SIEMENS EDA

Tessent™ BoundaryScan User's Manual

For Use With Tessent Shell

Software Version 2023.1
Document Revision 28

SIEMENS

About Siemens Digital Industries Software

Siemens Digital Industries Software is a global leader in the growing field of product lifecycle management (PLM), manufacturing operations management (MOM), and electronic design automation (EDA) software, hardware, and services. Siemens works with more than 100,000 customers, leading the digitalization of their planning and manufacturing processes. At Siemens Digital Industries Software, we blur the boundaries between industry domains by integrating the virtual and physical, hardware and software, design and manufacturing worlds. With the rapid pace of innovation, digitalization is no longer tomorrow’s idea. We take what the future promises tomorrow and make it real for our customers today. Where today meets tomorrow. Our culture encourages creativity, welcomes fresh thinking and focuses on growth, so our people, our business, and our customers can achieve their full potential.

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
28	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 2023
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

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 Tessent BoundaryScan

Boundary scan, also known as JTAG, is a design-for-testability (DFT) technique that facilitates the testing of interconnect circuitry on printed circuit boards and in multi-chip modules (MCMs). You can also test the devices on these boards and MCMs with boundary scan techniques. Boundary scan improves board-level testing and shortens the time required to develop manufacturing tests and diagnostics.

Note

JTAG is the Joint Test Action Group. This is the committee that formulated the IEEE 1149.1 standard that describes boundary scan. The most recent revision to the IEEE standard, made in 2013, is known as IEEE 1149.1-2013.

The IEEE 1149.6 specification adds support for differential and capacitively coupled pin testing. Tessent BoundaryScan fully supports IEEE 1149.1-2001 and IEEE 1149.6-2003. It does not support the extensions of newer versions of IEEE 1149.1 and IEEE 1149.6.

Benefits of Boundary Scan

In-circuit test of printed circuit boards has become less useful because of the prevalence of surface mount devices. Boundary scan provides the benefits of in-circuit testing without requiring physical access to the electrical network on the board.

Adding boundary scan logic enables you to detect the majority of board manufacturing process faults:

- Incorrect components
- Missing components
- Incorrectly oriented components
- Components with stuck, shorted, or open pins
- Failed wire bonds

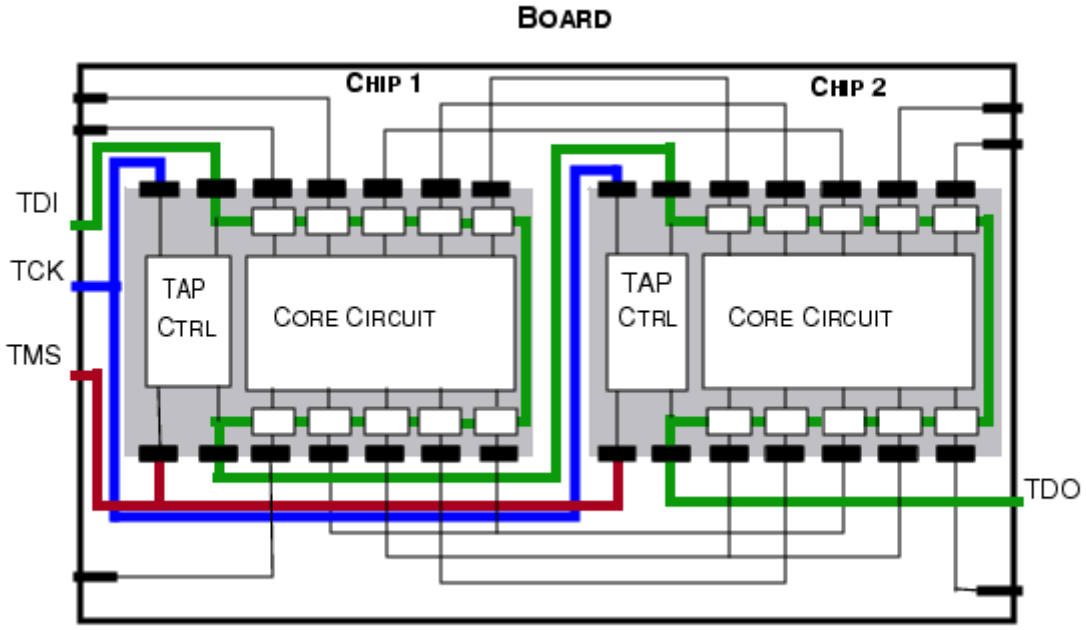
Although engineering costs may increase slightly because of the additional silicon and ports the boundary scan circuitry uses, implementing the IEEE 1149.1 standard can dramatically reduce design manufacturing costs.

Boundary Scan Overview

The primary use for boundary scan circuitry is in board-level testing. It can also control circuit-level test structures, such as built-in self-test (BIST) or internal scan. Add boundary scan circuitry to your design to create a standard interface for accessing and testing chips at the board level.

Figure 1-1 shows a board containing two chips with boundary scan circuitry.

Figure 1-1. Boundary Scan Chips on a Board



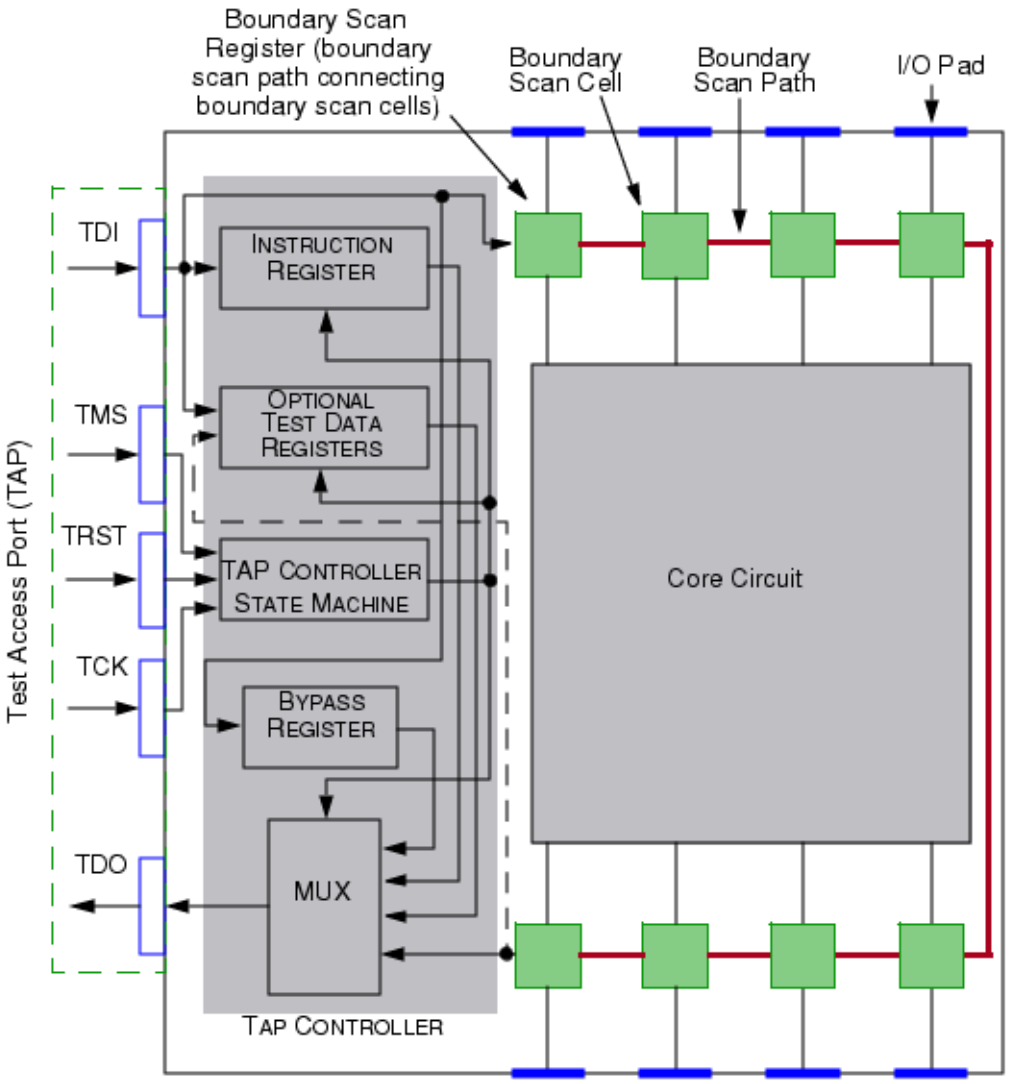
Using boundary scan on a board or MCM provides access to the chips' input and output ports by linking them together in a scan path. Data shifts along the scan path, starting at the board's test data in (TDI) port and ending at the board's test data out (TDO) port. The scan path connects all the board devices with boundary scan circuitry. The TDO of one chip feeds the TDI of the next, all the way around the board.

Test clock (TCK) and test mode select (TMS) connect globally to each boundary scan device in the board's scan path. Tessent BoundaryScan uses this configuration to enable you to test board interconnections, take a snapshot of system data, or test individual chips. The test access port (TAP) controller is a state machine that controls the operation of the boundary scan circuitry.

Boundary Scan Architecture

Tessent BoundaryScan inserts several structures in the chip architecture. Some of these structures also require additions to the instruction set.

Figure 1-2. Boundary Scan Architecture



This simple boundary scan architecture contains the following components:

- **Core Circuit** — This is the application logic of the original design before the addition of boundary scan logic. This logic may include internal scan circuitry or internal scan ports to facilitate building the boundary scan path.

- **Boundary Scan Cells** — These contain memory elements for capturing data from the circuit, loading data into the circuit, and shifting data to the next cell in the scan path. The tool places boundary scan cells between the core circuit and each bidirectional, input, and two- or three-state output pin. The set of boundary scan cells comprises a parallel-in, parallel-out shift register that runs along the periphery, or boundary, of the original design.
- **Test Access Port (TAP)** — This is a set of signals that control the boundary scan operation. The TAP consists of at least four signals of the test bus. These include the test clock (TCK), test data input (TDI), test data output (TDO), and the test mode selector (TMS). Also shown is the optional, active low asynchronous test reset signal (TRST).
- **TAP Controller State Machine** — This is a finite state machine that controls the operation of the instruction and test data registers. The TAP controller's next state depends on its current state and the TMS signal's value at each clock pulse.
- **Boundary Scan Register** — This is the primary test data register. The boundary scan register is a virtual shift register. It consists of the individual boundary scan cells joined in a path that can load and unload input and output data for the circuit. The loading and unloading can be done either serially or in parallel.
- **Bypass Register** — This is a register that shortens the path between TDI and TDO to one cell when there is no need to test a particular device. This shortened path bypasses the chip, providing more efficient test data shifting to other devices in the chain.
- **Optional Test Data Registers**:
  - **Device Identification Register** — This register contains a device identification code or other code used to check that the board contains the correct chips.
  - **Data-Specific Registers** — These registers provide access to the chip's test support features, such as BIST and internal scan paths.
- **Instruction Register** — This register controls the boundary scan circuitry by connecting a specific test data register between the TDI and TDO pins. It controls the operation affecting the data in that register using a predefined set of instructions. Some instructions are mandatory, and others are optional.

  Mandatory Instructions:

  - **EXTEST** — This instruction tests circuitry external to the devices, such as board interconnect. EXTEST is the main test instruction for boundary scan testing.
  - **SAMPLE/PRELOAD** — This instruction takes data from the I/O pads of the chips and latches it into the boundary scan register during normal board operation.

Note

The IEEE 1149.1-2001 standard has redefined this instruction as two separate instructions (SAMPLE and PRELOAD). The tool uses the PRELOAD instruction, if it is available, to load the boundary scan register. If PRELOAD is not available, SAMPLE/PRELOAD is used.

- **SAMPLE** — This instruction takes data from the I/O pads of the chips and latches it into the boundary scan register during normal board operation.
- **PRELOAD** — This instruction loads test data into the boundary scan register before selecting another instruction.
- **BYPASS** — This instruction enables the bypass of chips. Bypassed chips contribute only a single scan flop to the chain instead of all boundary scan registers, reducing the number of shifts.

Optional Instructions:

- **INTEST** — This instruction tests a chip's internal circuitry by applying a test vector to the application logic and capturing the output response.
- **IDCODE** — This instruction connects the device identification register between TDI and TDO. Use IDCODE when the device identification register contains the device ID code. This code verifies that the chip belongs on the board.
- **USERCODE** — This instruction also connects the device identification register between TDI and TDO, but the information in the register is user-defined and provides more detail than the IDCODE information.

Restriction

Tessent Shell does not support the USERCODE instruction.

- **CLAMP** — This instruction forces static 1s or 0s on selected nodes to block interfering signals, or to create a testable situation.
- **HIGHZ** — This instruction forces a chip's output and bidirectional pins into a high-impedance state so that an in-circuit tester can test the chip without risking overdrive damage.
- **RUNBIST** — This instruction runs the circuit's internal BIST procedure.

Restriction

Tessent Shell does not support the RUNBIST instruction.

- **EXTEST_PULSE** — This instruction tests circuitry external to the chips for designs that have AC pins. This instruction is mandatory when using AC cell types. For more details, refer to the IEEE 1149.6 standard.

- **EXTEST_TRAIN** — This instruction is optional when using AC cell types. It operates similarly to EXTEST_PULSE but can use multiple TCK cycles. For more details, refer to the IEEE 1149.6 standard.

Embedded Boundary Scan

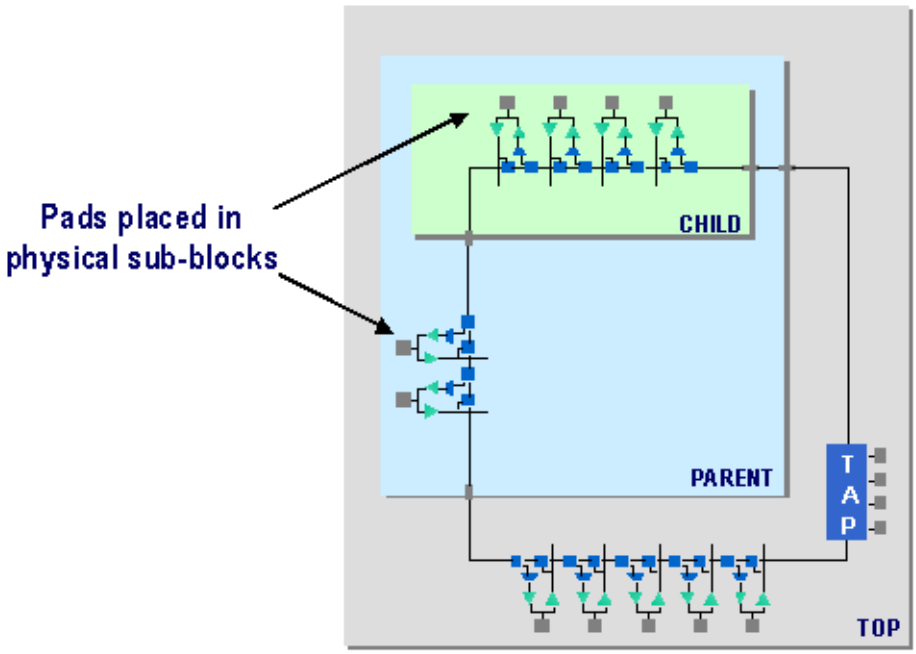
You can place chip I/O cells directly into cores to be closer to the logic that they service. This approach typically results in significant physical design benefits, including simplified signal routing and improved timing. The Tessent embedded boundary scan feature enables boundary scan cells to be integrated near their associated I/O cells within the core rather than at the top level of the chip.

You can add boundary scan cells to any core at any level of a design. See Figure 1-3 for a three-level example of an embedded boundary scan solution.

The embedded boundary scan capability of Tessent BoundaryScan automates the integration of the boundary scan cells and the verification of the resulting boundary scan segment within the core.

This approach maintains and extends the physical design benefits of placing I/O cells directly into cores. It enables a more efficient core reuse methodology because all design and DFT sign-off activity can occur at the core level.

Figure 1-3. Embedded Boundary Scan Implementation

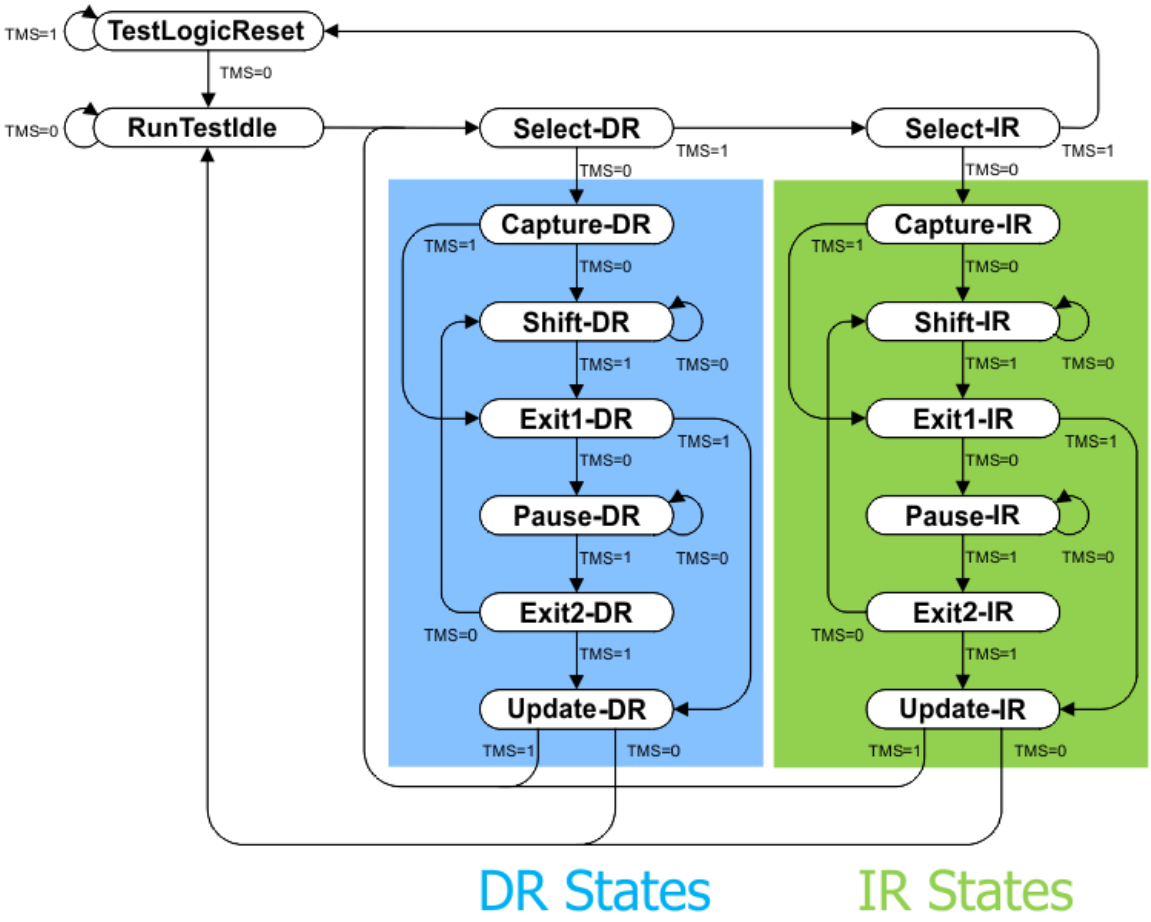


TAP Controller State Machine

The TAP controller is a synchronous finite state machine that controls the operation of the instruction and test data registers. The TAP controller's next state depends on its current state and the value of the TMS signal at the rising edge of each TCK clock pulse.

Figure 1-4 shows the finite state machine diagram for the TAP controller of an IEEE 1149.1 circuit.

Figure 1-4. TAP Controller Finite State Machine Diagram



The TMS signal (shown adjacent to each state transition) controls the state transitions on each rising edge of TCK. The rising edge of TCK also captures the TAP controller inputs.

The state encoding is as follows:

Table 1-1. State Encoding and TAP States

Encode Value	Corresponding TAP State
1111	TestLogicReset

Table 1-1. State Encoding and TAP States (cont.)

Encode Value	Corresponding TAP State
1100	RunTestIdle
0111	Select-DR
0110	Capture-DR
0010	Shift-DR
0001	Exit1-DR
0011	Pause-DR
0000	Exit2-DR
0101	Update-DR
0100	Select-IR
1110	Capture-IR
1010	Shift-IR
1001	Exit1-IR
1011	Pause-IR
1000	Exit2-IR
1101	Update-IR

These encodings appear on the state[3:0] output bus on the TAP.

Boundary Scan Insertion With Tessent BoundaryScan

Tessent BoundaryScan is the Siemens EDA boundary scan insertion tool. The tool creates and connects RTL-level boundary scan logic that is compliant with the IEEE 1149.1-2001 standard.

The Tessent BoundaryScan tool includes the following features:

- **Instruction Support —** Full support of the required IEEE 1149.1 standard instructions.
- **Extension Support —** Support of base extensions to IEEE 1149.1, such as the Device ID register.
- **Compliant Verilog —** Generation of Verilog (IEEE 1364-2001) code that is compliant with Questa® SIM, Synopsys Design Compiler, and other industry simulation and synthesis tools.

- **RTL-Level Boundary Scan Generation —** Insertion and connection of boundary scan circuitry at the RTL level, moving the generation of test circuitry earlier in the design flow.
- **Customized Boundary Scan —** Generation of default or user-customized boundary scan architectures.
- **Automatic Connection —** Automatic integration of boundary scan circuitry with internal scan logic.
- **Testbench Generation —** Generation of a testbench that enables testing the boundary scan circuitry after integration with the core application logic.
- **Test Vector Generation —** Generation of boundary scan test vectors in various common test data formats, including ASCII, binary, STIL, WGL, and others.
- **Setup File Generation —** Generation of ATPG setup files for designs with generated boundary scan circuitry.
- **Compliant BSDL —** Production of Boundary Scan Definition Language (BSDL) output compliant with the IEEE 1149.1-2001 standard.

Chapter 2
Getting Started With Tessent BoundaryScan

This chapter describes how to insert Tessent BoundaryScan within Tessent Shell and includes examples showing the most common scenarios and usages.

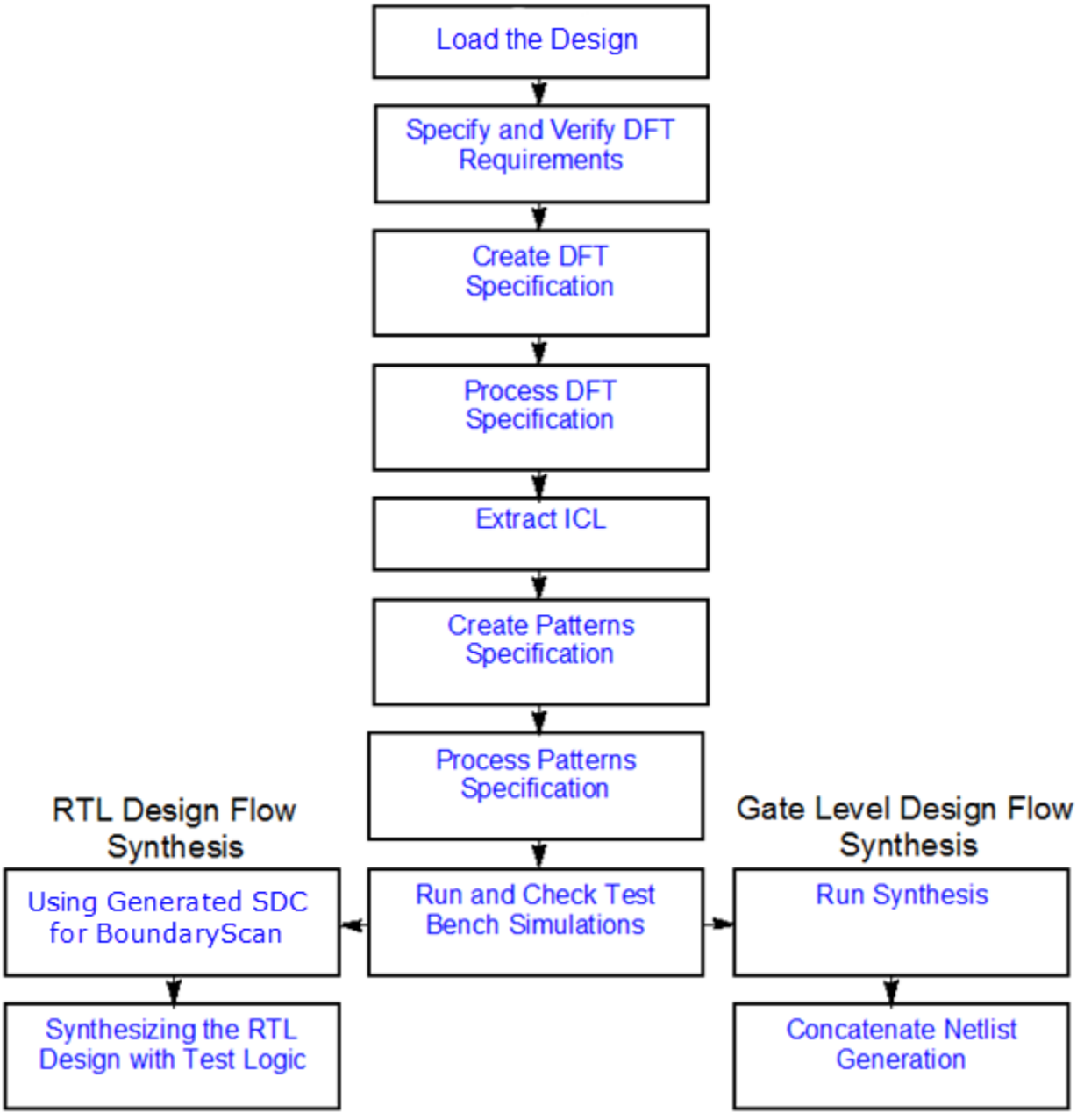
For a complete set of wrapper and property descriptions, refer to the "BoundaryScan" section of the "DftSpecification Configuration Syntax," "PatternsSpecification Configuration Syntax," and "DefaultsSpecification Configuration Syntax" sections in the *Tessent Shell Reference Manual*. The flow and steps are the same for inserting TAP, boundary scan, and embedded boundary scan. Each step describes any commands required for embedded boundary scan.

DFT Flow Using Tessent Shell

Tessent BoundaryScan has a basic, high-level flow sequence.

The following figure illustrates the high-level sequence of steps required to insert Tessent BoundaryScan into a design. Each step in the figure links to more detailed information about the design-for-test (DFT) flow, including examples.

Figure 2-1. Design Flow for Tessent BoundaryScan



Design Flow Prerequisites

To use the DFT design flow, you must have an RTL or gate-level netlist with IO pads inserted into the design.

For an RTL netlist, you must have the Tessent cell library or the pad library for the pad cells. For a gate-level netlist, you must have the Tessent cell library or the ATPG library for the standard cells, and the Tessent cell library for the IO pad cells.

Note

If the Tessent cell library for the IO pad cells is not available, Tessent Shell also natively supports the Tessent BoundaryScan-LV pad library.

Design Flow Dofile Example

The following example dofile shows how to set up a typical design flow.

Figure 2-1 on page 25 shows the design flow that the dofile follows.

Load the Design

```
set_context dft -rtl
read_cell_library ../library/adk_complete.tcelllib
read_verilog ../netlist/cpu_top.v
set_current_design cpu_top
```

Specify and Verify DFT Requirements

```
set_dft_specification_requirements -boundary_scan on
set_design_level chip
set_attribute_value tck_p -name function -value tck
set_attribute_value tdi_p -name function -value tdi
set_attribute_value tms_p -name function -value tms
set_attribute_value trst_p -name function -value trst
set_attribute_value tdo_p -name function -value tdo
set_boundary_scan_port_options ramclk_p -cell_options clock
set_boundary_scan_port_options reset_p -cell_options sample
check_design_rules
```

Create the DftSpecification

```
set spec [create_dft_specification]
report_config_data $spec
```

Process the DftSpecification

```
process_dft_specification
```

Extract ICL

```
extract_icl
```

Create the PatternsSpecification

```
create_patterns_specification
```

Process the PatternsSpecification

```
process_patterns_specification
```

Run and Check Testbench Simulations

```
set_simulation_library_sources -y ../library/verilog \
   -v ../library/pad_cells.v
run_testbench_simulations
check_testbench_simulations -report_status
```

Test Logic Synthesis

```
run_synthesis
```

Related Topics

Load the Design

Specify and Verify DFT Requirements

Create DFT Specification

Process DFT Specification

Extract ICL

Create Patterns Specification

Process Patterns Specification

Run and Check Testbench Simulations

Test Logic Synthesis

Pin Order File

Use a pin order file to map ports to package pins and define the boundary scan chain sequence.

Create the pin order file manually, optionally starting with the file created by the check_design_rules command. See "Pin Order Input File" in the *ETAssemble Tool Reference Manual* for this file's syntax.

Load the pin order file with the -pin_order_file option of the set_boundary_scan_port_options command.

Note

If you run set_boundary_scan_port_options command with the -pin_order_file option, the check_design_rules command does not create the file.

The first column of the pin order file lists the port names associated with the cells for the boundary scan chain. The order of the rows defines the ordering of the scan chain. The chain starts with the cell or cells created for the port listed in the first row of the pin order file, followed by the cell or cells created for the port listed in the second row, and so on.

The second column lists the pin IDs for mapping the device package pins in the BSDL file. You can use the -packed_pin_name option of the set_boundary_scan_port_options command to define the pin IDs if the pin order file is created automatically during check_design_rules.

The third column defines options for the ports. Do not use the pin order file to set these options. Instead, use the -cell_options option of set_boundary_scan_cell_options or BoundaryScanCellOptions in the BoundaryScan or the EmbeddedBoundaryScan wrapper to set them.

The fourth column specifies logical groups for the ports. Do not use the pin order file to define these groups. Instead, use LogicalGroups in the BoundaryScan or EmbeddedBoundaryScan wrapper to define them.

Related Topics

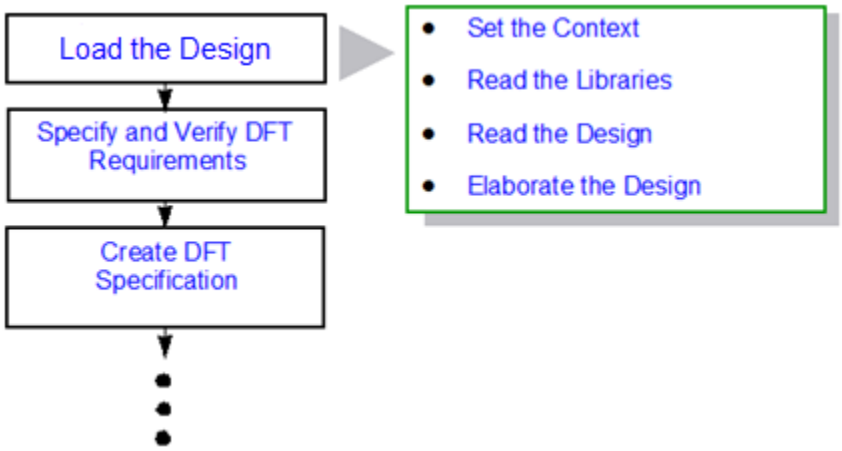
check_design_rules [Tessent Shell Reference Manual]

set_boundary_scan_port_options [Tessent Shell Reference Manual]

Load the Design

Loading the design is the first step in Tessent BoundaryScan insertion using Tessent Shell. The process consists of setting the correct context, reading libraries, reading the design, and elaborating the design.

Figure 2-2. Design Loading



Set the Context

In Tessent Shell, setting the context means two things. First and foremost, you must set the context to dft for boundary scan hardware to be created. Second, you must specify whether the design type to be read in is written in RTL. If so, you must specify the -rtl option. If the design to be read in is a gate-level Verilog netlist, you should specify the -no_rtl option.

When using the -no_rtl mode, a concatenated netlist is written out at the end of the dft insertion phase. In rtl mode, the file structure of the input design is preserved and only the modified design files are written out at the end of the dft insertion phase along with the newly created test IP. The netlist to be read in can be Verilog, VHDL, or mixed language.

If you used the Tessent Embedded Boundary Scan Flow, and Embedded Boundary Scan is integrated at the next level, you must open the Tessent Shell Data Base (TSDB) of the child (the embedded boundary scan's sub_block or physical_block) using the open_tsdb command. If you are using the same TSDB for both child and parent, you can reuse the TSDB (the default is tsdb_outdir), and you do not need to explicitly open the default TSDB because the existing content of the TSDB output directory is automatically visible to the tool. See the set_tsdb_output_directory command description for how to control the name and location of the TSDB output directory.

Examples

Example 1

The following example sets the context to dft and specifies that the design to be read in is written in RTL.

```
set_context dft -rtl
```

Example 2

The following example sets the context to dft and specifies that a gate-level netlist is to be read in.

```
set_context dft -no_rtl
```

Example 3

The following example opens a child's TSDB directory and therefore, exposes it at the parent level.

```
open_tsdb ../ebscan_tsdb_outdir
```

Read the Libraries

You can use the read_cell_library command to read in the library file for the pad IO macros that are instantiated in the design. If you are inserting Tessent BoundaryScan into an RTL netlist or design, reading the library for the pad IO macros is sufficient. If the Tessent cell libraries do not include the pad information, the legacy LV pad library format is natively supported by the read_cell_library command and can be used to augment the Tessent cell libraries with the pad information.

Examples

Example 1

The following example reads in the Tessent cell library file for the pad IO macros.

```
read_cell_library ../library/adk_complete.tcelllib
```

Example 2

The following example reads in the ATPG.lib files and the old pad library description when the Tessent cell libraries do not include the pad information.

```
read_cell_library ../library/atpg.lib
read_cell_library ../library/pad.library
```

Read the Design

In Tessent Shell, after setting the context and loading the required libraries, you can use the read_verilog command to read in the design.

Examples

Example 1

The following example reads in one netlist, which can be either RTL or gate level.

```
read_verilog ../netlist/cpu_top.v
```

Example 2

The following example reads in a Verilog file and directory of design files.

```
set_design_sources -format verilog -V ../design/top.v \
  -Y ../design -extensions {v gv}
```

Elaborate the Design

The next step in loading a design is to elaborate the design using the set_current_design command. The set_current_design command specifies the root of the design. If any module descriptions are missing, design elaboration identifies them. For Tessent BoundaryScan insertion, modules such as memory instances, PLL instantiations are not needed. You can specify the modules with the add_black_box -module command.

Example

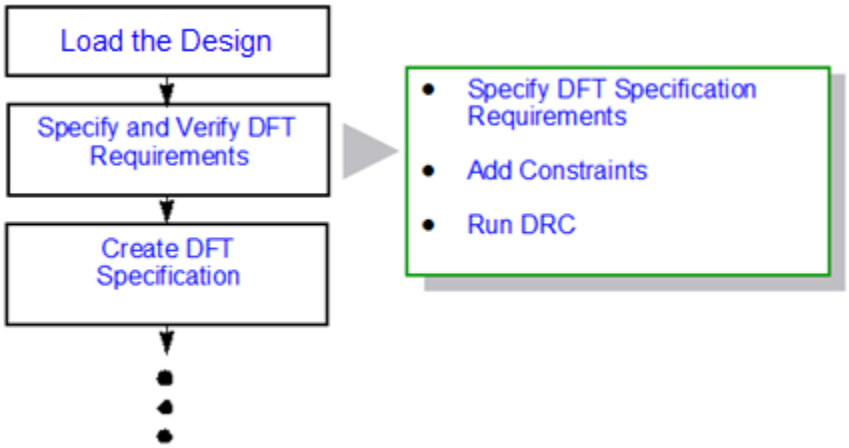
The following example shows how to use the set_current_design command.

```
set_current_design cpu_top
```

Specify and Verify DFT Requirements

The next step to insert Tessent BoundaryScan in Tessent Shell is to specify the DFT requirements, add constraints, and verify whether the DFT requirements specified are correct by running DRC (Design Rule Checking).

Figure 2-3. Specify and Verify DFT Requirements



Specify DFT Specification Requirements

To insert Tessent BoundaryScan, you must specify the DFT specification requirements with the set_dft_specification_requirements command. The DFT specification provides a template for logic and hardware you generate and insert into the design.

You specify chip with the set_design_level command when you are inserting boundary scan at the chip level. If you are using embedded boundary scan, you must specify either sub_block or physical_block. Also, you must use the set_boundary_scan_port_options command to specify the list of pad IO ports that need boundary scan, as shown in Example 2.

Examples

The following examples show how to specify DFT specification requirements for chip and sub_block levels.

Example 1

The following example shows how the boundary scan DFT specification requirements are specified and how the design is specified at the chip level.

```
set_dft_specification_requirements -boundary_scan On
set_design_level chip
```

Example 2

The following example shows, for embedded boundary scan, how to provide a Tcl list of pad IO ports that need boundary scan cells to be inserted. The example also shows how to set the design level to sub-block.

```
set_dft_specification_requirements -boundary_scan on

set_boundary_scan_port_options -pad_io_ports [list in1 in_diff_p in_\
  diff_n out1 out_diff_p out_diff_n clk A Y]

set_design_level sub_block
```

Note

To ensure the pad IO macros function properly when using embedded boundary scan, you may need to use the add_input_constraints command to constrain some ports to either a 1 or 0 value. Typically, these ports are driven to the proper values at the next higher level in the design.

Add Constraints

To insert Tessent BoundaryScan, four TAP pins (TDI, TCK, TMS, and TDO) must be available at the chip level and connected to pad IO macros. TRST, which is optional, can be an output pin of a power-up detector.

When all five TAP pins are connected to chip level pad IO macros, a 5-pin TAP is inserted. If TRST is connected to an internal power-on reset pin, a 4-pin TAP is inserted.

Note

If you need to specify internal pins for TDI, TMS, TCK, and TDO, and those pins connect to a chip level pad IO macro, you need two insertion passes. The TAP must be inserted in the first pass, and the boundary scan insertion is performed in a subsequent pass.

The TAP pins can be identified in the constraints section of your dofile or can be specified in the pin order file. Similarly, the power and ground pins can be identified in the constraints section or specified in the pin order file. If any special boundary-scan cell types are required, they can also be specified in the constraints section using the set_boundary_scan_port_options command.

If you are using embedded boundary scan and some ports must be constrained to a constant 1 or 0 value, use the add_input_constraints command. Typically, these values are properly driven at the next higher level in the design.

Examples

Example 1

The following example specifies the function and purpose of the five TAP pins (tck_p, tdi_p, tms_p, trst_p, tdo_p).

```
set_attribute_value tck_p -name function -value tck
set_attribute_value tdi_p -name function -value tdi
set_attribute_value tms_p -name function -value tms
set_attribute_value trst_p -name function -value trst
set_attribute_value tdo_p -name function -value tdo
```

The following specifies the power and ground pins.

```
set_attribute_value vdd* -name function -value power
set_attribute_value vss* -name function -value ground
```

The following specifies special boundary scan cell types.

```
set_boundary_scan_port_options ramclk_p -cell_options clock
set_boundary_scan_port_options reset_p -cell_options dont_touch
```

Example 2

The following example shows how to use the DefaultsSpecification wrapper to specify the five TAP pins if they are the standard port names used across all designs. The DefaultsSpecification wrapper is used when you run the create_dft_specification command as the next step.

```
DefaultsSpecification(group) { //Legal: company | group | user
  DftSpecification {
    IjtagNetwork {
      HostScanInterface {
        Chip {
          tck : tck_p;
          tdi : tdi_p;
          tdo : tdo_p;
          tms : tms_p;
          trst : trst_p;
        }
      }
    }
  }
}
```

You also can use the set_defaults_value command to set the DefaultsSpecification and the get_defaults_value command to see the specified value.

Example 3

The following example reads in the pin order file where the five TAP pins, power, and ground are specified.

```
set_boundary_scan_port_options -pin_order_file cpu_top.pinorder.my
```

Example 4

The following example shows how you can set some ports to a constant value if you are using embedded boundary scan.

```
add_input_constraints drive_strength[1] -C0
add_input_constraints drive_strength[2] -C0
add_input_constraints edriver1 -C1
add_input_constraints edriver2 -C1
```

Run DRC

The next step in Specify and Verify DFT Requirements is to run Design Rule Checking (DRC) to make sure all the constraints are correct. Once DRC is clean, Tessent Shell moves from the SETUP to the ANALYSIS prompt.

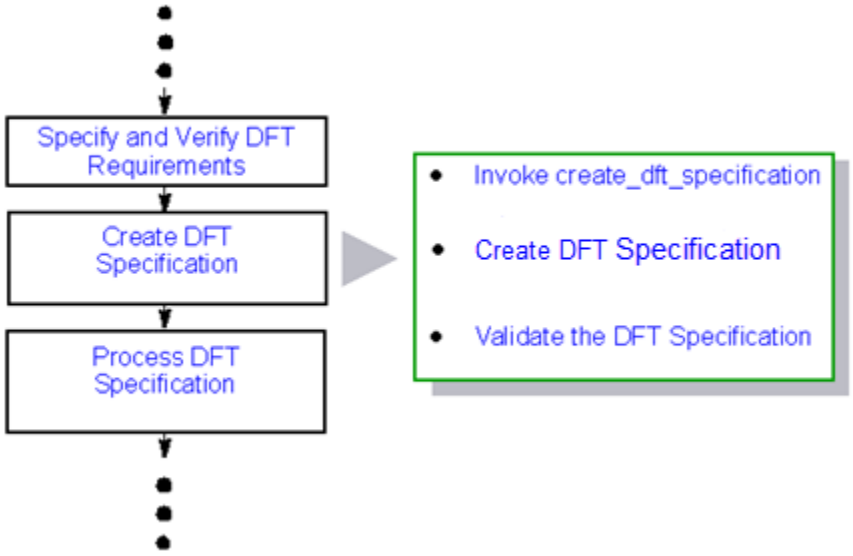
```
check_design_rules
```

Create DFT Specification

The next step in the design flow is to create a DFT specification.

The create_dft_specification command is used to create a default DFT specification based on the DFT requirements specified in the previous "Specify and Verify DFT Requirements" step. You can use the report_config_data command to report this default DFT specification. There are several methods available to edit or configure the DFT specification to meet your specific requirements.

Figure 2-4. Create DFT Specification



Invoke create_dft_specification

A DFT specification is automatically created using the create_dft_specification command. This DFT specification is stored in memory.

To report the DFT specification in memory, use the report_config_data command. The DFT specification uses IJTAG network infrastructure because this is the only supported method for incremental insertion passes. The IJTAG network is fully compliant with the 1149.1 IEEE standard. For further information about the Tessent IJTAG flow, refer to the *Tessent IJTAG User's Manual*.

To insert boundary scan into a preexisting TAP in your design, you must have an ICL for the TAP. The tool automatically uses this TAP to connect to the boundary scan chain if the TAP has ScanInterface host_bscan in the TAP ICL. You can also specify the host_bscan to connect to by

using the create_dft_specification -existing_host_bscan_scan_in command. The ICL for the preexisting TAP is read in automatically if it is in a location where the module description of the TAP is present. You also can use the read_icl command to read the ICL for a preexisting TAP. For requirements when using a preexisting TAP with boundary scan, refer to the "Requirements on a TAP to be usable for BoundaryScan" section in the *Tessent Shell Reference Manual*.

Examples

Example 1

In the following example, the DFT specification generated with the create_dft_specification is stored in a variable called dft_spec so that the variable can be used to report the DFT specification.

```
set dft_spec [create_dft_specification]
report_config_data $dft_spec
```

Example 2

The following example shows how to connect to the preexisting TAP when multiple TAPs are present. This only works if a ScanInterface host_bscan is in the ICL file for the preexisting TAP controller. If a single TAP controller is present, the tool automatically uses this controller, and you do not need to provide the host_bscan to connect to.

```
create_dft_specification -existing_host_bscan_scan_in \
  My_TAP_INST/fromBscan
report_config_data
```

My_TAP_INST in this example is the instance name of the preexisting TAP in the design, and fromBscan is the input port on the preexisting TAP where the boundary-scan chain needs to connect.

Configure the DFT Specification

There are two ways to configure the default DFT specification according to your requirements: using the Config Data Browser in Tessent Visualizer, or modifying the specification in memory. You do not need to edit the DFT specification if you want to use the default configuration.

Configure the DFT Specification With the Config Data Browser

One way to configure the DFT specification is to use the Config Data Browser in Tessent Visualizer. The edits you apply with this tool update the DFT specification in memory.

Prerequisites

- You have created a DFT specification as described in "Create DFT Specification" on page 36.

Procedure

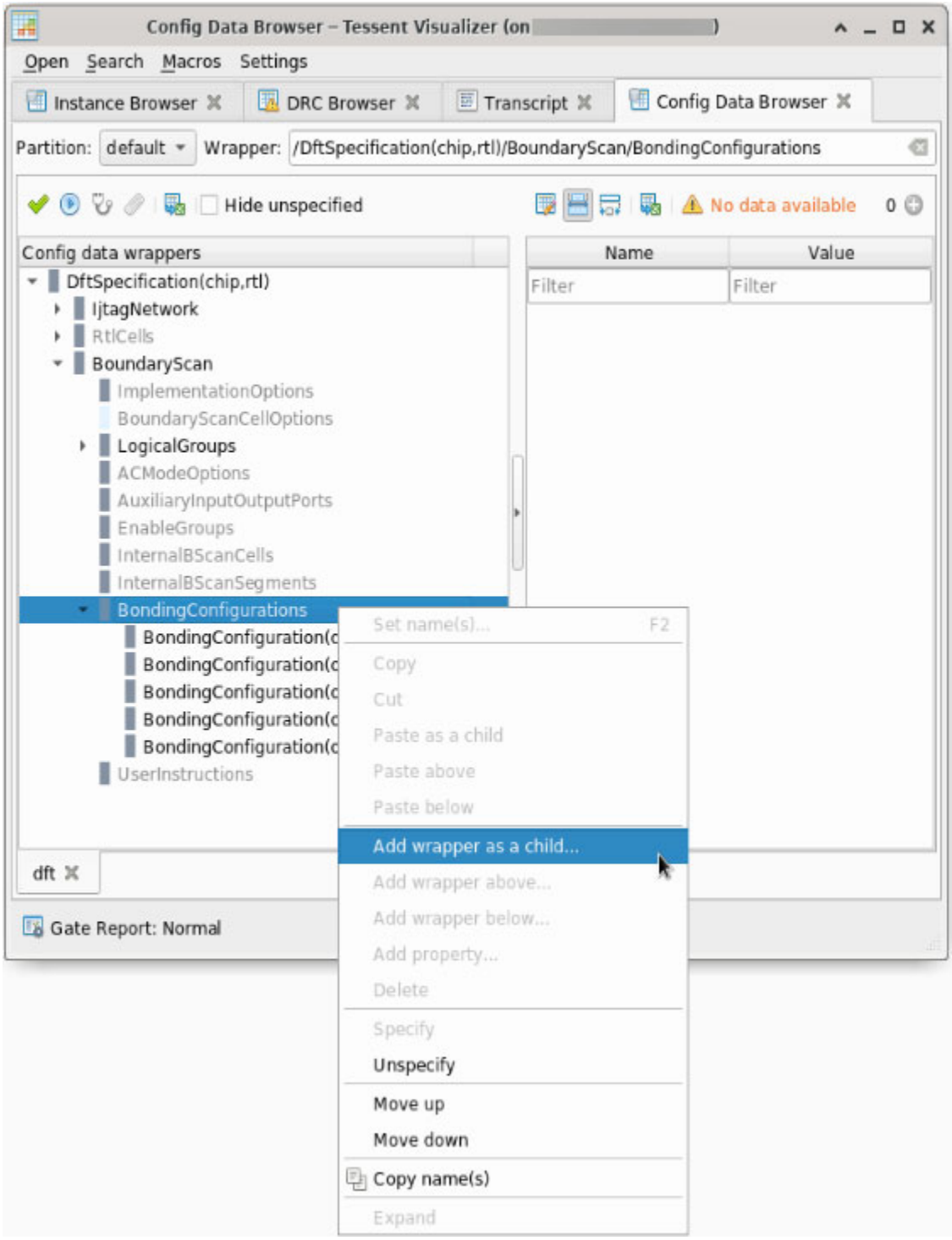
1. Open the Config Data Browser with the display_specification command.

   The Config Data Browser displays the DFT specification based on the DFT requirements specified in "Specify and Verify DFT Requirements" on page 32.

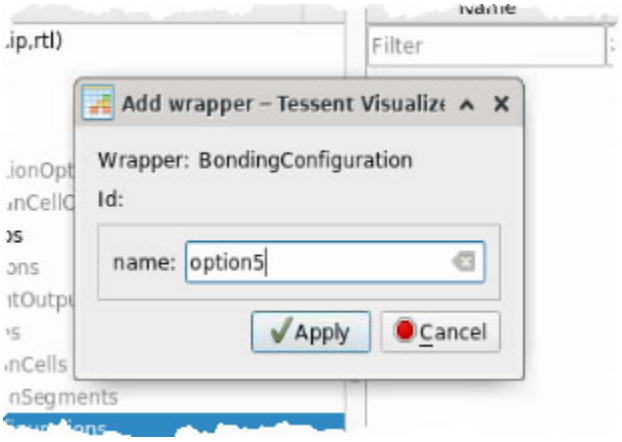
2. In the Configuration Tree pane, navigate through the tree to expose the BondingConfigurations wrapper.

3. Right-click the BondingConfigurations wrapper and choose **Add wrapper as a child**.

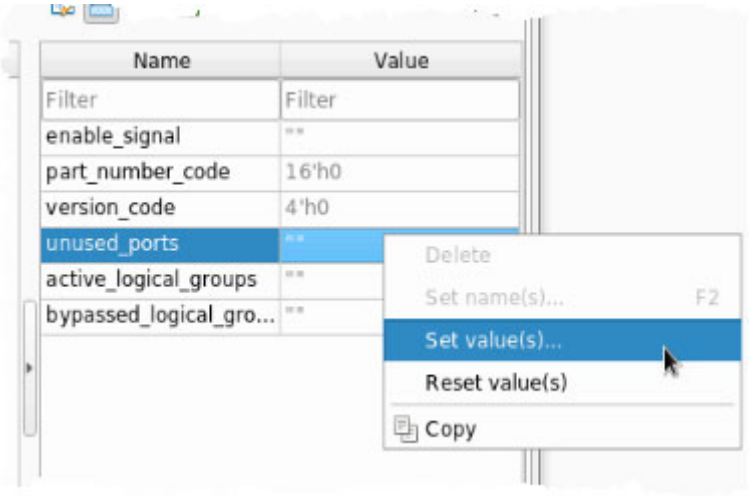
Figure 2-5. Editing the DFT Specification With the Config Data Browser



4. In the dialog box that is displayed, type a name for the new BondingConfiguration wrapper and click **Apply**.



5. Click a BondingConfiguration wrapper to show the Configuration Options pane
6. Add the ports to be excluded by right-clicking the unused ports filed and typing in the port names manually using the **Set value(s)** option.



7. Click **Apply** to update the Config Data Browser and the DFT specification in memory.
8. Repeat these steps to add additional modifications.

Results

This method demonstrates using the Config Data Browser to edit the DFT specification by adding multiple package bonding configurations that are defined by two BondingConfiguration wrappers.

To see the resulting configuration, use the report_config_data command.

```
> report_config_data $dft_spec

DftSpecification(car,gate) +{
  IjtagNetwork +{
    HostScanInterface(tap) +{
      Interface {
        tck : TCK;
        trst : TRST;
        tms : TMS;
        tdi : TDI;
        tdo : TDO;
      }
      Tap(main) +{
        HostIjtag(1) {
        }
        HostBscan {
        }
      }
    }
  }
  BoundaryScan {
    ijtag_host_interface : Tap(main)/HostBscan;
    BondingConfigurations {
      BondingConfiguration(default) {
      }
      BondingConfiguration(bonding1) {
        unused_ports : COORD2, COORD3, D1[2], D1[1], EN0, D1[0], D1[3];
      }
    }
  }
}
```

Examples

Using read_config_data, you can effectively cut and paste the manually entered Config Data Browser edits into the dofile as shown in the example below. Then for subsequent runs, the configuration edits are already present in the dofile, making the process repeatable with scripting.

```
read_config_data -in $dft_spec/BoundaryScan -from_string {
  BondingConfigurations +{
    BondingConfiguration(default) {
    }
    BondingConfiguration(bonding1) {
      unused_ports : COORD2, COORD3, D1[2], D1[1], EN0, D1[0], D1[3];
    }
  }
}
```

Related Topics

Configure the DFT Specification in Memory

Configure the DFT Specification in Memory

You can configure the DFT specification with the commands add_config_element and set_config_value. With this method, the dofile contains the commands and the modifications introduced, making the process repeatable.

Examples

The following examples show how to modify the DFT specification that is loaded in memory using the editing commands.

Example 1

The following example adds multiple package bonding configurations for Tessent BoundaryScan. The report_config_data command shows the DFT specification created.

```
> set spec [create_dft_specification]
> report_config_data $spec

DftSpecification(car,gate) {
  IjtagNetwork {
    HostScanInterface(tap) {
      Interface {
        tck : TCK;
        trst : TRST;
        tms : TMS;
        tdi : TDI;
        tdo : TDO;
      }
      Tap(main) {
        HostIjtag(1) {
        }
        HostBscan {
        }
      }
    }
  }
  BoundaryScan {
    ijtag_host_interface : Tap(main)/HostBscan;
  }
}
```

You can add the necessary BondingConfiguration wrappers and then use the report_config_data command to see how the DFT specification was updated.

```
> add_config_element $spec/BoundaryScan/BondingConfigurations
> add_config_element $spec/BoundaryScan/BondingConfigurations/\
    BondingConfiguration(default)

> add_config_element $spec/BoundaryScan/BondingConfigurations/\
    BondingConfiguration(bonding1)

> set_config_value $spec/BoundaryScan/BondingConfigurations/\
    BondingConfiguration(bonding1)/\
    unused_ports {COORD2 COORD3 D1[2] D1[1] EN0 D1[0] D1[3]}

> report_config_data

DftSpecification(car,gate) +{
  IjtagNetwork +{
    HostScanInterface(tap) +{
      Interface {
        tck : TCK;
        trst : TRST;
        tms : TMS;
        tdi : TDI;
        tdo : TDO;
      }
      Tap(main) +{
        HostIjtag(1) {
        }
        HostBscan {
        }
      }
    }
  }
  BoundaryScan {
    ijtag_host_interface : Tap(main)/HostBscan;
    BondingConfigurations {
      BondingConfiguration(default) {
      }
      BondingConfiguration(bonding1) {
        unused_ports : COORD2, COORD3, D1[2], D1[1], EN0, D1[0], D1[3];
      }
    }
  }
}
```

Example 2

The following example modifies the DFT specification using the editing commands, including the use of the delete_config_element command.

```
> set spec [create_dft_specification]
> report_config_data $spec

DftSpecification(cpu_top,rtl) {
  IjtagNetwork {
    HostScanInterface(tap) {
      Interface {
        tck : tck_p;
        trst : trst_p;
        tms : tms_p;
        tdi : tdi_p;
        tdo : tdo_p;
      }
      Tap(main) {
        HostIjtag(1) {
        }
        HostBscan {
        }
      }
    }
  }
  BoundaryScan {
    ijtag_host_interface : Tap(main)/HostBscan;
  }
}
```

You can add the necessary BondingConfiguration wrappers and then use the report_config_data to see how the DFT Specification was updated.

```
> add_config_element $spec/BoundaryScan/BondingConfigurations
> add_config_element $spec/BoundaryScan/BondingConfigurations/\
    BondingConfiguration(standard)

> add_config_element $spec/BoundaryScan/BondingConfigurations/\
    BondingConfiguration(package1)

> set_config_value $spec/BoundaryScan/BondingConfigurations/\
    BondingConfiguration(package1)/enable_signal cpu_top/cs

> set_config_value $spec/BoundaryScan/BondingConfigurations/\
    BondingConfiguration(package1)/unused_ports {D2, D3}

> delete_config_element enable_signal -in_wrapper $spec/BoundaryScan/\
    BondingConfigurations/BondingConfiguration(package1)

> delete_config_element unused_ports -in_wrapper $spec/BoundaryScan/\
    BondingConfigurations/BondingConfiguration(package1)

> set_config_value $spec/BoundaryScan/BondingConfigurations/\
    BondingConfiguration(package1)/\
    unused_ports {D1[2] D1[1] D1[0] D1[3]}

> report_config_data

> set spec [create_dft_specification]
> report_config_data $spec

DftSpecification(cpu_top,rtl) +{
  IjtagNetwork +{
    HostScanInterface(tap) +{
      Interface {
        tck : tck_p;
        trst : trst_p;
        tms : tms_p;
        tdi : tdi_p;
        tdo : tdo_p;
      }
      Tap(main) +{
        HostIjtag(1) {
        }
        HostBscan {
        }
      }
    }
  }
  BoundaryScan {
    ijtag_host_interface : Tap(main)/HostBscan;
    BondingConfigurations {
      BondingConfiguration(standard) {
      }
      BondingConfiguration(package1) {
        unused_ports : D1[2], D1[1], D1[0], D1[3];
      }
    }
  }
}
```

Related Topics

Configure the DFT Specification With the Config Data Browser

Validate the DFT Specification

In this optional step, you can validate the edits you made to the DFT specification to make sure no errors exist before you proceed to the next step.

Example

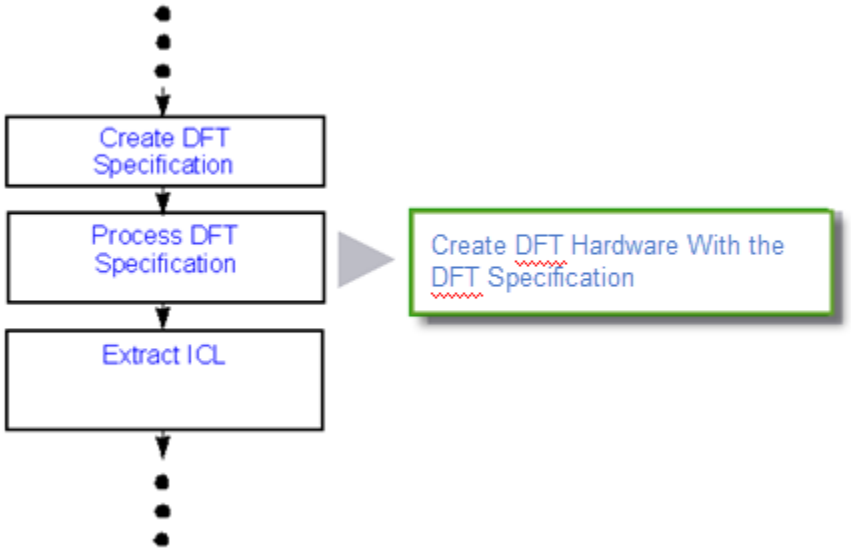
The following example validates your DFT specification.

```
process_dft_specification -validate_only
```

Process DFT Specification

The next step is to process the DFT specification that was created, edited, and validated in the previous step. This step creates and inserts the hardware for all the components that are in the DFT specification.

Figure 2-6. Process DFT Specification



Create DFT Hardware With the DFT Specification

Use the process_dft_specification command to generate and insert into the design all DFT hardware requested with the DFT specification. For Tessent BoundaryScan, the TAP controller and boundary-scan cells are inserted. IJTAG is used because boundary scan connects to the TAP, and the TAP is an IJTAG node.

Examples

Example 1

The following example generates and inserts into the design the hardware requested with the DFT specification.

```
process_dft_specification
```

Example 2

The following example generates the hardware requested with the DFT specification but does not insert the hardware into the design.

```
process_dft_specification -no_insertion
```

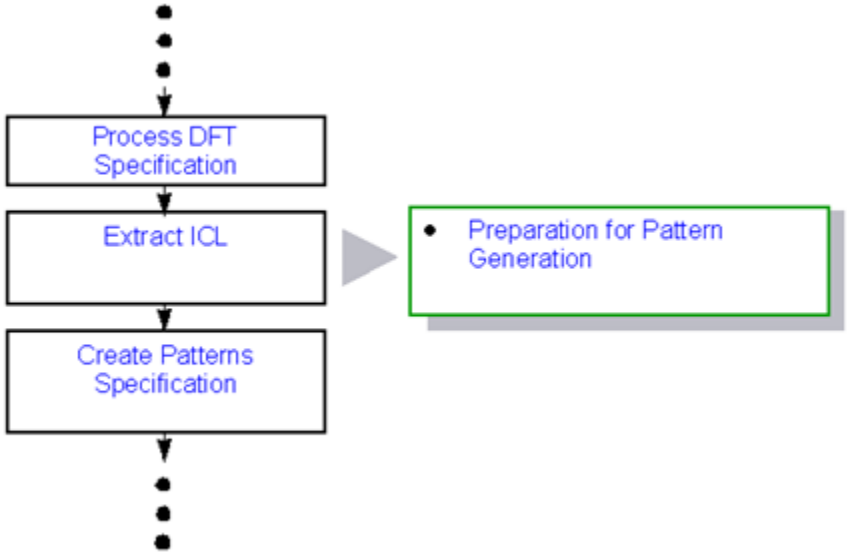
Extract ICL

The Extract ICL step verifies the proper connectivity of the ICL modules that were inserted with the process_dft_specification command. If no DRC violations are detected, the top-level ICL description is extracted.

The extract_icl command also creates an SDC file that can be used for synthesis. Refer to the "RTL Design Flow Synthesis" section for more information.

Downstream tools use the top-level ICL description for creating patterns. You can use the open_visualizer command to debug any ICL extraction DRC violations that are reported.

Figure 2-7. Extract ICL



Preparation for Pattern Generation

In this step, use the extract_icl command to find all modules (both Tessent instruments and non-Tessent instruments) with their associated ICL modules. If no DRC violations are detected, the ICL description of the root design is extracted.

The root of the design was specified with the set_current_design command during design elaboration in the Load the Design step. The Create Patterns Specification and Process Patterns Specification steps use the ICL description that was created for the root of the design. You can use the open_visualizer command to debug ICL extraction DRC violations. Refer to the "Tessent Visualizer" chapter in the *Tessent Shell User's Manual*.

Example

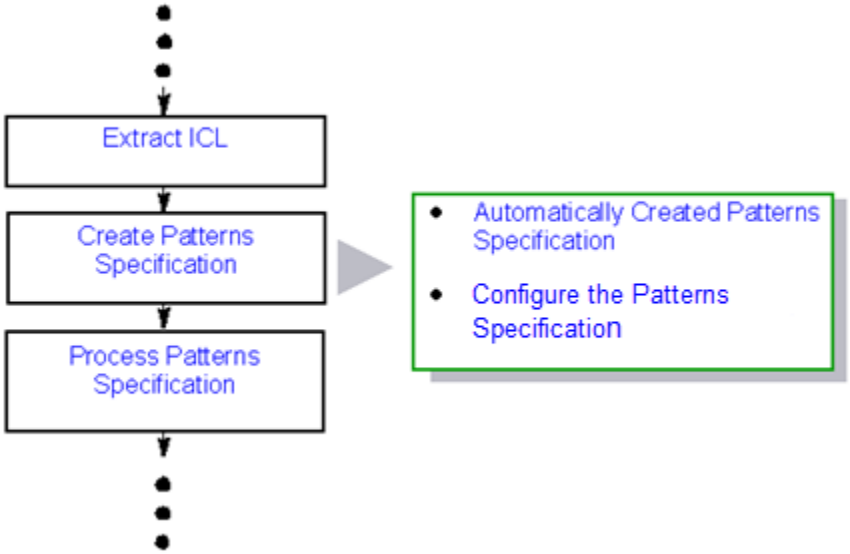
The following example extracts all ICL modules to the root of the design.

```
extract_icl
```

Create Patterns Specification

The Create Patterns Specification step creates the default patterns specification. The patterns specification is a configuration file that tells you what tests are being created. You can edit or configure the default patterns specification to generate the patterns specification you want.

Figure 2-8. Create Patterns Specification



Automatically Created Patterns Specification

The default patterns specification is created with the create_patterns_specification command. It is only stored in memory.

To see the specification, use the report_config_data command.

Example

The following example creates a default patterns specification using the create_patterns_specification command, stores the specification in a variable called pat_spec, and then uses this variable to report the patterns specification in memory. The optional Tcl variable pat_spec provides a convenient reference to the specification.

```
set pat_spec [create_patterns_specification]
report_config_data $pat_spec
```

For a full description of the patterns specification, see the section on the PatternsSpecification wrapper in the *Tessent Shell Reference Manual*.

The following example shows a PatternsSpecification wrapper for module “TOP” with design_identifier “gate”.

```
PatternsSpecification(TOP, gate, signoff) {
  Patterns(ICLNetwork) {
    ICLNetworkVerify(TOP) {
    }
  }
  Patterns(JtagBscanPatterns) {
    TestStep(JtagBscanTestStep) {
      BoundaryScan {
        bonding_configuration : option1;
        RunTest(test_logic_reset) {
        }
        RunTest(inst_reg) {
        }
        RunTest(id_reg) {
        }
        RunTest(bypass_reg) {
        }
        RunTest(bscan_reg) {
        }
        RunTest(input) {
        }
        RunTest(sample) {
        }
        RunTest(disabled_outputs) {
        }
        RunTest(highz_mode) {
        }
        RunTest(clamp) {
        }
        RunTest(output) {
        }
      }
    }
  }
}
```

Configure the Patterns Specification

This section describes methods you can use to configure your patterns specification. Use one of these methods to edit or create a patterns specification according to your requirements. Typically, you do not need to edit the signoff patterns specification; only the manufacturing patterns specification may need editing based on your requirements.

Method 1: Edit the Patterns Specification in Memory

We recommend this method because as you edit the patterns specification in memory, the commands that are used are specified in the Tcl or dofile. Therefore, the edits are repeatable for the next iteration by using only scripts.

Example

The following example shows an existing patterns specification you want to modify.

```
set pat_spec [create_patterns_specification]
report_config_data $pat_spec
```

```
PatternsSpecification(TOP, gate, signoff) {
  Patterns(ICLNetwork) {
    ICLNetworkVerify(TOP) {
    }
  }
  Patterns(JtagBscanPatterns) {
    TestStep(JtagBscanTestStep) {
      BoundaryScan {
        bonding_configuration : option1;
        RunTest(test_logic_reset) {
        }
        RunTest(inst_reg) {
        }
        RunTest(id_reg) {
        }
        RunTest(bypass_reg) {
        }
        RunTest(bscan_reg) {
        }
        RunTest(input) {
        }
        RunTest(sample) {
        }
        RunTest(disabled_outputs) {
        }
        RunTest(highz_mode) {
        }
        RunTest(clamp) {
        }
        RunTest(output) {
        }
      }
    }
  }
}
```

The following commands show how to make three types of changes.

```
# To edit a value
set_config_value $pat_spec/Patterns(JtagBscanPatterns)/TestStep(JtagBscanTestStep)/
BoundaryScan/bonding_configuration option2

# To delete an element
delete_config_element $pat_spec/Patterns(JtagBscanPatterns)/
TestStep(JtagBscanTestStep)/BoundaryScan/RunTest(test_logic_reset)

# To add an element
add_config_element RunTest(id_reg) -in_wrapper \
  $pat_spec/Patterns(JtagBscanPatterns)/TestStep(JtagBscanTestStep)/BoundaryScan/
```

The following shows the resulting patterns specification.

report_config_data $pat_spec

```
PatternsSpecification(TOP, gate, signoff) {
  Patterns(ICLNetwork) {
    ICLNetworkVerify(TOP) {
    }
  }
  Patterns(JtagBscanPatterns) {
    TestStep(JtagBscanTestStep) {
      BoundaryScan {
        bonding_configuration : option2;
        RunTest(inst_reg) {
        }
        RunTest(id_reg) {
        }
        RunTest(bypass_reg) {
        }
        RunTest(bscan_reg) {
        }
        RunTest(input) {
        }
        RunTest(sample) {
        }
        RunTest(disabled_outputs) {
        }
        RunTest(highz_mode) {
        }
        RunTest(clamp) {
        }
        RunTest(output) {
        }
        RunTest(id_reg) {
        }
      }
    }
  }
}
```

Method 2: Write Out the Patterns Specification, Edit the File, and Read the File Back In

This method may be easier, but every time the primary Tcl script or dofile runs, the patterns specification that is written out overwrites your edits. To make sure your edits are reusable and repeatable using scripts, make a copy of the patterns specification and then edit the specification before reading it back in.

Example

The following example writes the patterns specification into a file called *bonding1_config.pat_spec*. After making the edits in a copy of this file, this file is read back in. The file that is written out is different from the edited file that is read back in.

```
set pat_spec [create_patterns_specification]
report_config_data

write_config_data bonding1_config.patterns_spec -wrappers $pat_spec

# Edit your bonding1_config.patterns_spec file using an external text editor
# and save it under a new name

read_config_data bonding1_config_modified.patterns_spec
report_config_data
```

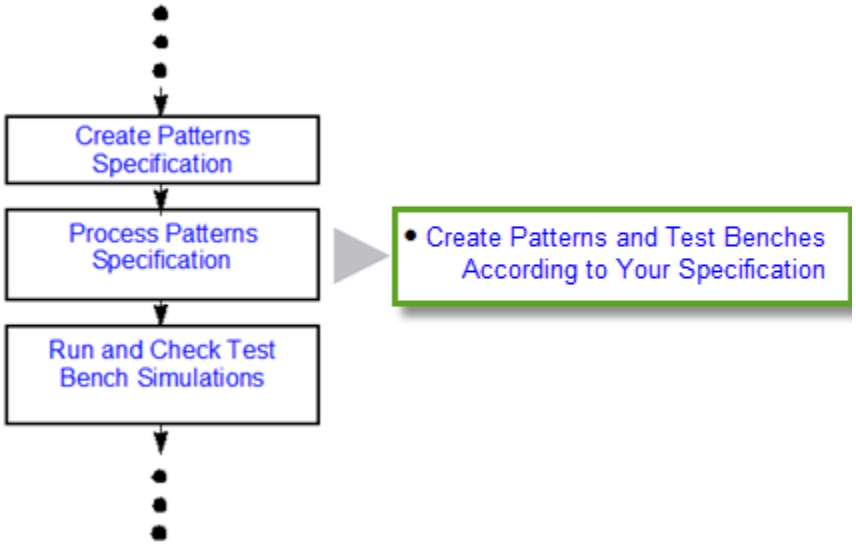
When you manually edit the patterns specification, we recommend that you use the process_patterns_specification command to validate the specification.

```
process_patterns_specification -validate_only
```

Process Patterns Specification

The process patterns specification step of the design flow creates the patterns and testbenches.

Figure 2-9. Process Patterns Specification



Create Patterns and Testbenches According to Your Specification

In this step of the design flow, you create the patterns or testbenches according to either the default patterns specification or to the edited patterns specification that you created in the previous step.

Example

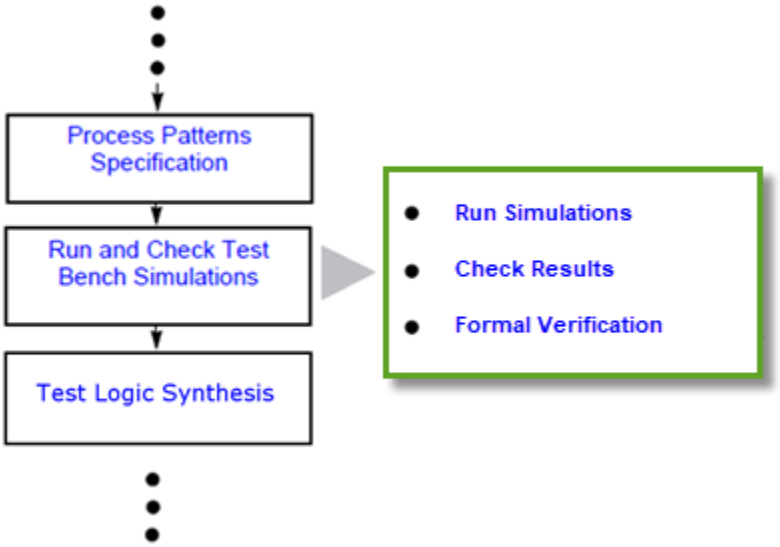
The following example generates the testbenches.

```
process_patterns_specification
```

Run and Check Testbench Simulations

Running and checking testbench simulations is the last step in the Tessent BoundaryScan insertion using Tessent Shell. In this step, you run simulations of the boundary scan verification and then check the results.

Figure 2-10. Run and Check Testbench Simulations



Run Simulations

Use the run_testbench_simulations command to invoke a simulation manager to run a set of simulation testbenches.

The run_testbench_simulations command compiles and simulates testbenches, generated for the TAP and boundary scan from the process_patterns_specification command, that are located at *tsdb_outdir/patterns/<design>.patterns_signoff*.

For a detailed description of the run_testbench_simulations command and its usage, see the *Tessent Shell Reference Manual*.

Example

The following example runs simulations of all patterns defined in the PatternsSpecification.

```
set_simulation_library_sources -y ../library/verilog \
   -v ../library/pad_cells.v
run_testbench_simulations
```

Check Results

Use the check_testbench_simulations command to check the status of the simulations that were previously launched by the run_testbench_simulations command.

For a detailed description of the check_testbench_simulations command and its usage, see the *Tessent Shell Reference Manual*.

Example

The following example checks the simulation results for errors.

```
check_testbench_simulations -report_status
```

Formal Verification

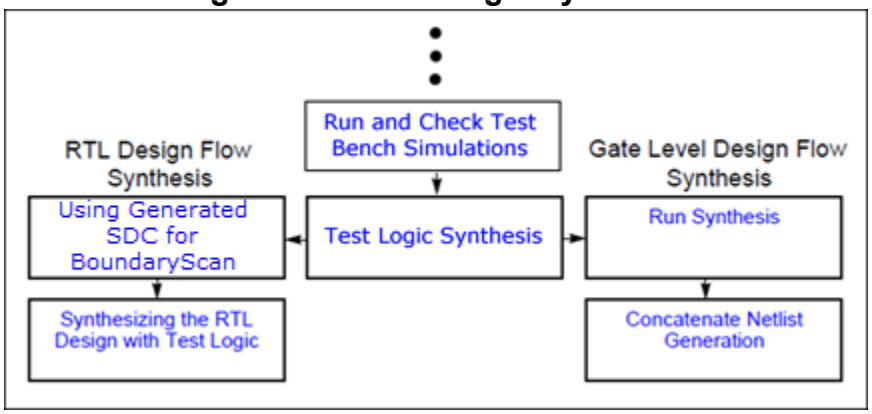
Tessent Shell-based products currently do not generate scripts for use with Synopsys® Formality®. You can however, set constraints in your design and manually create a script that is used with Formality.

For guidance on how this is accomplished, refer to the "Formal Verification" Appendix in the *Tessent Shell User's Manual*.

Test Logic Synthesis

The test logic synthesis process is different when handling a RTL or gate level design. The following sections outline the different options.

Figure 2-11. Test Logic Synthesis



RTL Design Flow Synthesis

The RTL synthesis flow that integrates the BoundaryScan RTL and associated test logic with the design RTL is an automated flow. The following sections outline the process.

Using Generated SDC for BoundaryScan

The extract_icl and extract_sdc commands both create a Synopsys Design Constraints (SDC) file that can be used for synthesis, layout and static timing analysis (STA). Because extract_sdc command requires ICL, it needs to be run after extract_icl.

When extract_icl is run on a physical block containing sub-blocks, the SDC constraints are generated for the physical module as well as the sub-blocks.

The created SDC is composed of several procedures that can be integrated into a design synthesis script.

- **tessent_set_default_variables —** This proc defines the default value of the variables used in instrument timing constraints. It is provided as a template. The user should override the values of these variables to correspond to their setup. For example, the values of the array "tessent_clock_mapping" should be overwritten with the user's chosen names for the functional clocks.
- **tessent_create_functional_clocks —** This proc defines the functional clocks used by the instrument constraints. This proc is not typically called because most users define their functional clocks in their own timing scripts.
- **tessent_set_non_modal —** This proc defines the constraints for the BoundaryScan instrument.

For more information and examples on how to use the generated SDC procs, refer to the "Timing Constraints SDC" chapter in the *Tessent Shell User's Manual*. Additional information is also provided specific to BoundaryScan Instrument proc usage.

Synthesizing the RTL Design With Test Logic

This process is automated by a script that can be created and then processed by a synthesis tool to synthesize an RTL design that has been DFT inserted.

The write_design_import_script command can be used to generate a script that can be processed by a synthesis tool to load the RTL design that has been DFT inserted. The script file written can be combined with the SDC generated during extract_icl to synthesize a physical block or chip design unit.

Gate Level Design Flow Synthesis

The gate level design flow synthesis is a fully automated flow and only requires the run_synthesis command to synthesize the test logic and integrate into the design,

Run Synthesis

The run_synthesis command only synthesizes test logic RTL contained within the TSDB.

When creating and inserting memory BIST, boundary scan or IJTAG logic, the generated RTL is automatically written to the TSDB during process_dft_specification.

The run_synthesis command to invokes a synthesis manager to perform synthesis of the test logic RTL

For a detailed description of the run_synthesis command and its usage, refer to the *Tessent Shell Reference Manual*.

Example

The following example performs synthesis for a design and can be run at the physical_block or top design level.

```
run_synthesis
```

Concatenate Netlist Generation

When run_synthesis completes successfully, a concatenated netlist of the design that contains the synthesized test logic and modified design modules is automatically created and placed in the dft_inserted_designs directory of the TSDB.

Chapter 3
Boundary Scan Specific Topics

Topics within this chapter cover a variety of subjects that describe cell library requirements as well as insertion of boundary scan cells in different design scenarios.

Pad Cell Library

To insert TAP and BoundaryScan using Tessent tools, you need a Tessent cell library for pad cells. The Tessent cell library is an integrated library that contains a functional description for simulation as well as attributes for test logic insertion for each cell.

Pad cell definitions take several forms depending on the function and structure of the pads. Examples of several common forms follow.

Input Pad Example

A simple input pad:

```
model INPAD
  (IO, FP)
(
  cell_type = pad;
  input (IO) (pad_from_io;  )
  output (FP) (pad_from_pad;  )
  {
    primitive = _buf mlc_buf_1 (IO, FP);
  }
) // end model INPAD
```

Output Pad Example

An output pad with an active high enable:

```
model OUTPAD_EN1
  (IO, TP, EN1)
(
  cell_type = pad;
  output (IO) (pad_to_io; )
  input (TP) (pad_to_pad; )
  input (EN1) (pad_enable_high; )
  (
    primitive = _tsh mlc_tsh_1 (TP, EN1, IO);
  )
) // end model OUTPAD_EN1
```

Bidirectional Pad Example

A bidirectional pad with an active low enable:

```
model IOPAD_EN0
  (IO, FP, TP, EN0)
(
  cell_type = pad;
  inout (IO) (pad_pad_io; )
  input (TP) (pad_to_pad; )
  output (FP) (pad_from_pad; )
  input (EN0) (pad_enable_low; )
  (
    primitive = _buf mlc_buf_1 (IO, FP);
    primitive = _tsl mlc_tsl_1 (TP, EN0, IO);
  )
) // end model IOPAD_EN0
```

Configurable Pad Example

This is an example of a pad that can be configured as an input or an output. The “modein” input controls the configuration:

```
model configurable_pad
   (io, fp, tp, en1, modein)
(
  cell_type = pad;
  mode(
    pin (io) (pad_pad_io; )
    pin (fp) (pad_from_pad; )
    pin (modein) (pad_tied1; )
  )
  mode(
    pin (io) (pad_pad_io; )
    pin (tp) (pad_to_pad; )
    pin (en1) (pad_enable_high; )
    pin (modein) (pad_tied0; )
  )

  input (tp) ( )
  input (en1) ( )
  input (modein) ( )
  inout (io) ( )
  output (fp) ( )
  (
    primitive = _inv mlc_inv_1 (modein, modeininv);
    primitive = _and mlc_and_1 (andin, modein, fp);
    primitive = _and mlc_and_2 (en1, modeininv, anden);
    primitive = _buf mlc_buf_1 (io, andin);
    primitive = _tsh mlc_tsl_1 (tp, anden, io);
  )
) // end model configurable_pad
```

Grouping Pad Ports

This is an example of a cell that contains two bidirectional pads. The *mode* wrapper is used to group the ports for each pad, and the pad enable *EN0* is common to both pads:

```
model two_pads (
    IO,
    TP,
    EN0,
    FP
)
(
  cell_type = pad;
  mode (
    pin (IO[0]) (pad_pad_io;)
    pin (TP[0]) (pad_to_pad;)
    pin (EN0) (pad_enable_low;)
    pin (FP[0]) (pad_from_pad;)
  )
  mode (
    pin (IO[1]) (pad_pad_io;)
    pin (TP[1]) (pad_to_pad;)
    pin (EN0) (pad_enable_low;)
    pin (FP[1]) (pad_from_pad;)
  )

  input (EN0) ()
  output (FP) (array = 1:0;)
  inout (IO) (array = 1:0;)
  input   (TP) (array = 1:0;)
  (
    primitive = _tsh outbuf0 (TPC[0], EN0, IO[0]) ;
    primitive = _buf inbuf0 (IO[0], FP[0]);
    primitive = _tsh outbuf1 (TP[1], EN0, IO[1]) ;
    primitive = _buf inbuf1 (IO[1], FP[1]);
  )
)  // end model two_pads
```

The function of a modeled pad is contained in the *pad_function* attribute. For example, to obtain the function of the *FP* pin of the input pad *INPAD* as defined above, you can use the get_attribute_value_list command:

```
SETUP> get_attribute_value_list -name pad_function \
    [get_port FP -of_modules INPAD]
from_pad
```

Note

The pad_function attribute is not set for configured or grouped pads as shown in the two examples above that include the *mode* wrapper.

For further information on how to generate a Tessent cell library, refer to the *Tessent Cell Library Manual*. For further information on the pad_function attribute, see Tessent Pin Function and Pad_Function Attributes in the *Tessent Cell Library Manual*.

The following sections describe alternate methods if you already have a *pad.library* file. This file contains pad IO cell descriptions and verilog simulation models for the pad cells.

Using pad.library and Verilog Simulation Models for the Pad Cells

Use this procedure if you have pad cell Verilog simulation models and a *pad.library* file that contains pad cell IO descriptions.

Prerequisites

- *pad.library* file containing pad IO cell descriptions.
- Verilog models for the pad IO cells describing the functional behavior.
- Tessent Shell is invoked and ready to begin the Read the Libraries step of Load the Design as shown in DFT Flow Using Tessent Shell of "DFT Flow Using Tessent Shell" on page 25.

Procedure

1. Read the *pad.library* file from the appropriate directory, as shown in the following example.

```
SETUP>read_cell_library ../library/pad.library
```

2. Read the Verilog models for the pad cells from the appropriate directory, as shown in the following example.

```
SETUP>read_verilog ../library/verilog/pads.v
```

Results

The attributes of the pad cells described within the *pad.library* file are applied on top of the Verilog description of the pad cells that were read in using the read_verilog command.

Creating a Tessent Cell Library from pad.library and Verilog Simulation Models

The Tessent Cell Library is not needed if you have a *pad.library* file and Verilog simulation models. There is an optional way of creating a Tessent Cell Library utilizing the *pad.library* and Verilog simulation models.

Prerequisites

- *pad.library* file containing pad IO cell descriptions

- Verilog models for the pad IO cells describing the functional behavior

Procedure

1. Invoke the LibComp tool from the unix shell prompt to compile a *libcomp.atpglib* file from the pad cell Verilog simulation models. The first parameter passed is the path to the Verilog simulation models for the pad cells.

   ```
   UNIX>libcomp pads.v -dofile
   ```

   Note

   You do not need to provide any dofiles with the -dofile switch.

   A *libcomp.atpglib* file is created for all the cells present in the *pads.v* file. Ensure that any includes using "include" in the *pads.v* file are properly read in.

   Note

   For more information about libcomp startup options, see libcomp in the *Tessent Cell Library Manual.*

2. Invoke Tessent Shell and optionally create a log file for reference.

   ```
   UNIX>tessent -shell -log create_cell_lib.log
   ```

3. Read the *pad.library* and the *libcomp.atpglib* file created in step 1 using read_cell_library at the Tessent Shell prompt.

   ```
   SETUP>read_cell_library pad.library
   SETUP>read_cell_library libcomp.atpglib
   ```

4. Write the new Tessent Cell Library using write_cell_library command.

   ```
   SETUP>write_cell_library pads.tcell_lib
   ```

Results

The created Tessent Cell Library can be read into Tessent Shell using the read_cell_library command.

If LibComp is unable to translate a cell in the *pads.v* file into an *atpg.lib* model, an empty blackbox model is created for that cell. In this situation, the translation of these cells into the Tessent Cell Library as functional library elements is not completed and they must be manually edited into valid functional models.

Related Topics

read_cell_library [Tessent Shell Reference Manual]

write_cell_library [Tessent Shell Reference Manual]

Customizing Boundary Scan Pin Order

The boundary scan pin order that was initially created can be modified and saved to satisfy design requirements and ensure repeatability of the BIST implementation.

The initial boundary scan pin order can be created from pin placement information provided in a Design Exchange Format (DEF) file that can be loaded during the Load the Design step. If a DEF file is not loaded, the initial pin order is obtained from the declaration order of the ports in the design file. Customization of the boundary scan pin order may be needed with a layout sorted pin order, and is most certainly needed for a port declaration ordered boundary scan chain to minimize routing.

Prerequisites

- If you want to have the initial boundary scan pin order based on port layout placement, the DEF file should be loaded during the Load the Design step. All the steps for loading a design should be completed.
- Specify and Verify DFT Requirements steps are completed.

Procedure

1. Edit the boundary scan pinorder file with the pin order you want, and save it to either the same or a different file name.

   The boundary scan pinorder file is created once an error-free Design Rule Checking (DRC) result is obtained with the check_design_rules command in the Run DRC step of the flow. The pinorder file is named <*design_name*>*.pinorder* and is located in the Tessent Shell Data Base (TSDB) tsdb_outdir/dft_inserted_designs folder. An example of a pinorder file is shown in Figure 3-1.

Figure 3-1. Example PINORDER File

```
//-----------------------------------------------------
//  File created by: Tessent Shell
//          Version:
//       Created on:
//-----------------------------------------------------


// PortName         PinName OptionList LogicalGroups
// --------         ------- ---------- -------------
   clk1_p           I1      -          -
   clk2_p           I2      -          -
   clk3_p           I3      -          -
   clk4_p           I4      -          -
   ramclk_p         I5      -          -
   reset_p          I6      -          -
   enable_p         I7      -          -
   paddr_p[10]      O1      -          -
   paddr_p[9]       O2      -          -
   paddr_p[8]       O3      -          -
   paddr_p[7]       O4      -          -
      .
      .
      .
```

2. On subsequent iterations, read in the customized pinorder file while in SETUP mode, prior to running check_design_rules as shown in this example:

   ```
   SETUP>set_boundary_scan_port_options -pin_order_file \
         ./tsdb_outdir/dft_inserted_designs/cpu_top.pinorder.modified
   ```

   The specified pinorder file is validated during the check_design_rules command rather than a pinorder file being created. The pinorder file that was specified is shown in the pin_order_file : *filename* property of the DftSpecification/BoundaryScan wrapper when the DftSpecification is created.

Results

Once the DFT hardware specified in the DftSpecification is created and inserted using process_dft_specification, the boundary scan chain order can be observed within the port listing of the BSDL file that is created. The BSDL file is located in the TSDB *<root_directory>/instruments/<design_name>_<design_id>_bscan.instrument* folder and is named *<design_name>.bsdl*.

Related Topics

Tessent Shell Data Base (TSDB) [Tessent Shell Reference Manual]

set_boundary_scan_port_options [Tessent Shell Reference Manual]

DftSpecification/BoundaryScan wrapper [Tessent Shell Reference Manual]

process_dft_specification [Tessent Shell Reference Manual]

Inserting Tessent Boundary Scan on a Custom or Preexisting TAP Controller

The Tessent Shell design flow natively supports a custom or preexisting TAP controller connection to Boundary Scan cells. Custom or preexisting TAP controllers can also be connected to other controllers, such as memory BIST or hybrid TestKompress/LBIST.

For Tessent Shell to work with the preexisting TAP controller, an ICL(Instrument Connectivity Language) description for the TAP controller needs to be created. The preexisting TAP controller must have the port functions described in the “Requirements on a TAP to be usable for BoundaryScan” section within the DftSpecification/BoundaryScan wrapper description in the *Tessent Shell Reference Manual*.

Prerequisites

- An ICL description for a the custom or preexisting TAP controller.
- A TAP controller that meets the requirements outlined in the DftSpecification/BoundaryScan wrapper description.

Procedure

1. Read in the Verilog model and the ICL description of the preexisting or custom TAP.

   ```
   SETUP>read_verilog design/my_custom_TAP.v
   SETUP>read_icl design/my_custom_TAP.icl
   ```

   If there is only a single HostBscan interface defined, the boundary scan chain is connected to the preexisting TAP controller that is read in if all the requirements of the TAP controller are met.

2. If the preexisting TAP controller has more than one HostBscan interface, you can explicitly specify the interface the boundary scan chain is to be connected to by using the following:

   ```
   ANALYSIS>create_dft_specification -existing_bscan_host_scan_in \
   my_custom_TAP_INST/fromBscan1
   ```

   my_custom_TAP_INST in the example above is the instance name of the preexisting custom TAP controller in the design and fromBscan1 is the port on the preexisting TAP where the boundary scan chain is to be connected.

Related Topics

create_dft_specification [Tessent Shell Reference Manual]

read_icl [Tessent Shell Reference Manual]

read_verilog [Tessent Shell Reference Manual]

Read the Design

Instrument Connectivity Language (ICL) [Tessent Shell Reference Manual]

Sharing TAP Ports Between a Preexisting TAP and Tessent TAP

TAP pins can be shared between a preexisting TAP and the Tessent TAP controller in situations where a preexisting TAP does not meet DftSpecification requirements or there is a design requirement to not disturb the preexisting TAP controller.

Follow this procedure to add a Tessent TAP controller to connect to the boundary scan chain in cases where:

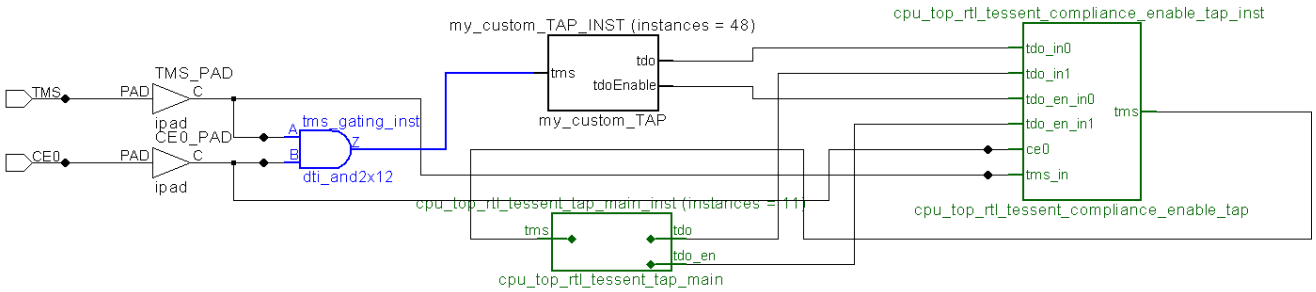
- A preexisting TAP controller is not to be disturbed.
- The preexisting custom TAP controller does not meet the requirements described in the DftSpecification/BoundaryScan wrapper description and cannot be used for boundary scan.

Prerequisites

- The IEEE 1687 standard requires that the state of a TAP controller should not change when its scan path is not selected. When implementing the methodology outlined in this procedure, the preexisting TAP has its own TMS gating. When the Tessent TAP is selected by its specified compliance enable signals, the preexisting TAP is gated off, preserving its state.

  Figure 3-2 shows the example implementation outlined in this procedure. The preexisting TAP module is my_custom_TAP, shown in black. The preexisting TAP does not need to be described in ICL. The TMS gating logic for this TAP is shown in blue, and must also pre-exist in the design. The complexity of the TMS gating logic depends on how many compliance enables are specified to enable the Tessent TAP. The Tessent TAP and compliance enable module inserted by Tessent Shell in this procedure are shown in green.

Figure 3-2. Sharing TAP Ports Example



Procedure

1. Specify the TAP pins during the "Specify DFT Specification Requirements" section of the design flow, as shown in the following example:

```
SETUP> set_attribute_value tck_p -name function -value tck
SETUP> set_attribute_value tdi_p -name function -value tdi
SETUP> set_attribute_value tms_p -name function -value tms
SETUP> set_attribute_value trst_p -name function -value trst
SETUP> set_attribute_value tdo_p -name function -value tdo
```

2. Specify the compliance enable (CE) pins during the "Create DFT Specification" section of the design flow:

```
ANALYSIS> create_dft_specification \
          -active_low_compliance_enables CEO \\<pin_port_list>
```

   For this example, there is a single active low compliance enable pin named "CE0".

   When specifying the compliance enable pins with create_dft_specification, the active_high_enables and active_low_enables properties within the HostScanInterface/ Interface/ComplianceEnable wrapper are populated with the named ports. They are also added to the BoundaryScan/ BoundaryScanCellOptions wrapper as compliance_enable1

and compliance_enable0, respectively. For this example there would only be active_low_enables and compliance_enable0 entries as shown in the following:

```
ANALYSIS> report_config_data
DftSpecification(cpu_top,rtl) {
  IjtagNetwork {
    HostScanInterface(tap) {
      Interface {
        tck : tck_p;
        trst : trst_p;
        tms : tms_p;
        tdi : tdi_p;
        tdo : tdo_p;
        ComplianceEnable {
          active_low_enables : CE0;
        }
      }
      Tap(main) {
        HostIjtag(1) {
        }
        HostBscan {
        }
      }
    }
  }
  BoundaryScan {
    ijtag_host_interface : Tap(main)/HostBscan;
    BoundaryScanCellOptions {
      CE0 : compliance_enable0;
    }
  }
}
DefaultsSpecification(user) {
}
```

If boundary scan is to be inserted, then those pins listed with the compliance_enable0/ compliance_enable1 attributes appear in the COMPLIANCE_PATTERNS attribute in the BSDL file. If an internal pin is provided in the list, then a warning is given that the BSDL file is not IEEE Std 1149.1 compliant.

3. Complete the Process DFT Specification section of the design flow. Prior to starting the Extract ICL step, use the add_ijtag_logical_connection command to enable extract_icl to bypass the TMS gating to the preexisting TAP.

   a. Manually change the context to patterns -ijtag for add_ijtag_logical_connection:

      ```
      INSERTION>set_context patterns -ijtag
      ```

   b. Specify the logical path across the TMS gating logic to the preexisting TAP. In this example it would be from the output of TMS_PAD to the output of tms_gating_inst as seen from Figure 3-2.

      ```
      SETUP>add_ijtag_logical_connection -to tms_gating_inst/Z -from
      TMS_PAD/C
      ```

If this step is not completed, the tool returns an I2 ICL Extraction error.

4. Complete the Extract ICL step and continue with the remaining steps of the design flow.

Results

The Tessent TAP is created and selected during boundary scan tests when the CE0 port is "0". This value is automatically set during boundary scan test simulations and is documented in the generated BSDL file. The Tessent TAP module is named <*design_name*>_<*design_id*>_tessent_tap_<*id*>.

A module named "<*design_name*>_<*design_id*>_tessent_compliance_enable_<*id*>" is created that muxes the TDO signal and gates the TMS signal to the Tessent TAP based on the compliance enable signal values.

If you are using a compliance enable pin and the logic to enable the TAP that is already present in the design, refer to Example 2 in the IjtagNetwork wrapper description in the *Tessent Shell Reference Manual*. This example shows the steps to follow when you want to connect the TAP controller to internal pins that are not directly on the pads associated to the TAP ports.

Related Topics

set_attribute_value [Tessent Shell Reference Manual]

create_dft_specification [Tessent Shell Reference Manual]

HostScanInterface/Interface [Tessent Shell Reference Manual]

BoundaryScanCellOptions [Tessent Shell Reference Manual]

AC JTAG

If there are any AC JTAG pad IO cells present in the design, then a Tessent Cell Library describing them needs to created and read in during the "Load the Design" portion of the design flow.

For more information, refer to Chapter 2 "Library Model Creation" of the *Tessent Cell Library Manual*.

If an old *pad.library* file is available and verilog models are present, then the procedure described in "Using pad.library and Verilog Simulation Models for the Pad Cells" can be used.

Related Topics

Load the Design

Embedded Boundary Scan Flow

The embedded boundary scan (EBscan) flow is typically used where the pad IO cells are present inside a module that is either a sub-block or a physical region. If the boundary scan cells need to

be present within this sub-block or the physical region, then the embedded boundary scan flow is used.

To implement this flow, the command set_boundary_scan_port_options is used to specify the ports that require embedded boundary scan cells. This process is performed in the Specify DFT Specification Requirements step after Load the Design completed. The following procedure shows an example of how this is done for both sub-block and physical region implementations.

Prerequisites

- Load the Design steps have been completed and the design is loaded and elaborated.

Procedure

1. Start the Specify and Verify DFT Requirements step by beginning to define the DFT specification requirements.

```
## Begin Specify and Verify DFT Requirements step
SETUP>set_dft_specification_requirements -boundary_scan on
```

2. Specify at what level boundary scan is to be inserted.

   For sub-block implementation:

```
SETUP>set_design_level sub_block
```

   For physical region implementation:

```
SETUP>set_design_level physical_block
```

3. Specify the list of pad IO ports that need embedded boundary scan cells by using the set_boundary_scan_port_options command.

```
## The following inserts embedded boundary scan cells
SETUP>set_boundary_scan_port_options -pad_io_ports [list in1 \
in_diff_p in_diff_n out1 out_diff_p out_diff_n clk A Y]
```

4. Use the add_input_constraints command to specify the constants that are required for the pad IO to operate.

   Note

   Some of the pad IO constants that are required may come from the next higher level, either through test setup or other initialization setup.

```
##Constraints specified for pins where the value comes from the next
level
SETUP>add_input_constraints vss -C0
SETUP>add_input_constraints Ten3[2] -C0
SETUP>add_input_constraints Ten3[1] -C0
SETUP>add_input_constraints Ten4[2] -C0
SETUP>add_input_constraints vdd -C1
SETUP>add_input_constraints Ten1 -C1
SETUP>add_input_constraints Ten2 -C1
SETUP>add_input_constraints Ten3[0] -C1
SETUP>add_input_constraints Ten4[1] -C1
```

5. Run check_design_rules to verify the implementation and then continue with the rest of the design flow.

```
##Running DRC
SETUP>check_design_rules
```

Related Topics

add_input_constraints [Tessent Shell Reference Manual]

set_dft_specification_requirements [Tessent Shell Reference Manual]

set_boundary_scan_port_options [Tessent Shell Reference Manual]

Adding a Test Data Register to the TAP Controller

User-defined bits for enhancing test control and status monitoring can be added through the creation of a Test Data Register (TDR) that is inserted and connected to the TAP controller.

The Tdr/DataInPorts wrapper specifies the number of data-in ports to create for test status monitoring on the TDR, the naming of the ports, and the connections to make to them. The Tdr/DataOutPorts wrapper specifies the number of data-out ports to create for test control on the TDR, the naming of the ports, and the connections to make to them. For more information, refer to the Tdr section in the *Tessent Shell Reference Manual*.

The following procedure provides an example of how you can create a default DFT specification and manually add a TAP using the Config Data Browser GUI. This method copies the DFT specification, inserting the TDR without using the Config Data Browser GUI and reading the DFT Specification back into memory. This facilitates the creation of dofiles to make the process repeatable without having to manually edit it within the Config Data Browser.

Prerequisites

- Load the Design steps have been completed and the design is loaded and elaborated.
- Specify and Verify DFT Requirements steps are completed.

Procedure

1. Create a DFT specification and also direct the output to a Tcl variable:

```
##To create a dft specification
set spec [create_dft_specification]
```

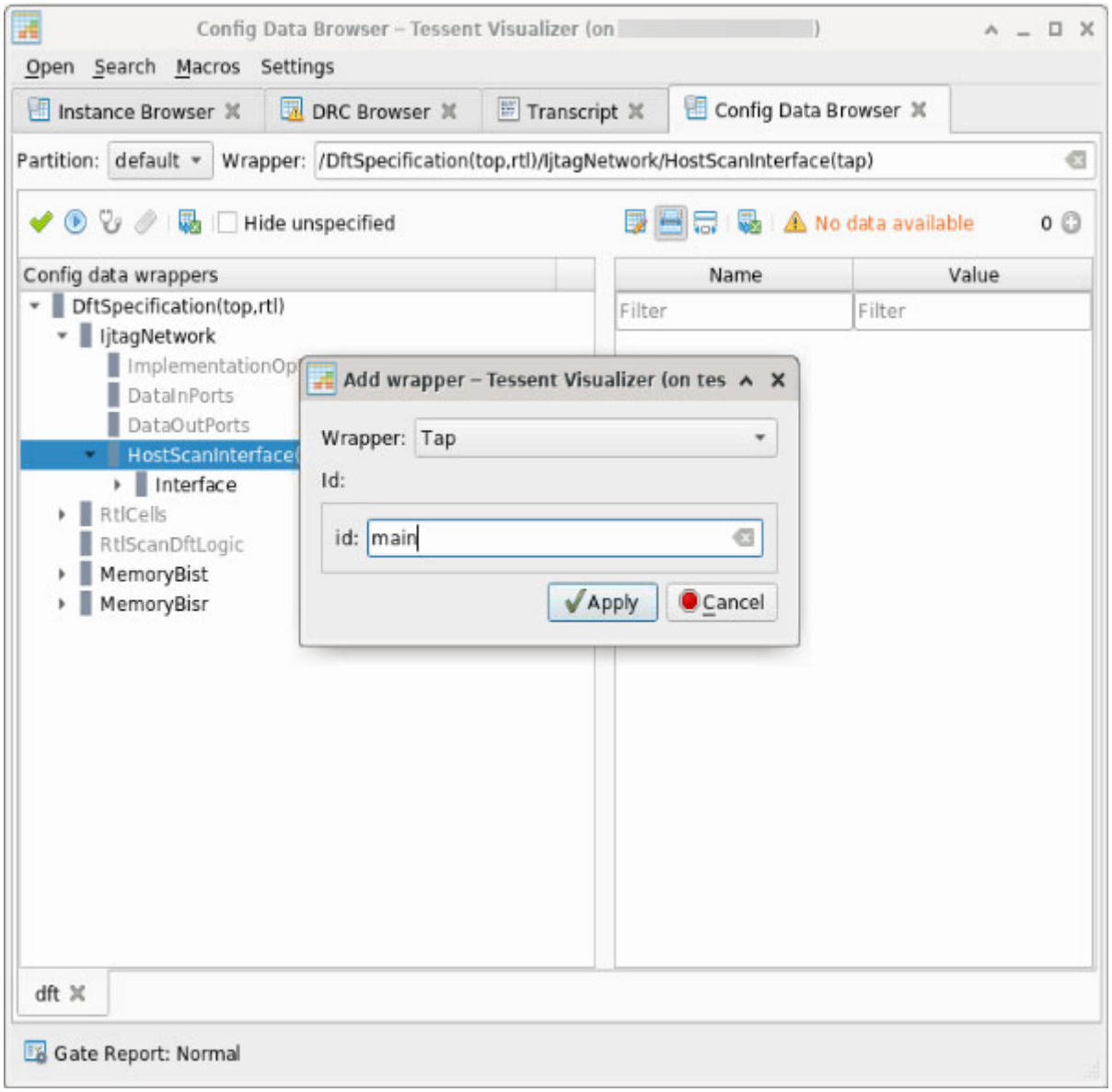
2. Invoke the Config Data Browser.

```
display_specification
```

3. Within the Config Data Browser, add in the TAP structure by selecting and highlighting the location within the DftSpecification tree, right clicking, and choosing **Add wrapper as a child**. In the dialog box that opens, enter a name for the TAP, as shown in Figure 3-3.

```
/IjtagNetwork/HostScanInterface(tap)/Tap(main)
```

Figure 3-3. Adding a TAP With the Config Data Browser



4. Continue using the tree view and the form to create a HostIjtag(1) wrapper as a child of the TAP.
5. In the shell window, use the report_config_data command to list the current structure.

   ```
   report_config_data $spec
   ```

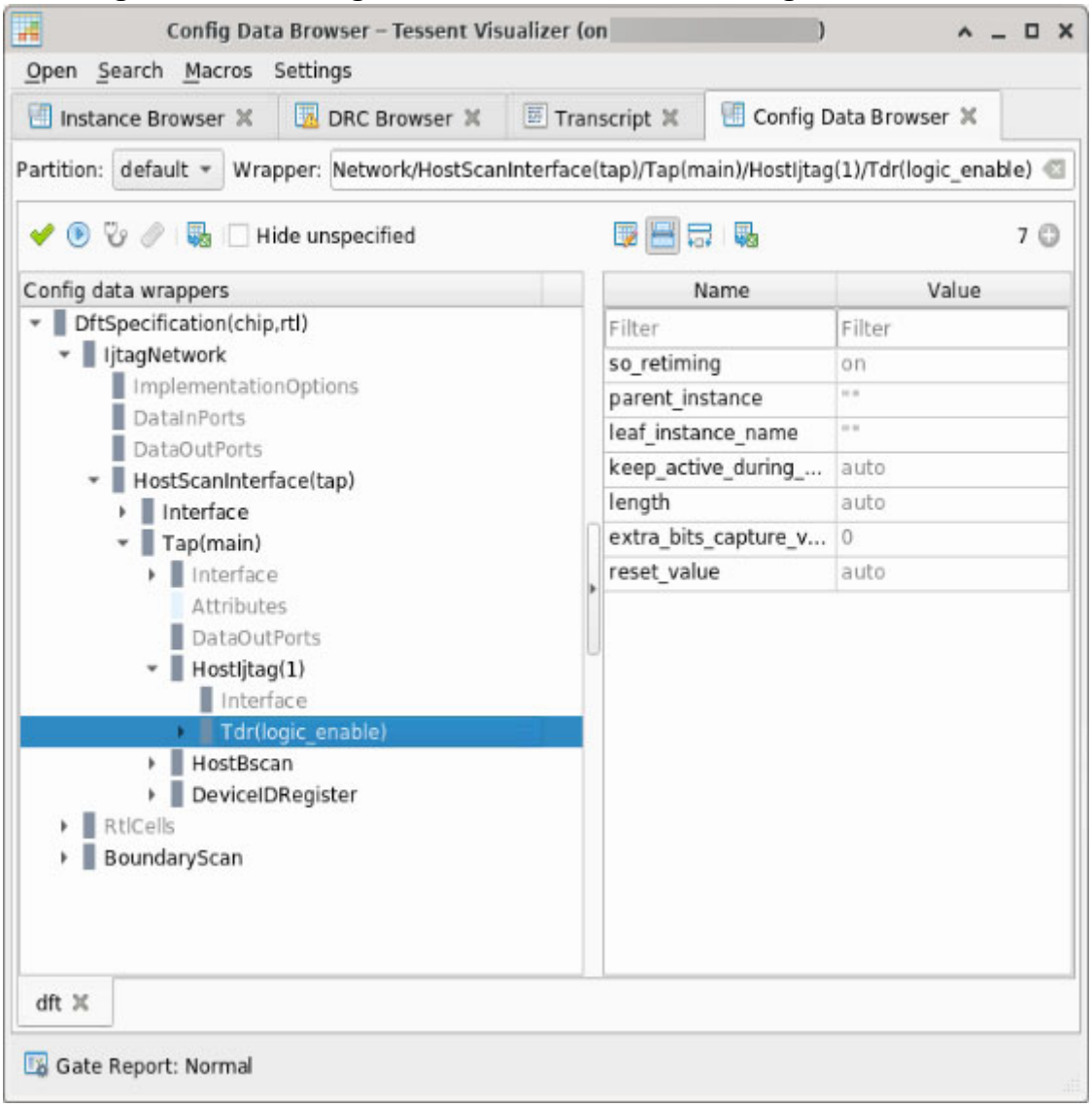
   The tool displays the configuration in the shell window.
6. Once the tool reports the configuration, copy and paste the report information into a dofile, implementing the TAP with the read_config_data command and adding the TDR manually, as shown in the following example. Then, for subsequent runs, the

configuration edits are already present in the dofile, making the process repeatable with scripting.

```
read_config_data -in_wrapper $spec/IjtagNetwork/
HostScanInterface(tap)/Tap(main)/HostIjtag(1) -from_string {
   Tdr(logic_enable) {
    DataOutPorts +{
      count       : 3;
      port_naming : ltest_en, bypass_en, low_power_en;
      }
   }
}
```

7. You can inspect the TDR that was added in the Config Data Browser, as shown in Figure 3-4:

Figure 3-4. Viewing the Added TDR in the Config Data Browser



8. Create the hardware from the DFT specification;

```
process_dft_specification
```

Related Topics

create_dft_specification [Tessent Shell Reference Manual]

display_specification [Tessent Shell Reference Manual]

report_config_data [Tessent Shell Reference Manual]

BSDL-Only Flow

You can generate boundary scan patterns in Tessent Shell using a Boundary Scan Description Language (BSDL) file created outside the Tessent environment. This pattern generation flow requires only the BSDL file. However, you can also include a Serial Vector Format (SVF) file in this flow if needed.

Prerequisites

- A valid BSDL file.

Procedure

1. Run a dofile to create and process the patterns specification for the BSDL-Only flow

```
set_context patterns -ijtag
set pat_spec [create_patterns_specification -bsdl_file ./EP432P12.bsdl]

report_config_data $pat_spec
process_patterns_specification
```

Note

By default, the tool searches for the BSDL package include directory in the same location as the BSDL file. If it is not there, specify the location with the bsdl_package_directories argument or create a symbolic link to the package directory.

The tool saves every successfully processed patterns specification in the TSDB in patterns directory (*./tsdb_outdir/patterns/*).

2. Use the PatternsSpecification/Patterns/TestStep/BoundaryScan wrapper to define the configuration data that specifies the patterns for the DFT components inserted into a design. Figure 3-5 shows an example of the wrapper contents. Save this to a file: for example, *example.patterns_spec_signoff*. Specify the third-party BSDL file in the BoundaryScan wrapper with the bsdl_file property.

Figure 3-5. Example PatternsSpecification for Third-Party BSDL

```
PatternsSpecification(EP432P12, gate, signoff) {
  Patterns(JtagBscanPatterns) {
    TestStep(JtagBscanTestStep) {
      BoundaryScan {
        bsdl_file : EP432P12.bsdl;
        RunTest(test_logic_reset) {
        }
        RunTest(inst_reg) {
        }
        RunTest(bypass_reg) {
        }
        RunTest(bscan_reg) {
        }
        RunTest(input) {
        }
        RunTest(sample) {
        }
        RunTest(disabled_outputs) {
        }
        RunTest(highz_mode) {
        }
        RunTest(clamp) {
        }
        RunTest(output) {
        }
      }
    }
  }
}
```

In most cases, the BSDL description should be sufficient to generate patterns, but if you need an initialization sequence, you can specify it as follows:

```
# Determine the path to Patterns(JtagBscanPatterns) wrapper
set JtagBscanPatterns_wrapper [get_config_elements \
  Patterns(JtagBscanPatterns) -in_wrapper $pat_spec -hierarchical]

# Add a ProcedureStep wrapper for an initialization sequence and
# define a path to the SVF initialization file
set init_wrapper [add_config_element ProcedureStep(init) -first \
  -in_wrapper $JtagBscanPatterns_wrapper]
set_config_value [get_config_element svf_file \
  -in_wrapper $init_wrapper] EP432P12.svf
report_config_data $pat_spec
process_patterns_specification
```

This results in the patterns specification with the following SVF initialization step.

```
PatternsSpecification(EP432P12,gate,signoff) {
  Patterns(JtagBscanPatterns) {
    ProcedureStep(init) {
      svf_file : EP432P12.svf;
    }
    TestStep(JtagBscanTestStep) {
      BoundaryScan {
        bsdl_file : EP432P12.bsdl;
        RunTest(test_logic_reset) {
        }
        RunTest(inst_reg) {
        }
        RunTest(bypass_reg) {
        }
        RunTest(bscan_reg) {
        }
        RunTest(input) {
        }
        RunTest(sample) {
        }
        RunTest(disabled_outputs) {
        }
        RunTest(highz_mode) {
        }
        RunTest(clamp) {
        }
        RunTest(output) {
        }
      }
    }
  }
}
```

For more explanation of the RunTest wrapper and parameters, refer to the BoundaryScan topic in the "Configuration-Based Specification" chapter of the *Tessent Shell Reference Manual*.

Results

The tool creates the *JtagBscanPatterns.v* testbench in the *tsdb_outdir/Patterns/EP432P12_gate.patterns_signoff* directory. For details on how the *tsdb_outdir* directory structure is organized, refer to "Tessent Shell Data Base (TSDB)" in the *Tessent Shell Reference Manual*.

Related Topics

BoundaryScan wrapper [Tessent Shell Reference Manual]

RunTest [Tessent Shell Reference Manual]

PatternsSpecification wrapper [Tessent Shell Reference Manual]

Dividing Boundary Scan for Logic Test

In most designs, the boundary scan chain needs to be included in the logic testing portion of the design to isolate the chip boundaries with a boundary scan chain.

Typically, a single boundary scan chain is too long as some compression or Built-In Self Test (BIST) is incorporated to test the logic portion of the design. Therefore, you need to divide (or segment) the boundary scan chain into shorter chain lengths to match the functional design flops' scan chain stitching. To properly segment the boundary scan chain, you can use the max_segment_length_for_logictest property in the BoundaryScan wrapper.

Block Level

This section demonstrates how to segment your boundary scan in a design at the physical block level. The segments start and end in the current logical group and do not span over different logical groups. For each logical group, the tool creates a tcd_scan and a WGL file that describe the logic test segments.

You can also use the logic test DFT signals to isolate the block, enabling you to run separate internal and external mode ATPG runs. During wrapper cell analysis, you can exclude ports from having wrapper cells and later use the existing boundary scan cell instead of a dedicated wrapper cell. See the analyze_wrapper_cells command reference.

Tip

If you need separate control over the boundary scan cells, you can use the bscan_clamp_enable and bscan_input_isolation_enable DFT signals. If not, you can use int_ltest_en and ext_ltest_en DFT signals. See the tables in the add_dft_signals command reference.

Prerequisites

- Complete the Load the Design steps and load/elaborate the block-level design.
- Define your DFT Specification requirements and constraints for the block-level design.

Procedure

1. Activate boundary scan.

```
SETUP>set_dft_specification_requirements -boundary_scan on
```

2. Specify a boundary scan cell on all the physical block ports with IO pads.

```
set_boundary_scan_port_options -pad_io_ports {ports_with_pads}
```

3. Run Design Rule Checking (DRC) to ensure all constraints are correct and if there are no errors, move Tessent Shell from Setup to Analysis mode.

```
SETUP>check_design_rules
```

4. Create the “DftSpecification(design_name,id)” configuration wrapper and copy the newly created wrapper object to the variable “spec” to customize the specification later.

```
ANALYZE>set spec [create_dft_specification]
```

5. Set the value for the max_segment_length_for_logictest parameter in the EmbeddedBoundaryScan wrapper to 200.

```
ANALYZE>set_config_value \
 $spec/EmbeddedBoundaryScan/max_segment_length_for_logictest 200
```

Note

Compared to the other examples in this section, we use the EmbeddedBoundaryScan wrapper instead of the BoundaryScan wrapper.

6. Create and insert the hardware for the TAP and boundary scan into the design.

```
ANALYZE>process_dft_specification
```

7. Extract the ICL, generate the patterns, then simulate the patterns.

```
set_system_mode setup

extract_icl

set_context patterns -ijtag

check_design_rules

set spec [create_patterns_specification]

process_patterns_specification

run_testbench_simulations
```

Results

This example created a tcd_scan and WGL file for each logical group. You can use the tcd_scan or WGL file to stitch the boundary scan cells into the scan chains created during scan insertion.

Chip Level With Scan Insertion

This section demonstrates how to segment your boundary scan in a chip-level design with scan insertion. The segments start and end in the current logical group and do not span over different logical groups. For each logical group, the tool creates a tcd_scan and a WGL file that describe the logic test segments.

Tip

If you need separate control over the boundary scan cells, you can use the bscan_clamp_enable and bscan_input_isolation_enable DFT signals. If not, you can use int_ltest_en and ext_ltest_en DFT signals. See the tables in the add_dft_signals command reference.

Prerequisites

- Complete the Load the Design steps and load/elaborate the chip-level design with scan insertion.
- Define your DFT Specification requirements and constraints for the chip-level design.

Procedure

1. Activate boundary scan.

```
SETUP>set_dft_specification_requirements -boundary_scan on
```

2. Run Design Rule Checking (DRC) to ensure all constraints are correct and if there are no errors, move Tessent Shell from Setup to Analysis mode.

```
SETUP>check_design_rules
```

3. Create the “DftSpecification(design_name,id)” configuration wrapper and copy the newly created wrapper object to the variable “spec” to customize the specification later.

```
ANALYZE>set spec [create_dft_specification]
```

4. Set the value for the max_segment_length_for_logictest parameter in the BoundaryScan wrapper to 200.

```
ANALYZE>set_config_value \
 $spec/BoundaryScan/max_segment_length_for_logictest 200
```

5. Set the value for the handle_logictest_segments_during_scan_insertion parameter in the BoundaryScan wrapper to on.

```
ANALYZE>set_config_value \
 $spec/BoundaryScan/handle_logictest_segments_during_scan_insertion
on
```

Note

This is the main difference between segmenting boundary scan chains with scan insertion and without. See Chip Level Without Scan Insertion for details.

6. Create and insert the hardware for the TAP and boundary scan into the design.

```
ANALYZE>process_dft_specification
```

7. Extract the ICL, generate the patterns, then simulate the patterns.

```
set_system_mode setup

extract_icl

set_context patterns -ijtag

check_design_rules

set spec [create_patterns_specification]

process_patterns_specification

run_testbench_simulations
```

Results

This example created a tcd_scan and WGL file for each logical group. You can use the tcd_scan or WGL file to stitch the boundary scan cells into the scan chains created during scan insertion.

Chip Level Without Scan Insertion

This section demonstrates how to segment your boundary scan in a chip-level design without scan insertion. Segments can span over multiple logical groups and the tool creates an instrument dictionary description instead of tcd_scan and WGL files.

The following example segments the boundary scan chain so that a maximum of 200 flops are present in any segment.

Prerequisites

- Complete the Load the Design steps and load/elaborate the chip-level design without scan insertion.
- Define your DFT Specification requirements and constraints for the chip-level design.

Procedure

1. Activate boundary scan.

```
SETUP>set_dft_specification_requirements -boundary_scan on
```

2. Run Design Rule Checking (DRC) to ensure all constraints are correct and if there are no errors, move Tessent Shell from Setup to Analysis mode.

```
SETUP>check_design_rules
```

3. Create the "DftSpecification(design_name,id)" configuration wrapper and copy the newly created wrapper object to the variable "spec" to customize the specification later.

```
ANALYZE>set spec [create_dft_specification]
```

4. Set the value for the max_segment_length_for_logictest parameter in the BoundaryScan wrapper to 200.

```
ANALYZE>set_config_value \   $spec/BoundaryScan/
max_segment_length_for_logictest 200
```

5. Create and insert the hardware for the TAP and boundary scan into the design.

```
ANALYZE>process_dft_specification
```

6. Inspect the results to confirm the boundary scan division into smaller chain segments for inclusion with logic testing and, optionally, save the output for later reference.

   a. List the currently available instrument dictionaries created by the process_dft_specification command.

```
ANALYZE>get_instrument_dictionary -list
```

      The tool returns a list similar to that shown below:

```
DftSpecification LibraryCells RtlCells
mentor::ijtag::DftSpecification
mentor::jtag_bscan::DftSpecification
mentor::memory_bisr
tshell_global
```

   b. List the scan chains and scan_in/scan_out ports created. This also identifies how many scan chains segments were created.

```
ANALYSIS>format_dictionary [get_instrument_dictionary \
mentor::jtag_bscan::DftSpecification  logic_test_scan_chains]
```

   c. (Optional) Save the output created in step 6.b for later reference using the following script.

```
set fp [open logic_test_scan_chains.dictionary w]
puts $fp "set logic_test_scan_chains {"
puts $fp [format_dictionary [get_instrument_dictionary
mentor::jtag_bscan::DftSpecification  logic_test_scan_chains ] ]
puts $fp "}"
close $fp
```

This saves the output to the *logic_test_scan_chains.dictionary* file.

7. Extract the ICL, generate the patterns, then simulate the patterns.

```
set_system_mode setup

extract_icl

set_context patterns -ijtag

check_design_rules

set spec [create_patterns_specification]

process_patterns_specification

run_testbench_simulations
```

Results

The single boundary scan chain between TDI and TDO is now segmented, with muxes added controlled by logic_test_enable. The single boundary scan chain connectivity is also maintained. You can use the instrument dictionary to include the boundary scan cells during EDT DFT insertion.

Related Topics

create_dft_specification [Tessent Shell Reference Manual]

set_config_value [Tessent Shell Reference Manual]

format_dictionary [Tessent Shell Reference Manual]

Pad Cell Input Path Considerations for Boundary Scan Testing

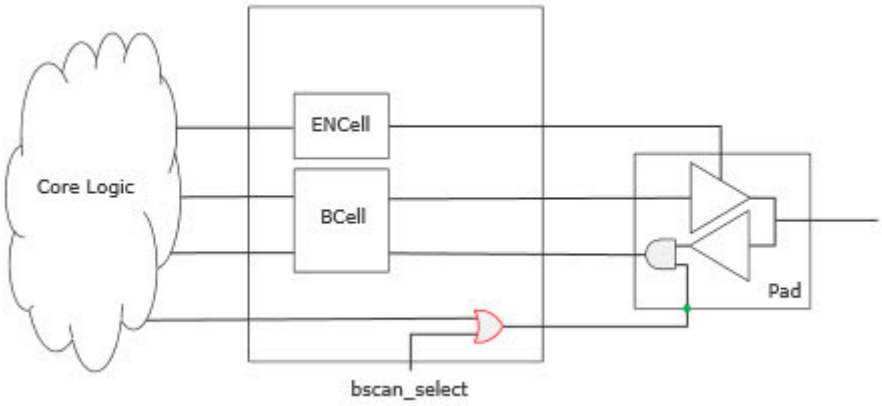
A simple pad cell input path consists of a plain buffer connecting the pad cell pin assigned the pad_pad_io pin attribute to the pin assigned the pad_from_pad attribute. Bidirectional pad cells also have a tri-state buffer on the output path. More complex input or bidirectional pad cells can have additional gating logic in the input path. The purpose of this additional gating logic in the input path is to prevent the core logic from toggling along with the signaling on the chip port.

During boundary scan testing, it is necessary to make the input path transparent so the value on the chip port can be observed in a boundary scan cell. Tessent Shell adds the necessary logic to

properly drive the pad cell input enable port during boundary scan testing. The appropriate pad_input_enable_high or pad_input_enable_low Pin Attributes need to be applied to the pad cell input enable pin for this to be implemented by Tessent Shell.

Figure 3-6 shows a bidirectional pad cell with an active high input enable pad cell pin, highlighted by a green connection dot. This pad cell needs to have the pad_input_enable_high pin attribute applied to this pin.

Figure 3-6. Bidirectional Pad Cell With Active High Input Enable



Tessent Shell automatically adds the appropriate logic, shown in red for this example, to achieve the transparency needed on the input path during boundary scan testing. If core logic drives the pad cell input enable, the added logic combines the boundary scan select with that signal. If the pad cell input enable was tied to a logic level, the added logic utilizes the same tie value. In the case of a pad cell configured with an auxiliary input, the auxiliary input enable signal is also logically combined to properly activate the pad cell input enable.

Pad Cell Library Attribute Considerations for Boundary Scan Testing

Certain pad cell attributes affect the way boundary scan logic is created and inserted into a design, depending on other considerations.

If a cell pad instance includes a port that has the pad_force_disable attribute, the corresponding pins on the instance are checked for the has_functional_source attribute when DFT hardware is generated with the process_dft_specification command. If this attribute is true, a mux is added during boundary scan insertion. The inputs are the functional source and force disable signals, and the select pin of the mux is connected to the boundary scan select signal.

If you use the “no insertion” flow (that is, the -no_insertion switch on the process_dft_specification command), you need to add this circuitry manually.

For a full description of pin attributes, see the Tessent Cell Library Manual.

Multiple Bonding Configurations

This section describes the process flow to add support for multiple package bonding configurations.

When you have multiple package bonding configurations, you need to edit the DftSpecification wrapper to add the various package options. One way, as demonstrated here, is to use the Config Data Browser to specify the bonding configurations and then read it into the main Tcl or dofile. This makes process repeatable and eliminates the need to invoke the Config Data Browser. You can also edit the DftSpecification as described in "Configure the DFT Specification in Memory" on page 42.

Prerequisites

- Load the Design steps have been completed and the design is loaded and elaborated.
- Specify and Verify DFT Requirements steps have been completed.

Procedure

1. Create a DFT specification and also direct the output to an environment variable. Report out the DFT specification for verification:

```
##To create a dft specification
ANALYSIS> set spec [create_dft_specification]

##Report on what dft spec was created
ANALYSIS> report_config_data $spec
```

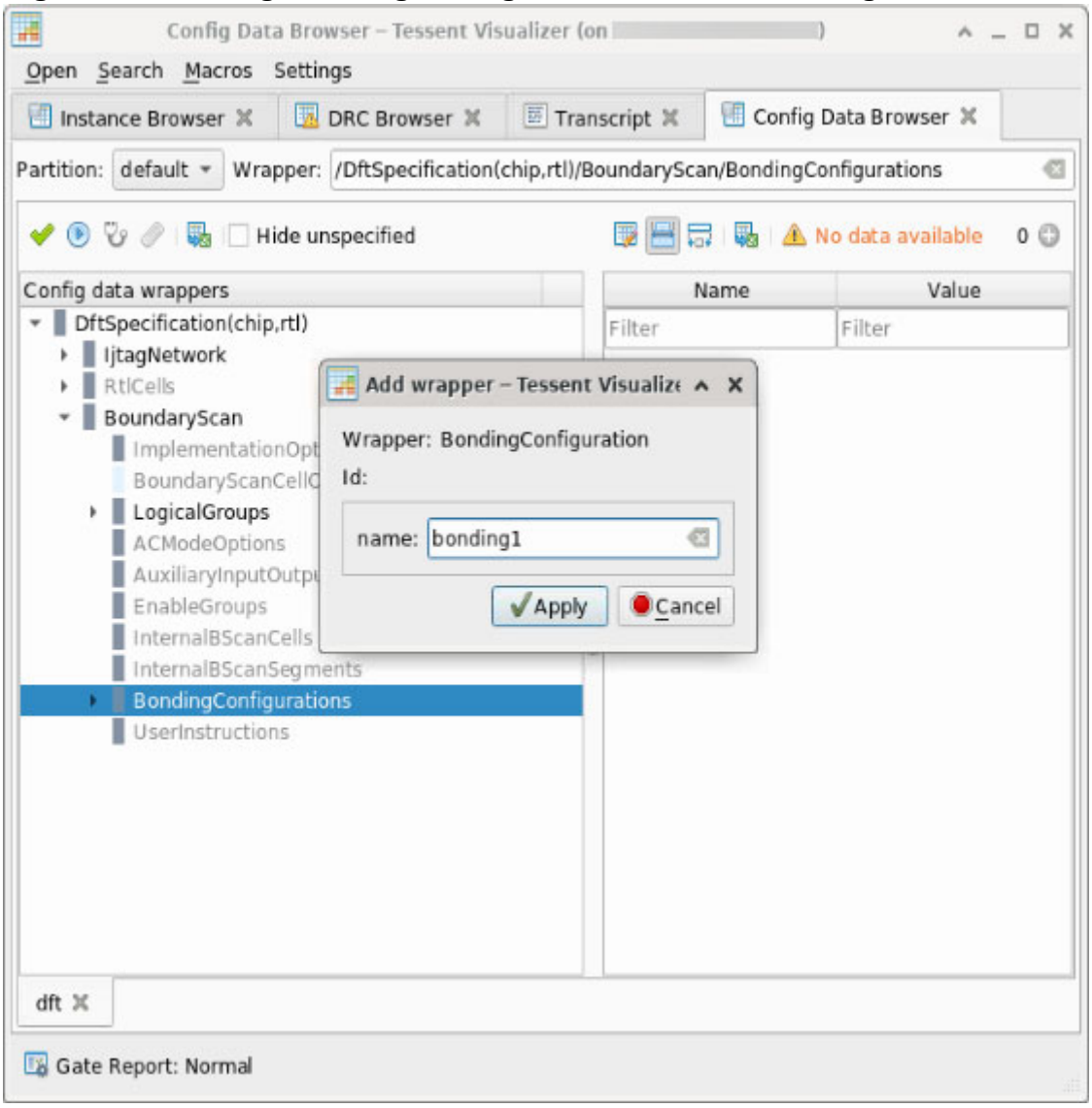
2. Invoke the Config Data Browser to modify the dft specification so multiple bonding configurations can be added. The Config Data Browser automatically opens and displays the DftSpecification(<design_name>,<design_id>) wrapper configuration when display_specification is run.

```
##Invoke the Config Data Browser and display the DftSpecification
ANALYSIS> display_specification
```

3. Expand the DftSpecification tree by clicking the triangle symbol to expand DftSpecification/BoundaryScan and display BondingConfigurations under BoundaryScan.
   a. Select BondingConfigurations in the tree and right click to display a menu of options available.
   b. Select **Add wrapper as a child** and type a name for the new BondingConfiguration in the dialog box that is displayed to add a configuration as shown in Figure 3-7.

      Repeat as needed for the number of bonding configurations required in your design. Two are added in this example.

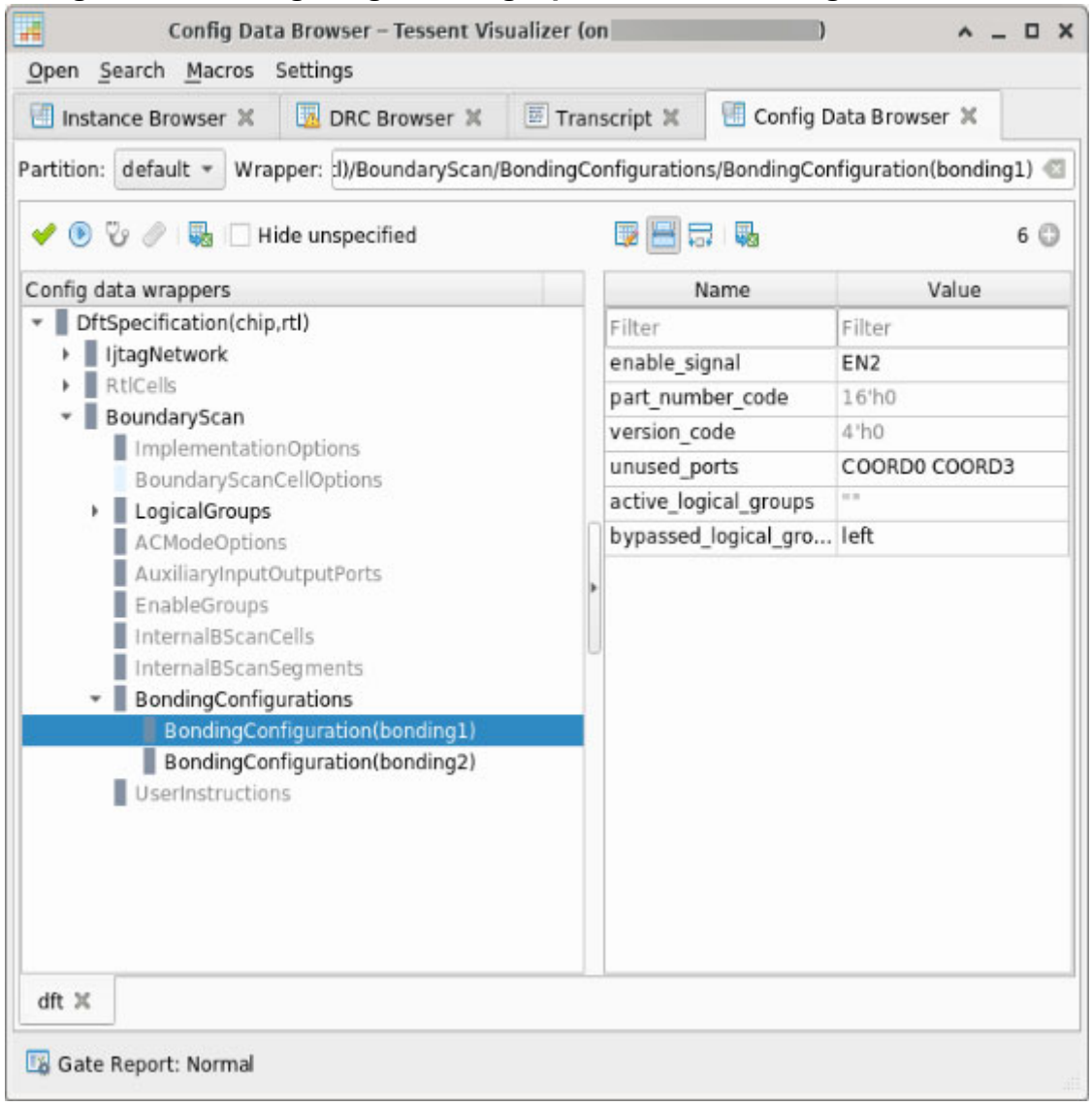
Figure 3-7. Adding Bonding Configurations With the Config Data Browser



4. Select a BondingConfiguration in the tree and, using the right mouse button context menu, fill in the options you want as shown in Figure 3-8.

   a. You must provide the names for a BondingConfiguration. This example uses “bonding1” and “bonding2.” Additionally, bonding1 has an enable signal EN2 and two unused ports, identified as COORD0 and COORD3. Finally, only the left logical group is bypassed during the bonding1 package option. If logical groups are bypassed, they must be provided in the pinorder file.

Figure 3-8. Configuring Bonding Options in the Config Data Browser



b. If needed, add additional unused ports and bypassed logical group in the same way.

For this example, the BondingConfiguration bonding2 has unused ports of D1[0], D1[1], D1[2] and D1[3].

5. Use report_config_data to list the current configuration:

```
ANALYSIS>report_config_data $spec
```

6. Once the configuration is reported, you can cut and paste this information into the dofile and read the Config Data Browser edits using the read_config_data command as shown in the following example. Then, for subsequent runs, the configuration edits are already present in the dofile, making the process repeatable with scripting.

```
read_config_data -in $spec/BoundaryScan -from_string {
BondingConfigurations +{
      BondingConfiguration(default) {
      }
      BondingConfiguration(bonding1) {
        enable_signal : EN2;
        unused_ports : COORD0, COORD3;
        bypassed_logical_groups : left;
      }
      BondingConfiguration(bonding2) {
        unused_ports : D1[2] D1[1] D1[0] D1[3];
      }
   }
}
```

7. If you are running Tessent Shell in the dft context with -no_rtl and the library does not have a clock_gating_and entry, the following operation needs to be completed to create the RTL for the AND clock gater.

   ```
   set_config_value /DftSpecification(car,gate)/use_rtl_cells On
   ```

8. Complete any other DftSpecification configurations that may be required and continue the rest of the design flow.

Custom Boundary Scan Cells

Custom boundary scan cells can be combined with pad cells in the overall boundary scan chain. Custom boundary scan cells need to already be inserted into the design and connected to the pad cell.

Note

Tessent Shell does not support the insertion of custom boundary scan cells.

The custom boundary scan cell is described using the same format as found in the *.tcd_bscan file. This file is normally created by the process_dft_specification when inserting a boundary scan chain into a sub or physical block, and the block contains a Core(<module_name>)/BoundaryScan wrapper. For this application, the wrapper is manually created to specify the custom boundary scan cell and interface. Tessent Shell automatically recognizes this description when matching modules within higher level parent modules and uses it to stitch the boundary scan chain in the parent module.

Examples

Example 1

This example shows a sample *.tcd_bscan* file for custom input cells, named *ipad_bscan_combo.tcd_bscan*, and the process used for stitching them with pad cells that are automatically inserted.

The contents of the custom *ipad_bscan_combo.tcd_bscan* file is listed below:

```
Core(ipad_bscan_combo){
 BoundaryScan {
  Interface {
    select_jtag_input : SJI_in;
    capture_shift_clock : bscan_clk;
    update_clock : update_bscan;
    shift_en : shift_bscan;
    scan_in : bscanIn;
    scan_out : bscanOut;
    scan_out_launch_edge : negedge;
  }
  ExternalPort(PAD) {
  }
  Cell(cell0) {
    function : input;
    bsdl_cell_type : BC_2;
    external_port : PAD;
  }
 }
}
```

The *ipad_bscan_combo.tcd_bscan* file is located in the netlist directory along with the *ipad_bscan_combo.v* Verilog file in this example.

```
SETUP>set_context dft -no_rtl

##Read the Tessent library for standard cells and pad cells.
SETUP>read_cell_library ../library/adk_complete.tcelllib

#Read the verilog
SETUP>read_verilog netlist/cpu_top.v
SETUP>set_design_sources -format verilog -Y netlist -extensions v
SETUP>read_verilog netlist/ipad_bscan_combo.v
SETUP>set_current_design cpu_top

##set_dft_specification_requirements cannot be specified until the design
has been read in.
ANALYSIS>set_dft_specification_requirements -boundary_scan On

##Need to set the design level before running check_design_rules
ANALYSIS>set_design_level chip
```

The remainder of the design flow steps are unchanged beyond this point, and would merge at the Add Constraints step within “Specify and Verify DFT Requirements” on page 32.

Example 2

This example shows a sample *.tcd_bscan* file for custom output cells. In this case, the netlist already has the output bscan cells inserted and connected, as well as a combinational cell for the enable. The process used for stitching the custom boundary scan outputs with pad cells that are automatically inserted is the same as that given in Example 1.

```
Core(opad_bscan_en_combo){
 BoundaryScan {
  Interface {
    select_jtag_output : SJO_in;
    force_disable      : forcedis;
    capture_shift_clock : bscan_clk;
    update_clock : update_bscan;
    shift_en : shift_bscan;
    scan_in : bscanIn;
    scan_out : bscanOut;
    scan_out_launch_edge : negedge;
  }
  ExternalPort(PAD) {
    buffer_type : three_state;
    control_cell : cell1;
  }
  Cell(cell0) {
    function : output;
    bsdl_cell_type : BC_2;
    external_port : PAD;
  }
  Cell(cell1) {
    function : control;
    bsdl_cell_type : BC_2;
    control_enable_value : 1;
  }
 }
}
```

Debugging Failing JtagBscan Simulations

Tessent Shell enables you to run and check testbench simulations. You can identify, view, and analyze failed simulation patterns. A series of examples is presented that outline the methods to accomplish each of these.

The command for running a set of simulation testbenches in Tessent Shell is run_testbench_simulations. This command is normally run with no arguments because it automatically uses the design name, design id, and pattern id found in the previously processed PatternsSpecification(design_name, design_id, pattern_id) wrapper.

The command check_testbench_simulations is used to check the status of simulations that were previously launched by run_testbench_simulations. Arguments can be passed to generate a status report or a status Tcl dictionary. If no argument is provided, the command updates a status line each second while running, and reports an error message for any failed simulations when completed.

Check Testbench Status Example

The following example runs a set of testbench simulations based on the design name, design id, and pattern id from the previously processed PatternsSpecification. Testbench simulations are checked and any failing pattern simulations are reported.

```
SETUP>run_testbench_simulation
SETUP>check_testbench_simulations
// Error: 1 out of 2 simulations failed:
// JtagBscanPatterns with 943 unexpected miscompares
```

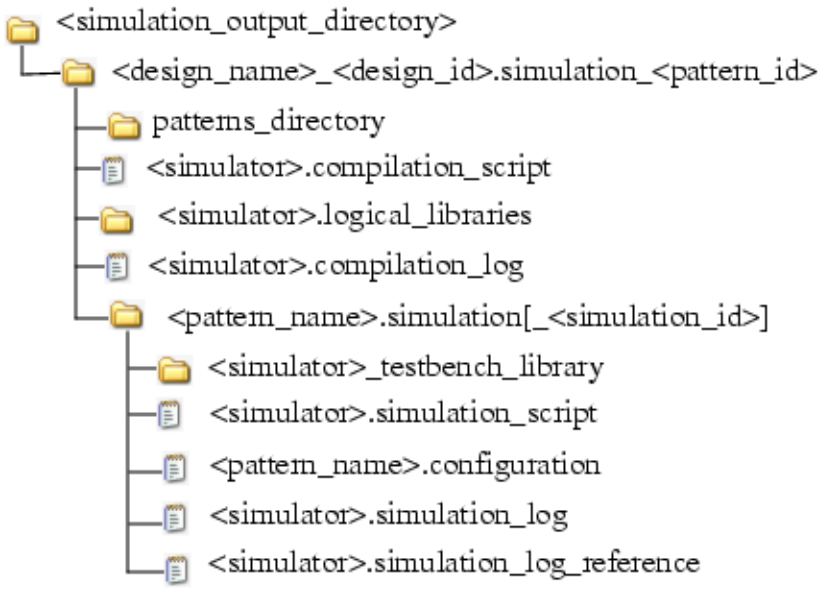
Identifying Detailed Simulation Status Example

This example shows another way to report testbench simulation status and the resulting output based on the same run_testbench_simulation shown in the previous example:

```
SETUP>check_testbench_simulations -report_status
// Simulation status for ./simulation_outdir/
cpu_top_gate.simulation_signoff
// ================================================================
// -----------------  ------  -----------  -----------  ------------
// Pattern Name       Status  Unexpected   Missing      Date
//                            Miscompares  Miscompares
// -----------------  ------  -----------  -----------  ------------
// ICLNetwork         pass       0            0         Thu May 07
// JtagBscanPatterns  fail       943          0         Thu May 07
```

The -report_status argument creates a report listing the simulation status of each pattern found inside the *<design_name>_<design_id>.simulation_<pattern_id>* directory as shown in Figure 3-9. This report identifies any pattern id that failed testbench simulation and needs to be re-run to capture and store waveform data for further debug in the waveform viewer.

Figure 3-9. Simulation Output Directory Contents



Viewing Simulation Waveforms for Analysis Example

The following example shows how you would run simulation on a specific pattern_id that failed testbench simulation and then save the simulation waveforms for viewing in the waveform window for further analysis.

Availability of the failing pattern and design environment should be confirmed, because a different PatternsSpecification could have been processed after the patterns that failed were run. The run_testbench_simulations command with the -report_list argument lists the patterns available in the *<design_name>_<design_id>.patterns_<pattern_id>/simulation.data_dictionary* file. This file is always updated upon successful validation and processing of the PatternsSpecification.

```
SETUP>run_testbench_simulations -report_list
List of pattern(s) for directory './tsdb_outdir/patterns/
cpu_top_gate.patterns_signoff':
  ICLNetwork  JtagBscanPatterns
```

Once the pattern availability is confirmed, as seen in the sample above with the listing of JtagBscanPatterns, the simulations can be re-run with waveform storage enabled.

```
SETUP>run_testbench_simulations -select JtagBscanPatterns \
       -store_simulation_waveforms on
```

Running run_testbench_simulations with the -store_simulation_waveforms argument enabled creates a *vsim.wlf* file in the directory where the JtagBscanPatterns were simulated.

In another Unix shell window, navigate to the folder where the *vsim.wlf* file was created and open it using the viewer in Questa SIM as shown in the example below:

```
UNIX>cd \
     simulation_outdir/cpu_top_gate.simulation_signoff/ \
     JtagBscanPatterns.simulation
UNIX>vsim -view vsim.wlf
```

While analyzing the waveforms of the failing patterns, you have the option of rerunning the simulation in the Tessent Shell window with new files or other options such as delay_mode_zero, unit_delay or others. After any adjustments are made, you can reload the new *vsim.wlf* file in the waveform window to view the results.

Restoring a Previous Simulation Analysis Session Example

If for some reason you quit a Tessent -shell session similar to that outlined in the previous example, and returned later to continue analysis, you can read the simulation library in the new

Tessent -shell session using the set_simulation_library_sources command. The session can be restored by following the sequence shown:

```
UNIX>tessent -shell -log simulation_debug.log
SETUP>set_context patterns
SETUP>set_simulation_library_sources \
      -v ../library/adk_complete.v -v ../library/picdram.v
SETUP>run_testbench_simulations -report_list
List of pattern(s) for directory './tsdb_outdir/patterns/
cpu_top_gate.patterns_signoff':
  ICLNetwork  JtagBscanPatterns

SETUP>run_testbench_simulations -select JtagBscanPatterns \
      -store_simulation_waveforms on -wait
```

In another unix shell window, navigate to the folder where the *vsim.wlf* file was created and open it using the viewer in Questa SIM as shown in the example below:

```
UNIX>cd \
     simulation_outdir/cpu_top_gate.simulation_signoff/ \
     JtagBscanPatterns.simulation
UNIX>vsim -view vsim.wlf
```

Related Topics

run_testbench_simulations [Tessent Shell Reference Manual]

check_testbench_simulations [Tessent Shell Reference Manual]

set_simulation_library_sources [Tessent Shell Reference Manual]

Chapter 4
MemoryBIST Insertion With BoundaryScan

This chapter describes the design flow that can be followed if TAP, BoundaryScan, and MemoryBIST are to be inserted for a design.

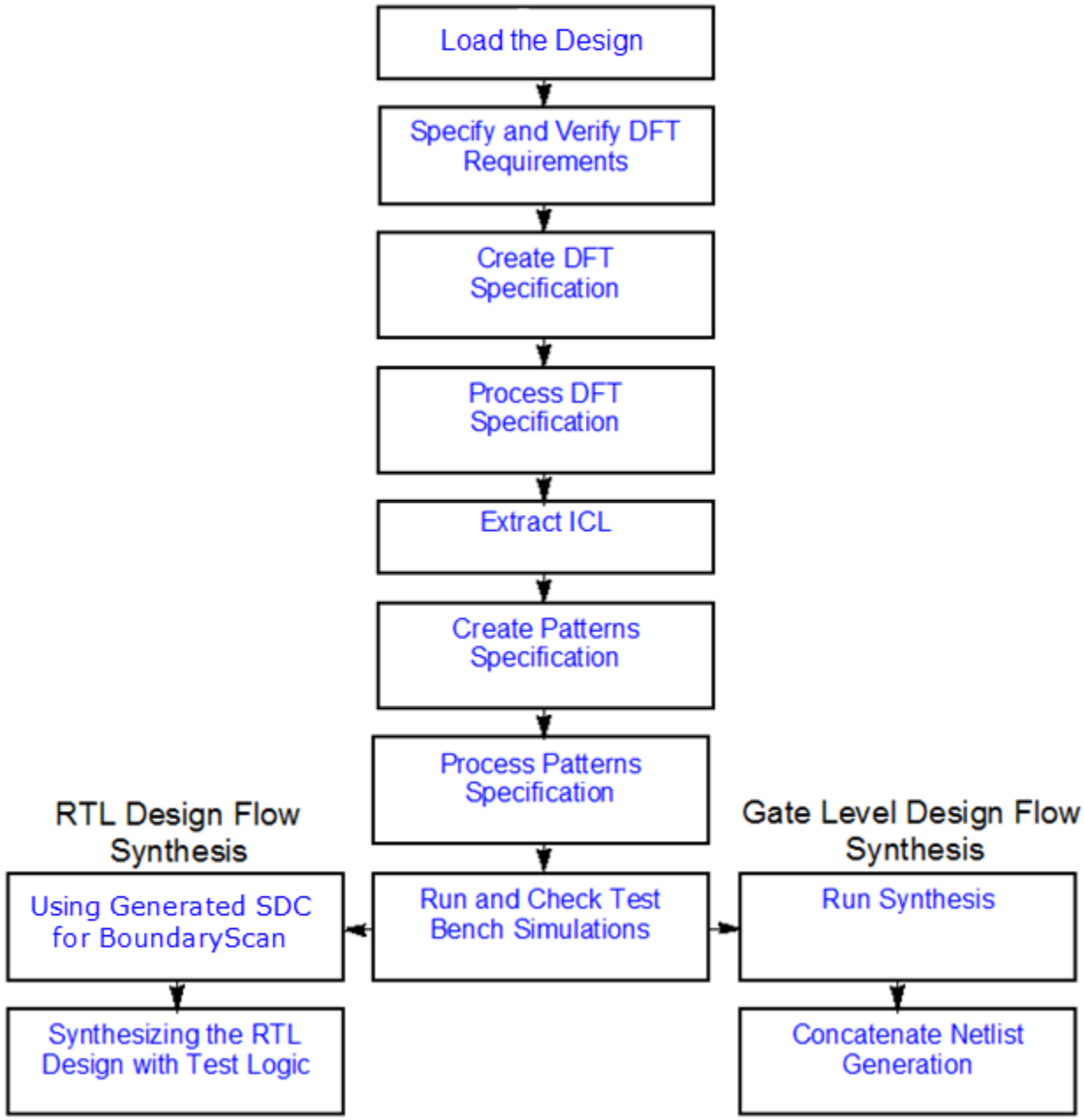
Overview

Two example design flows are described that show different methods of inserting MemoryBIST with BoundaryScan and TAP into a design.

One design flow inserts MemoryBIST, BoundaryScan and TAP in a single pass at the chip design level. The other flow inserts MemoryBIST in a first pass within a sub_block or physical_block, then inserting BoundaryScan and TAP in a second pass at the chip level. When MemoryBIST needs to be inserted inside a physical_block or a sub_block, then the design level needs to also be set to physical_block or a sub_block. A physical_block describes a design module where physical layout has been completed, whereas a sub_block describes a design module that can be instantiated inside another layout region. The layout region can be another design module or the entire chip.

The design flow that is used to insert MemoryBIST with BoundaryScan and TAP is the same as described in Getting Started With Tessent BoundaryScan and shown in Figure 4-1.

Figure 4-1. Design Flow for Tessent Shell



Related Topics

Tessent MemoryBIST User's Manual for use With Tessent Shell

TAP, BoundaryScan, and MemoryBIST

The example provided in this section covers the design implementation for MemoryBIST, BoundaryScan, and TAP all located at the chip design level. The example does this in a single

implementation pass where the tool generates and inserts the TAP, BoundaryScan, and MemoryBIST at the same time. You could perform the generation and insertion for each in three separate passes.

Load the Design steps. In this step, the design and libraries are loaded in:

```
set_context dft -no_rtl
read_cell_library ../library/adk_complete.tcelllib
read_cell_library ../library/memory.lib
set_design_sources -format tcd_memory -V ../library/picdram.memlib
read_verilog ../netlist/cpu_top.v
set_current_design cpu_top
```

Specify and Verify DFT Requirements steps. In the example below, set_dft_specification_requirements has both "-boundary_scan On" and "-memory_test On". The set_design_level command is set to "chip". The DRC is run when you run the command check_design_rules command.

```
set_design_level chip
set_dft_specification_requirements  -memory_test on \
                                    -boundary_scan on

set_attribute_value tck_p -name function -value tck
set_attribute_value tdi_p -name function -value tdi
set_attribute_value tms_p -name function -value tms
set_attribute_value trst_p -name function -value trst
set_attribute_value tdo_p -name function -value tdo
set_attribute_value vdd* -name function -value power
set_attribute_value vss* -name function -value ground
add_clocks 0 ramclk_p -Period 5ns
set_boundary_scan_port_options ramclk_p -cell_options clock
check_design_rules
```

Create DFT Specification steps:

```
set spec [create_dft_specification]
report_config_data $spec
//display_spec
set_config_value /DftSpecification(cpu_top,gate)/use_rtl_cells On
```

Process DFT Specification and Extract ICL steps:

```
process_dft_specification
extract_icl
```

Create Patterns Specification steps:

```
set pat_spec [create_pat_specification]
report_config_data $pat_spec
```

Process Patterns Specification step:

```
process_patterns_specification
```

Run and Check Testbench Simulations steps:

```
set_simulation_library_sources -v ./library/adk_complete.v \
                               -v ./library/picdram.v
run_testbench_simulations
check_testbench_simulations
check_testbench_simulations -report_status
```

Related Topics

set_simulation_library_sources [Tessent Shell Reference Manual]

run_testbench_simulations [Tessent Shell Reference Manual]

check_testbench_simulations [Tessent Shell Reference Manual]

MemoryBIST Insertion Before Tap and BSCAN

The example provided in this section covers the initial design insertion for MemoryBIST on a sub_block or physical_block, and finally the insertion of TAP and BoundaryScan at the chip level on a second pass.

If default tsdb_outdir is not used for MemoryBIST, then while inserting TAP and BoundaryScan, use the open_tsdb command to point to the tsdb outdir of where the MemoryBIST is inserted.

Procedure

1. MemoryBIST insertion: Load the Design steps:

   ```
   set_context dft -rtl
   read_cell_library ../library/adk.tcelllib
   set_design_sources -format verilog \
        -y {../data/design/mem ../data/design/rtl} -extension v
   read_verilog ../data/design/rtl/blockA.v
   set_current_design blockA
   report_memory_instances
   ```

2. MemoryBIST insertion: Specify and Verify DFT Requirements steps. MemoryBIST is inserted on a sub_block level.

   ```
   set_design_level sub_block
   set_dft_specification_requirements -memory_test on
   add_clock CLK -period 12ns -label clka
   check_design_rules
   ```

3. MemoryBIST insertion: Create DFT Specification steps:

```
create_dft_specification
report_config_data
report_config_syntax DftSpecification/MemoryBist
```

4. MemoryBIST insertion: Process DFT Specification and Extract ICL steps:

```
process_dft_specification
extract_icl
```

5. MemoryBIST insertion: Create Patterns Specification steps:

```
set spec [create_patterns_specification]
report_config_data $spec
```

6. MemoryBIST insertion: Process Patterns Specification step:

```
process_patterns_specification
```

7. MemoryBIST insertion: Run & Check Testbench Simulations steps:

```
run_testbench_simulations
check_testbench_simulations
```

8. The prior seven steps need to be repeated for any other sub_blocks or physical_blocks with memories that need MemoryBIST insertion before the design level is changed to chip level.
9. At the chip-level, if you have memories that need MemoryBIST insertion, you create a DFT Specification as shown below. This inserts the MemoryBIST as well as the TAP and BoundaryScan cells.

Note

If the default “tsdb_outdir” is not used for MemoryBIST insertion, then you must use the open_tsdb command to point to the “tsdb outdir” for where the MemoryBIST is inserted while inserting TAP and BoundaryScan cells.

10. BoundaryScan and Tap insertion: Load the Design steps:

```
set_context dft -rtl
read_cell_library ../library/adk.tcelllib
read_verilog ../data/design/rtl/top.v \
             ../data/design/fusebox/*.v

set_current_design top
```

11. BoundaryScan and Tap insertion: Specify and Verify DFT Requirements steps. The TAP signals can be specified here, as shown in Step 2 of TAP, BoundaryScan, and MemoryBIST, or the specified by the DefaultsSpecification as assumed in this example procedure.

```
set_dft_specification_requirements  -boundary_scan on
set_design_level chip
add_clocks clka -period 3ns
add_clocks clkb -period 12ns
set_attribute_value vddq -name function -value power
set_attribute_value vss -name function -value ground
check_design_rules
```

12. BoundaryScan and Tap insertion: Create DFT Specification steps:

```
set spec [create_dft_spec]
cat top.bisr_segment_order
report_conf_data $spec
```

13. BoundaryScan and Tap insertion: Process DFT Specification and Extract ICL steps:

```
process_dft_specification
extract_icl
```

14. BoundaryScan and Tap insertion: Create Patterns Specification steps:

```
set_defaults_value \
   PatternsSpecification/SignOffOptions/ \
    simulate_instruments_in_lower_physical_instances on
set spec [create_patterns_specification]
```

15. BoundaryScan and Tap insertion: Process Patterns Specification step:

```
process_patterns_specification
```

16. BoundaryScan and Tap insertion: Run and Check Testbench Simulations steps:

```
set_simulation_library_sources -v \
   ../library/verilog/adk.v
run_testbench_simulations
exit
```

Related Topics

run_testbench_simulations [Tessent Shell Reference Manual]

set_simulation_library_sources [Tessent Shell Reference Manual]

Chapter 5
BSDL Extraction

The BoundaryScan DFT insertion process adds new BoundaryScan cells and connects existing embedded BoundaryScan segments together. It creates and adds a BoundaryScan interface block. It adds a TAP controller during IJTAG DFT insertion if it does not already exist. In addition, it creates a Boundary Scan Description Language (BSDL) file.

When you run BoundaryScan DFT insertion at a design level other than the top level, the tool does not create a BSDL file. In this case, use the processes described in this document to create a BSDL file.

Prerequisites for BSDL Extraction

You must use version 2020.4 or later of Tessent Shell to use this flow. In addition, several structures must be in place before you can extract the BSDL file.

TAP Controller

A TAP controller must be present and connected to the test access ports. You must also create an ICL description with the tessent_instruction_reg attribute pointing to the ICL instruction register. If you instantiate the TAP controller into a design block, that block must be loaded with its full view so that the block is visible to the tool. The TAP controller must also meet all requirements for DFT insertion. See "Requirements on a TAP to be usable for BoundaryScan" in the *Tessent Shell Reference Manual* for more information.

Embedded BoundaryScan Segments

You must include a tcd_bscan description for all blocks containing BoundaryScan cells. Connect all pins of these blocks that are described with an ExternalPort wrapper to chip-level ports. Connect the scan-in and scan-out ports of embedded BoundaryScan ports to a scan chain that is also connected to the TAP ports and the TAP controller. Connect other interface pins to the TAP controller or to the TAP ports.

BondingConfiguration Bypasses

Any BondingConfiguration bypasses must be at the block level. Define these in the tcd_bscan file within a SegmentSelection wrapper. You can run Tessent Shell at the block level and use the BondingConfigurations wrapper to create these bypasses. For extraction, choose a configuration by applying the appropriate enable signal values. You can only extract one configuration at a time.

Extracting the BSDL File

Follow this procedure to extract the BSDL file.

Procedure

1. Set the DFT context.
2. Read the design.
3. Read the ICL description.
4. Read the tcd_bscan description.
5. Elaborate the design.
6. Enable the extraction:

   ```
   set_dft_specification_requirements -bsdl_extraction on
   ```

   Restriction

   You cannot extract Boundary Scan cells created during chip-level DFT insertion. The tool returns DRC violations if you attempt to extract these cells.

   You can also specify a suffix for the BSDL file with the -bonding_configuration switch (optional). This is useful if, for example, you have multiple bonding configurations and need to maintain several different BSDL files.

Results

The tool extracts the BSDL during check_design_rules, or when you set the system mode to “analysis.” The tool identifies and reports the TAP ports during the extraction. For example:

```
//  Identified tck port: jtag_in[2]
//  Identified tdi port: jtag_in[0]
//  Identified tdo port: jtag_out[0]
//  Identified tms port: jtag_in[1]
//  Identified trst port: jtag_in[3]
```

During BSDL extraction, perform quick synthesis and flatten the design. Tessent Shell preserves the TAP controllers, the tcd_bscan blocks, and all of the SegmentSelection enable signals. Finally, the tool performs the BoundaryScan design rule checks.

The tool identifies and reports the TAP controller. For example:

```
//  The identified TAP controller is
'blockdft_inst/blockdft_rtl_tessent_tap_main_inst' with the ICL module
description 'blockdft_rtl_tessent_tap_main'.
//  The ICL host scan interface of the TAP controller for BoundaryScan is
'host_bscan'.
```

The tool identifies and reports the BoundaryScan chain. For example:

```
//  Extracted tcd_bscan sequence:
//    core1_instance1
//    core2_instance2
//    core3_instance3
//    core1_instance4
//    core5_instance5
```

The tool identifies and reports the segment selection, if any. For example:

```
//  Selected SegmentSelection wrapper of tcd_bscan instances:
//    core2_instance2 uses segment selection 'Sel1'.
//    core5_instance5 uses segment selection 'Mode1'.
```

The tool reports the extracted BSDL file. For example:

```
//  Extracted BSDL file: ./tsdb_outdir/dft_inserted_designs/
top_rtl.dft_inserted_design/top.bsdl
```

If the *dft_inserted_designs* directory does not exist, the tool reports an error. When you run the create_patterns_specification command, the tool will find the file in this location if the -bsdl_files option is not specified. If you load the TSDB with the design ID and write the data with a new design ID, the tool transfers the BSDL file to your new *dft_inserted_designs* directory.

Related Topics

set_dft_specification_requirements [Tessent Shell Reference Manual]

Pattern Generation

Use the BSDL-only flow to generate patterns for the extracted BSDL files.

Procedure

1. Invoke Tessent Shell with or without loading the design data.
2. Set the patterns context.
3. Create the patterns specification:

   ```
   create_patterns_specification
   ```

4. Process the patterns specification.

See "BSDL-Only Flow" on page 79 for complete information.

AC Signal Generation

If your design includes IEEE 1149.6 compliant AC pads or boundary scan cells, the tool adds a BoundaryScan interface block to generate the ac_model_en, ac_signal, ac_init_clock1, and ac_init_clock0 signals during the standard chip-level DFT insertion flow. When you create the chip level by hand, you must also create this module using the create_ac_control property in the BoundaryScan wrapper.

The following is an example of a DftSpecification that includes the create_ac_control property:

```
DftSpecification(top,rtl) {
  IjtagNetwork {
    HostScanInterface(tap) {
      …
      Tap(main) {
        parent_instance : dft_block ;
        HostBscan {
        }
      }
    }
  }
  BoundaryScan {
    ijtag_host_interface : Tap(main)/HostBscan;
    interface_parent_instance : dft_block ;
    create_ac_control : on ;
    BoundaryScanCellOptions {
      * : dont_touch;
    }
  }
}
```

The tool creates a TAP controller and a dot6 control block. Both are instantiated in the dft_block instance, with the dot6 control block connected to the TAP controller.

Blocks With Multiple Embedded Boundary Scan Implementations

More than one BoundaryScan interface block can be present in a design. Each interface block has its own set of interface ports to handle different logical groups. However, to perform BSDL extraction in this situation, you must take some additional measures.

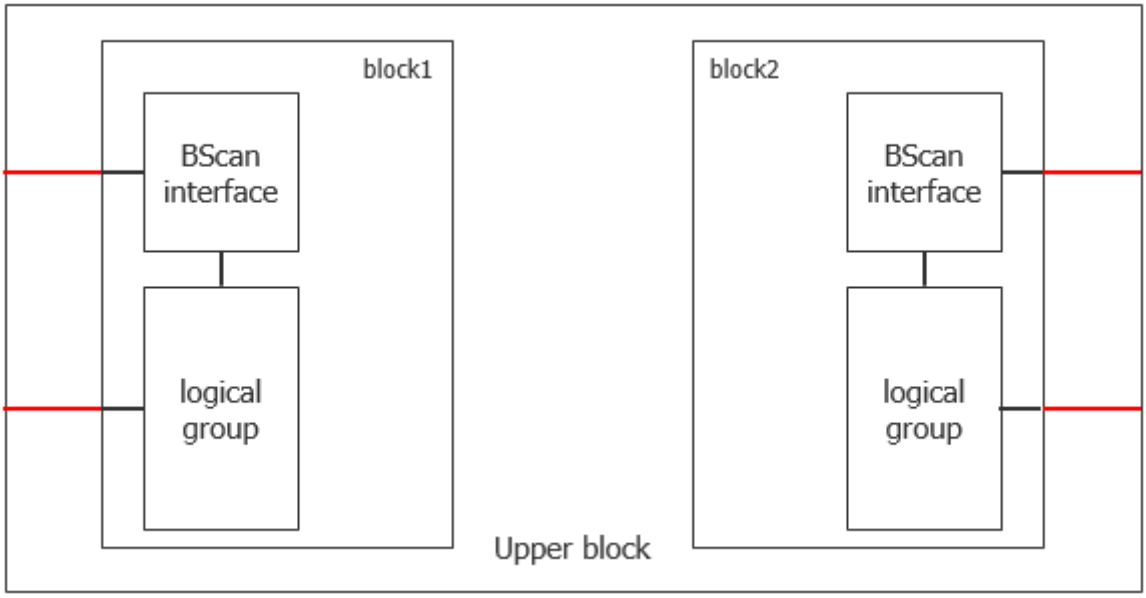
Multiple BoundaryScan hardware implementations are currently not supported in the standard DFT flow. To work around this limitation, you can run the BoundaryScan DFT insertion at a lower level of hierarchy so the tool can process one interface block at a time.

If your design does not have lower hierarchical levels such as block1 and block2 in Figure 5-1, you must add them. Connect the control and scan signals of the BoundaryScan interface blocks

to the boundary of the upper block, and from there to the external ports (shown in red in the figure).

Run the BoundaryScan DFT insertion on each block separately to obtain a tcd_bscan file for each block. Do not run BoundaryScan DFT insertion on the upper block. During extraction, load the full view of the upper block to make the lower levels visible to the tool. The tool finds them and uses the block1 and block2 tcd_bscan descriptions to perform the extraction.

Figure 5-1. Multiple BoundaryScan Interface Blocks

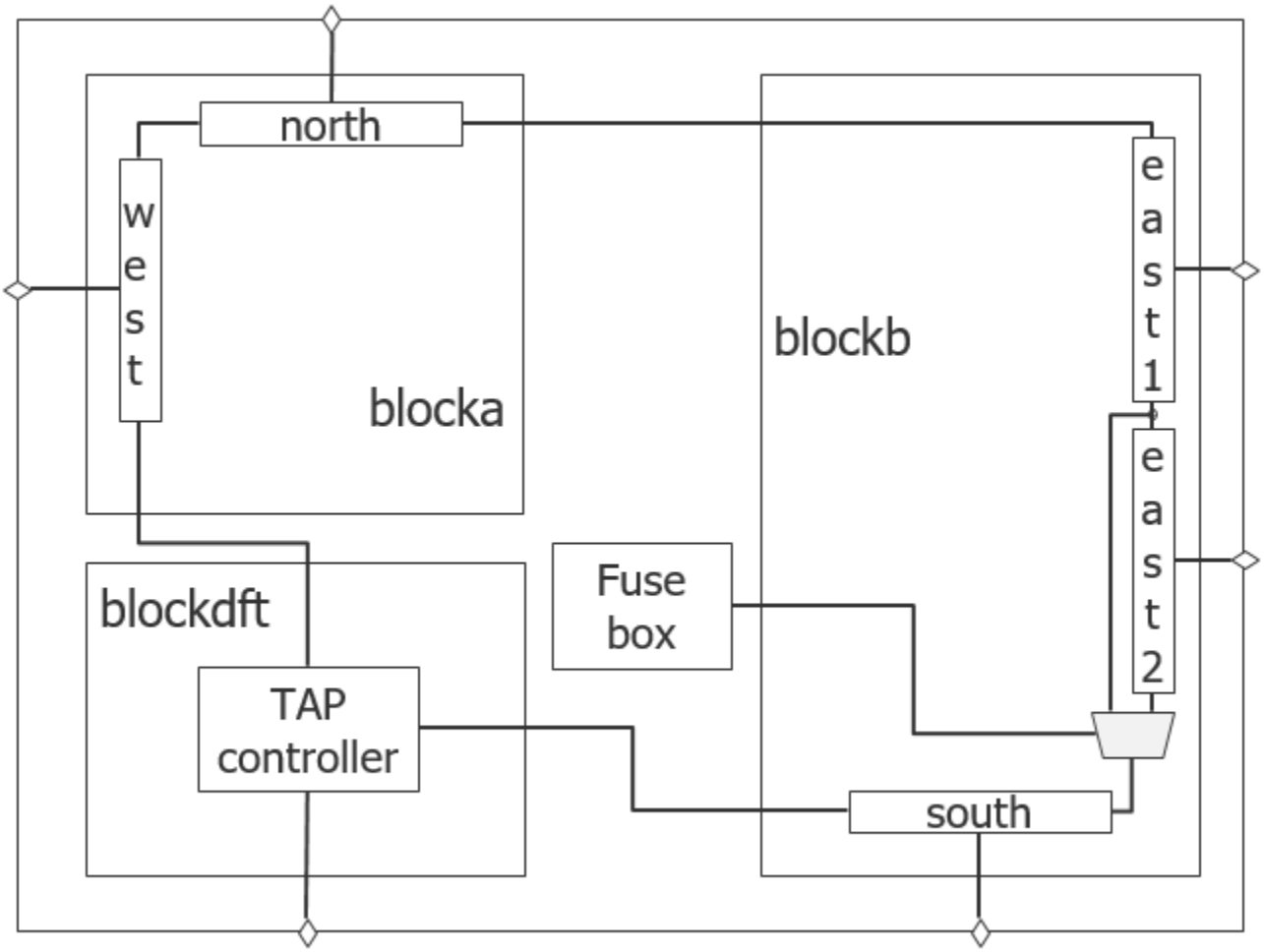


Example Test Case

This test case consists of three blocks.

- The blocks “blocka” and “blockb” include pad cells connected to chip-level ports.
- The block “blockdft” includes five pad cells that are connected to the chip-level TAP ports.

Figure 5-2. Example Test Case



The test case consists of nine steps:

1. The blocka/01.bscan_insertion/run script inserts BoundaryScan for blocka. The DFT insertion creates two logical groups:
   - The logical group “west” contains the BoundaryScan cells for io_west[3:0].
   - The logical group “north” contains the BoundaryScan cells for io_north[3:0].

   The tool extracts the ICL after DFT insertion.

```
#! /bin/sh -f
#\
exec tessent -shell -log $0.log -replace -dofile "$0"
# blocka/01.bscan_insertion/run
# Script does EBscan insertion for blocka
set_context dft -rtl -design_id rtl
# read the library containing the IOPAD_EN0 pad cell
read_cell_library ../../library/mentor/Pads.lib
# read and elaborate the design
read_verilog ../design/blocka.v
set_current_design blocka
set_design_level physical_block
set_tsdb_output_directory ../tsdb_outdir
# boundary scan settings
set_dft_specification_requirements -boundary_scan on
set_boundary_scan_port_options -pad_io_ports {io_west io_north}
# check the design and the boundary scan settings
check_design_rules
# DFT spec
set spec [create_dft_specification]
read_config_data -in_wrapper $spec/EmbeddedBoundaryScan -
from_string "
    Interface {
      select : bscan_select;
      bscan_clock : bscan_clock;
      force_disable : bscan_force_disable;
      select_jtag_input : bscan_select_jtag_input;
      select_jtag_output : bscan_select_jtag_output;
      capture_en : bscan_capture_en;
      shift_en : bscan_shift_en;
      update_en : bscan_update_en;
      scan_in : bscan_scan_in;
      scan_out : bscan_scan_out;
    }
    LogicalGroups {
      LogicalGroup(west) {
        first_port : io_west[3];
        parent_instance : pads_west;
      }
      LogicalGroup(north) {
        first_port : io_north[3];
        parent_instance : pads_north;
      }
    }
"
#report_config_data $spec
# BoundaryScan DFT insertion
process_dft_specification
# ICL extraction
set_system_mode setup
extract_icl
exit
```

2. The blocka/02.bscan_patterns/run script generates BoundaryScan patterns for blocka. This step creates the Verilog testbench for blocka and runs the simulation.

```
#! /bin/sh -f
#\
exec tessent -shell -log $0.log -replace -dofile "$0"
# blocka/02.bscan_patterns/run
# Script does EBscan pattern generation for blocka
set_context patterns -ijtag -rtl -design_id rtl
# read the library containing the IOPAD_EN0 pad cell
read_cell_library ../../library/mentor/Pads.lib
set_simulation_library_sources -extension v -y ../../library/
verilog
# read and elaborate the design
set_tsdb_output_directory ../tsdb_outdir
read_design blocka
set_current_design blocka
# check the design
check_design_rules
# patterns spec
set spec [create_patterns_specification]
#report_config_data $spec
# BoundaryScan patterns generation
process_patterns_specification
# testbench simulation
run_testbench_simulation
exit
```

3. The blockb/01.bscan_insertion/run script inserts BoundaryScan for blockb. The DFT insertion creates three logical groups:

   - The logical group “east1” contains the BoundaryScan cells for io_east1[3:0].
   - The logical group “east2” contains the BoundaryScan cells for io_east2[3:0].
   - The logical group “south” contains the BoundaryScan cells for io_south[3:0].

   The script defines two bonding configurations:

   - Bonding configuration “with_east2”. All three logical groups are part of the Boundary Scan chain.
   - Bonding configuration “without_east2”. The logical groups east1 and south are part of the Boundary Scan chain.

   When the enable signal port “bypass_east2” is set to 1, the logical group east2 is bypassed. The ICL is extracted after DFT insertion.

```
#! /bin/sh -f
#\
exec tessent -shell -log $0.log -replace -dofile "$0"
# blockb/01.bscan_insertion/run
# Script does EBscan insertion for blockb with a bypass for east2
set_context dft -rtl -design_id rtl
# read the library containing the IOPAD_EN0 pad cell
read_cell_library ../../library/mentor/Pads.lib
# read and elaborate the design
read_verilog ../design/blockb.v
set_current_design blockb
set_design_level physical_block
set_tsdb_output_directory ../tsdb_outdir
# boundary scan settings
set_dft_specification_requirements -boundary_scan on
set_boundary_scan_port_options -pad_io_ports {io_east1 io_east2
io_south}
# check the design and the boundary scan settings
check_design_rules
# DFT spec
set spec [create_dft_specification]
read_config_data -in_wrapper $spec/EmbeddedBoundaryScan -
from_string "
    Interface {
      select : bscan_select;
      bscan_clock : bscan_clock;
      force_disable : bscan_force_disable;
      select_jtag_input : bscan_select_jtag_input;
      select_jtag_output : bscan_select_jtag_output;
      capture_en : bscan_capture_en;
      shift_en : bscan_shift_en;
      update_en : bscan_update_en;
      scan_in : bscan_scan_in;
      scan_out : bscan_scan_out;
    }
    LogicalGroups {
      LogicalGroup(east1) {
        first_port : io_east1[3];
        parent_instance : pads_east1;
      }
      LogicalGroup(east2) {
        first_port : io_east2[3];
        parent_instance : pads_east2;
      }
      LogicalGroup(south) {
        first_port : io_south[3];
        parent_instance : pads_south;
      }
    }
    BondingConfigurations {
      BondingConfiguration(with_east2) {
      }
      BondingConfiguration(without_east2) {
        bypassed_logical_groups : east2;
        enable_signal : bypass_east2;
      }
    }
"
```

```
#report_config_data $spec
# BoundaryScan DFT insertion
process_dft_specification
# ICL extraction
set_system_mode setup
extract_icl
exit
```

4. The blockb/02.bscan_patterns/run script generates BoundaryScan patterns for blockb. This step creates two Verilog testbenches (one for each bonding configuration) for blockb and runs the simulations. The enable signal is the port bypass_east2 on blockb. The tool handles this enable port automatically, but you must ensure that the correct values are applied to the internal block enable signals.

```
#! /bin/sh -f
#\
exec tessent -shell -log $0.log -replace -dofile "$0"
# blockb/02.bscan_patterns/run
# Script does EBscan pattern generation for blockb
set_context patterns -ijtag -rtl -design_id rtl
# read the library containing the IOPAD_EN0 pad cell
read_cell_library ../../library/mentor/Pads.lib
set_simulation_library_sources -extension v -y ../../library/
verilog
# read and elaborate the design
set_tsdb_output_directory ../tsdb_outdir
read_design blockb
set_current_design blockb
# check the design
check_design_rules
# patterns spec
set spec [create_patterns_specification]
#report_config_data $spec
# BoundaryScan patterns generation
process_patterns_specification
# testbench simulation
run_testbench_simulation
exit
```

5. The blockdft/01.tap_controller/run script inserts the TAP controller for blockdft and creates a host scan interface for BoundaryScan. The script connects the TAP controller to the TAP ports and defines a post-insertion Tcl proc. This proc connects the BoundaryScan host scan interface to the block boundary so that the BoundaryScan interfaces of blocka and blockb can connect to it. The tool extracts the ICL after DFT insertion.

```
#! /bin/sh -f
#\
exec tessent -shell -log $0.log -replace -dofile "$0"
# blockdft/01.tap_controller/run
# Script creates a TAP controller with an boundary scan interface
set_context dft -rtl -design_id rtl
# read the library containing the INPAD and the OUTPAD_EN0 pad cell
read_cell_library ../../library/mentor/Pads.lib
set_simulation_library_sources -extension v -y ../../library/
verilog
# read and elaborate the design
read_verilog ../design/blockdft.v
set_current_design blockdft
set_design_level physical_block
set_tsdb_output_directory ../tsdb_outdir
# check the design
check_design_rules
# DFT spec
set spec DftSpecification(blockdft,rtl)
read_config_data -from_string "
DftSpecification(blockdft,rtl) {
  IjtagNetwork {
    HostScanInterface(tap) {
      Interface {
        tck : tck;
        tms : tms;
        trst : trst;
        tdi : tdi;
        tdo : tdo;
      }
      Tap(main) {
        HostIjtag(1) {
        }
        HostBscan {
        }
      }
    }
  }
}
"
proc process_dft_specification.post_insertion { root_wrapper args }
{
  array set extra_args $args
  # connect the boundary scan interface to the block level
  set tap_module [get_modules [list blockdft_rtl_tessent_tap_main]]
  set tap_instance [get_instances -of_modules $tap_module]
  create_connection [create_port bscan_clock -direction output]
[get_pins [list tck] -of_instances $tap_instance]
  create_connection [create_port bscan_tdi -direction output]
[get_pins [list tdi] -of_instances $tap_instance]
  set port_names [list \
    fsm_state host_bscan_to_sel host_bscan_from_so force_disable \
    select_jtag_input select_jtag_output extest_pulse extest_train \
    capture_dr_en shift_dr_en update_dr_en \
  ]
  # create the block level ports
  set already_created_bus [list]
  foreach_in_collection pin [get_pins $port_names -of_instances
```

```
$tap_instance] {
    if { [get_single_attribute_value -name is_bus $pin] } {
      set pin_bus_name [get_single_attribute_value -name bus_name
$pin]
      if { $pin_bus_name ni $already_created_bus } {
        lappend already_created_bus $pin_bus_name
        set li [get_single_attribute_value -name bus_left_index
$pin]
        set ri [get_single_attribute_value -name bus_right_index
$pin]
        set port_name "bscan_[get_single_attribute_value -name
bus_name [get_ports -of_pins $pin]]\[$li:$ri\]"
        create_port $port_name -direction
[get_single_attribute_value -name direction $pin]
      }
    } else {
      set port_name "bscan_[get_single_attribute_value -name name
[get_ports -of_pins $pin]]"
      create_port $port_name -direction [get_single_attribute_value
-name direction $pin]
    }
  }
  # connect the block level ports to the TAP controller
  foreach_in_collection pin [get_pins $port_names -of_instances
$tap_instance] {
    set port [get_ports [list "bscan_[get_single_attribute_value -
name leaf_name $pin]"]]
    if { [get_single_attribute_value -name direction $pin] eq
"input" } {
      delete_connections $pin
    }
    create_connection $port $pin
  }
}
# TAP controller generation and insertion
process_dft_specification
# ICL extraction
set_system_mode setup
extract_icl
exit
```

6. The top/01.stitching/run script creates the chip level. The chip level includes instances of blocka, blockb, blockdft, and a fusebox that controls the bonding configuration bypass of blockb. Depending on a Verilog macro bypass, the enable output of the module is tied to 0 or 1. This macro is used during the BSDL extraction (see step 7) and the Verilog testbench simulation (see step 9) to bypass the logical group east2. Finally, the script creates all necessary connections between the instances and the chip level.

```
#! /bin/sh -f
#\
exec tessent -shell -log $0.log -replace -dofile "$0"
# top/01.stitching/run
# Script does top level creation
set_context dft -rtl -design_id rtl
# read the library containing the pad cells
read_cell_library ../../library/mentor/Pads.lib
# read the blocks
open_tsdb ../../blocka/tsdb_outdir
read_design blocka
open_tsdb ../../blockb/tsdb_outdir
read_design blockb
open_tsdb ../../blockdft/tsdb_outdir
read_design blockdft
# fusebox to control bypass_east2
read_verilog ../../fusebox/design/fusebox.v
set_tsdb_output_directory ../tsdb_outdir
set_system_mode insertion
set_current_design [create_module top]
set_design_level chip
# create the instances of the three blocks
set blocka_inst [create_instance blocka_inst -of_module blocka]
set blockb_inst [create_instance blockb_inst -of_module blockb]
set blockdft_inst [create_instance blockdft_inst -of_module
blockdft]
set ebscan_instances $blocka_inst
append_to_collection ebscan_instances $blockb_inst
# bypass select signal
set fusebox_inst [create_instance fusebox_inst -of_module fusebox]
create_connections [get_pins [list bypass] -of_instances
$fusebox_inst] [get_pins [list bypass_east2] -of_instances
$blockb_inst]
# connect the TAP controller to the TAP ports
foreach_in_collection pin [get_pins [list tck tms trst tdi tdo] -
of_instances $blockdft_inst] {
  set port [create_port [get_single_attribute_value -name leaf_name
$pin] -direction [get_single_attribute_value -name direction $pin]]
  create_connection $port $pin
}
# create the chip level ports
set already_created_bus [list]
foreach_in_collection pin [get_pins [list io_*] -of_instances
$ebscan_instances] {
  if {[get_single_attribute_value -name is_bus $pin] } {
    set pin_bus_name [get_single_attribute_value -name bus_name
$pin]
    if { $pin_bus_name ni $already_created_bus } {
      lappend already_created_bus $pin_bus_name
      set li [get_single_attribute_value -name bus_left_index $pin]
      set ri [get_single_attribute_value -name bus_right_index $pin]
      set port_name [get_single_attribute_value -name bus_name
[get_ports -of_pins $pin]]\[$li:$ri\]
      create_port $port_name -direction [get_single_attribute_value
-name direction $pin]
    }
  } else {
    set port_name [get_single_attribute_value -name name [get_ports
```

```
-of_pins $pin]]
    create_port $port_name -direction [get_single_attribute_value -
name direction $pin]
  }
}
# connect the pads of blocka and blockb to the chip level
foreach_in_collection pin [get_pins [list io_*] -of_instances
$ebscan_instances] {
  set port [get_ports [get_attribute_value_list -name leaf_name
$pin]]
  create_connection $port $pin
}
# connect the BScan interface of blocka and blockb to the TAP
controller
set bscan_connections [list \
  bscan_host_bscan_to_sel bscan_select \
  bscan_clock bscan_clock \
  bscan_force_disable bscan_force_disable \
  bscan_select_jtag_input bscan_select_jtag_input \
  bscan_select_jtag_output bscan_select_jtag_output \
  bscan_capture_dr_en bscan_capture_en \
  bscan_shift_dr_en bscan_shift_en \
  bscan_update_dr_en bscan_update_en \
]
foreach { blockdft_name block_ebscan_name } $bscan_connections {
  set block_dft_pin [get_pins [list $blockdft_name] -of_instances
$blockdft_inst]
  set block_ebscan_pins [get_pins [list $block_ebscan_name] -
of_instances $ebscan_instances]
  create_connections $block_dft_pin $block_ebscan_pins
}
# boundary scan chain stitching - blocka -> blockb -> blockdft
create_connection [get_pins [list bscan_tdi] -of_instances
$blockdft_inst] [get_pins [list bscan_scan_in] -of_instances
$blocka_inst]
create_connection [get_pins [list bscan_scan_out] -of_instances
$blocka_inst] [get_pins [list bscan_scan_in] -of_instances
$blockb_inst]
create_connection [get_pins [list bscan_scan_out] -of_instances
$blockb_inst] [get_pins [list bscan_host_bscan_from_so] -
of_instances $blockdft_inst]
write_design -tsdb
exit
```

7. The top/02.bsdl_extraction/run script extracts the BSDL. This step extracts two BSDL files:

   - *../tsdb_outdir/instruments/top_rtl_bscan.instrument/top_with_east2.bsdl* contains all BoundaryScan cells in the design.
   - *../tsdb_outdir/instruments/top_rtl_bscan.instrument/top_without_east2.bsdl* does not contain the cells from blockb for the logical group east2.

   The script reads the interface view for blocka and blockb. The tcd_bscan data describes all required information about the content of these blocks. The script reads the blockdft

block with the full view. This is necessary because the block contains the TAP controller.

After reading the design, the script sets the function attribute to define the TAP ports. For the extraction of the first BSDL file, it does not set the macro bypass (the fusebox_inst/bypass pin has a 0 value). This selects the SegmentSelection(with_east2) of blockb_inst. The script uses the set_dft_specification command is used to activate the extraction flow and set the suffix for the BSDL file to “with_east2.”

Use the check_design_rules command to extract the BSDL file, and repeat for the other bonding configuration of blockb. This time, set the macro bypass (the fusebox_inst/bypass pin has a 1 value). This selects the SegmentSelection(without_east2) of blockb_inst. Use the set_dft_specification_requirements command to activate the extraction flow again, with the suffix for the BSDL file defined to “without_east2.” Use the check_design_rules command to extract the BSDL file again.

```
#! /bin/sh -f
#\
exec tessent -shell -log $0.log -replace -dofile "$0"
# top/02.bsdl_extraction
# Script does the BSDL file extraction for top
set_context dft -rtl -design_id rtl
# read the library containing the pad cells
read_cell_library ../../library/mentor/Pads.lib
# read the blocks
open_tsdb ../../blocka/tsdb_outdir
read_design blocka -design_id rtl -view interface
open_tsdb ../../blockb/tsdb_outdir
read_design blockb -design_id rtl -view interface
open_tsdb ../../blockdft/tsdb_outdir
read_design blockdft -design_id rtl -view full
# read the top
set_tsdb_output_directory ../tsdb_outdir
read_design top -design_id rtl
set_current_design top
set_design_level chip
# mark the TAP ports
set_attribute_value -name function -value tck tck
set_attribute_value -name function -value tms tms
set_attribute_value -name function -value trst trst
set_attribute_value -name function -value tdi tdi
set_attribute_value -name function -value tdo tdo
# enable the BSDL file extraction
set_dft_specification_requirements -bsdl_extraction on -
bonding_configuration with_east2
puts "bypass enable signal from the fusebox:
[get_single_attribute_value -name tie_value [get_pins fusebox_inst/
bypass]]"
# BSDL file creation
check_design_rules
set_system_mode setup
# enable the bypass by setting the macro and re-reading the fusebox
module
set_design_macro bypass
read_verilog ../../fusebox/design/fusebox.v -force
set_current_design top
set_design_level chip
# second BSDL file with the bypass enabled
set_dft_specification_requirements -bsdl_extraction on -
bonding_configuration without_east2
puts "bypass enable signal from the fusebox:
[get_single_attribute_value -name tie_value [get_pins fusebox_inst/
bypass]]"
# BSDL file creation
check_design_rules
exit
```

8. The top/03.bscan_patterns/run script performs BoundaryScan pattern generation and creates the BoundaryScan testbench for the chip level. It invokes a new Tessent Shell session here to run the BSDL-only flow.

Note

Pattern generation with the ICL description of the chip level is not currently supported.

```
#! /bin/sh -f
#\
exec tessent -shell -log $0.log -replace -dofile "$0"
# top/03.bscan_patterns/run
# Script does pattern generation with the BSDL only flow
set_context patterns -ijtag -rtl -design_id rtl
set_tsdb_output_directory ../tsdb_outdir
set spec [create_patterns_specification -bsdl_files { \
  ../tsdb_outdir/instruments/top_rtl_bscan.instrument/
top_with_east2.bsdl \
  ../tsdb_outdir/instruments/top_rtl_bscan.instrument/
top_without_east2.bsdl \
}]
process_patterns_specification
exit
```

9. The top/04.simulation/run script performs the Verilog simulation. This step simulates the Verilog testbenches created earlier. The output value on fusebox_inst/bypass is set with a compilation macro bypass. This value controls the bypass multiplexer in the blockb_inst instance. You must be sure to apply the correct select value. Each simulation is done in a separate simulation output directory, because the compilation macro creates different versions of the fusebox module.

```
#! /bin/sh -f
#\
exec tessent -shell -log $0.log -replace -dofile "$0"
```

```
#! /bin/sh -f
#\
exec tessent -shell -log $0.log -replace -dofile "$0"
# top/04.simulation/run
# Script does the simulation for top
set_context patterns -ijtag -rtl -design_id rtl
# needed Verilog sources for the simulation
set_simulation_library_sources -extension v \
  -y ../../library/verilog \
  -y ../../blocka/tsdb_outdir/dft_inserted_designs/
blocka_rtl.dft_inserted_design/modified_rtl_files \
  -y ../../blocka/tsdb_outdir/instruments/
blocka_rtl_cells.instrument \
  -y ../../blocka/tsdb_outdir/instruments/
blocka_rtl_bscan.instrument \
  -v ../../blocka/tsdb_outdir/instruments/
blocka_rtl_bscan.instrument/blocka_rtl_tessent_bscan_cells.v \
  -y ../../blockb/tsdb_outdir/dft_inserted_designs/
blockb_rtl.dft_inserted_design/modified_rtl_files \
  -y ../../blockb/tsdb_outdir/instruments/
blockb_rtl_cells.instrument \
  -y ../../blockb/tsdb_outdir/instruments/
blockb_rtl_bscan.instrument \
  -v ../../blockb/tsdb_outdir/instruments/
blockb_rtl_bscan.instrument/blockb_rtl_tessent_bscan_cells.v \
  -y ../../blockdft/tsdb_outdir/dft_inserted_designs/
blockdft_rtl.dft_inserted_design/modified_rtl_files \
  -y ../../blockdft/tsdb_outdir/instruments/
blockdft_rtl_cells.instrument \
  -y ../../blockdft/tsdb_outdir/instruments/
blockdft_rtl_ijtag.instrument \
# point to the tsdb for top
set_tsdb_output_directory ../tsdb_outdir
# extra directories to force the new compilation of fusebox
run_testbench_simulations -design_name top -select
JtagBscanBonding1Patterns -simulation_output_directory
simulation_outdir_with_east2
run_testbench_simulations -design_name top -select
JtagBscanBonding2Patterns -simulation_output_directory
simulation_outdir_without_east2 -compilation_macro_definitions
bypass
exit
```

Chapter 6
Tap, BoundaryScan and LPCT Type 2 TestKompress

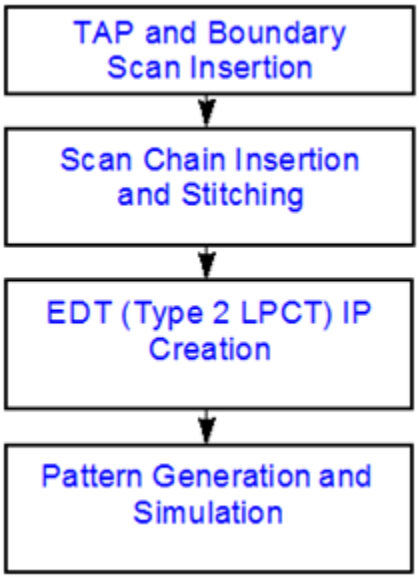
You can use the TAP to control TestKompress, where the TAP's TDI and TDO pins are used as the EDT channel in and channel out pins. The BoundaryScan chain is segmented into smaller Reduced Pin Count Test (RPCT) segments so they can also be part of the logic testing.

Overview

The benefit of implementing TestKompress with a Tap and Type 2 Low Pin Count Test (LPCT) controller and segmenting boundary scan cells into RPCT segments is that the combinational logic that is present between the boundary scan cell and the first tier or level of functional flops is tested during logic testing with TestKompress. An example design is provided that implements this architecture.

The design flow shown in Figure 6-1 shows the high-level overview of the steps that are implemented.

Figure 6-1. Tap, Boundary Scan, and LPCT Type 2 TK Design Flow



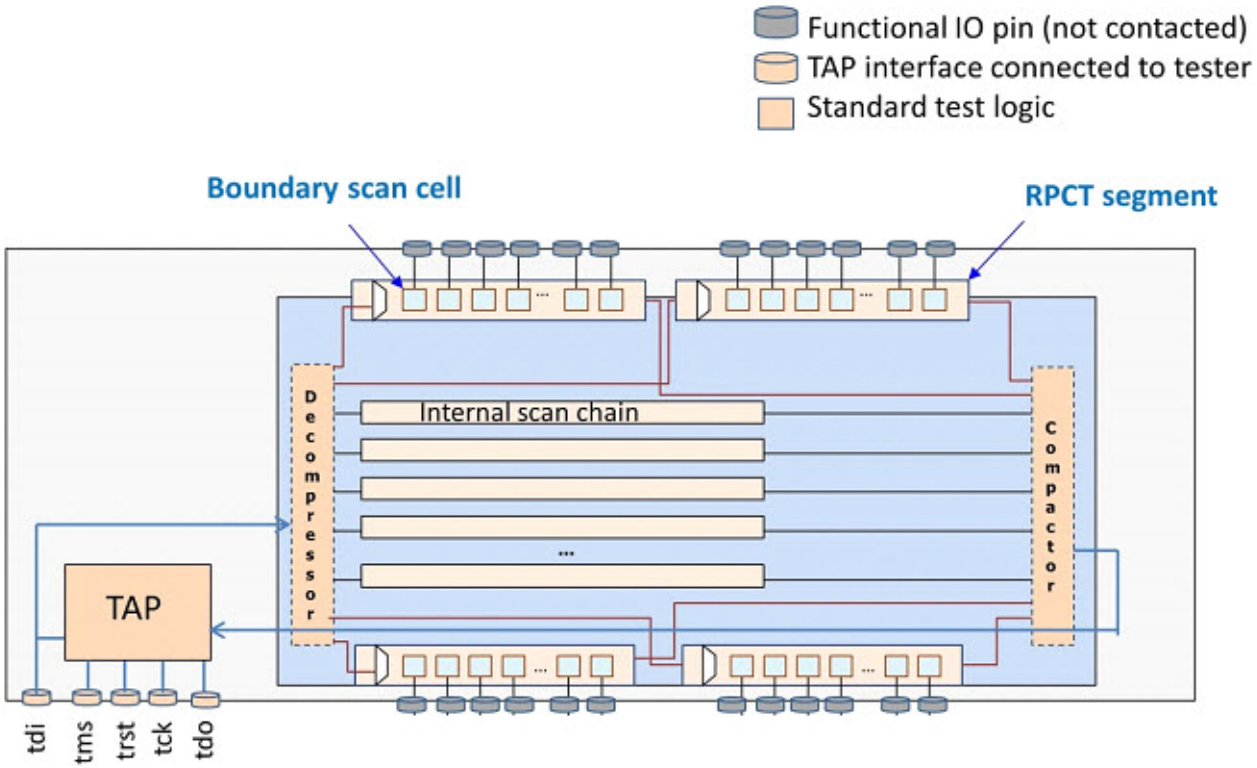
The TAP and boundary scan chains are inserted first, followed by scan insertion and stitching of the functional design flops. The segmented RPCT boundary scan chains are declared as preexisting scan chains for the scan insertion step.

In the IP creation step, the signals from the TAP controller that are needed to be connected to the Type 2 LPCT controller are specified. The TAP controller's TDI and TDO pins are used as the EDT channel in and channel out pins respectively.

Finally, patterns are created with TDI and TDO as the single channel for TestKompress. The TAP states are then cycled and validated via verilog simulation to ensure the design's integrity.

Figure 6-2 shows how a single boundary scan chain that is connected between the TDI and TDO of the TAP controller, gets divided into smaller RPCT (Reduced Pin Count Test) segments and how it is connected between the decompressor and the compactor of the TestKompress IP. For further information, refer to "Type 2 LPCT Controller" in the *Tessent TestKompress User's Manual*.

Figure 6-2. Tap, Boundary Scan and Type 2 LPCT TestKompress Implementation



Design Flow for TAP Control of TestKompress

The design steps, considerations and example scripts to implement this architecture are described in the following sections.

TAP and Boundary Scan Insertion

Insert the JTAG TAP and boundary scan chain.

The example script provides specific details that you can use as a guide for your design. This example follows the DFT Flow Using Tessent Shell steps.

Procedure

1. Estimate the scan chain length for the functional design cells.

   If the scan chain length for the functional design cells is unknown, pick a number for the boundary scan chain length to ensure it is not the longest chain, which could impact compression.

2. Apply the estimated scan chain length to segment the boundary scan chain into smaller segments.

   Use the max_segment_length_for_logictest property in the BoundaryScan wrapper to specify how many boundary scan cells are to be in each scan chain.

Examples

The following example script inserts a TAP and boundary scan chain following the DFT Flow Using Tessent Shell steps. The script also connects a TDR to the TAP controller that controls the boundary scan chain to be either a single chain or segmented and used with logic testing. The max_segment_length_for_logictest property sets the boundary scan chain length to 250 cells.

If you need to perform custom editing between the insertion of DFT components and the saving of the design, you can define a Tcl procedure with the name *process_dft_specification.post_insertion*. The tool calls this procedure automatically during process_dft_specification after it has inserted all DFT components and before you run the write_design command.

Note

The example script uses this technique to insert clock control logic that provides the proper clocking for segmented boundary scan chains during logic testing. Figure 6-3 shows the contents of this file, which you can customize to meet your design requirements. For more information on this method, refer to the process_dft_specification command description.

```
## Design loading
set_context dft -no_rtl
read_cell_library ../../libs/adk_complete.tcelllib
read_cell_library ../../libs/memory.lib
read_verilog ../netlist/cpu_top.v
set_current_design cpu_top

## Specify and verify DFT requirements
set_dft_specification_requirements -boundary_scan On
set_design_level chip
set_attribute_value tck_p -name function -value tck
set_attribute_value tdi_p -name function -value tdi
set_attribute_value tms_p -name function -value tms
set_attribute_value trst_p -name function -value trst
set_attribute_value tdo_p -name function -value tdo
set_boundary_scan_port_options ramclk_p -cell_options clock
check_design_rules

## Create DftSpecification
set spec [create_dft_specification]
set_config_value $spec/BoundaryScan/max_segment_length_for_logictest 250
read_config_data -in $spec/IjtagNetwork/HostScanInterface(tap)/ \
    Tap(main)/HostIjtag(1) -from_string {
          Tdr(logic_enable) {
            DataOutPorts {
              count       : 3;
              port_naming : ltest_enable, bypass_enable, \
                            low_power_enable;
            }
          }
    }
report_config_data $spec
set_config_value /DftSpecification(cpu_top,gate)/use_rtl_cells On

## Custom proc to be run after process_dft_specification
source post_dft_insertion_procedure.tcl

## Process DftSpecification
process_dft_specification

## Inspect the results to confirm the boundary scan is now segmented
## into smaller chain segments, as described in Step 5 of
## Chip Level Without Scan Insertion. The maximum
## length was set by max_segment_length_for_logictest to 250 flops.
get_instrument_dictionary -list
format_dictionary [get_instrument_dictionary
mentor::jtag_bscan::DftSpecification  logic_test_scan_chains]

## Save the above into a separate file called
## logic_test_scan_chains.dictionary for later reference.
set fp [open logic_test_scan_chains.dictionary w]
puts $fp "set logic_test_scan_chains {"
puts $fp [format_dictionary [get_instrument_dictionary
mentor::jtag_bscan::DftSpecification  logic_test_scan_chains ] ]
puts $fp "}"
close $fp
```

```
## Extract ICL
extract_icl

## Create PatternsSpecification
set pat_spec [create_patterns_specification]

## Without this the select of the mux on the functional clocks is X,
## causing simulation failures
set_config_value $pat_spec/AdvancedOptions/ConstantPortSettings/scan_en1
0
## Process PatternsSpecification
process_patterns_specification

## Run and check testbench simulation
set_simulation_library_sources -v ../../libs/adk_complete.v \
    -v ../../libs/pads.v -v ../../libs/ram.v
run_testbench_simulations
check_testbench_simulations
```

Figure 6-3. post_dft_insertion_procedure.tcl Example File (design name = cpu_top)

```
proc process_dft_specification.post_insertion {cpu_top args} {

## The following is needed for controlling the clocks for segmented
## boundary scan chain during logic/scan test

## Create a scan_en1 port
create_port scan_en1
## Delete the existing connection on logic_test_enable port
delete_connection cpu_top_gate_tessent_bscan_logical_group_DEF_inst/
logic_test_enable
## Create connection from the TDR to control logic_test_enable
## When this TDR is set to 1, we are in logic testing mode
create_connection cpu_top_gate_tessent_tdr_logic_enable_inst/ltest_enable
cpu_top_gate_tessent_bscan_logical_group_DEF_inst/logic_test_enable
```

```
## The clockbscan, updatebscan, and shiftBscan2Edge are intercepted by
## a mux. The other inputs of the clockbscan, updatebscan is tck
## whereas the other input of shiftBscan2Edge is scan_en1.
set tck_tap_point \
    [get_fanins cpu_top_gate_tessent_bscan_interface_I/tck]
intercept_connection cpu_top_gate_tessent_bscan_interface_I/ \
        to_bscan_capture_shift_clock \
    -cell_function_name mux \
    -input2 $tck_tap_point \
    -select cpu_top_gate_tessent_bscan_logical_group_DEF_inst/ \
        logic_test_enable \
    -leaf_instance_prefix clockBscan_
intercept_connection cpu_top_gate_tessent_bscan_interface_I/ \
        to_bscan_update_clock \
    -cell_function_name mux \
    -input2 $tck_tap_point \
    -select cpu_top_gate_tessent_bscan_logical_group_DEF_inst/ \
        logic_test_enable \
    -leaf_instance_prefix updateBscan_
intercept_connection cpu_top_gate_tessent_bscan_interface_I/ \
        to_bscan_shift_en \
    -cell_function_name mux \
    -input2 scan_en1 \
    -select cpu_top_gate_tessent_bscan_logical_group_DEF_inst/ \
        logic_test_enable \
    -leaf_instance_prefix shiftBscan2Edge_

## Here we do not have OCCs, so we add muxes to the functional clocks
## in the design.
intercept_connection /inpad0/C -cell_function_name mux \
    -input2 $tck_tap_point -select scan_en1 \
    -leaf_instance_prefix clk1_p_
intercept_connection /inpad1/C -cell_function_name mux \
    -input2 $tck_tap_point -select scan_en1 \
    -leaf_instance_prefix clk2_p_
intercept_connection /inpad2/C -cell_function_name mux \
    -input2 $tck_tap_point -select scan_en1 \
    -leaf_instance_prefix clk3_p_
intercept_connection /inpad3/C -cell_function_name mux \
    -input2 $tck_tap_point -select scan_en1 \
    -leaf_instance_prefix clk4_p_
intercept_connection /inpad4/C -cell_function_name mux \
    -input2 $tck_tap_point -select scan_en1 \
    -leaf_instance_prefix ramclk_p_
```

Related Topics

TAP and Boundary Scan Insertion

add_scan_chains [Tessent Shell Reference Manual]

read_icl [Tessent Shell Reference Manual]

Scan Chain Insertion and Stitching

“Reduced Pin Count Requirements” [Tessent TestKompress User's Manual]

"Generating and Verifying Test Patterns" [Tessent TestKompress User's Manual]

EDT (Type 2 LPCT) IP Creation

set_lpct_pins [Tessent Shell Reference Manual]

Creation of the EDT Logic [Tessent TestKompress User's Manual]

Scan Chain Insertion and Stitching

The second step is to insert scan chains for the rest of the functional design logic.

Prerequisites

- Complete TAP and Boundary Scan Insertion steps.

Procedure

1. Use the steps outlined in the example script provided as a guide to complete this stage of the design flow. Supporting scripts for the example are provided in Figure 6-4 through Figure 6-7.
2. Use the add_scan_chains -internal property to declare the boundary scan chains as preexisting, as shown by the red highlighted lines in the example script.

   The ICL file for the design created in TAP and Boundary Scan Insertion is read in using read_icl, and synthesized along with the functional design scan chains. A PDL file is also created and read in for the TAP controller and the TDR that is used to enable a segmented boundary scan chain.

Examples

```
set_context dft -scan
read_cell_library ../library/adk_complete.tcelllib
read_verilog ../from_step1/cpu_top.v_full
read_verilog ../from_step1/synthesized/boundary_scan_cells.v
read_verilog ../from_step1/synthesized/tessent_tap_main.v

##Reading icls and pdls before set_current_design
read_icl ../from_step1/tsdb_outdir/dft_inserted_designs/ \
    cpu_top_gate.dft_inserted_design/cpu_top.icl
dofile ./dofiles/cpu_top_gate_tessent_tap_main.pdl
dofile ./dofiles/cpu_top_gate_tessent_tdr_logic_enable.pdl

set_current_design cpu_top

## Add clock definitions
add_clocks 0 tck_p
add_clocks 0 clk1_p
add_clocks 0 clk2_p
add_clocks 0 clk3_p
add_clocks 0 clk4_p
add_clocks 0 ramclk_p


add_input_constraints trst_p -C1
add_input_constraints tms_p -C0

##Scan Enable scan_en1 as preExisting
set_scan_enable scan_en1
add_input_constraints scan_en1 -C0
add_nonscan_instances cpu_top_gate_tessent_tap_main_inst
add_nonscan_instances cpu_top_gate_tessent_tdr_logic_enable_inst

## Reading in preexisting bscan chains
dofile ./dofiles/e_chains.dofile

set_system_mode analysis
insert_test_logic -max_length 250 -clock merge \
    -edge merge -new_scan_po
report_test_logic
report_scan_chains
write_design -output generated/cpu_top_scan.v -replace
write_atpg_setup generated/cpu_scan -replace
```

Figure 6-4. Example cpu_top_gate_tessent_tap_main.pdl File

```
## Called from the Test_Setup procedure in Figure 6-7.
iProcsForModule cpu_top_gate_tessent_tap_main
iProc tap_main_setup { } {
//SelectJtagOutput is 0 so the value from the core is captured into
//the output boundary scan cells.
    iWrite select_jtag_output 0b0
//SelectJtagInput is 1 so the value from the boundary scan register
//or cell is captured into the first level of functional flops.
    iWrite select_jtag_input 0b1
iApply
}
```

Figure 6-5. Example cpu_top_gate_tessent_tdr_logic_enable.pdl File

```
## Called from the Test_Setup procedure in Figure 6-7.
iProcsForModule cpu_top_gate_tessent_tdr_logic_enable
iProc logic_enable { } {
    iWrite tdr[2:0] 0b100
iApply
}
```

Figure 6-6. Example e_chains.dofile

```
add_scan_groups group1 existing_chains.testproc
add_scan_chains -internal chain0 group1 \
    cpu_top_gate_tessent_bscan_logical_group_DEF_inst/\
    clk1_p_logic_scanin \
    cpu_top_gate_tessent_bscan_logical_group_DEF_inst/\
    CELL249_BSCAN_SO
add_scan_chains -internal chain1 group1 \
    cpu_top_gate_tessent_bscan_logical_group_DEF_inst/\
    expdout_p_6_logic_scanin \
    cpu_top_gate_tessent_bscan_logical_group_DEF_inst/\
    CELL0_BSCAN_SO
```

Figure 6-7. Example existing_chains.testproc File

```
set time scale 1ns;
alias int_clocks = clk1_p, clk2_p, clk3_p, clk4_p;
timeplate global =
  force_pi 0;
  measure_po 15;
  pulse tck_p 25 50; // tck_p
  pulse clk1_p 25 50; //
  pulse clk2_p 25 50; //
  pulse clk3_p 25 50; //
  pulse clk4_p 25 50; //
  period 100;
end; // timeplate global


procedure Test_Setup = // Describes the setup phase to be applied
                       //once at the beginning of test
timeplate global;
iCall cpu_top_gate_tessent_tap_main_inst.tap_main_setup ;
iCall cpu_top_gate_tessent_tdr_logic_enable_inst.logic_enable ;
end; // procedure Test_Setup }}}


procedure Shift = // Describes one clock cycle of shift.
  timeplate global;
  cycle =
    force scan_en1      1;
    force trst_p       1; // trst_p
    pulse tck_p;          // tck_p
    force tms_p        0; // tms_p
    force_sci;
    measure_sco;
  end;
end; // procedure Shift


procedure Load_Unload =//Proceedure to load/unload the scan chains.
timeplate global;
   cycle =
      force trst_p        1; // trst_p
      force tck_p         0;  // tck_p
      force tdi_p         0;  // tdi_p
      force int_clocks    0;
      force tms_p         0;  // tms_p
end;
  apply Shift 250; //Auto adjusted by FastScan to match the longest
                   //scan chain
end; // procedure Load_Unload
```

EDT (Type 2 LPCT) IP Creation

The third design flow step for implementing TAP control of TestKompress is to create the Embedded Deterministic Testing (EDT) compression logic IP.

Prerequisites

- Complete TAP and Boundary Scan Insertion steps.
- Complete Scan Chain Insertion and Stitching steps.

Procedure

1. Follow the steps outlined in the example script provided below as a guide to complete this stage of the design flow. The example shows the settings needed for LPCT Type 2 IP creation.

   Caution

   You must specify the test_logic_reset signal from the Tessent TAP as active low because it is driven as active low. Failure to do this results in simulation failures and a condition that is difficult to debug.

2. Synthesize the EDT IP logic RTL that was generated into core netlist Verilog gates using the Synopsys Design Complier script generated during the IP creation. For more information, see "Creation of the EDT Logic" in the *Tessent TestKompress User's Manual*.

Examples

```
##Using a Type 2 LPCT Controller
set_lpct_controller On -TAp_controller_interface On \
    -Generate_scan_enable On
set_lpct_controller -shift_control clock

##LPCT Pin connections from LPCT controller
set_lpct_pins output_scan_en scan_en1
set_lpct_pins TEST_Clock_connection \
    clk1_p_mux/A1 clk2_p_mux/A1 clk3_p_mux/A1 clk4_p_mux/A1 \
    clockBscan_mux/A1 updateBscan_mux/A1

##Use the TAP's TDI and TDO as the external channel to drive EDT
set_edt_options -location internal -channels 1
set_edt_pins input_channel 1 tdi_p tdi_i/C
set_edt_pins output_channel 1 tdo_p tdo_i/I

##LPCT Pin connections to LPCT controller pins
set_lpct_pins reset - cpu_top_gate_tessent_tap_main_inst/\
   test_logic_reset -active low
set_lpct_pins capture_dr - cpu_top_gate_tessent_tap_main_inst/\
   capture_dr_en
set_lpct_pins shift_dr - cpu_top_gate_tessent_tap_main_inst/\
   shift_dr_en
set_lpct_pins update_dr - cpu_top_gate_tessent_tap_main_inst/\
   update_dr_en
set_lpct_pins test_mode - cpu_top_gate_tessent_tdr_logic_enable_inst/\
   ltest_enable
set_lpct_pins tms tms_p tms_i/C
set_lpct_pins clock tck_p tck_i/C
```

Related Topics

"Reduced Pin Count Requirements" [Tessent TestKompress User's Manual]

Creation of the EDT Logic [Tessent TestKompress User's Manual]

Pattern Generation and Simulation

The final step for implementing TAP control of TestKompress is generating the compressed test patterns and verifying the integrity of the inserted test circuitry and scan chains through simulation.

Guidelines and tips for successful implementation of this step for implementing this architecture are presented. For additional information on generating and verifying EDT test patterns, refer to "Generating and Verifying Test Patterns" in the *Tessent TestKompress User's Manual*.

For the architecture implementation described in this chapter, make sure that during the EDT (Type 2 LPCT) IP Creation phase, the set_lpct_pins TEST_clock_connection property is used as described in the example for that section. If there are no On-Chip-Clock (OCC) generators in the design, then insert TCK for all the functional clocks. The TCK clock needs to be controlled from the output of the LPCT controller.

Note

The following needs to be understood if you plan on inserting OCC in your design.

The OCC generator is inserted only for functional clocks. The TCK clock has an added clock gating function and does not receive an OCC generator.

There are two modes in which patterns can be generated during pattern generation - a SLOW and FAST capture mode. During SLOW capture mode, the slow speed clock is used for both shift and capture during "stuck-at" testing. In this mode (SelectJtagOutput = 0 and SelectJtagInput =1), the logic states into and out of the core are being sourced and captured by the boundary scan registers. However, during the FAST capture mode (SelectJtagOutput = 1 and SelectJtagInput = 1), the logic state from the core side is not captured in the output boundary scan cells. If not set up correctly, what ATPG predicts happens does not match simulation results, creating errors that are very difficult to diagnose.

In some designs, it may be that the "at-speed" coverage is very low due to the fact that any interaction between the boundary scan cells and the core is not covered. If the boundary scan cells can be synthesized to the same frequency as the fast capture clock, then during transition testing the interaction can be tested "at-speed". During boundary scan test the clock used is the low-speed TCK, so this is not typically wanted.

Appendix A
Tessent Core Description

This section describes the configuration data syntax used to describe the following macro module types: core and boundary scan segments.

This appendix uses the following syntax conventions when documenting wrappers and properties used in the library descriptions.

Table A-1. Conventions for Command Line Syntax

Convention	Example	Usage
UPPercase	-STatic	Required argument letters are in uppercase; in most cases, you may omit lowercase letters when entering literal arguments, and you need not enter in uppercase. Arguments are normally case insensitive.
Boldface	set_fault_mode **Uncollapsed** | **Collapsed**	A boldface font indicates a required argument.
[ ]	exit [-force]	Square brackets enclose optional arguments. Do not enter the brackets.
Italic	dofile ***filename***	An italic font indicates a user-supplied argument.
{ }	add_fault_sites {**-ALl** | **-UNDEFINED_Cells** } [-VERBose]	Braces enclose arguments to show grouping. Do not enter the braces.
|	add_fault_sites {**-ALl** | **-UNDEFINED_Cells** } [-VERBose]	The vertical bar indicates an either/or choice between items. Do not include the bar in the command.
Underline	set_au_analysis **ON** | **OFf**	An underlined item indicates either the default argument or the default value of an argument.
…	add_clocks ***off_state*** ***primary_input_pin***… [-Internal]	An ellipsis follows an argument that may appear more than once. Do not include the ellipsis when entering commands.

Table A-2. Syntax Conventions for Configuration Files

Convention	Example	Usage
Italic	***scan_in :*** *port_pin_name*;	An italic font indicates a user-supplied value.

Table A-2. Syntax Conventions for Configuration Files (cont.)

Convention	Example	Usage
Underline	wgl_type : generic | lsi;	An underlined item indicates the default value.
|	logic_level : both | high | low;	The vertical bar separates a list of values that you must choose from. Do not include the bar in the configuration file.
...	port_naming : *port_naming*, ...;	Ellipses indicate a repeatable value. The comment "// repeatable" also indicates a repeatable value.
//	// default: ijtag_so	The double slash indicates the text immediately following is a comment and tells the tool to ignore the text.

Core

In Tessent Shell, descriptions of core elements, such as the memory library, the boundary scan information, or the fuse box interface, are presented to the tool in form of TCD files (Tessent Core Description files). After loading, they are hierarchically organized under the “Core” root entry. This is unique for a given module name.

Usage

```
Core(module_name) {
  Memory {
  }
  BoundaryScan {
  }
  FuseBoxInterface  {
  }
}
```

Description

The Core wrapper collects all TCD data read into the tool. Such descriptions are automatically read in during module matching. See the set_design_sources -format tcd_memory command description for information about where they are looked for. See the read_core_descriptions command description to learn how to read them in explicitly.

You can also report on the loaded TCD information. You do this using the report_config_data command. An example is "report_config_data Core(ModuleName)/Memory -partition tcd". This reports the contents of the Memory entries under Core. To see the supported syntax, use the report_config_syntax command, for example "report_config_syntax Core/Memory".

Arguments

- *module_name*

  The name of the module, equivalent of the current design module name. You don't need to specify this when loading a memory TCD file. The tool with auto-generate and auto-configure the Core-level wrapper for you.

Related Topics

set_design_sources [Tessent Shell Reference Manual]

read_core_descriptions [Tessent Shell Reference Manual]

report_config_data [Tessent Shell Reference Manual]

report_config_syntax [Tessent Shell Reference Manual]

set_module_matching_options [Tessent Shell Reference Manual]

BoundaryScan

Specifies the embedded boundary scan chain already implemented into the design module_name.

Usage

```
Core(module_name) {
  BoundaryScan {
    logic_test_segments_present : on | off ;
    Interface {
    }
    CustomBsdlCellInfo {
      cell_type_id : cell_info_description; // repeatable
    }
    ExternalPort(port_name) {
    }
    Cell(id) {
    }
    NonScannableInstances {
      instance_name ; // repeatable
    }
    Segment(segment_name) {
      first_cell_id : cell_id;
      last_cell_id  : cell_id;
    }
    SegmentSelection(selection_name) {
      SegmentNames {
        segment_name; //repeatable
      }
      EnableSignals {
        pin_port_or_net_name : value; // repeatable
      }
    }
    TapController {
      icl_instance         : instance_name ;
      host_scan_interface  : host_scan_interface ;
    }
    DftSignalMapping {
      DesignObject(design_object) {
        dft_signal : dft_signal;
        instance   : block_instance;
      }
    }
  }
}
```

Description

This wrapper describes the pad and boundary scan cell contents of a module (*module_name*) already instantiated in the design. Such descriptions are automatically read in during module matching.

- See the "set_design_sources -format tcd_bscan" command description for information about where the tool looks for them.

- See the read_core_descriptions command description to learn how to read them in explicitly.
- See the set_module_matching_options command description for information about the name-matching process.

Note

The legacy LogicVision *.lvbscan* format is supported natively and is automatically translated into this format when read.

To see the content of a read-in Core(ModuleName)/BoundaryScan, use the "report_config_data Core(ModuleName)/BoundaryScan -partition tcd" command.

To see the supported syntax, use the "report_config_data [get_config_value Core/BoundaryScan -partition meta:tcd -object]" command.

Arguments

- logic_test_segments_present : on | off ;

  An optional property that specifies whether the module includes embedded BoundaryScan hardware with logic test support so the tool can handle it properly at higher levels. If set to on, the tool bypasses the module instance when creating logic scan segments at the higher level. This ensures that the tool excludes embedded BoundaryScan blocks with logic test support from higher-level scan chain hookup.

  Restriction

  Do not add instances with embedded BoundaryScan logic test support to the Core/BoundaryScan/NonScannableInstances wrapper. Instead, add an empty wrapper to specify that no instances are marked as is_non_scannable.

Interface

This section describes the common ports of the boundary scan hardware of the module module_name.

Usage

```
Core(module_name) {
  BoundaryScan {
    Interface {
      select                    : port_name ;
      reset                     : port_name ;
      force_disable             : port_name ;
      select_jtag_input         : port_name ;
      select_jtag_output        : port_name ;
      select_jtag_enable        : port_name ;
      capture_shift_clock       : port_name ;
      capture_shift_clock_inv   : port_name ;
      bscan_clock               : port_name ;
      update_clock              : port_name ;

      capture_en                : port_name ;
      shift_en                  : port_name ;
      update_en                 : port_name ;

      scan_in                   : port_name ;
      scan_out                  : port_name ;
      scan_out_launch_edge      : negedge | posedge ;

      ac_init_clock0            : port_name ;
      ac_init_clock1            : port_name ;
      ac_signal                 : port_name ;
      ac_mode_enable            : port_name ;
    }
  }
}
```

Description

This section describes the common ports of the boundary scan hardware of the module module_name. Theses ports usually connect to a higher level boundary scan interface block or to a TAP.

Arguments

- select : *port_name*

  This signal is used to enable the boundary-scan register logic in the module. If this signal is active the capture, shift and update enable signals effect the boundary scan register. The signal is furthermore used to gate the clocks bscan_clock, capture_shift_clock and capture_shift_clock_inv. The signal is optional and it assumed to be active high.

- reset : *port_name*

  This signal is used to reset the boundary scan register logic in the module. The signal is active low.

- force_disable : *port_name*

  This disables all pad cell drivers. The external ports show a Z signal. This is limited to those pad cells that can be disabled. The signal overrides the settings from the enable boundary scan cells. The signal is active high. The signal is needed to support the HIGHZ instruction.

- select_jtag_input : *port_name*

  This is the select signal of the SJI multiplexer that switches between the pad cells from_pad port and the output of the boundary scan cell. The output of the multiplexer is connected to the core. When the signal is high it selects the boundary scan cell output. This signal is mandatory, in case the module contains input boundary scan cells.

- select_jtag_output : *port_name*

  This is the select signal of the SJO multiplexer that switches between the core signal and the output of the boundary scan cell. The output of the mux is connected to the pad cells to_pad port. When the signal is high it selects the boundary scan cell output. The signal is mandatory, in case the module contains output or bidirectional boundary scan cells.

- select_jtag_enable : *port_name*

  This is the select signal of the SJI multiplexer that switches between the functional pad enable and the output of the enable boundary scan cell. The output of the multiplexer is connected to the pad enable port. When this signal is high it selects the output of the boundary scan cell. This entry is optional. The select of the SJE multiplexer can be connected to select_jtag_output instead of this dedicated signal, when this signal is not specified.

- capture_shift_clock: *port_name*

  This is a gated version of the test clock. The clock should be active during shift and capture cycles. The inactive state of the clock is the low state. You need to specify either bscan_clock or capture_shift_clock or capture_shift_clock_inv.

- capture_shift_clock_inv : *port_name*

  Inverted version of capture_shift_clock. This is the legacy clock timing that is used in ETAssemble.You need to specify either bscan_clock or capture_shift_clock or capture_shift_clock_inv.

- bscan_clock : *port_name*

  Clock that needs to be continuously running during the boundary scan test. This is usually the test clock or a gated version of the test clock that is disabled outside of the boundary scan test. You need to specify either bscan_clock or capture_shift_clock or capture_shift_clock_inv.

- update_clock : *port_name*

  A gated clock that pulses the update register in the boundary scan cell. The disabled stated of the clock is the low state. You need to specify this clock when using the capture_shift_clock or the capture_shift_clock_inv clock. You need to specify either this entry or update_en.

- capture_en : *port_name*

  When this signal is high, the boundary scan cells capture the value from either pad, the core, or the update register dependent on the type of the boundary scan cell. You need to specify this signal when you use bscan_clock. The signal is active high.

- shift_en : *port_name*

  This is the shift enable of the boundary scan register. It is the select of the scan multiplexer inside the boundary scan cells, and a high value here connects the boundary scan cell to a scan chain. This entry is mandatory.

- update_en : *port_name*

  When this signal is high the update elements in the boundary scan cells take over the value of the boundary scan register. This entry is needed when you use bscan_clock. You need to specify either this entry or update_clock.

- scan_in : *port_name*

  This is the scan input of the boundary scan segment inside the module. It is used to shift in data from the TAP or a boundary scan interface block or from other parts of the boundary scan register.

- scan_out : *port_name*

  This is the scan output of the boundary scan segment inside the module. It is used to shift out data to the TAP or a boundary scan interface block or to other parts of the boundary scan register.

- scan_out_launch_edge : negedge | posedge

  This entry is only valid if scan_out is defined. It specifies the edge where the data on the scan output is valid.

- ac_init_clock0 : *port_name*

  You can use this clock to trigger the initialization of the test receiver in AC input and AC bi-directional pad cells. The tool initializes the test receiver's hysteretic memory element on the falling edge of the clock or by a logic 0 on this port.

- ac_init_clock1 : *port_name*

  You can use this clock to trigger the initialization of the test receiver in AC input and AC bidirectional pad cells. The tool initializes the test receiver's hysteretic memory element on the rising edge of the clock or by a logic 1 on this port.

- ac_signal : *port_name*

  This signal gives the pulse(s) for the extest pulse or train according the the IEEE 1149.6 standard. This signal is mandatory when the module contains AC output or inout cells.

- ac_mode_enable: *port_name*

  This enables the AC functionality in the described module. This signal enables the input AC functionality and the output functionality for those ports that don't have an AC select cell.

CustomBsdlCellInfo

This wrapper defines custom boundary scan cell types and describes their capture behavior.

Usage

```
Core(module_name) {
  BoundaryScan {
    CustomBsdlCellInfo {
      cell_type_id : cell_info_description ; // repeatable
    }
  }
}
```

Description

The CustomBsdlCellInfo wrapper is used to define custom boundary scan cell types and describe their capture behavior. The CustomBsdlCellInfo descriptions are saved to the TS_BSCAN_CELLS BSDL package file along side the BSDL file in the Tessent Shell Data Base (TSDB) and be used when generating IO test patterns.

Standard 1149.1 built-in cell types, such as those prefixed with BC_ or AC_, as well as custom cell types, are specified in the BoundaryScan/Cell wrapper to describe boundary scan cells in the design. The custom boundary scan cell types specified in the BoundaryScan/Cell wrapper must also be described in the CustomBsdlCellInfo wrapper.

Arguments

- *cell_type_id* : *cell_info_description* ;

  A repeatable label string and complex string pair that defines and describes the custom cell type.

  The cell_type_id string value is user-defined, and is specified in the BoundaryScan/Cell/bsdl_cell_type property to identify the custom cell type for a boundary scan cell.

  The cell_info_description syntax must exactly match the CELL_INFO syntax as defined in the IEEE 1149.1 standard, “Annex B.10 User-supplied BSDL packages” section. Note that the cell_info_description must be enclosed in quotes because the description contains spaces.

Examples

The following example shows how to define, and use, a custom boundary scan cell type that is a variation of the standard BC_7 cell type.

The standard BC_7 cell type is described by the following CELL_INFO in the IEEE 1149.1 standard:

```
constant BC_7 : CELL_INFO :=
 ((BIDIR_IN, EXTEST, PI), (BIDIR_OUT, EXTEST, PO),
  (BIDIR_IN, SAMPLE, PI), (BIDIR_OUT, SAMPLE, PI),
  (BIDIR_IN, INTEST, UPD), (BIDIR_OUT, INTEST, PI));
```

We want to create a custom cell type that differs in the highlighted INTEST behavior. For the standard cell, this portion can be read as “for this cell used as a bidirectional cell acting as an input (BIDIR_IN) while INTEST is in effect, the capture flip-flop loads the value of the Update flip-flop (or latch) data (UPD) during Capture-DR controller state”.

The custom cell to be defined has the following behavior for the highlighted section:

```
(BIDIR_IN, INTEST, X)
```

This indicates an unknown value is loaded rather than the UPD value, as is done in the standard cell type.

The example below shows the definition and use of the described custom cell type:

```
Core(core) {
  BoundaryScan {
    CustomBsdlCellInfo {
      my_BC_7 : "((BIDIR_IN, EXTEST, PI), (BIDIR_OUT, EXTEST, PO),
                  (BIDIR_IN, SAMPLE, PI), (BIDIR_OUT, SAMPLE, PI),
                  (BIDIR_IN, INTEST, X),  (BIDIR_OUT, INTEST, PI))";
    }
    Cell(my_bidir){
      function : bidir ;
      bsdl_cell_type : my_BC_7 ;
      ...
    }
  ...
  }
}
```

ExternalPort

This section describes the properties of a port that is to be connected directly to a top level port and the connected pad cell.

Usage

```
Core(module_name) {
  BoundaryScan(module_name) {
    ExternalPort(port_name) {

      differential_inverse_of     : port_name;
      differential_type           : voltage | current;

      buffer_type                 : three_state | two_state |
                                    low_only | high_only;
      highz_during_force_disable  : on | off | auto;
      pull_resistor               : high | low | none;

      control_cell                : cell_id;

      ac_hp_time                  : time;
      ac_lp_time                  : time;
      ac_hp_location              : on_chip | off_chip;

      auxiliary_output            : port_name;
      auxiliary_output_enable     : port_name;
      auxiliary_input             : port_name;
      auxiliary_input_enable      : port_name;

      from_core_pin               : pin_name;
      from_core_port              : port_name;
      to_core_pin                 : pin_name;
      to_core_port                : port_name;
    }
  }
}
```

Description

This section describes the properties of a port port_name on the tcd_bscan module that is to be directly connected to a top level port. The pad cell is inside the tcd_bscan segment and this wrapper describes the properties of the pad cell.

Arguments

- differential_inverse_of : *port_name*

  This entry is used for the associated port of a differential pair. The port_name refers to the ExternalPort(port_name) wrapper of the representative port. This entry is exclusive with all other entries. The settings for the differential pair need to be done in the wrapper of the representative port.

- differential_type : voltage | current

  Specifies the nature of the differential pair. If this is set to voltage it means that the signals are similar to the other logic signals. In case of current the differential pair is directly accessible from the tester and the tester needs to determine how to deal with this differential port. If this entry is specified there needs to be a ExternalPort() wrapper referring to this wrapper with the differential_inverse_of property to this port.

- buffer_type : three_state | two_state | low_only | high_only

  This specifies the values the pad driver can drive and if the driver can be disabled. See the table for an overview of the combinations.

Table A-3. Buffer Types and Their Available State

buffer_type	can drive a 0	can drive a 1	can be disabled to Z
three_state	yes	yes	yes
two_state	yes	yes	no
low_only	yes	no	yes
high_only	no	yes	yes

  A three_state port can drive 0 and 1 values and is enables and disabled by an extra enable cell. A two_state pot can drive a 0 and a 1, but cannot be disabled. Note that you cannot implement a fully IEEE 1149.1 HIGHZ instruction if you use such a driver in your design, because this buffer type lacks the needed ability to disable the pad driver. The types low_only and high_only are asymmetric drivers like open emitter and open collector driver. The enable signal of a driver can be overridden by the force disable signal that switches off all driver except the two_state ones. This entry is only valid for output or inout ports.

- highz_during_force_disable : on | off | auto

  Specifies if the pad driver can be disabled with the force disable signal. This entry is only valid for output or inout ports. In case of auto it is assumed that driver of the buffer_type three_state, low_only and high_only can be disabled.

- pull_resistor : high | low | none

  Specifies if a pull resistor is present in the pad.

Table A-4. Valid Combinations of the pull_resistor and buffer_type

pull_resistor	buffer_type	bsdl disable result
none	three_state	Z
none	two_state	-
none	low_only	weak1
none	high_only	weak0
high	low_only	pull1
low	high_only	pull0

The preceding table shows the valid combinations. This entry is only valid for output and inout ports.

- control_cell : *cell_id*

  This refers to a Cell(*cell_id*) wrapper that contains the description of the control cell that enables and disables the pad. This entry is only valid for output and inout ports.

- ac_hp_time : *time*

  This describes the AC high pass behavior of the pad according to the IEEE 1149.6 standard.

- ac_lp_time : *time*

  This describes the AC low pass behavior of the pad according to the IEEE 1149.6 standard.

- ac_hp_location : on_chip | off_chip

  This described if the AC high pass is on the chip or needs to be added on the outside.

- auxiliary_output : *port_name*

  This specifies a port that can be multiplexed with the data from the core or the data bscan cell. The output of the multiplexer is connected to the pad of the external port. The external port needs to be an output or inout port. You need to specify also the auxiliary_output_enable entry.

- auxiliary_output_enable : *port_name*

  This is the multiplexer enable for the auxiliary_output above. You need to specify both entries together.

- auxiliary_input : *port_name*

  This specifies a port that gets the value from the pad of the external port, if enabled with the auxiliary_input_enable. The external port needs to be an input or inout port. You need to specify also the auxiliary_input_enable entry.

- auxiliary_input_enable : *port_name*

  This is the enable for the auxiliary_input above. You need to specify both entries together.

- from_core_pin : *pin_name*

  This property specifies the pin within the module that corresponds to the from_core function of the associated output or inout port.

- from_core_port :*port_name*

  This property specifies a port that is in the controlling fan-in of the from_core_pin when the embedded boundary scan chain and the pad buffers are the only logic in a sub_block wrapper module. To connect to the core side of a port equipped with a pad buffer wrapped in a sub_block module, use this property to identify the input port on the module associated with an output or inout port.

- to_core_pin : *pin_name*

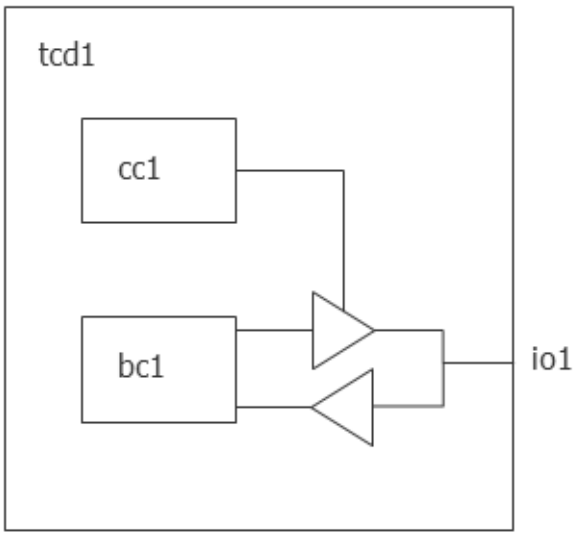
  This property specifies the pin within the module that corresponds to the to_core function of the associated input or inout port.

- to_core_port : *port_name*

  This property specifies a port that is in the controlling fanout of the to_core_pin when the embedded boundary scan chain and the pad buffers are the only logic in a sub_block wrapper module. To connect to the core side of a port equipped with a pad buffer wrapped in a sub_block module, use this property to identify the output port on the module associated with an input or inout port.

Examples

Figure A-1. Bidirectional Pad With Data and Enable Cell



The core description of the bidirectional pad with boundary scan and enable cell :

```
Core(tcd1) {
   BoundaryScan {
      Interface { // not shown in the figure
         force_disable        : force_disable1;
         select_jtag_input    : select_jtag_input1;
         select_jtag_output   : select_jtag_output1;
         bscan_clock          : bscan_clock1;
         capture_en           : capture_enable1;
         shift_en             : shift_enable1;
         update_en            : update_enable1;
         scan_in              : scan_input1;
         scan_out             : scan_output1;
         scan_out_launch_edge : negedge;
      }
      ExternalPort(io1) {
         buffer_type          : three_state;
         control_cell         : cc1;
      }
      Cell(cc1) {
         function             : control;
         bsdl_cell_type       : BC_2;
         control_enable_value : 1;
         safe_value           : 0;
      }
      Cell(bc1) {
         function             : bidir;
         bsdl_cell_type       : BC_7;
         external_port        : io1;
         safe_value           : 0;
      }
   } }
```

Cell

This section describes a boundary scan cell.

Usage

```
Core(module_name) {
  BoundaryScan {
    Cell(id) {
      function             : output | input | bidir |
                             control | control_reset | internal |
                             observe_only | clock | ac_select;
      bsdl_cell_type       : cell_type_id;
      external_port        : port_name;
      control_enable_value : 0 | 1;
      ac_select_cell       : cell_id;
      safe_value           : 0 | 1 | x;
      ac_type              : on | off;
    }
  }
}
```

Description

This section describes a boundary scan cell.

Arguments

- function : output | input | bidir | control | control_reset | internal | observe_only | clock | ac_select

  The following table shows the function of each boundary scan cell.

Table A-5. Cell functions

function	description	BSDL function
output	A data cell that drives an output external port. The cell may also be able to observe the value from the core.	OUTPUT2 or OUTPUT3
input	A data cell that observes an input external port. The cell may also be able to drive the signal to the core.	INPUT
bidir	A data cell for an inout external port.	BIDIR
control	A control cell that enables and disables a group of output or inout external ports.	CONTROL
control_reset	Same as control, but forced to the disabled state during the Test-Logic_reset TAP state.	CONTROLR

Table A-5. Cell functions (cont.)

function	description	BSDL function
internal	Cell not associated with an external port.	INTERNAL
observe_only	A cell that captures values from input, output or inout external ports.	OBSERVE_ONLY
clock	A cell that observes the state of a clock external port.	CLOCK
ac_select	A cell that enables the AC functionality on a group of output or inout external ports.	INTERNAL

- bsdl_cell_type : *cell_type_id*

  This is the entry that is used to describe the cell in the BSDL file. This can either be a build in type like the BC_# or AC_# or a custom cell type described in a package file.

- external_port : *port_name*

  This refers to an ExternalPort(port_name) wrapper. This cell is a databoundary scan cell of the external port.

- control_enable_value : 0 | 1

  This specifies the enable value for cells with the function control, control_reset and ac_select. For those cells the entry is mandatory.

- ac_select_cell : *cell_id*

  This refers to another Cell(cell_id) wrapper. The other cell is an AC select cell and enables and disables the AC functionality of this cell.

- safe_value : 0 | 1 | x

  This specifies the safe value of the cell. The safe value is used in the BSDL file. The default is x.

- ac_type : on | off

  Specifying this to on implies that this is an AC data boundary scan cell. It needs to have the function input, output, bidir, or observe_only. With the function property set to input or observe_only, the cell observes the IEEE 1169.9 test receiver on the port described by the external_port property. With the function property set to output or bidir, the cell determines the polarity of the IEEE 1149.6 pulse or train on the port described by the external_port property. The default for ac_type is off.

NonScannableInstances

This wrapper defines BoundaryScan instances that must have the is_non_scannable attribute set to true.

Usage

```
Core(module_name) {
  BoundaryScan {
    NonScannableInstances {
      instance_name ; // repeatable
    }
  }
}
```

Description

The NonScannableInstances wrapper specifies the BoundaryScan instances that must have the is_non_scannable attribute set to true. If the NonScannableInstances wrapper is not present, all instances in *module_name* are set as non-scannable. If the NonScannableInstances wrapper is present, but empty, all instances in *module_name* are scannable. If the NonScannableInstances wrapper is present and contains instances, those listed instances that are within the module *module_name* are set as non-scannable.

Arguments

- *instance_name* ;

  A repeatable string that specifies a BoundaryScan instance within the design *module_name* that must have the is_non_scannable attribute set to true.

Examples

The non-scannable BoundaryScan instances can be queried by using two attributes:

- is_non_scannable = true
- is_non_scannable_reason = “imported_from_bscan”

Using these attributes, the following command shows the BoundaryScan instances that are non-scannable:

```
get_instances -filter
{is_non_scannable&&is_non_scannable_reason=="imported_from_bscan"}
```

Note that the attributes apply to all child instances.

Segment

Specifies boundary scan cells that define a segment of a scan chain.

Usage

```
Core(core_name) {
  BoundaryScan {
    Segment(segment_name) {
      first_cell_id : cell_id;
      last_cell_id : cell_id;
    }
    SegmentSelection(selection_name) {
      SegmentNames {
        segment_name; // repeatable
      }
      EnableSignals {
        pin_port_or_net_name : value; // repeatable
      }
    }
  }
}
```

Description

The Segment wrapper specifies two boundary scan cells that have a fixed sequence. These two cells define, as the start and end points, a segment in the boundary scan chain. This segment is either part of the scan path or is bypassed as a whole.

The corresponding SegmentSelection wrapper specifies the segments that are part of the boundary scan chain and the enable signals that control them.

Arguments

- ***segment_name***

  A string that uniquely identifies the segment.

- first_cell_id : ***cell_id***

  A property that specifies the cell_id of the first boundary scan cell in the segment.

- last_cell_id : ***cell_id***

  A property that specifies the cell_id of the last boundary scan cell in the segment.

- ***selection_name***

  A string that uniquely identifies a segment selection wrapper.

- ***pin_port_or_net_name*** : ***value***

  A name and value pair that specifies an enable signal and the logic value that selects the segment.

Examples

Embedded BoundaryScan Segment With Four Boundary Scan Cells

You can bypass the cell3, cell2, and cell0 cells with the bonding_enable1 port. The boundary scan cell, cell1, is part of the boundary scan register.

```
Core(mycore) {
  BoundaryScan {
    Interface {
      ...
    }
    Cell(cell3) {
      ...
    }
    Cell(cell2) {
      ...
    }
    Cell(cell1) {
      ...
    }
    Cell(cell0) {
      ...
    }
    Segment(segment1) {
      first_cell_id : cell3;
      last_cell_id : cell2;
    }
    Segment(segment2) {
      first_cell_id : cell1;
      last_cell_id : cell1;
    }
    Segment(segment3) {
      first_cell_id : cell0;
      last_cell_id : cell0;
    }
    SegmentSelection(complete) {
      SegmentNames {
        segment1;
        segment2;
        segment3;
      }
      EnableSignals {
        bonding_enable1 : 0;
      }
    }
    SegmentSelection(bypass_cells) {
      SegmentNames {
        segment2;
      }
      EnableSignals {
        bonding_enable1 : 1;
      }
    }
  }
}
```

TapController

Specifies the TAP controller and the ICL host scan interface that control the BoundaryScan hardware.

Usage

```
Core (core_name) {
  BoundaryScan {
    TapController {
      icl_instance : instance_name ;
      host_scan_interface : host_scan_interface ;
    }
  }
}
```

Description

Use the TapController wrapper when an embedded boundary scan block contains the TAP controller that controls the BoundaryScan hardware.

Arguments

- ***icl_instance***

  A string that identifies the instance name of the TAP controller.

- ***host_scan_interface***

  A string that identifies the ICL host scan interface name for BoundaryScan. The host scan interface must either be named “host_bscan,” or its “tessent_is_bscan_host” attribute must be set to “true.”

Examples

This example defines a TAP controller with instance name “tap_wrapper_tap_instance” and host scan interface “host_bscan.”

```
Core(corea) {
  BoundaryScan {
    Interface {
      scan_in : bscan_scan_in;
      scan_out : bscan_scan_out;
      scan_out_launch_edge : negedge;
    }
    TapController {
      icl_instance : tap_wrapper_tap_instance;
      host_scan_interface : host_bscan;
    }
    Segment(before_tap) {
      first_cell_id : cell11;
      last_cell_id : cell8;
    }
    ...
  }
}
```

DftSignalMapping

Specifies the DftSignals used to control bonding configuration bypasses.

Usage

```
Core (core_name) {
  BoundaryScan {
    DftSignalMapping {
      DesignObject(design_object) {
        dft_signal : dft_signal;
        instance   : block_instance;
      }
    }
  }
}
```

Description

Use the DftSignalMapping wrapper to map which DftSignals control the bonding configuration bypass muxes.

Arguments

- **DesignObject** (***design_object***)

  A required property that identifies the physical location of the DftSignal used in the SegmentSelection/EnableSignals.

- **dft_signal** : ***dft_signal***;

  A required property and string pair that identifies the name of the DftSignal.

- instance : *block_instance*;

  An optional property and string pair to specify if the DftSignal is not in the current design. For example, in a sub-block.

Examples

The following example shows a BoundaryScan wrapper with a DftSignalMapping wrapper for two DFT signals used as bonding enables. Two DesignObject wrappers are identified with SegmentSelection/EnableSignals.

```
Core(mycore) {
  BoundaryScan {
    Interface {
      ...
    }
    Segment(segment3) {
      ...
    }
    Segment(segment2) {
      ...
    }
    Segment(segment1) {
      ...
    }
    Segment(segment0) {
      ...
    }
    SegmentSelection(bypass2) {
      SegmentNames {
        segment3;
        segment1;
        segment0;
      }
      EnableSignals {
        mycore_rtl_tessent_tdr_sri_ctrl_inst/mybypass2 : 1;
        mycore_rtl_tessent_tdr_sri_ctrl_inst/mybypass1 : 0;
      }
    }
    SegmentSelection(bypass1) {
      SegmentNames {
        segment3;
        segment2;
        segment0;
      }
      EnableSignals {
        mycore_rtl_tessent_tdr_sri_ctrl_inst/mybypass2 : 0;
        mycore_rtl_tessent_tdr_sri_ctrl_inst/mybypass1 : 1;
      }
    }
    DftSignalMapping {
      DesignObject(mycore_rtl_tessent_tdr_sri_ctrl_inst/mybypass2) {
        dft_signal : mybypass2;
      }
      DesignObject(mycore_rtl_tessent_tdr_sri_ctrl_inst/mybypass1) {
        dft_signal : mybypass1;
      }
    }
  }
}
```

Appendix B
Support For AC Pins (IEEE 1149.6)

This section describes the commands and templates Tessent uses to create device circuitry compliant with the IEEE 1149.6 standard for differential or capacitively coupled pins.

Specifying AC Pins in Your Design

AC boundary scan is inserted in your design when AC pins are present. This can be controlled via several Tessent Shell commands.

The set_dft_specification_requirements command can be used to include or exclude AC boundary scan in your design. The optional -ac_boundary_scan switch on this command controls whether AC boundary scan is inserted. When this switch is set to “off,” AC boundary scan (IEEE 1149.6) is not implemented. If the command is not used, or if “-ac_boundary_scan auto” (the default) is set, AC boundary scan is inserted.

AC Pins in Configuration Specifications

The wrappers and properties of the Tessent tool's configuration specifications affect how AC boundary scan is implemented in your design.

The DftSpecification and PatternsSpecification wrappers can be used to control how IEEE 1149.6 AC pins and test patterns are handled in your design. This method provides a means of defining and maintaining alternate configurations from a variety of sources.

For a full description of how to configure test structures in your design using the configuration data syntax, see the "Configuration-Based Specification" chapter in the *Tessent Shell Reference Manual*.

AC Pins in the DftSpecification Wrapper

Several wrappers and properties in the DftSpecification wrapper affect the way AC pins are managed in your design.

For a full description of this wrapper, see the DftSpecification section of the *Tessent Shell Reference Manual*.

The Interface wrapper inside the EmbeddedBoundaryScan wrapper includes the following arguments:

- **ac_init_clock0** — Defines the name of the ac_init_clock0 port. Default: "bscan_ac_init_clk0".
- **ac_init_clock1** — Defines the name of the ac_init_clock1 port. Default: "bscan_ac_init_clk1".
- **ac_signal** — Defines the name of the ac_signal port. Default: "bscan_ac_signal".
- **ac_mode_en** — Defines the name of the ac_mode_en port. Default: "bscan_ac_mode_en".

These ports are only created when AC pads are present in the design.

The BoundaryScanCellOptions wrapper inside the BoundaryScan and EmbeddedBoundaryScan wrappers includes the following argument:

- **add_dot6_from_pad_cell** — identifies the AC input and inout ports where the from_pad data signal is to be intercepted with an additional boundary scan cell.

These ports can also be defined using the set_boundary_scan_port_options command.

The ACModeOptions wrapper inside the BoundaryScan and EmbeddedBoundaryScan wrappers can be used to specify a number of IEEE 1149.6 properties of the boundary scan chain. See "ACModeOptions" in the *Tessent Shell Reference Manual* for complete information.

AC Pins in the PatternsSpecification Wrapper

Several wrappers and properties in the PatternsSpecification wrapper affect the way AC pins are managed in your design.

For complete information on this wrapper, see "PatternsSpecification" in the *Tessent Shell Reference Manual*.

The LoadBoardInfo wrapper includes the dot6_ttest argument. This argument (referred to as TTest in the IEEE 1149.6 standard) specifies the minimum time required by the slowest coupling capacitor, among all AC Loopbacks, to fully discharge. It is used mainly by DC BoundaryScan tests such as dot6_dc_input and dot6_dc_output to ensure that DC levels are not captured by the test receiver.

The default value of dot6_ttest is three times the slowest time constant among all AC pads that have an on-chip high-pass or low-pass filter (as recommended by the IEEE 1149.6 standard). If no on-chip filter is present or if no BSDL file can be found, the value defaults to three times the TCK period.

The ACLoopbacks wrapper inside the LoadBoardInfo wrapper specifies connection loopbacks from AC source ports to AC destination ports through a coupling capacitor. This wrapper takes arguments in pairs (*destination_port* : *source_port*) with the following restrictions:

- *source_port* and *destination_port* must be output and input respectively. Inout ports are not supported.
- 1149.1 ports are not supported.
- To loop back differential pairs, you can specify either of:
  - The output positive leg and the input positive leg
  - The output negative leg and the input positive leg

  You can also specify only the positive leg's port names. In this case, the negative leg is inferred from the BSDL file or taken from the differential_inverse_of properties of the specified ports.
- The *destination_port* must be the positive leg.

  The connection between the other two legs; that is, between the positive leg of the source and the negative leg of the destination, is inferred.

Value	Description
dot6_ac_00	This test measures the AC parameters Vhyst_edge and Thyst on contacted 1149.6 input pins by applying a valid logic 0 to the input pins and expecting to capture a logic 0. This test applies AC waveforms to contacted inputs and uses the EXTEST_PULSE instruction.
dot6_ac_01	This test measures the AC parameters Vhyst_edge and Thyst on contacted 1149.6 input pins by applying an invalid logic 0 to the input pins and expecting to capture a logic 1. This test applies AC waveforms to contacted inputs and uses the EXTEST_PULSE instruction. This test is meant for manufacturing patterns and cannot pass simulation, because it assumes that the test receiver reacts correctly to an invalid 0 or 1.
dot6_ac_10	This test measures the AC parameters Vhyst_edge and Thyst on contacted 1149.6 input pins by applying an invalid logic 1 to the input pins and expecting to capture a logic 0. This test applies AC waveforms to contacted inputs and uses the EXTEST_PULSE instruction. This test is meant for manufacturing patterns and cannot pass simulation, because it assumes that the test receiver reacts correctly to an invalid 0 or 1.
dot6_ac_11	This test measures the AC parameters Vhyst_edge and Thyst on contacted 1149.6 input pins by applying a valid logic 1 to the input pins and expecting to capture a logic 1. This test applies AC waveforms to contacted inputs and uses the EXTEST_PULSE instruction.
dot6_ac_input	This test verifies the AC operation of the 1149.6 receivers. It performs the following tasks: • Applies a checkerboard pattern to contacted AC input/bidirectional pins. Disables all output/bidirectional pads. • Applies a valid AC waveform to contacted input and inout pins. • Tests the “no transition” detection capability, if present, by applying two consecutive test patterns while keeping the AC pin level constant. If the pad contains either a Low-Pass filter or an embedded High-Pass filter, the absence of transition in the second pattern makes its corresponding boundary-scan cell capture its default value. This test uses both the EXTEST_PULSE and EXTEST_TRAIN instructions. **Note:** inputs with loopbacks are not tested with this test, but rather with the dot6_ac_output test.

Value	Description
dot6_ac_output	This test verifies the AC operation of the 1149.6 transmitters and loopbacks. It performs the following tasks: • Enables all AC output pads. Ignores and turns off all non-AC pads, if possible. • Alternatively disables and enables ACSelect cells. • Runs the test on one Enable Group at a time. • Strobes AC waveforms on all AC contacted outputs. This test uses both EXTEST_PULSE and EXTEST_TRAIN instructions, and both AC and DC loopbacks (if present).
dot6_ac_select_cells	This test verifies that the netlist properly implements the AC grouping described in the BSDL. Only one ACSelect cell is enabled at a time, and its action is verified by checking that all AC output pins in its fanout properly toggle during the RunTestIdle state when the EXTEST_PULSE instruction is loaded. This test is necessary only during netlist verification in the design flow. In manufacturing, the dot6_ac_output test ensures that all AC select cells are properly operating by enabling or disabling them all at once.
dot6_dc_00	This test measures DC parameters Vthreshold and Vhyst_level on contacted 1149.6 input pins by applying a valid logic 0 to the input pins and expecting to capture a logic 0. All AC pad drivers are disabled for this test.
dot6_dc_01	This test measures DC parameters Vthreshold and Vhyst_level on contacted 1149.6 input pins by applying an invalid logic 0 to the input pins and expecting to capture a logic 1. All AC pad drivers are disabled for this test. This test is meant for manufacturing patterns and cannot pass simulation, because it assumes that the test receiver reacts correctly to an invalid 0 or 1.
dot6_dc_10	This test measures DC parameters Vthreshold and Vhyst_level on contacted 1149.6 input pins by applying an invalid logic 1 to the input pins and expecting to capture a logic 0. All AC pad drivers are disabled for this test. This test is meant for manufacturing patterns and cannot pass simulation, because it assumes that the test receiver reacts correctly to an invalid 0 or 1.
dot6_dc_11	This test measures DC parameters Vthreshold and Vhyst_level on contacted 1149.6 input pins by applying a valid logic 1 to the input pins and expecting to capture a logic 1. All AC pad drivers are disabled for this test.

Value	Description
dot6_dc_input	This 1149.6 test verifies the behavior of all 1149.6 AC input pins when operating in EXTEST mode. Just as the non-AC (in other words, DC) pins “input” test, this pattern applies zero and one logic values at the AC pins and checks that their 1149.6 test-receiver's associated bscan cells capture the same value. In addition, whenever the target tester supports it, the test also verifies that all AC pins can reliably detect invalid input voltage levels sitting in between the “zero” and “one” threshold voltages.
dot6_dc_output	This 1149.6 test is equivalent to the existing output IO test for normal DC pads. Just as in the output test, the dot6_dc_output test applies checkerboard values, activates output cells one enable group at a time, and uses load board loopbacks as well as IO internal loopbacks. In addition, this test also covers 1149.6-specific features such as coupling capacitors discharge time, ACSelect cells, and output inversion during the RunTestIdle state of the TAP.

IEEE 1149.6 Hardware

Additional hardware is needed when AC pins under the IEEE 1149.6 standard are present in your design.

TAP Controller

When AC pads are present in your design, Tessent BoundaryScan generates a modified Tessent TAP controller that provides the IEEE 1149.6 instructions and control circuit.

Two variants of the EXTEST instruction are defined in the InstructionCodes section of the DftSpecification/IjtagNetwork/Tap/HostBScan wrapper: EXTEST_PULSE and EXTEST_TRAIN. For example:

```
DftSpecification(mychip,rtl) {
  IjtagNetwork {
    HostScanInterface(tap) {
      Tap(main) {
        bypass_instruction_codes : 2'b00 ;
        HostBscan {
          InstructionCodes {
            CLAMP          : unused ;
            EXTEST         : 2'b01 ;
            EXTEST_PULSE   : 2'b10 ;
            EXTEST_TRAIN   : 2'b11 ;
            INTEST         : unused ;
            SAMPLE_PRELOAD : unused ;
            HIGHZ          : unused ;
          }
        }
      }
    }
  }
}
```

Note

EXTEST_PULSE is required. EXTEST_TRAIN is optional.

The modified TAP controller has outputs corresponding to these instructions. Names for these output ports can be defined in the Interface section of the wrapper if the default names are not wanted.

AC Control Signals

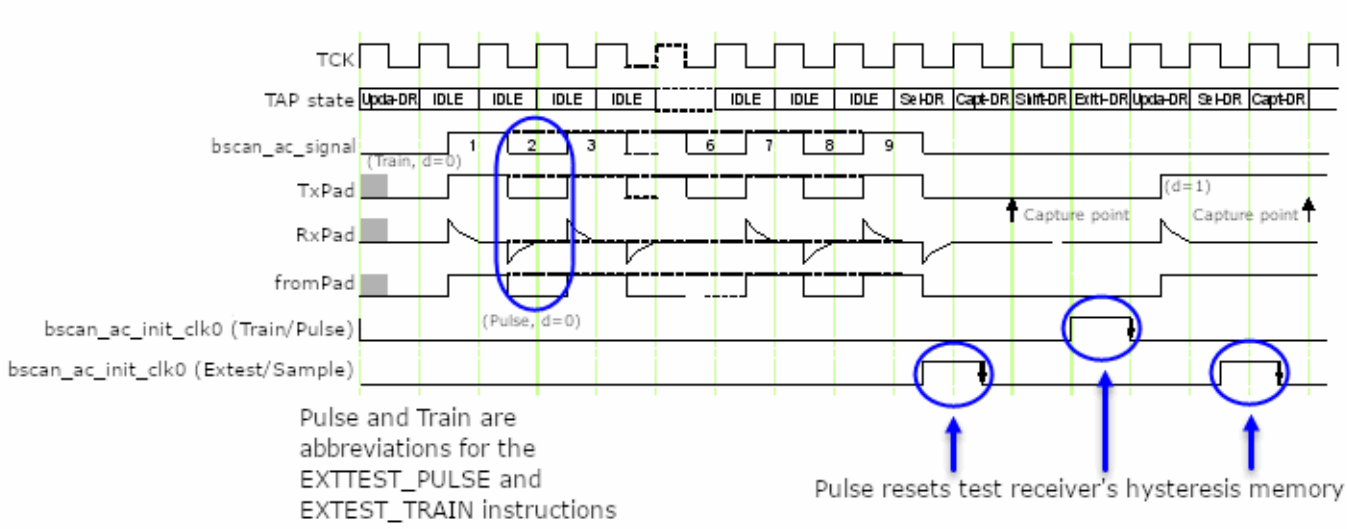
When Tessent BoundaryScan creates the modified TAP controller to support the IEEE 1149.6 standard, it creates several dedicated control signals as well.

The modified TAP controller includes an interface module with output ports for each of the three mandatory IEEE 1149.6 control signals:

- *to_bscan_ac_mode_en* — Indicates that either the EXTEST_PULSE or EXTEST_TRAIN instruction is active.
- *to_bscan_ac_signal* — Carries the output state inversion signal used by all IEEE 1149.6 output boundary-scan cells.
- *to_bscan_ac_init_clock0* and *to_bscan_ac_init_clock1* — Indicates the initialization *initClk* signal used by all AC input pads.

These signals are described in the IEEE 1149.6 standard. Their waveforms are illustrated in Figure B-1 for reference.

Figure B-1. Waveforms Illustrating IEEE 1149.6 Timing



Boundary Scan Cells

Tessent BoundaryScan generates several boundary scan AC cell types:

- Input-only, output-only, or bidirectional pad.
- Pads where the JTAG muxes might be located inside the pad, or inside the core, or inside the boundary scan cell.
- Single-ended or differential pad.

You can also specify more options on a per boundary scan cell basis:

- Input or bidirectional boundary-scan cells can either ignore, sample, or intercept the functional input signal going to the core.
- Group output AC pads so as to enable the AC test mode of only one group at a time.

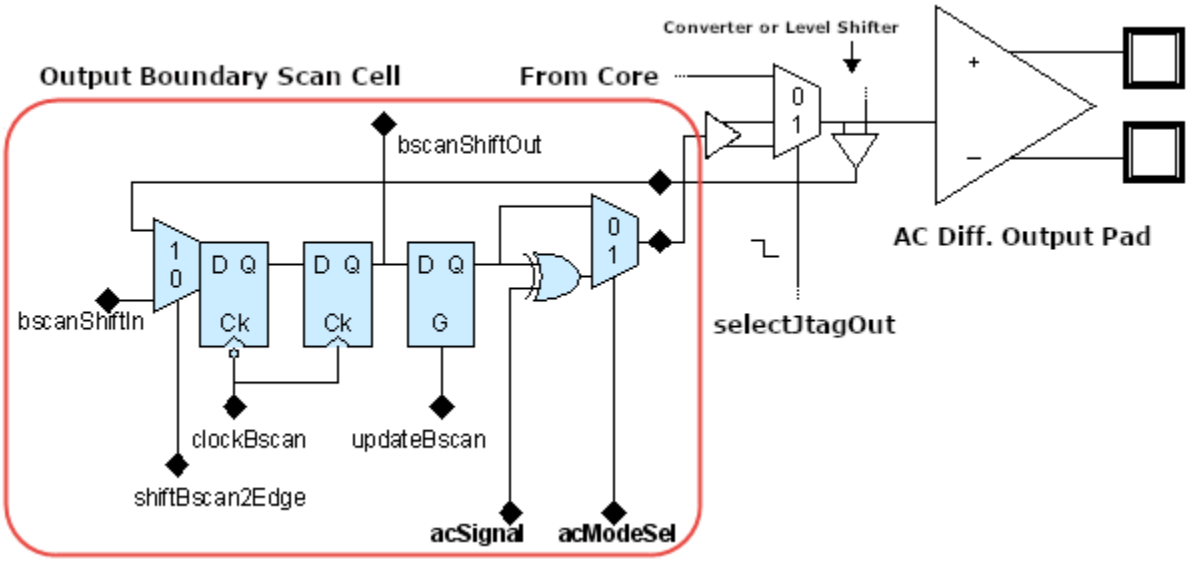
This subsection describes only one example of differential input and one example of differential output AC-pads, along with their corresponding boundary-scan cells. Bidirectional IEEE 1149.6 boundary-scan cells are built from a concatenation of one output and one input IEEE 1149.6 boundary-scan cell.

Differential Output Pad Example

This example describes a boundary scan cell for differential outputs.

Figure B-2 shows the schematic of the boundary scan cell for differential outputs with example connections to an output pad cell. This boundary scan cell is slightly different from a standard Tessent output boundary scan cell—an EXOR and multiplexer have been added.

Figure B-2. Output Boundary Scan Cell (One Per Differential Pair)



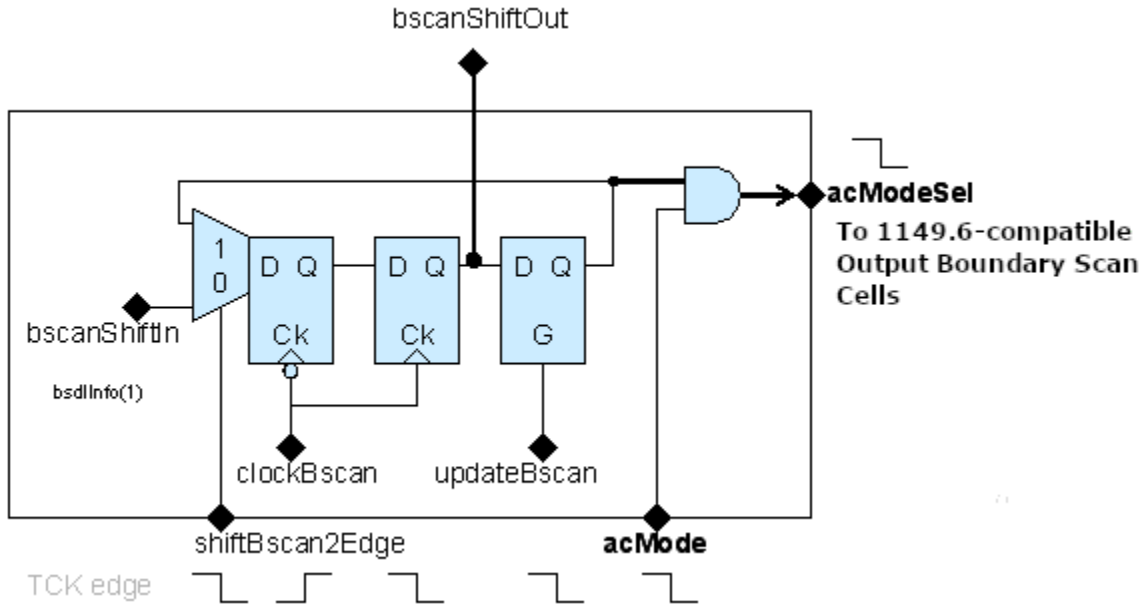
Typically, the multiplexer in the functional signal path is not in the boundary scan cell because the multiplexer conveys *GHz* signals in functional mode. If the multiplexer conveys differential core signals, you also need a differential-to-single-ended converter. The output of the update latch is EXORed with the *acSignal* when this boundary scan cell is in *AC* mode (either one of the two IEEE 1149.6 instructions is active and the output group is enabled for *AC* mode).

AC Select Cell Example

The AC select cells provide a way to group individual output pins into ACGroups.

Figure B-3 shows an example AC select cell. In these groups, you can turn the *AC* mode output pin inversion on or off during RTL on a per-group basis. Its output acModeSel pin can connect to many 1149.6-compatible output boundary scan cells of the same-named input pins.

Figure B-3. AC Select Cell

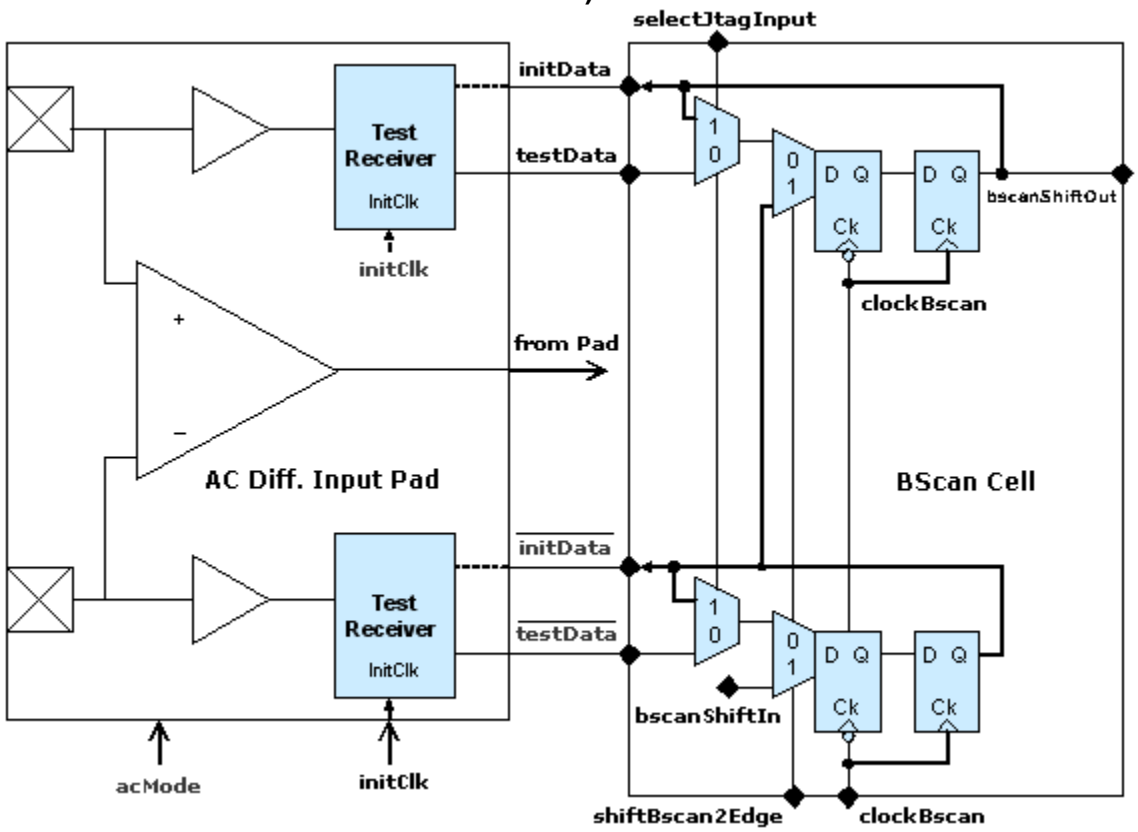


Differential Input Pad Example

This example describes the boundary scan cell for differential inputs. The cell contains two observe-only boundary scan registers: one for the positive pin and one for the negative pin.

The input AC pad communicates with the sample-only boundary scan cells through the initData(inv) and the testData(inv) signals. One sample-only cell captures the value of the positive pin, and the other captures the value of the negative pin. For single-ended AC-coupled inputs, you need only one sample cell. A special *Test Receiver* circuit processes the *testData* signals, which is mandatory in the IEEE 1149.6 standard. Tessent BoundaryScan supports only the IEEE 1149.6 pads where the hysteretic memory element is embedded within the test receivers. Figure B-4 shows an example of a test receiver with an embedded hysteretic memory element.

Figure B-4. Input Boundary Scan Cell (One Sample-Only Cell Per Differential Pin)



The following summarizes the IEEE 1149.6 signals:

- *acMode* (optional) — This pin controls the behavior of the test receiver's input comparator. When the pin is 1, the input test receiver is in AC mode. This pin can either enable an internal low-pass filter or change the reference voltage for the hysteretic comparator. Refer to Figure B-4 for an example of this test receiver.
- *initClk* — The hysteresis of the test receiver is initialized while this signal is logic 1 or when the signal transitions from 0 to 1.
- *initData*, *initDataInv* — The hysteresis of the test receiver is initialized as if these data values were the most recent output of the hysteretic receiver. *initData* is associated with the positive pin, and *initDataInv* to the negative pin.
- *testData*, *testDataInv* — The pin value of positive and negative pins, after processing by the test receivers, captured by the boundary scan cells.
- *fromPad* — This is the functional mode signal in high-speed SerDes pads. The output is normally connected to a deserializer circuit and is typically not available for boundary scan observation.

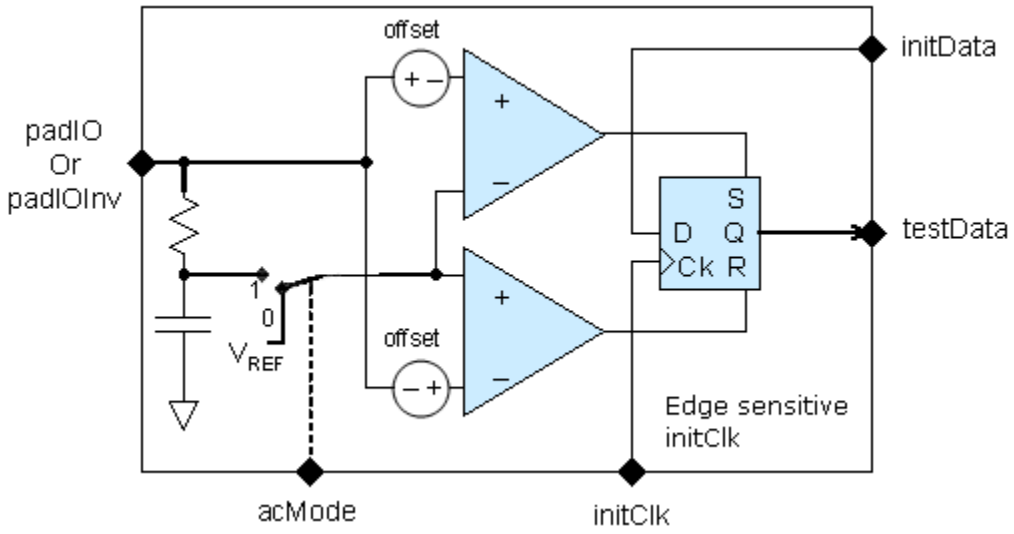
Test Receiver Example

This example describes the Tessent BoundaryScan support for test receivers with an embedded hysteretic memory element.

Tessent BoundaryScan supports only IEEE 1149.6 pads where the hysteretic memory element is embedded within the test receivers, as shown in Figure B-5. In the example, an optional *acMode* input pin selects the reference for the input comparator between a fixed voltage (in DC mode) and a low-pass filter output (in AC mode).

The test receiver's hysteretic memory element is a positive-edge flip-flop. Tessent BoundaryScan also supports test receivers with a memory element made of either a positive- or negative-edge flip-flop, or a gated-low or gated-high latch. For more test receiver circuit examples, refer to the IEEE 1149.6 standard.

Figure B-5. Example of a Test Receiver With an Embedded Hysteretic Memory Element



Appendix C
Getting Help

There are several ways to get help when setting up and using Tessent software tools. Depending on your need, help is available from documentation, online command help, and Siemens EDA Support.

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 —** You 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.