift_en_in	Input	Clock controller scan enable pins.

Table D-2. Clock Controller Pins (cont.)

Pin Name	Type	Description
scan_en: shift_en_out	Output	Clock controller scan enable pins.
shift_capture_clock: shift_capture_clock_out	Output	When issuing the "add_dft_signals edt_clock shift_capture_clock -create_from_other_signals" command during LBIST, supply the Tessent OCC slow_clock pins with a gated shift capture clock.
shift_capture_clock: shift_capture_clock_in	Input	During ATPG mode, shift_capture_clock_out provides gated shift capture clock or PI clock source, via this port.
capture_en: capture_en_in	Input	OCC capture enable pins.
capture_en: capture_en_out	Output	OCC capture enable pins.
edt_clock_en	Input	Supplied by the LPCT during EDT mode, this port is used by the clock gater that creates the edt_lbist_clock signal that drives the EDT controller(s).

EDT/LogicBIST Wrapper Pins

The following table lists the pin names for the LogicBIST-related pins on the EDT module.

Table D-3. EDT/LogicBIST Wrapper Pins

Pin Name	Type	Description
edt_clock	Input	Clock for shared EDT/BIST IP.
lbist_en	Input	Active high enable signal that indicates BIST mode of operation.
misr_accumulate_en	Input	Active high enable signal that enables MISR to compress input.
lbist_misr	Output	A parallel output bus that combines all the MISRs inside the EDT block. For example, if a design contains two 24-bit MISRs, then this pin is a 48-bit output. The tool does not automatically connect this signal to any design pins.
lbist_prpg_en	Input	Active high enable signal that enables decompressor to run as PRPG.
lbist_reset	Input	Synchronous reset signal from the LogicBIST controller.

Table D-3. EDT/LogicBIST Wrapper Pins (cont.)

Pin Name	Type	Description
ijtag_si	Input	Shift register input for LogicBIST register seeding.
ijtag_so	Output	Shift register output for LogicBIST register seeding.

Segment Insertion Bit Signals

The following table lists the IJTAG-style signals for controlling the SIBs inside the shared EDT/LogicBIST IP.

Table D-4. SIB Pins

Pin Name	Type	Description
sib_capture_en	Input	Active high signal that indicates TAP is in capture-DR state.
sib_en	Input	Active high enable SIB enable signal, connected to decoded LogicBIST instruction from TAP controller.
sib_reset	Input	Active low signal that indicates TAP is in test logic reset state.
sib_shift_en	Input	Active high signal that indicates TAP is in shift-DR state.
sib_update_en	Input	Active high signal that indicates TAP is in update-DR state.
tck	Input	1149.1 test clock input.

Appendix E
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 *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 Digital Industries Software 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.