Tessent® TestKompress® User’s Manual

Software Version v2013.4

December 2013

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

Table of Contents

Chapter 8

Chapter 9

Appendix A

Appendix B

Appendix C

List of Figures

List of Tables

Chapter 1
Getting Started

This manual describes how to integrate Tessent® TestKompress® into your design process. More information can be found in the following manuals:

- *Tessent Shell Reference Manual* — Contains information on Tessent TestKompress commands and information for all DRCs including the Tessent TestKompress-specific EDT Rules.
- *Tessent Shell User's Manual*— Contains information about the Tessent Shell environment in which you use Tessent TestKompress.

For a complete list of Mentor Graphics Tessent-specific terms, including Tessent TestKompress-specific terms, refer to the *Tessent Glossary*.

Tessent TestKompress

Tessent TestKompress is a Design-for-Test (DFT) product that creates test patterns and implements compression for the testing of manufactured ICs. Advanced compression reduces ATE memory and channel requirements and reduced data volume results in shorter test application times and higher tester throughput than with traditional ATPG. TestKompress also supports traditional ATPG.

Tessent TestKompress creates and embeds compression logic (EDT logic) and generates compressed test patterns as follows:

- **Test patterns** — Compressed test patterns are generated and loaded onto the Automatic Test Equipment (ATE).
- **Embedded logic** — EDT logic is generated and embedded in the IC to:

  a. Receive the compressed test patterns from the ATE and decompress them.

  b. Deliver the uncompressed test patterns to the core design for testing.

  c. Receive and compress the test results and return them to the ATE.

Tessent TestKompress is command-line driven from Tessent Shell:

- The IP Creation phase of Tessent TestKompress is executed in the Tessent Shell "dft -edt" context.
- The Pattern Generation phase of Tessent TestKompress is executed in the Tessent Shell "patterns -scan" context.

Supported Test Patterns

Tessent TestKompress supports all types of test patterns except:

- Random pattern generation.
- Tessent FastScan™ MacroTest. You can only apply MacroTest patterns to a design with Tessent TestKompress by accessing the scan chains directly, bypassing the EDT logic.

Supported Scan Architectures

Tessent TestKompress logic supports mux-DFF and LSSD or a mixture of the scan architectures as listed in Table 1-1.

Table 1-1. Supported Scan Architecture Combinations

EDT Logic	Supported Scan Architectures
DFF-based	LSSD, Mux-DFF, and mixed
Latch-based	LSSD

Tessent TestKompress Inputs

You need the following components to use Tessent TestKompress:

- Scan-inserted gate-level Verilog netlist.
- Synthesis tool.
- Compatible Tessent cell library of the models used for your design scan circuitry. If necessary, you can convert Verilog libraries to a compatible Tessent cell library format with the LibComp utility. For more information, see “Using LibComp to Create Tessent ATPG Models” in the *Tessent Cell Library Manual*.
- Timing simulator such as ModelSim.

Potential Affects of Tessent TestKompress on the Design

Depending on the configuration and placement of the EDT logic, your design may be affected as follows:

- **Extra level of hierarchy** — If you place the EDT logic outside the core design, you must add a boundary scan wrapper which adds a level of hierarchy.
- **Minimal physical space** — The size of the EDT logic is roughly about 25 gates per internal scan chain. The following examples can be used as guidelines to roughly estimate the size of the EDT logic for a design:
    - For a one million gate design with 200 scan chains, the logic BIST controller including PRPG, MISR and the BIST controller, is 1.25 times the size of the EDT logic for 16 channels.
    - For a one million gate design configured into 200 internal scan chains, the EDT logic including decompressor, compactor, and bypass circuitry with lockup cells

requires less than 20 gates per chain. The logic occupies an estimated 0.35% of the area. The size of the EDT logic does not vary significantly based on the size of the design.

- For 8 scan channels and 100 internal scan chains, the EDT logic was found to be twice as large as a TAP controller, and 19% larger than the MBIST™ controller for a 1k x 8-bit memory.

EDT Technology

Embedded Deterministic Testing (EDT) is the technology used by Tessent TestKompress. EDT technology is based on traditional, deterministic ATPG and uses the same fault models to obtain similar test coverage using a familiar flow. EDT extends ATPG with improved compression of scan test data and a reduction in test time.

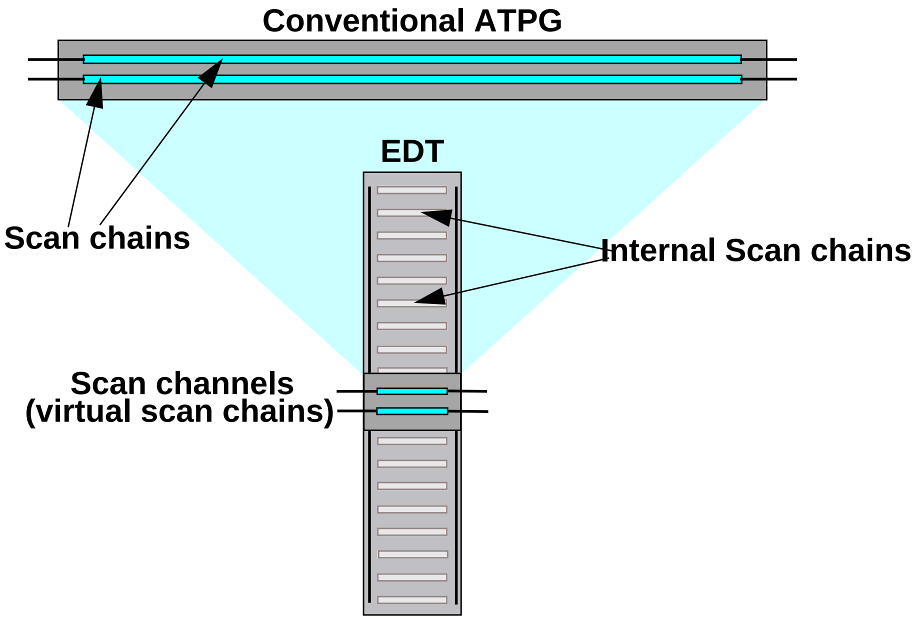
Tessent TestKompress achieves compression of scan test data by controlling a large number of internal scan chains using a small number of scan channels. Scan channels can be thought of as *virtual* scan chains because, from the point of view of the tester, they operate exactly the same as traditional scan chains. Therefore, any tester that can apply traditional scan patterns can apply compressed patterns as described in the following topics:

- Scan Channels
- Structure and Function
- Test Patterns

Scan Channels

With Tessent TestKompress, the number of internal scan chains is significantly larger than the number of external *virtual* scan chains the EDT logic presents to the tester. Figure 1-1 illustrates conceptually how a design tested with EDT technology is seen from the tester compared to the same design tested using conventional scan and ATPG.

Figure 1-1. EDT as Seen from the Tester



Under EDT methodology, the *virtual* scan chains are called *scan channels* to distinguish them from the scan chains inside the core. Their number is significantly less than the number of internal scan chains. The amount of compression is determined by two parameters:

- number of scan chains in the design core
- number of scan channels presented to the tester

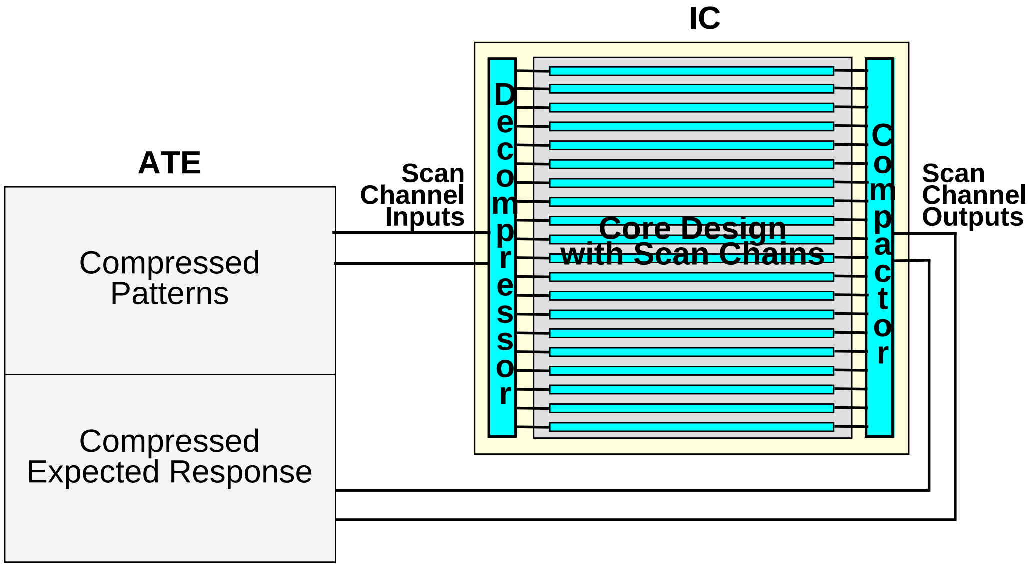
For more information on establishing a compression target for your application, see "Effective Compression" on page 20 and "Compression Analysis" on page 48.

Structure and Function

EDT technology consists of logic embedded on-chip, new EDT-specific DRCs, and a deterministic pattern generation technique.

The embedded logic includes a decompressor located between the external scan channel inputs and the internal scan chain inputs and a compactor located between the internal scan chain outputs and the external scan channel outputs. See Figure 1-2.

Figure 1-2. Tester Connected to a Design with EDT



You have the option of including bypass circuitry for which a third block (not shown) is added. No additional logic (test points or X-bounding logic) is inserted into the core of the design. Therefore, EDT logic affects only scan channel inputs and outputs, and thus has no effect on functional paths.

Figure 1-2 shows an example design with two scan channels and 20 short internal scan chains. From the point of view of the ATE, the design appears to have two scan chains, each as long as the internal scan chains. Each compressed test pattern has a small number of *additional shift cycles*, so the total number of shifts per pattern would be slightly more than the number of scan cells in each chain.

Note

The term *additional shift cycles* refers to the sum of the initialization cycles, masking bits (when using Xpress), low-power bits (when using a low-power decompressor), and user-defined pipeline bits.

You can use the following equation to predict the number of initialization cycles the tool adds to each pattern load. (In this equation, *ceil* indicates the ceiling function that rounds a fraction to the next highest integer.) This equation applies except when you have very few channels in which case there are four extra cycles per scan load. (Note, this equation does not factor in additional shift cycles added to support masking and low-power.)

$$\text{Number of initialization cycles} = ceil\left(\frac{\text{decompressor size}}{\text{number of channels}}\right)$$

For example, if a design has 16 scan channels, 1250 scan cells per chain, and a 50-bit decompressor, we can calculate the number of initialization cycles as 4 by using the above formula. Since each chain has 1,250 scan cells and each compressed pattern requires four initialization cycles, the tester sees a design with 16 chains requiring 1,254 shifts per pattern.

Note

The EDT IP creation phase and ATPG generation phase may report a different number of initialization cycles depending on whether low power is enabled. Enabling low power increases the number of initialization cycles in the EDT IP creation phase.

Test Patterns

Tessent Shell generates compressed test patterns specifically for on-chip processing by the EDT logic. For a given testable fault, a compressed test pattern satisfies ATPG constraints and avoids bus contention, similar to conventional ATPG.

A set of compressed test patterns is stored on the ATE and each test pattern applies data to the inputs of the decompressor and holds the responses observed on the outputs of the compactor. The ATE applies the compressed test patterns to the circuit through the decompressor, which lies between the scan channel pins and the internal scan chains. From the perspective of the tester, there are relatively few scan chains present in the design.

The compressed test patterns, after passing through the decompressor, create the necessary values in the scan chains to guarantee fault detection. The functional input and output pins are directly controlled (forced) and observed (measured) by the tester, same as in conventional test. On the output side of the internal scan chains, hardware compactors reduce the number of internal scan chains to feed the smaller number of external channels. The response captured in the scan cells is compressed by the compactor and the compressed response is compared on the tester. The compactor ensures faults are not masked and X-states do not corrupt the response.

You define parameters, such as the number of scan channels and the insertion of lockup cells, which are also part of the RTL code. The tool automatically determines the internal structure of the EDT hardware based on the parameters you specify, the number of internal scan chains, the length of the longest scan chain, and the clocking of the first and last scan cell in each chain. Test patterns include parallel and serial test benches for Verilog as well as parallel and serial WGL, and most other formats supported formats.

TestKompress Compression Logic

Tessent TestKompress generates hardware in blocks in VHDL or Verilog RTL. You integrate the compression logic (EDT logic) into your design by using Tessent Shell with the core level of the design. The tool then generates the following three components:

- Decompressor — Feeds a large number of scan chains in your core design from a small number of scan channels, and decompresses EDT scan patterns as they are shifted in.

The decompressor resides between the channel inputs (connected to the tester) and the scan chain inputs of the core. Its main parts are an LFSM and a phase shifter.

- Compactor — Compacts the test responses from the scan chains in your core design into a small number of scan output channels as they are shifted out.

  The compactor resides between the core scan chain outputs and the channel outputs connected to the tester. It primarily consists of spatial compactor(s) and gating logic.

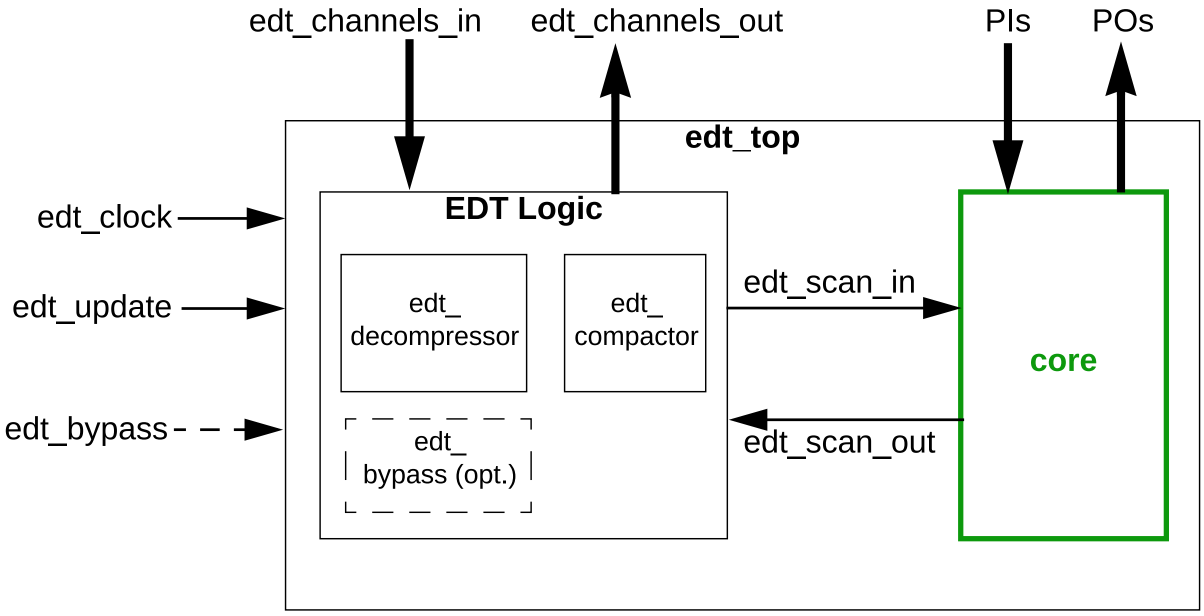
- Bypass Module (Optional) — Bypasses the EDT logic by using multiplexers (and lockup cells if necessary) to concatenate the internal scan chains into fewer, longer chains. Enables you to access the internal scan chains directly through the channel pins. Generated by default.

  If you choose to implement bypass circuitry, the tool includes bypass multiplexers in the EDT logic. Chapter 7, "Compression Bypass Logic," discusses bypass mode. You can also insert the bypass logic in the netlist at scan insertion time to facilitate design routing. For more information, see "Insertion of Bypass Chains in the Netlist" on page 40.

These three components are all contained within the EDT logic block that, by default, is instantiated in a top-level "wrapper" module. The design core is also instantiated in the top-level wrapper. This is illustrated conceptually in Figure 1-3.

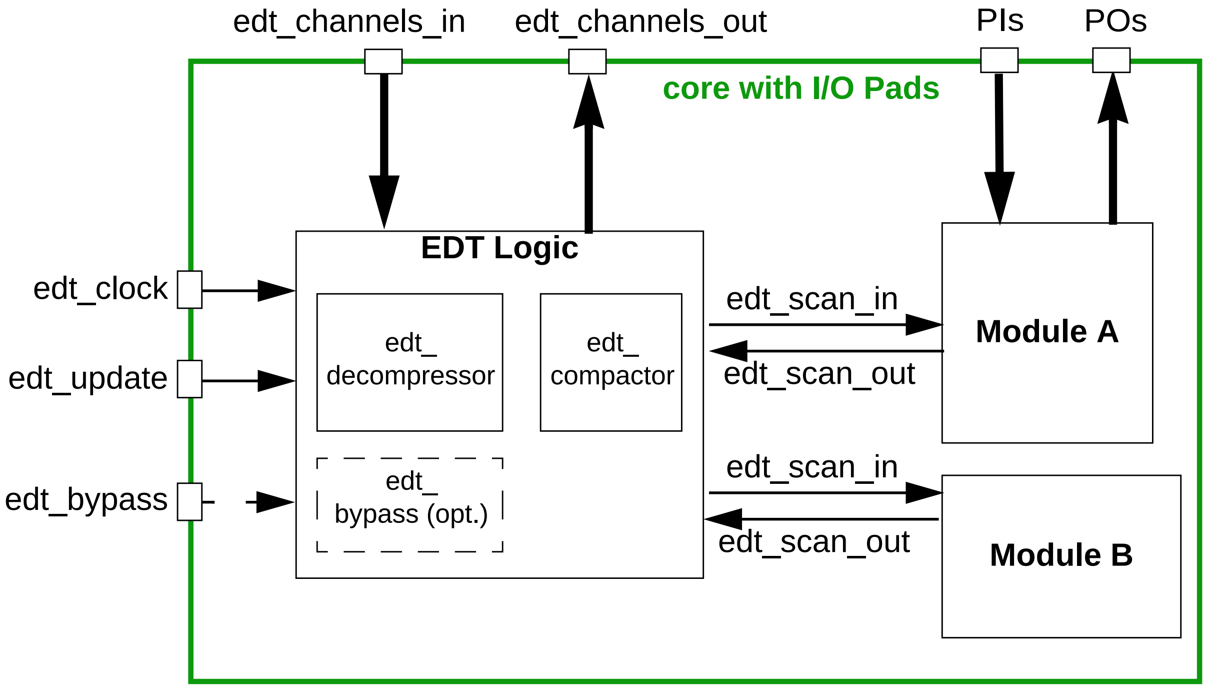
You insert pads and I/O cells on this new top level. Because the EDT logic is outside the core design (that is, outside the netlist used in Tessent Shell), the tool flow you use to implement this configuration is referred to as the external EDT logic location flow, or simply the "external flow."

Figure 1-3. EDT logic Located Outside the Core (External Flow)



Alternatively, you can invoke Tessent Shell and use a design that already contains I/O pads. For these designs, the tool enables you to insert the EDT logic block in the existing top level within the original design. This is shown conceptually in Figure 1-4. Because the EDT logic is instantiated within the netlist used in Tessent Shell, this configuration is referred to as the internal EDT logic location flow or simply the "internal flow."

Figure 1-4. EDT logic Located Within the Core (Internal Flow)



By default, the tool automatically inserts lockup cells as needed in the EDT logic. They are placed within the EDT logic, between the EDT logic and the design core, and in the bypass circuitry that concatenates the scan chains. The section, "Understanding Lockup Cells," describes in detail how the tool determines where to insert lockup cells.

DRC Rules

Tessent TestKompress performs the same ATPG design rules checking (DRC) after design flattening that Tessent FastScan performs. A detailed discussion of DRC is included in "ATPG Design Rules Checking" in the *Tessent Scan and ATPG User's Manual*.

In addition, Tessent TestKompress also runs a set of DRCs specifically for EDT. For more information, see "Design Rule Checks" on page 75."

Internal Control

In many cases, it is preferable to use internal controllers (JTAG or test registers) to control EDT signals, such as edt_bypass, edt_update, scan_en, and to disable the edt_clock in functional mode. For detailed information about how to do this with boundary scan, refer to "Uncompressed ATPG (External Flow) and Boundary Scan" on page 181.

Logic Clocking

The default EDT logic contains combinational logic and flip-flops. All the flip-flops, except lockup cells, are positive edge-triggered and clocked by a dedicated clock signal that is different from the scan clock. There is no clock gating within the EDT logic, so it does not interfere with the system clock(s) in any way.

You can set up the clock to be a dedicated pin (named edt_clock by default) or you can share the clock with a functional non-clock pin. Such sharing may cause a decrease in test coverage because the tool constrains the clock pin during test pattern generation. You must not share the edt_clock with another clock or RAM control pin for several reasons:

- If shared with a scan clock, the scan cells may be disturbed when the edt_clk is pulsed in the load_unload procedure during pattern generation.
- If shared with RAM control signals, RAM sequential patterns and multiple load patterns may not be applicable.
- If shared with a non-scan clock, test coverage may decline because the edt_clk is constrained to its off-state during the capture cycle.

Because the clock used in the EDT logic is different than the scan clock, lockup cells can be inserted automatically between the EDT logic and the scan chains as needed. The tool inserts lockup cells as part of the EDT logic and never modifies the design core.

Note

You can set the EDT clock to pulse before the scan chain shift clocks and avoid having lockup cells inserted. For more information, see "Pulse EDT Clock Before Scan Shift Clocks" on page 64.

Latch-based EDT logic uses two clocks (a master and a slave clock) to drive the logic. For reasons similar to those listed above for DFF-based logic, you must not share the master EDT clock with the system master clock. You can, however, share the slave EDT clock with the system slave clock.

Note

During the capture cycle, the system slave clock, which is shared with the slave EDT clock, is pulsed. This does not affect the EDT logic because the values in the master latches do not change. Similarly, in the load_unload cycle, although the slave EDT clock is pulsed, the value at the outputs of the system slave latches is unchanged because the slave latches capture old values.

In a skew load procedure, when a master clock is only pulsed at the end of the shift cycle (so different values can be loaded in the master and slave latches), the EDT logic is unaffected because the master EDT clock is not shared.

ASCII and Binary Patterns

Compressed ATPG test patterns can be written out in ASCII and binary formats, and can also be read back into the tool. As with uncompressed patterns, you use these formats primarily for debugging simulation mismatches and archiving. However, there are some differences with compressed and uncompressed patterns as follows:

- Compressed and uncompressed ASCII patterns are different in several ways. When you create patterns with compression, the captured data is stored with respect to the internal scan chains and the load data is stored with respect to the external scan channels. The load data in the pattern file is in compressed format—the same form it is fed to the decompressor.
- With the simulation of compressed patterns, Xs may not be due to capture; they may result from the emulation of the compactor. For a detailed discussion of this effect and how masking is done with compressed patterns, refer to "Understanding Scan Chain Masking in the Compactor" on page 218.

Fault Models and Test Patterns

For compression, the tool uses fault-model independent and pattern-type independent compression algorithms. The compression technology supports all fault models (stuck-at, transition, Iddq, and path delay) and deterministic pattern types (combinational, RAM sequential, clock-sequential, and multiple loads) supported and/or generated by uncompressed ATPG.

To summarize, the compression technology:

- Accepts the same fault models as uncompressed ATPG.
- Accepts the same deterministic pattern types as uncompressed ATPG with the exception of MacroTest which is not supported.
- Produces the same test coverage as uncompressed ATPG.

Effective Compression

Effective compression is the actual compression achieved for a specific test application. The effective compression is determined by balancing the EDT compression characteristics with the test environment/design needs.

The effective compression is limited by many parameters, including the following:

- number of scan chains in your design core
- number of scan channels presented to the tester

Use the following ratio to determine the chain to channel ratio for an application:

$$Chaintochannelratio \cong \frac{\text{\# of Scan Chains}}{\text{\# of Scan Channels}}$$

The effective compression achieved for a design is always less than the chain to channel ratio because the EDT technology generates more test patterns than traditional ATPG. With EDT technology, compression is achieved by reducing the amount of data per test pattern and not by reducing the number of test patterns generated. Consequently, additional test patterns require additional shift cycles that reduce the overall compression.

Note

The term *additional shift cycles* refers to the sum of the initialization cycles, masking bits (when using Xpress), and low-power bits (when using a low-power decompressor).

It is also important to balance the compression target with the testing resources and design needs. Using an unnecessarily large compression target may have an adverse affect on compression, testing quality, and design layout as follows:

- **Lower test coverage** — Higher compression ratios increase the compression per test pattern but also increase the possibility of generating test patterns that cannot be compressed and can lead to lower test coverage.
- **Decrease in overall compression** — Higher compression ratios also decrease the number of faults that dynamic compaction can fit into a test pattern. This can increase the total number of test patterns and, therefore, decrease overall compression.
- **Routing congestion** — There is no limit to the number of internal scan chains, however, routing constraints may limit the compression ratio. Most practical configurations will not exceed the compression capacity.

For more information on determining the right compression for your design, see "Compression Analysis" on page 48.

TestKompress Flow Overview

This section describes the default Tessent TestKompress flow by briefly introducing the steps required to incorporate EDT into a gate-level Verilog netlist. This summary is intended to provide an overview of the three main phases in the flow:

- Creating the EDT Logic
- Synthesizing the EDT Logic
- Generating Test Patterns

Creating the EDT Logic

1. Invoke Tessent Shell.

```
<Tessent_Tree_Path>/bin/tessent -shell -dofile edt_ip_creation.do \
     -logfile ../transcripts/edt_ip_creation.log -replace
```

2. Provide Tessent Shell commands. For example:

Tip: The following commands can be located in the dofile used for invocation in step 1.

```
// Set context, read library, read and set current design
set_context dft -edt
read_verilog gatelevel_netlist.v
read_cell_library atpg.lib
set_current_design top

// Setup Scan Chains and Clocks
add_scan_groups grp1 ../generated/atpg.testproc
add_scan_chains chain1 grp1 edt_si1 edt_so1
add_scan_chains chain2 grp1 edt_si2 edt_so2
...
add_scan_chains chain5 grp1 edt_si5 edt_so5
analyze_control_signals -auto_fix

// Specify the number of scan channels.
set_edt_options -channels 1

// Flatten the design, run DRCs.
set_system_mode analysis

// Verify the EDT configuration is as expected.
report_edt_configurations -verbose

// Generate the RTL EDT logic and save it.
write_edt_files created -verilog -replace

// At this point, you can optionally create patterns (without saving them)
// to get an estimate of the potential test coverage.
create_patterns

// Create reports
report_statistics
report_scan_volume

// Close the session and exit.
exit
```

Synthesizing the EDT Logic

1. Run Design Compiler.

Note
The Design Compiler synthesis script referenced in the following invocation line is output from “write_edt_files” in preceding step 2.

```
dc_shell -f ../created_dc_script.scr |& tee ../transcripts/dc_edt.log
```

Generating Test Patterns

1. Invoke Tessent Shell.

Note

The netlist *created_edt_top_gate.v* referenced in the following invocation line is output from Design Compiler (see the previous section, “Synthesizing the EDT Logic”).

```
<Tessent_Tree_Path>/bin/tessent -shell -dofile edt_pattern_gen.do \
      -logfile ../transcripts/edt_pattern_gen.log -replace
```

2. Provide Tessent Shell commands. For example:

```
// Set context, read library, read and set current design
set_context patterns -scan
read_verilog created_edt_top_gate.v
read_cell_library atpg.lib
set_current_design top

// Run the *_edt.dofile output from “write_edt_files” when creating
// the EDT logic.
dofile ../created_edt.dofile

// Flatten the design, run DRCs.
set_system_mode analysis

// Verify the EDT configuration.
report_edt_configurations

// Generate patterns.
create_patterns

// Create reports.
report_statistics
report_scan_volume

// Save the patterns in ASCII format.
write_patterns ../generated/patterns_edt.ascii -ascii -replace

// Save the patterns in parallel and serial Verilog format.
write_patterns ../generated/patterns_edt_p.v -verilog -replace -parallel
write_patterns ../generated/patterns_edt_s.v -verilog -replace -serial
   -sample 2

// Save the patterns in tester format; WGL for example.
write_patterns ../generated/test_patterns.wgl -wgl -replace

// Close the session and exit.
exit
```

Tessent Shell User Interface

Tessent Shell is a Unix/Linux command line driven tool that provides access to Tessent FastScan for uncompressed ATPG and to Tessent TestKompress for compressed ATPG.

Invocation

You can invoke Tessent Shell from the command line as described in “TestKompress Flow Overview. To exit Tessent Shell and return to the operating system, type “exit” at the command line:

```
prompt> exit
```

For more information on invoking Tessent Shell, see the tessent command in the *Tessent Shell Reference Manual*.

Uncompressed and Compressed ATPG

For uncompressed ATPG, you use Tessent Shell in the “patterns -scan” context.

For compressed ATPG, you use Tessent Shell in the “dft -edt” context to create the EDT logic, and in the “patterns -scan” context to generate compressed test patterns.

EDT must be on whenever you are creating test patterns or EDT logic. You can use the report_environment command to check the tool status. You can use the set_edt_options command to enable compression.

For more information about Tessent Shell and contexts, see “Tessent Shell Introduction” in the *Tessent Shell User’s Manual*.

Supported Design Format

For pattern generation, you can read in a scan-inserted gate-level Verilog netlist and a compatible Tessent cell library of the models used for the scan circuitry.

For more information on the Tessent cell library, see “Creating ATPG Models” in the *Tessent Cell Library Manual*.

Batch Mode

You can run Tessent Shell in batch mode by using a *dofile* to pipe commands into the application. Dofiles let you automatically control the operations of the tool. The dofile is a text file you create that contains a list of application commands that you want to run, but without entering them individually. If you have a large number of commands, or a common set of commands you use frequently, you can save time by placing these commands in a dofile.

If you place all commands, including the exit command, in a dofile, you can run the entire session as a batch process from the command line. Once you generate a dofile, you can run it at invocation.

For example, to run a dofile as a batch process using the commands contained in the dofile *my_dofile.do*, enter:

```
<Tessent_Tree_Path>/bin/tessent -shell -dofile my_dofile.do
```

The following shows an example Tessent Shell dofile:

```
// my_dofile.do
//
// Dofile for EDT logic Creation Phase.

// Execute setup script from Tessent Scan.
   dofile edt_ip_creation.do

// Set up EDT.
set_edt_options -channels 2

// Run DRC.
set_system_mode analysis

// Report and write EDT logic.
report_edt_configurations
report_edt_pins
write_edt_files created -verilog -replace

// Exit.
exit
```

By default, if the tool encounters an error when running one of the commands in the dofile, it stops dofile execution. However, you can turn this setting off or specify to exit to the shell prompt by using the set_dofile_abort command.

Log Files

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

You can generate log files by using the -Logfile switch when you invoke the tool. When setting up a log file, you can instruct Tessent Shell to generate a new log file, replace an existing log file, or append information to a log file that already exists.

You can also use the set_logfile_handling command to generate a log file during a tool session.

Note

A log file created during a tool session only contains notes, warnings, and error messages that occur after you issue the set_logfile_handling command. Therefore, you should enter it as one of the first commands in the session.

UNIX Commands

You can run UNIX operating system commands within Tessent Shell by using the "system" command. For example, the following command executes the UNIX operating system command **date** within a Tessent Shell session:

```
prompt> system date
```

Disk Space

To conserve disk storage space, the tools can read and write disk files using either the UNIX **compress** or the GNU **gzip** command. When you provide a filename with the appropriate filename extension (".Z" for compress, or ".gz" for gzip), the tools automatically process the file using the appropriate utility. Two commands control this capability:

- set_file_compression - Turns file compression on or off. This command applies to *all* files that the tool reads from and writes to.
- set_gzip_options - Specifies which GNU **gzip** options to use when the tool is processing files that have the .gz extension.

Note

The *file* compression used by the tools to manage disk storage space is unrelated to the *pattern* compression you apply to test pattern sets in order to reduce the pattern count. You will see many references to the latter type of compression throughout the DFT documentation.

Chapter 2
The Compressed Pattern Flows

The flows shown in Figures 2-1 and 2-2 compare the basic steps and tools used for an uncompressed ATPG top-down design flow with the steps and tools used to incorporate compressed patterns in both an external and an internal flow. These flows primarily show the typical top-down design process flow using a structured compression strategy.

This manual discusses the steps shown in grey in Figures 2-1 and 2-2; it also mentions certain aspects of other design steps, where applicable. For more information on the ATPG flow, see the *Tessent Scan and ATPG User's Manual*; that information is not repeated in this section.

Figure 2-1. Top-Down Design Flow - External

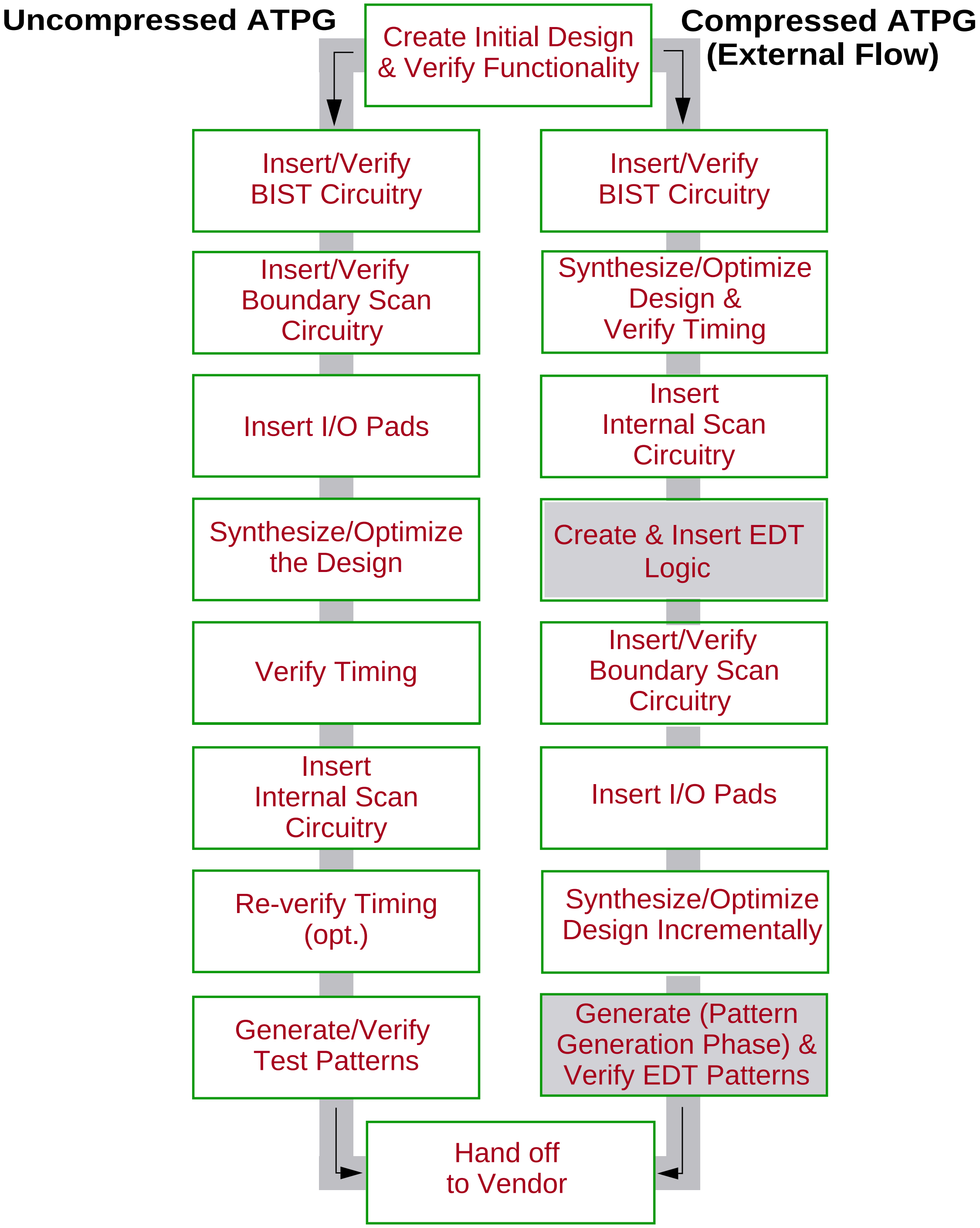
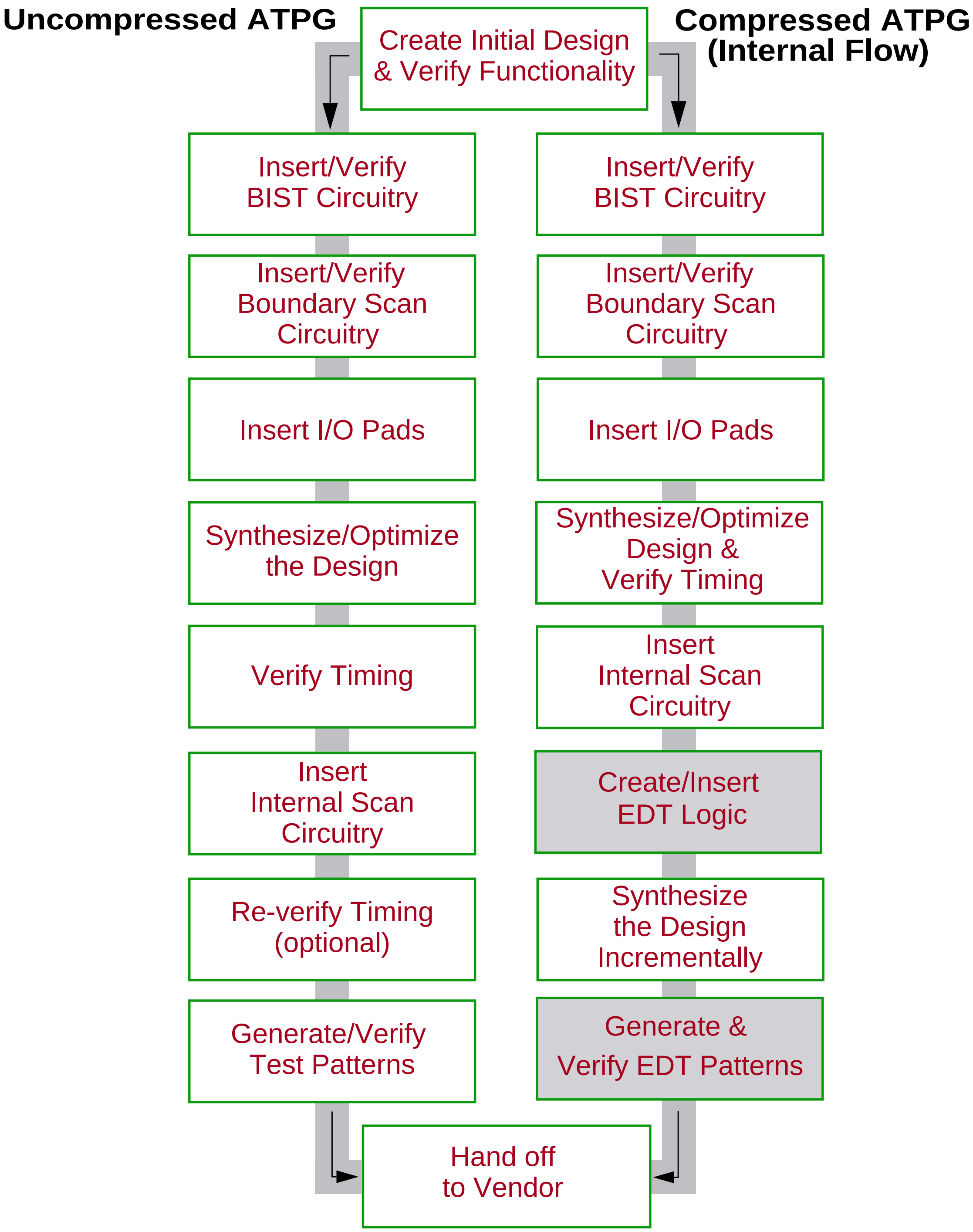

Uncompressed ATPG
Create Initial Design & Verify Functionality
Compressed ATPG (External Flow)
Insert/Verify BIST Circuitry
Insert/Verify Boundary Scan Circuitry
Insert I/O Pads
Synthesize/Optimize the Design
Verify Timing
Insert Internal Scan Circuitry
Re-verify Timing (opt.)
Generate/Verify Test Patterns
Insert/Verify BIST Circuitry
Synthesize/Optimize Design & Verify Timing
Insert Internal Scan Circuitry
Create & Insert EDT Logic
Insert/Verify Boundary Scan Circuitry
Insert I/O Pads
Synthesize/Optimize Design Incrementally
Generate (Pattern Generation Phase) & Verify EDT Patterns
Hand off to Vendor

Figure 2-2. Top-Down Design Flow - Internal


Uncompressed ATPG
Create Initial Design & Verify Functionality
Compressed ATPG (Internal Flow)
Insert/Verify BIST Circuitry
Insert/Verify BIST Circuitry
Insert/Verify Boundary Scan Circuitry
Insert/Verify Boundary Scan Circuitry
Insert I/O Pads
Insert I/O Pads
Synthesize/Optimize the Design
Synthesize/Optimize Design & Verify Timing
Verify Timing
Insert Internal Scan Circuitry
Insert Internal Scan Circuitry
Create/Insert EDT Logic
Re-verify Timing (optional)
Synthesize the Design Incrementally
Generate/Verify Test Patterns
Generate & Verify EDT Patterns
Hand off to Vendor

Top-Down Design Flows

The first task in any design flow is to create the initial register transfer level (RTL) design, using whatever means you choose. If your design is in Verilog format and contains memory models, you can add built-in self-test (BIST) circuitry to your RTL design. You then choose to use either an uncompressed or a compressed pattern flow.

Uncompressed ATPG Flow

Commonly, in an ATPG flow that does *not* use compression, you would next insert and verify I/O pads and boundary scan circuitry. Then, you would synthesize and optimize the design using the Synopsys Design Compiler tool or another synthesis tool, followed by a timing verification with a static timing analyzer such as PrimeTime.

After synthesis, you are ready to insert internal scan circuitry into your design using Tessent Scan. In the uncompressed ATPG flow, after you insert scan, you could optionally re-verify the timing because you added scan circuitry. Once you were sure the design is functioning as desired, you would generate test patterns using Tessent FastScan and generate a test pattern set in the appropriate format.

Compressed Pattern Flows

By comparison, a compressed pattern flow can take one of two paths:

- External Flow (External Logic Location Flow) — Differs from the uncompressed ATPG flow in that you do not insert I/O pads and boundary scan until *after* you run Tessent Shell with the scan-inserted core to insert the EDT logic. The EDT logic is located external to the design netlist.
- Internal Flow (Internal Logic Location Flow) — Similar to an uncompressed ATPG flow, you may insert and verify I/O pads and boundary scan circuitry before you synthesize and optimize the design. The EDT logic is instantiated in the top level of the design netlist, permitting the logic to be connected to internal nodes (I/O pad cells or an internal test controller block, for example) or to the top level of the design. Typically, the EDT logic is connected to the internal nodes of the pad cells used for channel and control signals and you would run Tessent Shell with the scan-inserted core that includes I/O pads and boundary scan.

Choosing a Compressed Pattern Flow

You should choose between the external and internal flows based on whether the EDT logic signals need to be connected to nodes internal to the design netlist read into the tool (internal nodes of I/O pads, for example), or whether the EDT logic can be connected to the design using a wrapper.

In the external flow, after you insert scan circuitry the next step is to insert the EDT logic. Following that, you insert and verify boundary scan circuitry if needed. Only then do you add I/O pads. Then, you incrementally synthesize and optimize the design using either Design Compiler or another synthesis tool.

In the internal flow, you can integrate I/O pads and boundary scan into the design before the scan insertion step. Then, after you create and insert the EDT logic, use Design Compiler with the script created by Tessent Shell to synthesize the EDT logic.

In either flow, once you are sure the design is functioning as desired, you generate compressed test patterns. In this step, the tool performs extensive DRC that, among other things, verifies the synthesized EDT logic.

You should also verify that the design and patterns still function correctly with the proper timing information applied. You can use ModelSim or another simulator to achieve this goal. You may then have to perform a few additional steps required by your ASIC vendor before handing off the design for manufacture and testing.

Note

It is important to check with your vendor early in your design process for requirements and restrictions that may affect your compression strategy. Specifically, you should determine the limitations of the vendor's test equipment. To plan effectively for using EDT, you must know the number of channels available on the tester and its memory limits.

The Compressed Pattern Flows

This section presents the requirements for a compressed pattern flow and then provides an overview of the steps you follow in each of the two compressed pattern flows.

- Design Requirements for a Compressed Pattern Flow
- Compressed Pattern External Flow
- Compressed Pattern Internal Flow

The chapters that follow describe each of the steps in the flow in detail.

Note

Tessent Shell supports mux-DFF and LSSD scan architectures, or a mixture of the two, within the same design. The tool creates DFF-based EDT logic by default. However, you can direct the tool to create latch-based IP for pure LSSD designs. Table 1-1 on page 12 summarizes the supported scan architecture combinations.

Design Requirements for a Compressed Pattern Flow

Before you begin a compressed pattern flow, you must ensure that your design satisfies a set of prerequisites.

The prerequisites are:

- Format — Your design input must be in gate-level Verilog. The logic created by Tessent TestKompress is in Verilog or VHDL RTL.
- Pin Access — The design needs to allow access to all clock pins through primary input pins. There is no restriction on the number of clocks.
- I/O Pads — I/O pad requirements for the two flows are quite different as described here:
    - External Flow — The tool creates the EDT logic as a collar around the circuit (see Figure 1-3). Therefore, the core design ready for logic insertion must consist of only the core *without* I/O pads. In this flow, the tool cannot insert the logic between scan chains and I/O pads already in the design.

Note

Add the I/O pads around the collar after it is created, but before logic synthesis. The same applies to boundary scan cells: add them after the EDT logic is included in the design.

The design may or may not have I/O pads when you generate test patterns. To determine the expected test coverage, you can perform a test pattern generation trial run on the core when the EDT logic is created *before* inserting I/O pads.

Note

You should not save the test patterns generated when the EDT logic is created; these patterns do not account for how I/O pads are integrated into the final synthesized design.

When producing the final patterns for a whole chip, run Tessent Shell on the synthesized design *after* inserting the I/O pads. For more information, refer to the procedure for managing pre-existing I/O pads in section “Preparation For the External Flow” on page 38.

- Internal Flow — The core design, ready for EDT logic insertion, may include I/O pad cells for all the I/Os you inserted before or during initial synthesis. The I/O pads, when included, can be present at any level of the design hierarchy and do not necessarily have to be at the top level. If the netlist includes I/O pads, there should also be some pad cells reserved for EDT control and channel pins that are not going to be shared with functional pins. Refer to “Functional/EDT Pin Sharing” in Chapter 4 for more information about pin sharing.

Note

The design may have I/O pads; it is not a requirement. When EDT logic is inserted in the netlist, you can connect it to any internal design nodes or top level of the design netlist.

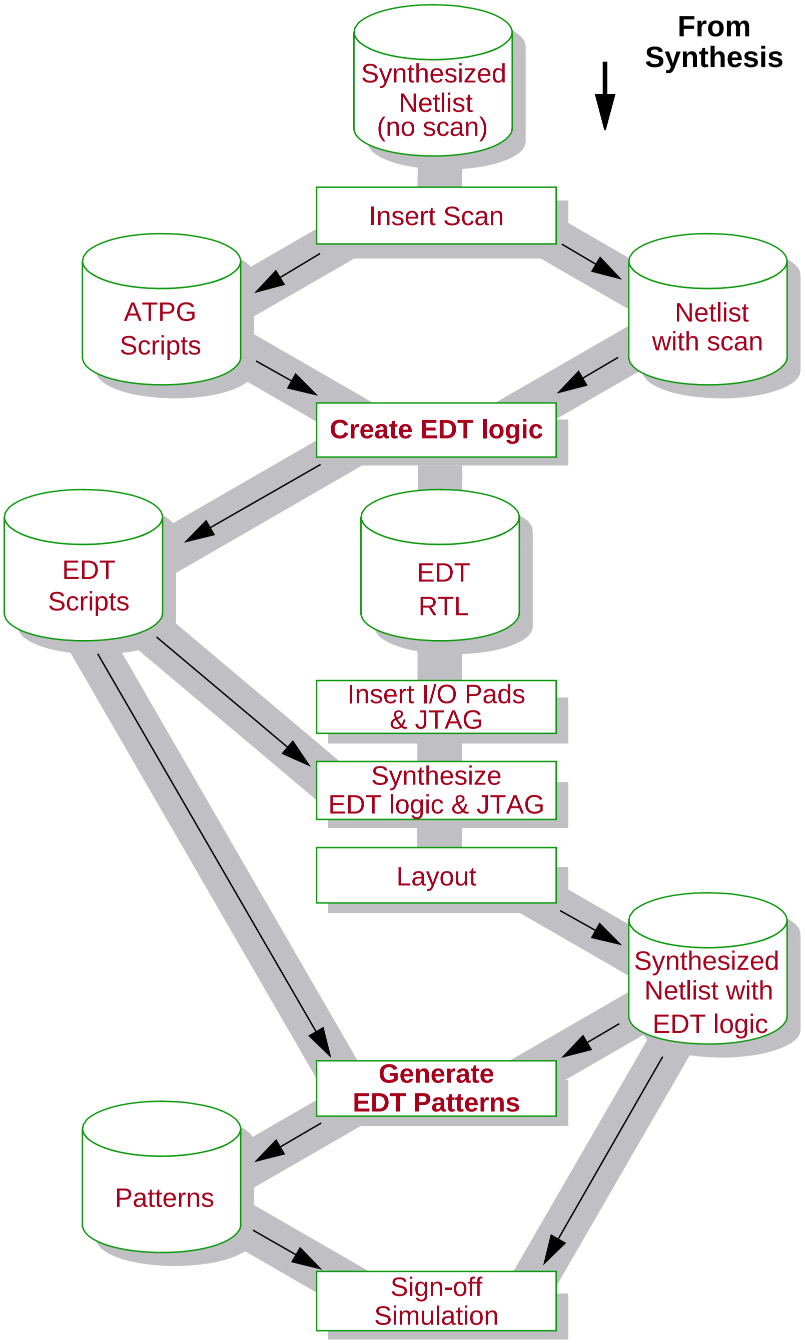
Compressed Pattern External Flow

The compressed pattern external flow is focused on EDT logic creation and EDT pattern generation.

Figure 2-3 expands the steps shown in grey in Figure 2-1, and shows the files used in the tool's external flow. The basic steps in the flow are summarized in the following list.

1. Prepare and synthesize the RTL design.
2. Insert an appropriately large number of scan chains using Tessent Scan or a third-party tool. For information on how to do this using Tessent Scan, refer to "Inserting Internal Scan and Test Circuitry," in the *Tessent Scan and ATPG User's Manual*.
3. Optionally, perform an ATPG run on the scan-inserted design without EDT. Use this run to ensure there are no basic issues such as simulation mismatches caused by an incorrect library. If you want, you can run Tessent Shell in "patterns -scan" context to perform this step.
4. Optionally, simulate the patterns created in step 3.
5. **EDT Logic Creation Phase**: Invoke Tessent Shell with the scan-inserted gate-level description of the core without I/O pads or boundary scan. Create the RTL description of the EDT logic.
6. Insert I/O pads and boundary scan (optional).
7. Incrementally synthesize the I/O pads, boundary scan, and EDT logic.
8. **EDT Pattern Generation Phase**: After you insert I/O pads and boundary scan, and synthesize all the added circuitry (including the EDT logic), invoke Tessent Shell with the synthesized top-level Verilog netlist and generate the EDT test patterns. You can write test patterns in a variety of formats including Verilog and WGL.
9. Simulate the compressed test patterns that you created in the preceding step 8. As for regular ATPG, the typical practice is to simulate all parallel patterns and a sample of serial patterns.

Figure 2-3. Compressed Pattern External Flow



Compressed Pattern Internal Flow

The compressed pattern internal flow is also focused on EDT logic creation and EDT pattern generation.

Figure 2-4 details the steps shown in grey in Figure 2-2, and shows the files used in the tool's internal flow. The basic steps in the flow are summarized in the following list.

1. Prepare and synthesize the RTL design, including boundary scan and I/O pads cells for all I/Os. Provide I/O pad cells for any EDT control and channel pins that will *not* be shared with functional pins.

Note

In this step, you must know how many EDT control and channel pins are needed, so you can provide the necessary I/O pads.

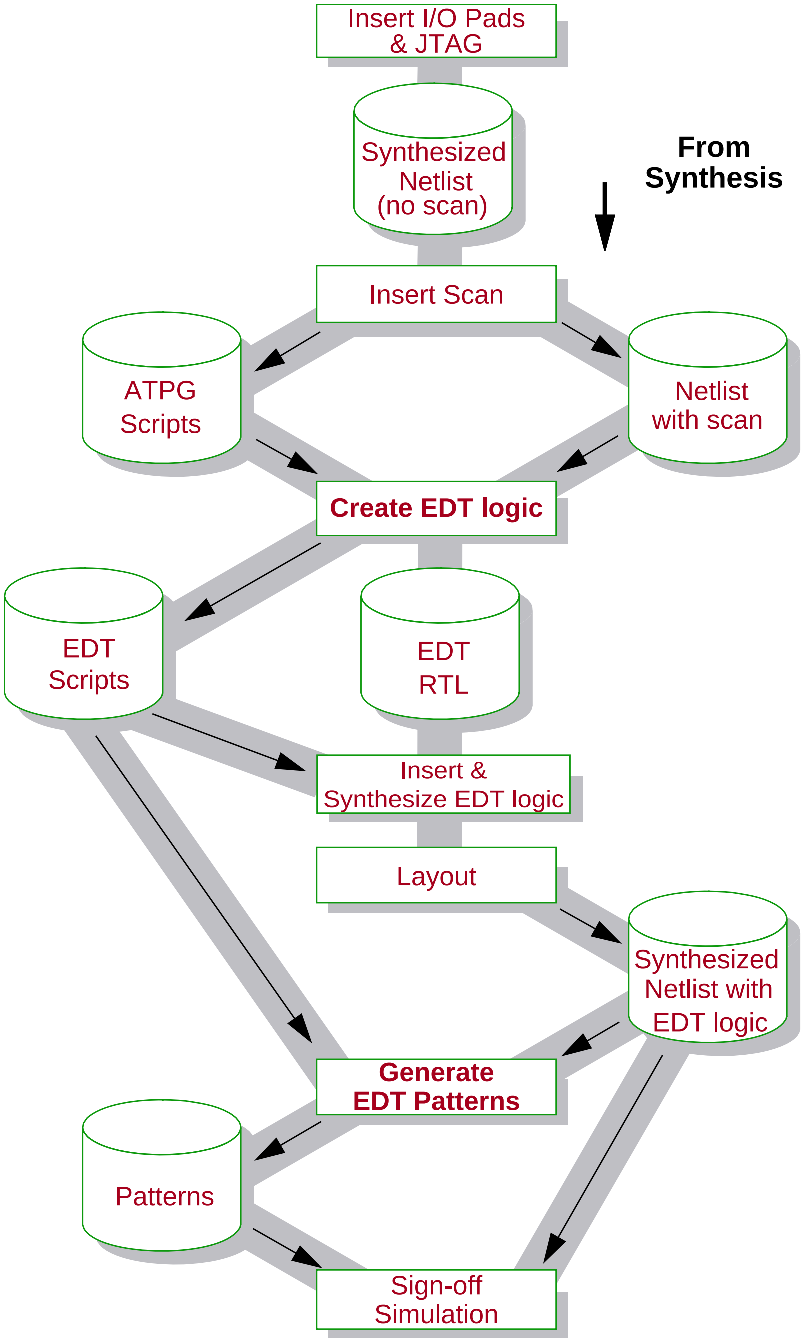
2. Insert an appropriately large number of scan chains using Tessent Scan or a third-party tool. Be sure to add new primary input and output pins for the scan chains to the top level of the design. These new pins are only temporary; the signals to which they connect will become internal nodes and the pins removed when the EDT logic is inserted into the design and connected to the scan chains. For information on how to insert scan chains using Tessent Scan, refer to "Inserting Internal Scan and Test Circuitry," in the *Tessent Scan and ATPG User's Manual*.

Note

As the new scan I/Os at the top level are only temporary, take care not to insert pads on them.

3. Perform an ATPG run on the scan-inserted design without EDT (optional). Use this run to ensure there are no basic issues such as simulation mismatches caused by an incorrect library.
4. Simulate the patterns created in step 3. (optional).
5. **EDT logic Creation Phase**: Invoke Tessent Shell with the scan-inserted gate-level description of the core. Create the RTL description of the EDT logic. The tool creates the EDT logic, inserts it into the design, and generates a Design Compiler script to synthesize the EDT logic inside the design.
6. Run the Design Compiler script to incrementally synthesize the EDT logic.
7. **EDT Pattern Generation Phase**: After you insert the EDT logic, invoke Tessent Shell with the synthesized top-level Verilog netlist and generate the EDT test patterns. You can write test patterns in a variety of formats including Verilog and WGL.
8. Simulate the compressed test patterns that you created in the preceding step. As for regular ATPG, the typical practice is to simulate all parallel patterns and a sample of serial patterns.

Figure 2-4. Compressed Pattern Internal Flow

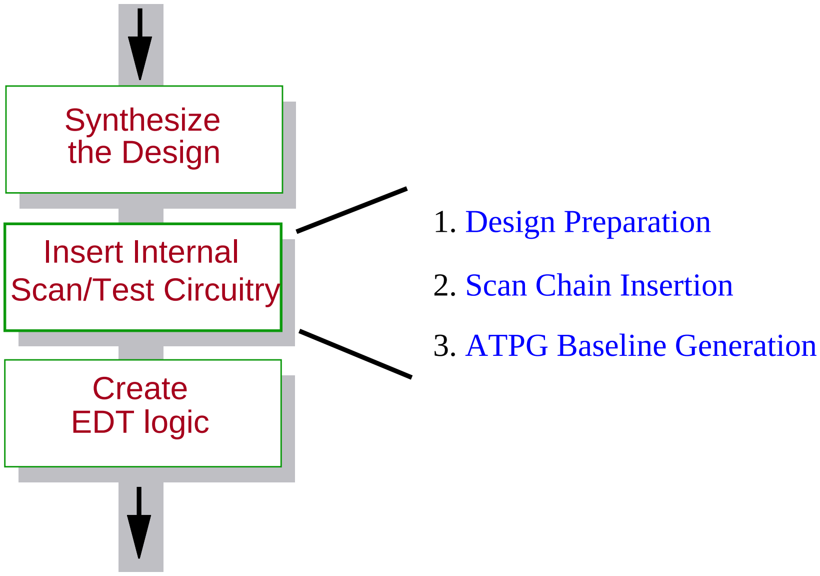


Chapter 3
Scan Chain Synthesis

For ATEs with scan options, the number of channels is usually fixed and the only variable parameter is the number of scan chains. In some cases, the chip package rather than the tester may limit the number of channels. Therefore, scan insertion and synthesis is an important part of the compressed ATPG flow.

You can use Tessent Scan or another scan insertion product to insert scan chain circuitry in your design before generating EDT logic. You can also generate the EDT logic before scan chain insertion. For more information, see the "Integrating Compression at the RTL Stage" on page 225 of this document.

Figure 3-1. Scan Chain Insertion and Synthesis Procedure



This chapter discusses the tasks related to performing scan insertion that are outlined in Figure 3-1.

Design Preparation

As a prerequisite to reading this section, you should understand the information in "Inserting Internal Scan and Test Circuitry" in the *Tessent Scan and ATPG User's Manual*. The following sub-sections assume you are familiar with that information and only cover the EDT-specific issues you need to be aware of before you insert test structures into your design.

Preparation For the External Flow

- **Managing Pre-existing I/O Pads**

  Because the synthesized hardware is added as a collar around the core design, the core should not have I/O pads when you create the EDT logic. If the design has I/O pads, you need to extract the core or remove the I/O pads.

Note
If you must insert I/O pads prior to or during initial synthesis, consider using the internal flow, which does not require you to perform the steps described in this section.

  If the core and the I/O pads are in separate blocks, removing the I/O pads is simple to do as described here:

  a. Invoke Tessent Shell and read in the design.

  b. Set the current design to the core module using the set_current_design command.

  c. Write out the core using the write_design command.

  d. Insert scan into the core and synthesize the EDT logic around it.

  e. Reinsert the EDT logic/core combination into the original circuit in place of the core you extracted, such that it is connected to the I/O pads.

  If your design flow dictates that the I/O pads be inserted prior to scan insertion, you can create a blackbox as a place holder that corresponds to the EDT block. You can then stitch the I/O pads and, subsequently, the scan chains to this block. Once the RTL model of the block is created, you use the RTL model as the new architecture or definition of the blackbox placeholder. The port names of the EDT block must match those of the blackbox already in the design, so only the architectures need to be swapped.

- **Managing Pre-existing Boundary Scan**

  If your design requires boundary scan, you must add the boundary scan circuitry outside the top-level wrapper created by Tessent Shell. The EDT logic is typically controlled by primary input pins and not by the boundary scan circuitry. In test mode, the boundary scan circuitry just needs to be reset.

Note
If you must insert boundary scan prior to or during initial synthesis, consider using the internal flow, which is intended for pre-existing boundary scan or I/O pads.

If the design already includes boundary scan, you need to extract the core or remove the boundary scan. This is the same requirement, described in the preceding section, that applies to pre-existing I/O pads. Use the procedure for managing pre-existing I/O pads in section "Preparation For the External Flow" on page 38 to do this.

Note

Boundary scan adds a level of hierarchy outside the EDT wrapper and requires you to make certain modifications to the generated dofile and test procedure file that you use for the test pattern generation.

For more complete information about including boundary scan, refer to "Boundary Scan" on page 101.

- **Synthesizing a Gate-level Version of the Design** — As a prerequisite to starting the compressed ATPG flow, you need a synthesized gate-level netlist of the core design without scan. As explained earlier, the design must not have boundary scan or I/O pads. You can synthesize the netlist using any synthesis tool and any technology.

Preparation For the Internal Flow

The EDT logic is connected between the I/O pads and the core so the core should have I/O pad cells in place for all the design I/Os. You must also add I/O pads for any EDT control and channel pins that you do not want to share with the design's functional pins.

There are three mandatory EDT control pins: edt_clock, edt_update, and edt_bypass unless you disable bypass circuitry during setup. There are *2n* channel I/Os where *n* is the number of external channels for the netlist. Refer to "EDT Control and Channel Pins" in Chapter 4 for detailed information about EDT control and channel pins.

Scan Chain Insertion

You should insert an appropriately large number of scan chains. For testers with the scan option, the number of channels is usually fixed, and the variable is the number of chains. Therefore, scan configuration is an important part of the compressed ATPG flow. Refer to the next section "Determining How Many Scan Chains to Use" for more information.

The scan chains can be connected to dedicated top-level scan pins. In designs that implement hierarchical scan insertion, the scan chains can be defined at internal pins on the block instances. In such a case, there is no need to bring these block scan chains to dedicated scan pins at the top level. For more information, see "Scan Chain Pins" on page 41.

The following limitations exist for the insertion of scan chains:

- Only scan using the mux-DFF or LSSD scan cell type (or a mixture of the two) is supported. The tool creates DFF-based EDT logic by default; however, you can direct it to create latch-based logic for pure LSSD designs. Table 1-1 on page 12 summarizes the EDT logic/scan architecture combinations the tool supports. For information about specific scan cell types, refer to “Scan Architectures” in the *Tessent Scan and ATPG User’s Manual*.

- Both prefixed and bused scan input and output pins are allowed; however, the buses for bused pins must be in either ascending or descending order (not in random order).
- Unlike uncompressed ATPG, "dummy" scan chains are not supported in compressed ATPG. This is because EDT logic is dependent on the scan configuration, particularly the number of scan chains. Uncompressed ATPG performance is independent of the scan configuration and can assume that all scan cells are configured into a single scan chain when dummy scan chains are used.

Insertion of Bypass Chains in the Netlist

Tessent Shell can generate EDT logic for netlists that contain two sets of pre-defined scan chains. This enables you to insert both the bypass chains for bypass mode and the core scan chains for compression mode into the netlist with a scan-insertion tool before the EDT logic is generated.

You can use any scan insertion tool, but you must adhere to the following rules when defining the scan chains:

- Scan chains and bypass chains must use the same I/O pins.
- If the control pin used to select bypass or compression mode is shared with the edt_bypass pin, the bypass chains must be active when the edt_bypass pin is at 1, and the scan chains must be active when the edt_bypass pin is at 0.
- Test procedure file for the EDT logic must set up the mux select, so the shortened internal scan chains can be traced.

Inserting bypass chains with a scan insertion tool ensures that lockup cells and multiplexers used for bypass mode operation are fully integrated into the design netlist to allow more effective design routing.

For more information, see “Compression Bypass Logic” on page 172.

Inclusion of Uncompressed Scan Chains

Uncompressed scan chains (scan chains not driven by or observed through EDT logic) are permitted in a design that also uses EDT logic. You can insert and synthesize them like any other scan chains, but you do not define them when creating the EDT logic.

You must define the uncompressed scan chain during test pattern generation using the add_scan_chains command without the *-Internal* switch.

You can set up uncompressed scan chains to share top-level pins by defining existing top-level pins as equivalent or physically defining multiple scan chains with the same top-level pin. For more information, see the add_scan_chains command in the *Tessent Shell Reference Manual*.

For additional information, refer to the following sections:

- "Preparation for EDT Logic Creation" on page 52
- "Test Pattern Generation Files" on page 86
- "Preparation for Test Pattern Generation" on page 109
- "Each EDT block must have a discrete set of scan chains — Scan chains cannot be shared between blocks." on page 245

Determining How Many Scan Chains to Use

Although you generally determine the number of scan chains based on the number of scan channels and the desired compression, routing congestion can create a practical limitation on the number of scan chains a design can have. With a very large number of scan chains (usually more than a thousand), you can run into problems similar to those for RAMs, where routing can be a problem if several hundred scan chains start at the decompressor and end at the compactor.

Other reasons to decrease the number of scan chains might be to limit the number of incompressible patterns and/or to reduce the pattern count. For more information, see "Effective Compression" on page 20.

For testers with a scan option, the number of channels is usually fixed and the variable you modify will be the number of chains. Because the effective compression will be slightly less than the ratio between the two numbers (the chain-to-channel ratio), in most cases it is sufficient to do an approximate configuration by using slightly more chains than indicated by the chain-to-channel ratio. How many more depends on the specific design and on your experience with the tool. For example, if the number of scan channels is 16 and you need five times (5X) effective compression, you can configure the design with 100 chains (20 more than indicated by the chain-to-channel ratio). This typically results in 4.5 to 6X compression.

Scan Groups

EDT supports the use of exactly one scan group. A scan group is a grouping of scan chains based on operation. For more information, see the "Scan Groups" section of the *Tessent Scan and ATPG User's Manual*.

Scan Chain Pins

When you perform scan insertion, you must not share any scan chain pins with functional pins. You can connect the inserted scan chains to dedicated pins you create for them at the top level.

If you use the external flow, these dedicated pins become internal nodes when the tool creates the additional wrapper. If you use the internal flow, the dedicated pins are removed when the EDT logic is instantiated in the design and connected. Therefore, using dedicated pins does not increase the number of pins needed for the chip package. To ensure the scan chains have dedicated output pins, use the -*Output New* option with the insert_test_logic command in Tessent Scan.

You can also leave the scan chains anchored to internal scan pins instead of connecting them to the top level.

Note

You can share functional pins with the external decompressor scan *channel* pins. Remember, these *channels* become the new “virtual” scan chains seen by the tester. You specify the number of channels, as well as any pin sharing, in a later step when you set up Tessent Shell for inserting the EDT logic. Refer to “EDT Control and Channel Pins” in Chapter 4 for more information.

Note

If a scan cell drives a functional output, avoid using that output as the scan pin. If that scan cell is the last cell in the chain, you must add a dedicated scan output.

About Reordered Scan Chains

The EDT logic (including bypass circuitry) depends on the clocking of the design. When necessary to prevent clock skew problems, the tool automatically includes lockup cells in the EDT logic. If, after you create the EDT logic, you reorder the scan chains incorrectly, the automatically inserted lockup cells will no longer behave correctly. The following are potential problem areas:

- Between the decompressor and the scan chains (between the EDT clock and the scan clock(s))
- Between the scan chain output and the compactor when there are pipeline stages (between the scan clock(s) and the EDT clock)
- In the bypass circuitry where the internal scan chains are concatenated (between different scan clocks)

You can avoid regenerating the EDT logic by ensuring the following are true after you reorder the scan chains:

- The first and last scan cell of each chain have the same clock and phase.

  To satisfy this condition, you should reorder within each chain and within each clock domain. If both leading edge (LE) triggered and trailing edge (TE) triggered cells exist in the same chain, do not move these two domains relative to each other. After reordering, the first and last cell in a chain do not have to be precisely the same cells that occupied those positions before reordering, but you do need to have the same clock domains (clock pin and clock phase) at the beginning and end of the scan chain.
- If you use a lockup cell at the end of each scan chain and if all scan cells are LE triggered, you do not have to preserve the clock domains at the beginning and end of each scan chain.

When all scan cells in the design are LE triggered, the lockup cell at the end of each chain enables you to reorder however you want. You can move clock domains and you can reorder across chains. But if there are both LE and TE triggered flip-flops, you must maintain the clock and edge at the beginning and end of each chain. Therefore, the effectiveness and need of the lockup cell at the end of each chain depends on the reordering flow, and whether you are using both edges of the clock.

For flows where re-creating the EDT logic is unnecessary, you still must regenerate patterns (just as for a regular ATPG flow). You should also perform serial simulation of the chain test and a few patterns to ensure there are no problems. If you include bypass circuitry in the EDT logic (the default), you should also create and serially simulate the bypass mode chain test and a few patterns.

Scan Insertion Dofile Example

The scan chains must have dedicated pins. To ensure this is the case for the outputs, you must use the insert_test_logic -*Output New* command and option in Tessent Scan (dft -scan context). The following is an example dofile for inserting scan chains with Tessent Scan. Notice the use of "-output new" (shown in bold font) and the single scan group:

```
// tscan.do
//
// Tessent Scan dofile to insert scan chains for EDT.

// Set context, read library, read and set current design, etc.
...

// Set up required scan type and methodology: mux-DFF, scan.
set_scan_type mux_scan

// Set up control signals.
add_clocks 0 clk1 clk2 clk3 clk4 ramclk

// Define test logic for lockup cells.
add_cell_models inv02 -type inv
add_cell_models latch -type dlat CLK D -active high
set_lockup_cell on

// Set up Test Control Pins.
set_scan_insertion -sen scan_en
set_scan_insertion -ten test_en

// Set up scan chain naming.
set_scan_pins Input -prefix edt_si -initial 1 -modifier 1
set_scan_pins Output -prefix edt_so -initial 1 -modifier 1

// Flatten design, run DRCs, and identify scan cells.
set_system_mode analysis
report_statistics
add_clock_groups grp1 clk1 clk2 clk3 clk4
run

// Insert scan chains and test logic.
insert_test_logic -edge merge -clock merge -number 16 -output new
```

```
// “-output new” is required to ensure separate scan chain outputs.

// Report information.
report_scan_chains
report_test_logic

// Write output files.
write_design my_gate_scan.v - verilog -replace
write_atpg_setup my_atpg -replace

exit
```

You should obtain the following outputs from Tessent Scan:

- Scan-inserted gate-level netlist of the design
- Test procedure file that describes how to operate the scan chains
- Dofile that contains the circuit setup and test structure information

ATPG Baseline Generation

You can generate an ATPG baseline after scan chain insertion.

An ATPG baseline can be used to:

- Estimate the final test coverage early in the flow, before you insert the EDT logic.
- Obtain the scan data volume for the test patterns pre-compression. You can then compare the scan data volume for test patterns before and after compression to evaluate the effects of compression.

Note

Directly comparing pattern counts is not meaningful because EDT patterns are much smaller than ATPG patterns. This is because the relatively short scan chains used in EDT require many fewer shift cycles per scan pattern.

- Provide additional help for debugging. You can simulate the patterns you generate in this step to verify that the non-EDT patterns simulate without problems.
- Find other problems, such as library errors or timing issues in the core, before you create the EDT logic.

Note

If you include bypass circuitry, you also can run regular ATPG after you insert the EDT logic.

This run is like any ATPG run and does not have any special settings; the key is using the same settings (pattern types, constraints, and so on) used to create the compressed test patterns.

The test procedure file used for this ATPG run can be identical to the one generated by scan insertion. However, it should be modified to include the same timing, specified by the tester, that is used to generate the compressed test patterns. By using the same timing information, you ensure simulation comparisons are realistic. To avoid DRC violations when you save test patterns, update the test procedure file with information for RAM clocks and for non-scan-related procedures.

Use the report_scan_volume command to report test data before and after compression and compare the data to evaluate the effect of compression.

Save the patterns if you want to simulate them. You can use any Verilog timing simulator.

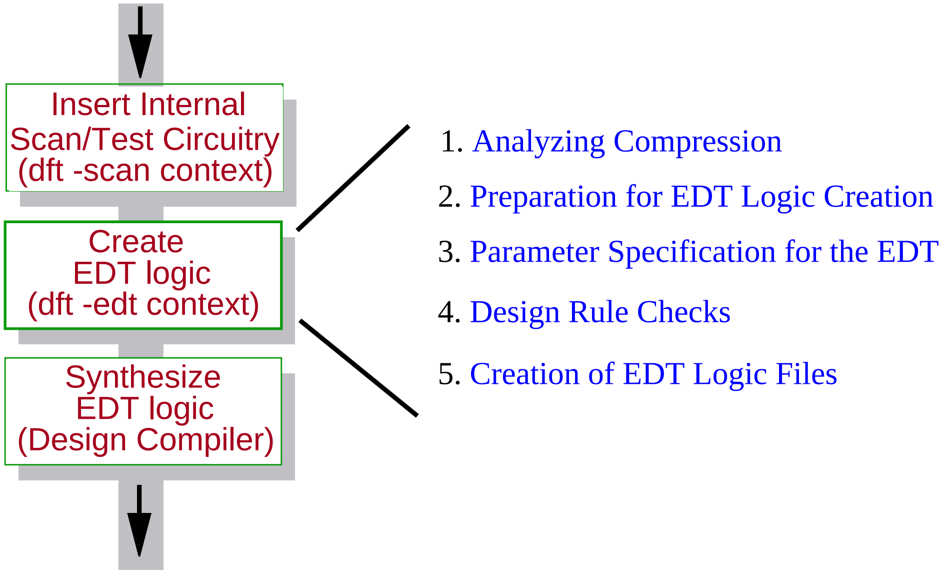
Note

This ATPG run is intended to provide test coverage and pattern volume information for traditional ATPG. Save the patterns if you want to simulate them, but be aware that they have no other purpose. The final compressed test patterns are generated and saved after the EDT logic is inserted and synthesized.

Chapter 4
Creating EDT Logic

This chapter describes how to create and insert EDT logic into a scan-inserted design. Figure 4-1 shows the layout of this chapter as it applies to the process of creating and inserting the EDT logic.

Figure 4-1. EDT Logic Creation Process



1. Analyzing Compression
2. Preparation for EDT Logic Creation
3. Parameter Specification for the EDT
4. Design Rule Checks
5. Creation of EDT Logic Files

For more information on specific commands, see the *Tessent Shell Reference Manual*.

Compression Analysis

You need to determine a scan chain to scan channel ratio (chain:channel ratio) for your application before you create the EDT logic. The chain:channel ratio determines the compression for an application. For more information, see "Effective Compression" on page 20.

Usually the number of scan channels are dictated by hardware resources such as test channels on the ATE and the top-level design pins available for test. However, you can usually vary the number of scan chains to optimize the compression for an application.

You can determine the optimal chain:channel ratio for an application by varying the number of scan channels or scan chains and then generating test patterns and evaluating the following elements:

- **Test coverage** — Determine if the test effectiveness is adequate for the application.
- **Data volume** — Determine how much test pattern data is generated after compression and whether it is within the test hardware limitations.
- **ATPG baseline (optional)** — Compare the test data statistics for the ATPG baseline with the compressed test pattern statistics. See "ATPG Baseline Generation" on page 44.

You can use the analyze_compression command to explore the effects of different chain:channel ratios on test data without making modifications to your design. For more information, see "Analyzing Compression" on page 48.

Related Topics

Effective Compression	Analyzing Compression

Analyzing Compression

Use this procedure to explore chain:channel ratios, test coverage, and test data volume for an EDT application. You can perform this procedure before or after the EDT logic is created and on block-level or chip-level architecture designs.

Note

This procedure is used for analysis only and does not permanently alter design configurations or produce any test patterns.

Prerequisites

- Scan-inserted gate-level netlist. Can be any scan chain configuration. The tool disregards the configuration if other settings are specified for the analysis. For more information, see the analyze_compression command.

- It is recommended to use the reset_state command to discard existing test patterns and restore fault population before analyzing the design.

Procedure

1. Invoke Tessent Shell on your design. The tool invokes in setup mode. For more information, see "Supported Design Format" on page 24.

   ```
   <Tessent_Tree_Path>/bin/tessent -shell
   ```

2. Provide Tessent Shell commands. For example:

   set_context patterns -scan
   read_verilog my_gate_scan.v
   read_cell_library my_lib.aptg
   set_current_design top

3. Define scan chains and add clocks using the add_scan_chains and add_clocks commands.

4. Analyze the design to determine the maximum chain:channel configurations that can be used for your design. Use this step to analyze both chip-level and block-level designs. For example:

   set_fault_type stuck
   set_fault_sampling 5
   analyze_compression

   The tool analyzes the design and returns a range of chain:channel ratio values beginning with the ratio where a negligible drop in fault coverage occurs and ending with the ratio where a 1% drop in fault coverage occurs as follows:

   ```
   // For stuck-at_faults
   //
   //  Chain:Channel Ratio  PredictedFaultCoverageDrop
   //  -------------------  -------------------------------
   //  153                  negligible fault coverage drop
   //  154                  0.01 % - 0.05 % drop
   //  160                  0.10 %
   //  168                  0.15 %
   //  171                  0.20 %
   …
   //  CPU time is 155 seconds.
   ```

   The design is analyzed for the fault type specified by the set_fault_type command before the analyze_compression command is executed. The analyze_compression command uses the current fault population. If no faults are added, the tool operates on all faults or a subset of sampled faults that are determined by the fault sampling rate specified by the "set_fault_sampling <rate>" command

For example, if you want to analyze transition faults with a 10% fault sampling rate, you would use the following commands:

```
set_fault_type transition
set_fault_sampling 10
analyze_compression
```

For more information, see the analyze_compression command.

5. Select a chain:channel ratio from the list and calculate how many scan chains and scan channels to use for your first trial run. For more information, see "Compression Analysis" on page 48.

6. Depending on the chip architecture, specify the chain:channel ratios, emulate the EDT logic, and generate test patterns as follows:

   - Emulating a virtual single block EDT configuration

     ```
     set_fault_type stuck
     analyze_compression -chains 270 -channels 9
     ```

   - Emulating a virtual modular EDT configuration
     If you are analyzing compression for a block-level design, you may need to manually determine how to allocate chains and channels across blocks to achieve the selected chain:channel ratio before you perform this step. For example:

     ```
     set_fault_sampling 80
     analyze_compression -Edt_block BLK1 -CHAINs 400 - CHANNELs 8
        -Edt_block BLK2 -CHAINs 200 -CHANNELs 4 -SHift_power_control
        -SWitching_threshold_percentage 20
     ```

   The tool emulates the EDT logic with the specified sampling rate, fault type, and chain:channel ratio, generates temporary test patterns, and displays a statistics report similar to the following:

```
               Statistics Report
                Stuck-at Faults
----------------------------------------------
Fault Classes                      #faults
                                   (total)
----------------------------  ----------------
  FU (full)                        2173901
  --------------------------  ----------------
  UC (uncontrolled)                  729 ( 0.03%)
  UO (unobserved)                  17523 ( 0.81%)
  DS (det_simulation)            1696097 (78.02%)
  DI (det_implication)            342047 (15.73%)
  PU (posdet_untestable)            1099 ( 0.05%)
  PT (posdet_testable)               633 ( 0.03%)
  UU (unused)                      12547 ( 0.58%)
  TI (tied)                        25920 ( 1.19%)
  BL (blocked)                     18120 ( 0.83%)
  RE (redundant)                   29870 ( 1.37%)
  AU (atpg_untestable)             29316 ( 1.35%)
----------------------------------------------
Untested Faults
```

```
  --------------------------
  AU (atpg_untestable)
    PC   (pin_constraints)        186 ( 0.01%)
    Unclassified                29130 ( 1.34%)
  UC+UO
    AAB  (atpg_abort)            6619 ( 0.30%)
    UNS  (unsuccess)            11633 ( 0.54%)
----------------------------------------------
Coverage
  --------------------------
  test_coverage                        97.68%
  fault_coverage                       93.79%
  atpg_effectiveness                   99.15%
----------------------------------------------
#test_patterns                           2285
  #basic_patterns                        2108
  #clock_po_patterns                        3
  #clock_sequential_patterns              174
#simulated_patterns                      4544
CPU_time (secs)                        4755.1
----------------------------------------------

Note: The reported statistics are based on a 80% fault sample.

//  CPU time to analyze_compression is 4751 seconds.
//
//  -----------------------------------------------------------
//  Scan volume report.
//  -------------------
//     channels     : 12
//     shift cycles : 145
//  -----------------------------------------------------------
//  pattern             # test  # scan                   volume
//  type              patterns   loads  (cell loads or unloads)
//  ----------------  --------  ------  -----------------------
//  setup_pattern            2       2                     3480
//  chain_test              71      71                   123540
//  basic                 2108    2108                  3667920
//  clock_po                 3       3                     5220
//  clock_sequential       174     174                   302760
//  ----------------  --------  ------  -----------------------
//  total                2358    2358                 4102920 (4.1M)
//
Power Metrics                Min.   Average   Max.
--------------------------  ------  -------  ------
WSA                          0.08%   27.39%  46.28%
State Element Transitions    0.00%   30.48%  50.67%
--------------------------  ------  -------  ------
Peak Cycle
--------------------------
WSA                          0.08%   28.26%  46.28%
State Element Transitions    0.00%   31.55%  50.67%
--------------------------  ------  -------  ------
Load Shift Transitions       7.32%   15.97%  19.91%
Response Shift Transitions   9.85%   33.69%  50.51%
```

7. Review the statistics report to determine whether the chain:channel ratio is adequate as follows:

    - If the chain:channel ratio yields adequate results, insert the scan chains and create the EDT logic. See "Scan Chain Synthesis" on page 37 and "Preparation for EDT Logic Creation" on page 52.
    - If the data volume and/or test coverage is unacceptable, repeat steps 3, 4, and 5 until you determine the optimal chain:channel ratio to use for your application.

Related Topics

analyze_compression	Compression Analysis
If Compression is Less Than Expected	If Test Coverage is Less Than Expected

Preparation for EDT Logic Creation

Depending on your application, the following subsections discuss the steps needed to prepare for creating/inserting EDT logic into your design. You can create the EDT logic immediately after you insert scan chains, or you can run traditional ATPG and simulate the resulting patterns first, as described in the "ATPG Baseline Generation" on page 44.

EDT must be on whenever you are creating test patterns or EDT logic. You can use the report_environment command to check the tool status. You can use the set_edt_options command to enable compression.

Scan Chain Definition

You must define the clocks and scan chain information. You can include these commands in a dofile or invoke the dofile that Tessent Scan generates to define clocks and scan chains. For example:

dofile my_atpg.dofile

The following shows an example setup dofile generated by Tessent Scan:

```
add_scan_groups grp1 my_atpg_setup.testproc
add_scan_chains chain1 grp1 edt_si1 edt_so1
add_scan_chains chain2 grp1 edt_si2 edt_so2
add_scan_chains chain3 grp1 edt_si3 edt_so3
...
add_scan_chains chain98 grp1 edt_si98 edt_so14
add_scan_chains chain99 grp1 edt_si99 edt_so15
add_scan_chains chain100 grp1 edt_si100 edt_so16
add_write_controls 0 ramclk
add_read_controls 0 ramclk
add_clocks 0 clk
```

These commands are explained in "Defining the Scan Data" in the *Tessent Scan and ATPG User's Manual*.

Internal Scan Chains in Tessent Shell IP Creation

You can add internal scan chains in the EDT IP creation phase (dft -edt context). Internal scan chains are scan chains where the scan input and output signals are not brought to the top level of the design and connected to top-level pins. This supports the hierarchical scan insertion flow and removes the requirement to bring core-level scan pins to the top level.

You use the add_scan_chains -internal command to define internal scan chains during IP creation as shown in the following example. Note, the K4, K9, and K10 IP creation DRCs do not apply to internal scan pins and are skipped. These DRCS will still be run for top-level scan pins.

Note

TestKompress not invoked from Tessent Shell still requires top-level scan pins during IP creation.

This example shows IP creation in a design with three EDT blocks: cpu and alu have internal scan chains, whereas TOP has top-level scan chains. Note, the scan pins for the cpu and alu blocks are defined at the respective instance pins and not brought to the top level. The scan pins for the TOP block are defined at the top level.

```
set_context dft -edt
add_clock 0 clk
add_scan_group grp1 scan_setup.testproc
set_edt_options -location internal
//
// EDT block: cpu
add_edt_block cpu
add_scan_chain -internal cpu_chain1 grp1 /cpu/scan_in1 /cpu/scan_out1
…
add_scan_chain -internal cpu_chain100 grp1 /cpu/scan_in100 \
   /cpu/scan_out100
set_edt_options -channel 5
//
// EDT block: alu
add_edt_block alu
add_scan_chain -internal alu_chain1 grp1 /alu/scan_in1 /alu/scan_out1
…
add_scan_chain -internal alu_chain60 grp1 /alu/scan_in60 /alu/scan_out60
set_edt_options -channel 3
//
// EDT block: TOP
add_edt_block TOP
add_scan_chain TOP_chain1 grp1 scan_in1 scan_out1
…
add_scan_chain TOP_chain20 grp1 scan_in20 scan_out20
set_edt_options -channel 1
//
//System mode transition - perform DRC
set_system_mode analysis
write_edt_files created -verilog -replace
```

Tessent Shell (dft -edt) supports internal scan chains during IP creation. However, non-Tessent Shell TestKompress does not. If you were to define internal scan chains during IP Creation using the following dofile commands in non-Tessent Shell TestKompress:

```
set_edt_options -location internal
add_scan_chains -internal chain1 grp1 /u1/scan_in1 /u1/scan_out1
add_scan_chains -internal chain2 grp1 /u1/scan_in2 /u1/scan_out2
   …
set_system_mode atpg
```

The tool would infer the Pattern Generation phase and would possibly fail with pattern generation DRCs like those shown here:

```
// ---------------------------------------------------------------------
// Begin EDT setup and rules checking.
// ---------------------------------------------------------------------
// Running EDT Pattern Generation Phase.
// Error: Defined pin "edt_clock" for EDT clock signal is not in design.
//    Violation safe to ignore, correct operation verified by subsequent
      DRCs. (K5-1)
// Error: Defined pin "edt_update" for EDT update signal is not in design.
//    Violation safe to ignore, correct operation verified by subsequent
      DRCs. (K5-2)
// Error: Defined pin "edt_channels_in1" for channel input 1 signal is not
      in design. (K5-3)
// Error: Defined pin "edt_channels_out1" for channel output 1 signal is
      not in design. (K5-4)
// Error: Defined pin "edt_channels_in2" for channel input 2 signal is not
      in design. (K5-5)
// Error: Defined pin "edt_channels_out2" for channel output 2 signal is
      not in design. (K5-6)
// Error: 6 defined EDT pin(s) not in design. (K5)
// EDT setup and rules checking aborted, CPU time=0.00 sec.
// Error: Rules checking unsuccessful, cannot exit SETUP mode.
```

Parameter Specification for the EDT Logic

You use the set_edt_options command to set parameters for the EDT logic. The two most important parameters are the position of the EDT logic, internal or external to the design core, and the number of scan channels.

For a basic run to create external EDT logic (the default), you only need to specify the number of channels. For example, the following command sets up external EDT logic with two input channels and two output channels:

set_edt_options -channels 2

Other parameters specify whether to create DFF-based or latch-based EDT logic and whether to include bypass circuitry in the EDT logic, lockup cells in the decompressor, and/or pipeline stages in the compactor.

By default, Tessent Shell generates:

- EDT logic external to the design core
- DFF-based EDT logic
- Lockup cells in the decompressor, compactor, and bypass logic
- An Xpress compactor without pipeline stages
- Bypass logic

For more information, see the set_edt_options command in the *Tessent Shell Reference Manual*.

The following topics describe other commonly used parameter settings:

- Dual Compression Configurations
- Defining Dual Compression Configurations
- Asymmetric Input and Output Channels
- Latch-Based EDT logic
- Pipeline Stages in the Compactor
- Compactor Type
- Longest Scan Chain Range
- EDT Logic Reset
- EDT Architecture Version
- Generating EDT Logic When Bypass Logic is Defined in the Netlist
- Specifying Hard Macros
- Pulse EDT Clock Before Scan Shift Clocks

Dual Compression Configurations

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

When two distinct configurations are defined, an additional EDT pin is generated to select the active configuration: edt_configuration. For more information on EDT pins, see "EDT Control and Channel Pins" on page 65.

Separate ATPG dofiles and procedure files are created for each configuration. A single dofile and test procedure file is generated for the bypass mode. These ATPG files are then used to generate test patterns for each configuration separately as you would with a single compression configuration.

In the modular flow, you should coordinate compression configuration usage between design groups to ensure the compression configurations are defined and set up properly for each block as follows:

- A maximum of two compression configurations can be defined for the entire design, across all EDT blocks, although the configuration parameters can be different for different EDT blocks belonging to that design.
- Channel parameters for each of the two configurations can vary from block to block.

  In the following example, blocks *b1* and *b2* have the same *config_high* configuration name but have different parameters: in b1, *config_high* has two input channels and 4 output channels parameters and, in b2, *config_high* has 1 input and 1 output channel:

  ```
  set_current_edt_block b1
  set_current_edt_configuration config_high
  set_edt_options -input 2 -output 4
  set_current_edt_configuration config_low
  set_edt_options -input 4 -output 5

  set_current_edt_block b2
  set_current_edt_configuration config_high
  set_edt_options -input 1 -output 1
  set_current_edt_configuration config_low
  set_edt_options -input 3 -output 3
  ```

- The configuration with the highest compression ratio must always have the highest compression ratio for each of the EDT blocks.
- To create a single compression configuration for a block, only define parameters for one of the compression configurations.

Limitations

- A configuration with a higher number of input channels than the other configuration must also have an equal or higher number of output channels than the other configuration. For example:

  The following configurations are valid because in each case the configuration with a higher input channel count also has an equal or higher number of output channels:

  Config1 = 4 input channels and 2 output channels
  Config2 = 2 input channels and 1 output channels

  Config1 = 2 input channels and 2 output channels
  Config2 = 4 input channels and 2 output channels

The following configurations are *not* valid because in each case the configuration with a higher input channel count has a lower output channel count:

Config1 = 4 input channels and 1 output channels
Config2 = 2 input channels and 2 output channels

Config1 = 2 input channels and 2 output channels
Config2 = 4 input channels and 1 output channels.

- The channels for the high compression configuration cannot be explicitly specified. By default, the high-compression configuration uses the first channels defined for the low-compression configuration. This applies to both input and output channels.
- Bypass mode is supported for the lowest-compression configuration only. You can define the number of bypass chains in either of the configurations as long as the specified number does not exceed the number of input/output channels of the lowest-compression configuration. For example,

```
Configuration 1 = 2 input channels and 2 output channels
Configuration 2 = 4 input channels and 4 output channels
The maximum number of bypass chains = 4
```

  For more information on bypass mode, see "Compression Bypass Logic" on page 172.
- You cannot generate test patterns during EDT logic creation to determine the test coverage. analyze_compression does not support dual compression configurations.
- The Basic compactor does not support more than one configuration. By default the tool generates logic that contains the Xpress compactor. For more information on compactors, see "Understanding Compactor Options" on page 215.
- There are no DRCs specific to dual compression configurations, so you must run DRC on each configuration in the test pattern generation phase. For more information, see "Generating Test Patterns" on page 120.

Defining Dual Compression Configurations

Use this procedure to create EDT logic with two compression configurations for a single design block.

Prerequisites

- Scan chains must be defined. For more information, see "Scan Chain Definition" on page 52.

Procedure

1. Invoke Tessent Shell. For example:

   ```
   <Tessent_Tree_Path>/bin/tessent -shell
   ```

   Tessent Shell invokes in setup mode.

2. Provide Tessent Shell commands. For example:

   ```
   set_context dft -edt
   read_verilog my_gate_scan.v
   read_cell_library my_lib.aptg
   set_current_design top
   ```

3. Define the first compression configuration. For example:

   ```
   add_edt_configurations config1
   set_edt_options -input_channels 6 -output_channels 5
   ```

4. Define the second configuration. For example:

   ```
   add_edt_configurations config2
   set_edt_options -input_channels 3 -output_channels 3
   ```

   To create a single compression configuration for a block, only define parameters for one of the compression configurations.

5. Define the remaining parameters for the EDT logic. See "Parameter Specification for the EDT Logic" on page 54.

6. Run DRC and fix any violations. See "Design Rule Checks" on page 75. You must run DRC on each configuration.

7. Generate the EDT logic. For more information, see "Creation of EDT Logic Files" on page 77. A separate dofile and procedure file is created for each configuration. The configuration name is appended to the prefix specified with the write_edt_files command:

   ```
   <filename_prefix>_<configuration_name>_edt.dofile
   <filename_prefix>_<configuration_name>_edt.testproc
   ```

Examples

The following example uses a dofile to create dual compression configurations for a single block.

```
set_context dft -edt
read_verilog my_gate_scan.v
read_cell_library my_lib.aptg
set_current_design top

// edt_ip_creation.do
//
// Dofile for EDT logic Creation Phase
// Execute setup script from Tessent Scan
dofile scan_chain_setup.dofile

// Set up EDT configurations
add_edt_configurations my_pkg_test_config
set_edt_options -channels 16
add_edt_configurations my_wafer_test_config
set_edt_options -channels 2

// Set bypass pin
set_edt_pins bypass my_bypass_pin

//set_edt_options configuration pin
set_edt_pins edt_configuration my_configuration_pin
set_system_mode analysis

// Report and write EDT logic.
report_edt_configurations -all //reports configurations for all blocks.
report_edt_pins //reports all pins including compression configuration
               // specific pins.
write_edt_files created -verilog -replace //Create dofiles and
                                          //testproc files for both the
                                          //configs and bypass mode
```

The following example shows a dofile that sets up modular EDT blocks with dual compression configurations at the top-level.

```
// Set up dual compression configurations
add_edt_configuration manufacturing_test
add_edt_blocks B1
set_edt_options -pipe 2 -channels 4
add_edt_blocks B2
set_edt_options -channels 1
add_edt_blocks B3
set_edt_options -channels 2

add edt configuration system_test
set_current_edt_block B1
set_edt_options -channels 2
set_current_edt_block B2
set_edt_options -channels 1
set_current_edt_block B3
set_edt_options -channels 1
```

```
// Set up top-level clocks and channel pins for each block
set_current_edt_block B1
add_clocks 0 clk
add_clocks 0 reset

dofile scan/atpg1.dofile_top
set_edt_pins in 1 coreA_channel_in1
set_edt_pins out  1 coreA_channel_out1
set_edt_pins in 2 coreA_channel_in2
set_edt_pins out  2 coreA_channel_out2
set_edt_pins in 3 coreA_channel_in3
set_edt_pins out  3 coreA_channel_out3
set_edt_pins in 4 coreA_channel_in4
set_edt_pins out 4  coreA_channel_out4

set_current_edt_block B2
dofile scan/atpg2.dofile2
set_edt_pins in 1 coreB_channel_in1
set_edt_pins out  1 coreB_channel_out1

set_current_edt_block B3
dofile scan/atpg3.dofile3
set_edt_pins in 1 coreC_channel_in1
set_edt_pins out  1 coreC_channel_out1
set_edt_pins in 2 coreC_channel_in2
set_edt_pins out  2 coreC_channel_out

//Run DRC
set_system_mode analysis

//Report EDT configuration and generate EDT logic
report_edt_configurations -all -verbose

write_edt_files   ./edt_ip/created1_core_top -verilog -synth dc_shell
-replace -rtl_prefix chip_level

exit -force
```

Related Topics

add_edt_configurations

delete_edt_configurations

report_edt_configurations

set_current_edt_configuration

Asymmetric Input and Output Channels

You can specify a different number of input versus output channels for the EDT logic with the -*Input_Channels* and -*Output_Channels* switches of the set_edt_options command.

Bypass Scan Chains

You can use the set_edt_options -*BYPASS_Chains* ***integer*** to specify how many bypass chains the EDT logic is configured to support. By default, the number of bypass chains created equals the number of input/output channels. If the number of input and output channels differ, the smaller number is used.

You can only specify a number of bypass chains equal to or less than the number of bypass chains created by default. For dual configuration applications, you can only specify the bypass chains after both configurations are defined.

For more information on bypass mode, see "Compression Bypass Logic" on page 172.

Latch-Based EDT logic

Tessent Shell supports mux-DFF and LSSD scan architectures, or a mixture of the two, within the same design. The tool creates DFF-based EDT logic by default. If you have a pure LSSD design and prefer the logic to be latch-based, you can use the -Clocking switch to get the tool to create latch-based EDT logic.

Compactor Type

Use the *-COMpactor_type* switch to specify which compactor is used in the generated EDT logic. By default, the Xpress compactor is used. For more information, see "Understanding Compactor Options" on page 215.

Pipeline Stages in the Compactor

The EDT logic can be set up to include pipeline stages between logic levels within the compactor. The *-PIpeline_logic_levels_in_compactor* switch allows you to specify a maximum number of logic levels (XOR gates) in a compactor before pipeline stages are inserted. By default, no pipeline stages are inserted. For more information, see "Using Pipeline Stages in the Compactor" on page 186.

Pipeline Stages Added to the Channel

When generating the EDT IP, if output channel pipeline stages will be added later, you must specify "set_edt_pins -change_edge_at_compactor_output trailing_edge" to ensure that the compactor output changes consistently on the trailing edge of the EDT clock. Output channel pipeline stages should then start with leading-edge sequential elements.

Longest Scan Chain Range

Sometimes, you may need to change the length of the scan chains in your design after generating the EDT logic. Ordinarily, you must regenerate the EDT logic when such a change alters the length of the longest scan chain.

During setup, before you generate the EDT logic, you can optionally specify a length range for the longest scan chain using the -Longest_chain_range switch. As long as any subsequent scan chain modifications do not result in the longest scan chain exceeding the boundaries of this range, you will not have to regenerate the EDT logic because of a shortening or lengthening of the longest chain.

Note

This applies only to scan chain length. Other scan chain changes, such as reordering the scan chains may require EDT logic regeneration. For more information, see "About Reordered Scan Chains" on page 42".

EDT Logic Reset

The EDT logic may optionally include an asynchronous reset signal that resets all the sequential elements in the logic. Use "-reset_signal asynchronous" with the set_edt_options command if you want the EDT logic to include this signal. If you choose to include the reset, the hardware will also include a dedicated control pin for it (named "edt_reset" by default).

EDT Architecture Version

To ensure backward compatibility between older EDT logic architectures (created with older versions of the tool) and pattern generation in the current version of the tool, use the -Ip_version switch. This enables you to specify the version of the EDT architecture the tool should expect in the design. In the EDT logic creation phase, the tool writes a dofile containing EDT-specific commands for used for ATPG. Any set_edt_options commands included in this dofile will also use this switch to specify the EDT architecture version; therefore, you usually do not need to explicitly specify this switch.

Note

The logic version is incremented only when the hardware architecture changes. If the software is updated, but the logic generated is still functionally the same, only the software version changes.

You can generate test patterns for the older EDT logic architectures, but by default, the EDT logic version is assumed to be the currently supported version.

Specifying Hard Macros

You can specify the hard macros in a design so the tool recognizes and avoids modifying them while tracing clock paths for EDT logic bypass mode. For more information on EDT logic bypass mode, see "Compression Bypass Logic" on page 172.

When one of the specified hard macros are encountered, the tool uses tap points identified from the boundary of the macro cells to drive the bypass lockup cell clocks.

In cases where localized clock gaters are used, a tap point identified for one scan cell may not be appropriate for another scan cell even when they use the same top-level clocks. So, in cases where localized clock gaters are involved, the tool routes the clock pin of each scan cell involved with bypass lockup cells to the EDT logic to avoid clock skew.

Note

This functionality does not effect the type or quantity of lockup cells inserted for bypass mode.

Limitations

- Compression must be used to insert the EDT logic in the design core before synthesis.

Prerequisites

- Tessent Shell is invoked with a design netlist containing hard macros.

Procedure

1. Set up the EDT logic to be inserted internal to the design core. For example:

```
add_clocks 0 –internal pll/clk1
add_clocks 0 –internal pll/clk2
set_edt_options –location internal
add_scan_chains chain1 grp1 scan_in1 scan_out1
add_scan_chains chain2 grp1 scan_in2 scan_out2
```

2. Set up any additional EDT logic requirements for your test application.
3. Identify each hard macro inside the design. For example:

```
set_attribute_value SCBcg1 SCBcg2 -name hard_macro -value true
```

4. Run DRC and fix any errors. For example:

```
set_system_mode analysis
```

5. Create the EDT logic RTL and insert it in the design core netlist. For example:

```
write_edt_files created -replace
```

Related Topics

- Compressed Pattern Internal Flow
- get_attribute_value_list
- write_edt_files

Pulse EDT Clock Before Scan Shift Clocks

You can set up the EDT clock to pulse before the scan chain shift clocks with the *-pulse_edt_before_shift_clocks* switch of the set_edt_options command.

By default, the EDT and scan chain shift clocks are pulsed simultaneously. Setting the EDT logic up this way makes it independent of the scan chain clocking and provides the following benefits:

- Makes creating EDT logic for a design in the RTL stage easier because scan chain clocking information is not required. For more information on creating EDT logic at the RTL stage, see "Integrating Compression at the RTL Stage" on page 225.
- Removes the need for lockup cells between scan chains and the EDT logic because correct timing is ensured by the clock sequence. Only a single lockup cell between pairs of bypass scan chains is necessary. For more information, see "Understanding Lockup Cells" on page 191.
- Simplifies clock routing because the lockup cells used for bypass scan chains are driven by the EDT clock instead of a system clocks. This eliminates the need to route system clocks to the EDT logic.

To use this functionality, the shift speed must be able to support two independent clock pulses in one shift cycle, which may increase test time.

Reporting the EDT Logic Configuration

You can report the current EDT logic configuration with the report_edt_configurations command. This command lists configuration details including the number of scan channels and logic version.

For example:

report_edt_configurations

```
// IP version:2
// External scan channels:2
// Longest chain range:600 - 700
// Bypass logic:On
// Lockup cells:On
// Clocking:edge-sensitive
```

Note

Because the report_edt_configurations command needs a flat model and DRC results to produce the most useful information, you usually use this command in other than setup mode. For an example of the command's output when issued after DRC, see "DRC when EDT Pins are Shared with Functional Pins" later in this chapter.

EDT Control and Channel Pins

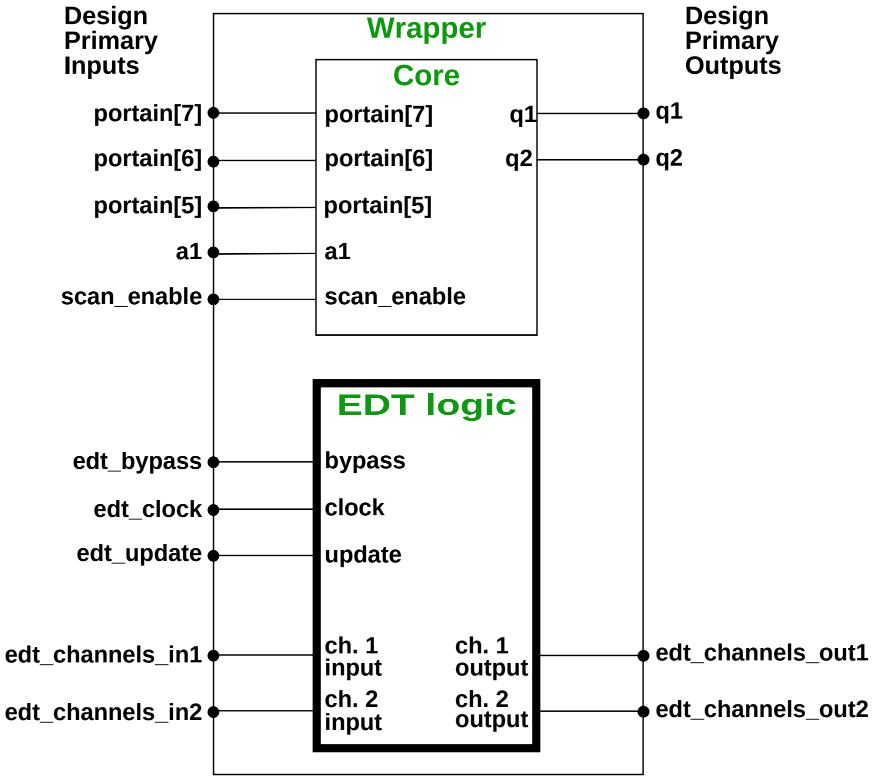
EDT logic includes both control and channel pins.

EDT logic includes the following pins:

- Scan channel input pins
- Scan channel output pins
- EDT clock
- EDT update
- Scan-enable (optional—included when any scan channel output pins are shared with functional pins)
- Bypass mode control
- Reset control (optional—included when you specify an asynchronous reset for the EDT logic)
- EDT_configuration (optional—included when you specify multiple configurations)

Figure 4-2 shows the basic configuration of these pins for an example design when the EDT logic is instantiated externally and configured with bypass circuitry and two scan channels. External EDT logic is always instantiated in a top-level EDT wrapper.

Figure 4-2. Default EDT Logic Pin Configuration with Two Channels



The default configuration consists of pins for the EDT clock, update, and bypass inputs. There are also two additional pins (one input and one output) for each scan channel. If you do not rename an EDT pin or share it with a functional pin, as described in the "Functional/EDT Pin Sharing" section, the tool assigns the default EDT pin names shown.

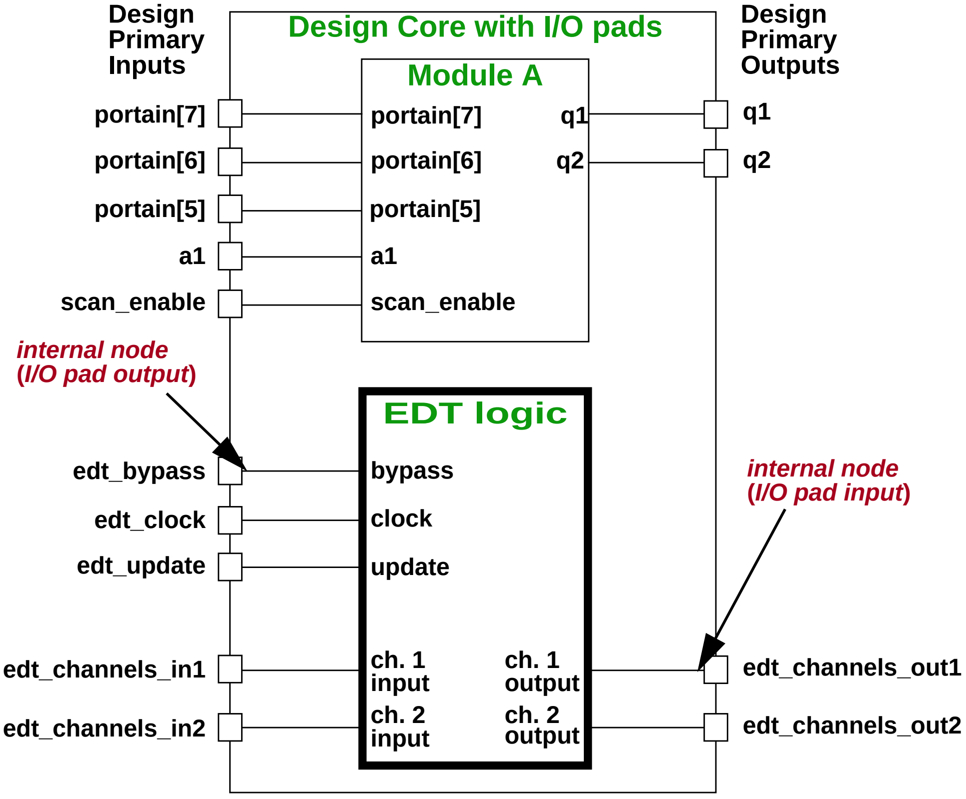
To see the names of the EDT pins, issue the report_edt_pins command:

report_edt_pins

```
//     Pin description          Pin name              Inversion
//     ---------------          --------              ---------
//     Clock                    edt_clock                 -
//     Update                   edt_update                -
//     Bypass mode              edt_bypass                -
//     Scan channel 1 input     edt_channels_in1          -
//      "       "     " output  edt_channels_out1         -
//     Scan channel 2 input     edt_channels_in2          -
//      "       "     " output  edt_channels_out2         -
```

Figure 4-3 shows how the preceding pin configuration looks if the EDT logic is inserted into a design netlist that includes I/O pads (internal EDT logic location). Notice that the EDT control and channel I/O pins are now connected to internal nodes of I/O pads that are part of the core design. You set up these connections by specifying an internal node for each EDT control and channel I/O pin. For more information, see Connections for EDT Pins (Internal Flow only).

Figure 4-3. Example of a Basic EDT Pin Configuration (Internal EDT Logic)



Functional/EDT Pin Sharing

EDT pins can be shared with functional pins, with a few restrictions. You use the set_edt_pins command to specify sharing of an EDT pin with a functional pin and to specify whether a signal is inverted in the I/O pad for the pin. For more information, see the set_edt_pins command.

When you share a channel output pin with a functional pin, the tool inserts a multiplexer before the output pin. This multiplexer is controlled by the scan_enable signal, and you must define the scan_enable signal with the set_edt_pins command. If you do not define the scan_enable signal, the tool defaults to “scan_en”, and adds this pin if it does not exist. During DRC, all added pins are reported with K13 DRC messages. You can report the exact names of added pins using the report_drc_rules command.

For channel input pins and control pins, you use the -Inv switch to specify (on a per pin basis) if a signal inversion occurs between the chip input pin and the input to the EDT logic. For example, if an I/O pad you intend to use for a channel pin inverts the signal, you must specify the inversion when creating the EDT logic. The tool requires the pin inversion information, so the generated test procedure file operates correctly with the full netlist for test pattern generation.

If bypass circuitry is implemented, you need to force the bypass control signal to enable or disable bypass mode. When you generate compressed EDT patterns, you disable bypass mode by setting the control signal to the off state. When you generate regular ATPG patterns for example, you must enable the bypass mode by setting the bypass control signal to the on state. The logic level associated with the on or off state depends on whether you specify to invert the signal. The bypass control pin is forced in the automatically generated test procedure.

In all cases, EDT pins shared with bidirectional pins must have the output enable signal configured so that the pin has the correct direction during scan. The following list describes the circumstances under which the EDT pins *can* be shared.

- **Scan channel input pin** — No restrictions.
- **Scan channel output pin** — Cannot be shared with a pin that is bidirectional or tri-state at the *core* level. This is because the tool includes a multiplexer between the compactor and the output pad when a channel output pin is shared, and tri-state values cannot pass through the multiplexer. A scan channel output pin that later will be connected to a pad and is bidirectional at the top level is allowed.

Note
Scan channel output pins that are bidirectional need to be forced to Z at the beginning of the load_unload procedure. Otherwise, the tool is likely to issue a K20 or K22 rule violation during DRC, without indicating the reason.

- **EDT clock** — Must be defined as a clock and constrained to its defined off state. If shared with a bit of a bus, problems can occur during synthesis. For example, Design Compiler (DC) does not accept a bit of a bus being a clock. The EDT clock pin must only be shared with a non-clock pin that does not disturb scan cells; otherwise, the scan cells will be disturbed during the load_unload procedure when the EDT clock is pulsed. This restriction might cause some reduced coverage. You should use a dedicated pin for the EDT clock or share the EDT clock pin only with a functional pin that controls a small amount of logic. If any loss of coverage is not acceptable, then you must use a dedicated pin.

- **EDT reset** — Should be defined as a clock and constrained to its defined off state. If shared with a bit of a bus, problems can occur during synthesis. For example, DC does not accept a bit of a bus being a clock. The EDT reset pin must only be shared with a non-clock pin that does not disturb scan cells. This restriction might cause some reduced coverage. You should use a dedicated pin for the EDT reset, or share the EDT reset pin only with a functional pin that controls a small amount of logic. If any loss of coverage is not acceptable, then you must use a dedicated pin.
- **EDT update** — Can be shared with any non-clock pin. Because the EDT update pin is not constrained, sharing it has no impact on test coverage.
- **Scan enable** — As for regular ATPG, this pin must be dedicated in test mode; otherwise, there are no additional limitations. EDT only uses it when you share channel output pins. Because it is not constrained, sharing it has no impact on test coverage.
- **Bypass** (optional) — Must be forced during scan (forced on in the bypass test procedures and forced off in the EDT test procedures). It is not constrained, so sharing it has no impact on test coverage. For more information on bypass mode, see "Compression Bypass Logic" on page 172.
- **Edt_configuration** (optional) — The value corresponding with the selected configuration must be forced on during scan chain shifting.

Note

RTL generation allows sharing of control pins. The preceding restrictions ensure the EDT logic operates correctly and with only negligible loss, if any, of test coverage.

Shared Pin Configuration

The synthesis methodology does not change when you specify pin sharing. You do, however, need to add a step to the EDT logic creation phase. In this extra step, you define how pins are shared.

For example, you are using the external flow with two scan channels and you want to share three of the channel pins, as well as the EDT update and EDT clock pins, with functional pins. Assume the functional pins have the names shown in Table 4-1.

Table 4-1. Example Pin Sharing

EDT Pin Description	Functional Pin Name
Input 1 (Channel 1 input)	portain[7]

Table 4-1. Example Pin Sharing

EDT Pin Description	Functional Pin Name
Output 1 (Channel 1 output)	edt_channels_out1 (new pin, default name)
Input 2 (Channel 2 input)	portain[6]
Output 2 (Channel 2 output)	q2
Update	portain[5]
Clock	a1
Bypass	my_bypass (new pin, non-default name)

You can see the names of the EDT pins, prior to setting up the shared pins, by issuing the report_edt_pins command:

report_edt_pins

```
//    Pin description          Pin name            Inversion
//    ---------------          --------            ---------
//    Clock                    edt_clock               -
//    Update                   edt_update              -
//    Bypass mode              edt_bypass              -
//    Scan channel 1 input     edt_channels_in1        -
//     "       "     " output  edt_channels_out1       -
//    Scan channel 2 input     edt_channels_in2        -
//     "       "     " output  edt_channels_out2       -
```

You can use the set_edt_pins command to specify the functional pin to share with each EDT pin. With this command, you can specify to tap an EDT pin from an existing core pin. You can also use the command to change the name of the new pin the tool creates for each dedicated EDT pin. Figure 4-4 on page 71 illustrates both of these cases conceptually.

Note

In the external flow, the specified pin sharing is implemented in the wrapper generated when the EDT logic is created. The "Top-level Wrapper" section contains additional information about this wrapper. In the internal flow, the pin sharing is implemented when you create and insert the EDT logic into the design before synthesis.

If a specified pin already exists in the core, the tool shares the EDT signal with that pin. Figure 4-4 shows an example of this for the EDT clock signal. The command "set_edt_options clock a1", will cause the tool to share the EDT clock with the a1 pin instead of creating a dedicated pin for the EDT clock. If you specify a pin name that does not exist in the core, a dedicated EDT pin with the specified name is created. For example, "set_edt_pins bypass my_bypass" will cause the tool to create the new pin my_bypass and connect it to the EDT bypass pin.

For each EDT pin you do not share or rename using the set_edt_pins command, if its default name is unique, the tool creates a dedicated pin with the default name. If the default name is the same as a core pin name, the tool automatically shares the EDT pin with that core pin. Table 4-2 lists the default EDT pin names.

Table 4-2. Default EDT Pin Names

EDT Pin Description	Default Name
Clock	edt_clock
Reset	edt_reset
Update	edt_update
Scan Enable	scan_en
Bypass mode	edt_bypass
Scan Channel Input	“edt_channels_in” followed by the index number of the channel
Scan Channel Output	“edt_channels_out” followed by the index number of the channel
EDT configuration select	edt_configuration

When you share a pin between an EDT channel output and a core output, the tool includes a multiplexer in the circuit together with the EDT logic, but in a separate module at the top level. An example is shown in red in Figure 4-4 for the shared EDT channel output 2 signal, and the core output signal q2. As previously mentioned, the multiplexer is controlled by the defined scan enable pin. If a scan enable pin is not defined, the tool adds one with the EDT default name, “scan_en.” Here are the commands that would establish the example pin sharing shown in Table 4-1:

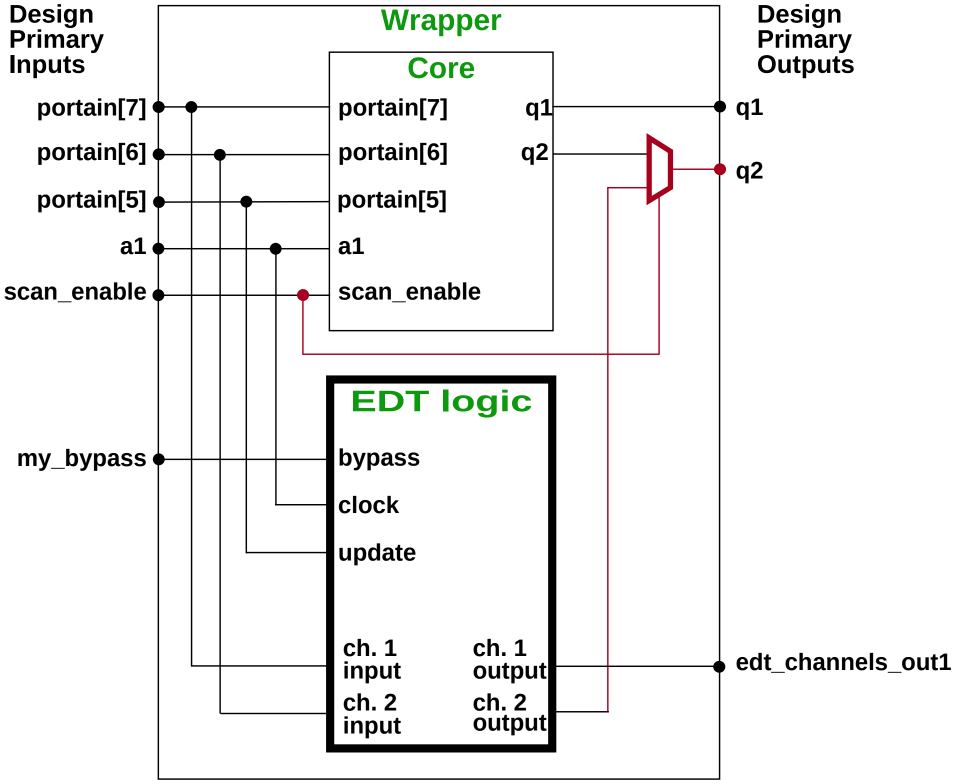
```
set_edt_pins input 1 portain[7]
set_edt_pins input 2 portain[6]
set_edt_pins output 2 q2
set_edt_pins update portain[5]
set_edt_pins clock a1
set_edt_pins bypass my_bypass
```

If you report the EDT pins using the “report_edt_pins” command after issuing the preceding commands, you will see that the shared EDT pins have the same name as the functional core pins. You will also see, for each pin, whether the pin’s signal was specified as inverted. Notice that the listing now includes the scan enable pin because of the shared EDT output pin:

```
report_edt_pins
```

```
//    Pin description           Pin name            Inversion
//    ---------------           --------            ---------
//    Clock                     a1                      -
//    Update                    portain[5]              -
//    Scan enable               scan_enable             -
//    Bypass mode               my_bypass               -
//    Scan channel 1 input      portain[7]              -
//     "       "    " output    edt_channels_out1       -
//    Scan channel 2 input      portain[6]              -
//     "       "    " output    q2                      -
```

Figure 4-4. Example with Pin Sharing Shown in Table 4-1(External EDT Logic)



After DRC, you can use the *report_drc_rules k13* command to report the pins added to the top level of the design to implement the EDT logic.

report_drc_rules k13

```
//   Pin my_bypass will be added to the EDT wrapper. (K13-2)
//   Pin edt_channels_out1 will be added to the EDT wrapper.
        (K13-3)
```

Connections for EDT Pins (Internal Flow only)

For the internal flow, you must specify the name of each internal node (instance pin name) to connect each EDT control and channel pin. For more information, see the set_edt_pins command.

Note

Before specifying internal nodes, you must specify internal logic placement with the set_edt_options -*location* internal command.

For every EDT pin, you should provide the name of a design pin and the corresponding instance pin name for the internal node that corresponds to it. The latter is the input (or output) of an I/O pad cell where you want the tool to connect the output (or input) of the EDT logic. For example:

set_edt_pins clock pi_edt_clock edt_clock_pad/po_edt_clock

The first argument “clock” is the description of the EDT pin; in this case the EDT clock pin. The second argument “pi_edt_clock” is the name of the top-level design pin on the I/O pad instance. The last argument is the instance pin name of the internal node of the pad. The pad instance is “edt_clock_pad” and the internal pin on that instance is “po_edt_clock”.

If you specify only one of the pin names, the tool treats it as the I/O pad pin name. If you specify an I/O pad pin name, but not a corresponding internal node name, the EDT logic is connected directly to the top-level pin, ignoring the pad. This may result in undesirable behavior.

If you do not specify either pin name, and the tool does not find a pin at the top level by the default name, it adds a new port for the EDT pin at the top level of the design. You will need to add a pad later that corresponds to that port.

For the internal flow, the report_edt_pins command lists the names of the internal nodes the EDT pins are connected. For example (note that the pin inversion column is omitted for clarity):

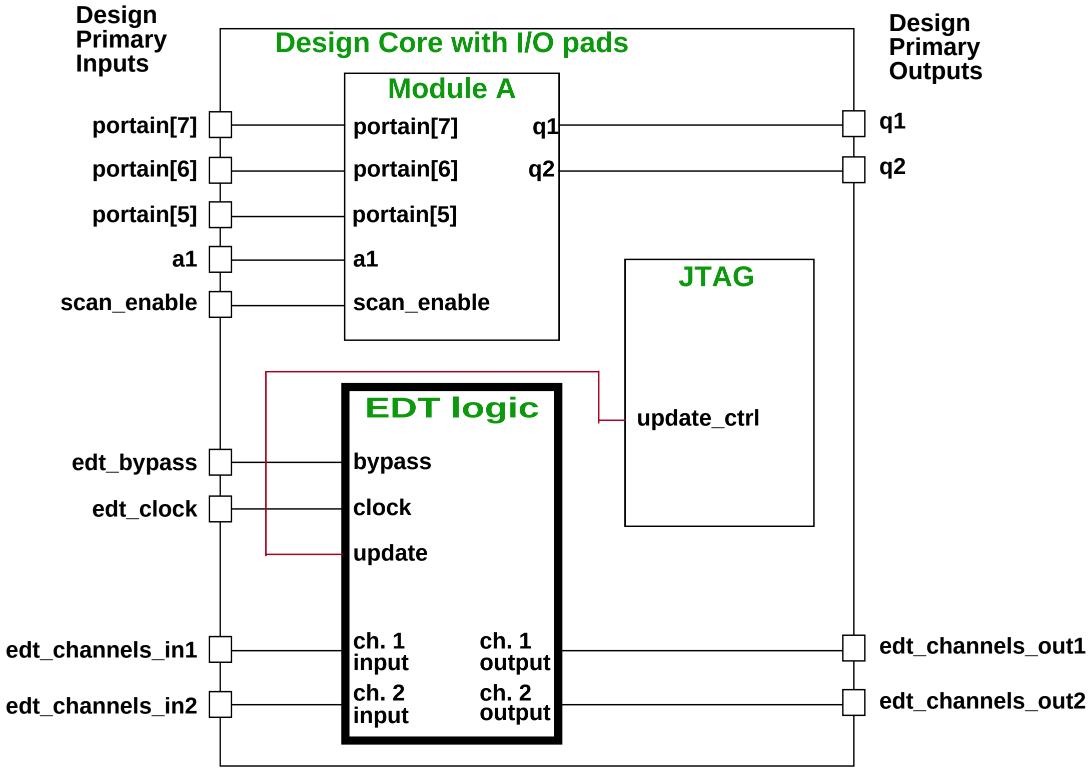
report_edt_pins

```
//
//    Pin description          Pin name           Internal connection
//    ---------------          --------           -------------------
//    Clock                    edt_clock          edt_clock_pad/Z
//    Update                   edt_update         edt_update_pad/Z
//    Bypass mode              edt_bypass         edt_bypass_pad/Z
//    Scan ch... 1 input       edt_ch..._in1      channels_in1_pad/Z
//     "    "    " output      edt_ch..._out1     channels_out1_pad/Z
//    Scan ch... 2 input       edt_ch..._in2      channels_in2_pad/Z
//     "    "    " output      edt_ch..._out2     channels_out2_pad/Z
//
```

Internally Driven EDT Pins

When an EDT control pin is driven internally by JTAG or other control registers (Figure 4-5), you should use the set_edt_pins command to specify that no top-level pin exists for the control pin.

Figure 4-5. Internally Driven edt_update Control Pin



Specifying these types of pins prevents false K5 DRC violations. You should specify internally driven pins in one of the following ways:

- **EDT logic creation**

  a. Specify the internal node that drives the control pin during logic creation. For example:

  ```
  set_edt_options -location internal
  set_edt_pins update - JTAG/update_ctrl
  set_system_mode analysis
  write_edt_files my_design -verilog -replace
  ```

  Where *JTAG/update_ctrl* is the internal node driving the *update* control pin.

  b. Edit the test procedure file to include any procedures or pin constraints needed to drive the specified internal node (*JTAG/update_ctrl*) to the correct value.

- **Pattern generation**

  a. Specify the internally driven control pin has no top-level pin during test pattern generation. For example:

  ```
  set_edt_pins update -
  set_system_mode analysis
  add_faults /my_design
  create_patterns
  ```

Note

All input and output channels must have a corresponding top-level pin.

Structure of the Bypass Chains

When bypass logic is generated, the connections for each bypass chain are automatically configured. These interconnections are fine for most designs.

However, you can specify custom chain connections with the set_bypass_chains command. For more information, see "Compression Bypass Logic" on page 172.

Decompressor and Compactor Connections

After you specify the number of scan channels in the EDT logic (see "Parameter Specification for the EDT Logic"), the tool automatically determines which scan chain outputs to compact into each channel output.

You can modify the tool's default connections using one of the following methods:

Note

Redefining compactor connections for a channel that has already been defined overwrites the previous settings for that channel.

- Reorder the add_scan_chains commands — When generating the EDT IP, the tool uses the sequence of add_scan_chains commands to connect the EDT hardware to the scan chains. You can change the order of the add_scan_chains commands in your dofile to change how they are connected to the decompressor and compactor. Note, this method changes both the decompressor and compactor connections for a particular chain.
- Specify new connections using the set_compactor_connections command — You can use the set_compactor_connections command to override the tool's default connections and explicitly define the connections between scan chains and compactor. This method allows you to change the compactor connections without changing the default decompressor connections to those chains.

  If you have dual configurations, you can still define the compactor connections using the set_compactor_connections command but only for the configuration that uses all scan channels as shown in the following example. (set_compactor_connections is not tied to a specific configuration since you only need to define connections once for each channel.)

```
set_current_edt_configuration config_high
set_edt_options -input 1 -output 1

set_current_edt_configuration config_low
set_edt_options -input 3 -output 3

set_compactor_connections -channel 1 -chains ...
set_compactor_connections -channel 2 -chains ...
set_compactor_connections -channel 3 -chains ...
```

Design Rule Checks

DRC is performed automatically when you leave setup mode by issuing the "set_system_mode analysis" command.

Tessent Shell provides a class of EDT-specific "K" rules. The section, "EDT Rules (K Rules)" in the *Tessent Shell Reference Manual* provides reference information on each EDT-specific rule.

Notice the DRC message describing the EDT rules in the following example transcript. This transcript is for the design with two scan channels shown in Figure 4-2, in which none of the EDT pins are shared with functional pins:

```
// ------------------------------------------------------
// Begin EDT rules checking.
// ------------------------------------------------------
// Running EDT logic Creation Phase.
// 7 pin(s) will be added to the EDT wrapper. (K13)
// EDT rules checking completed, CPU time=0.01 sec.
// All scan input pins were forced to TIE-X.
// All scan output pins were masked.
// ----------------------------------------------------------
```

These messages indicate the tool will add seven pins, which include scan channel pins, to the top level of the design. The last two messages refer to pins at both ends of the core-level scan chains. Because these pins are not connected to the top-level wrapper (external flow) or the top level of the design (internal flow), the tool does not directly control or observe them in the capture cycle when generating test patterns.

To ensure values are not assigned to the internal scan input pins during the capture cycle, the tool automatically constrains all internal scan chain inputs to X (hence, the "TIE-X" message). Similarly, the tool masks faults that propagate to the scan chain output nodes. This ensures a fault is not counted as observed until it propagates through the compactor logic. The tool only adds constraints on scan chain inputs and outputs added within the tool as PIs and POs.

Note

To properly configure the internal scan chain inputs and outputs so that the tool can constrain them as needed, you must use the -Internal switch with the add_scan_chains command when setting up for pattern generation in the Pattern Generation phase. This switch does, on internal scan nodes, the same thing that the add_primary_inputs and add_primary_outputs commands do on non-EDT designs.

DRC when EDT Pins are Shared with Functional Pins

If you specified to share any EDT pin with a functional pin, DRC will include messages for K rules affected by the sharing. Here is DRC output for the design shown in Figure 4-2, after it is re-configured to share certain EDT pins with functional pins, as illustrated in Figure 4-4:

```
// ----------------------------------------------------------
// Begin EDT rules checking.
// ----------------------------------------------------------
// Running EDT logic Creation Phase.
// Warning: 1 EDT clock pin(s) drive functional logic. May
      lower test coverage when pin(s) are constrained. (K12)
// 2 pin(s) will be added to the EDT wrapper. (K13)
// EDT rules checking completed, CPU time=0.00 sec.
// All scan input pins were forced to TIE-X.
// All scan output pins were masked.
// ----------------------------------------------------------
```

Notice only two EDT pins are added, as opposed to seven pins before pin sharing. Shared pins can create a test situation in which a pin constraint might reduce test coverage. The K12 warning about the shared EDT clock pin points this out to you. For details, refer to "Functional/EDT Pin Sharing" on page 67.

If you report the current configuration with the report_edt_configurations command after DRC, you will get more useful information. For example:

report_edt_configurations

```
// IP version:                 1
// Shift cycles:               381, 373 (internal scan length)
                               + 8 (additional cycles)
// External scan channels:     2
// Internal scan chains:       16
// Masking registers:          1
// Decompressor size:          32
// Scan cells:                 5970
// Bypass logic:               On
// Lockup Cells:               On
// Clocking:                   edge-sensitive
// Compactor pipelining:       Off
```

Notice that the number of shift cycles (381 in this example) is more than the length of the longest chain. This is because the EDT logic requires additional cycles to set up the decompressor for each EDT pattern (eight in this example). The number of extra cycles is dependent on the EDT logic and the scan configuration.

Creation of EDT Logic Files

By default, the tool writes out the RTL files in the same format as the original netlist.

Once you have specified the EDT logic parameters, you use the write_edt_files command to create the files that make up the EDT logic. For example:

```
write_edt_files created -replace
```

Where *created* is the name string prepended to the files and *-replace* is a switch that allows the tool to overwrite any existing files with the same name.

Depending on the EDT logic placement, the following EDT logic files are created:

- **created_edt_top.v** (external EDT logic only) — Top-level wrapper that instantiates the core, EDT logic circuitry, and channel output sharing multiplexers.
- **created_edt_top_rtl.v** (internal EDT logic only) — Core netlist with an instance of the EDT logic connected between I/O pads and internal scan chains but without a gate-level description of the EDT logic.
- **created_edt.v** — EDT logic description in RTL.
- **created_core_blackbox.v** (external EDT logic only) — Blackbox description of the core for synthesis.
- **created_dc_script.scr** — DC synthesis script for the EDT logic.
- **created_rtlc_script.scr** — RTL Compiler synthesis script for the EDT logic.
- **created_edt.dofile** — Dofile for test pattern generation.
- **created_edt.testproc** — Test procedure file for test pattern generation.
- **created_bypass.dofile** — Dofile for uncompressed test patterns (bypass mode)
- **created_bypass.testproc** — Test procedure file for uncompressed test patterns (bypass mode)

You can also use the write_edt_files command to create IJTAG files that describe the static configuration inputs of the TestKompress IP. These static configuration inputs set, enable, or disable certain features of the TestKompress IP: edt bypass, single chain bypass, low power, and edt configuration. For example:

```
write_edt_files created -IJTAG data_ports -replace
```

For details on how to use the IJTAG files for TestKompress ATPG, see“ Setting up EDT IP for IJTAG Integration” in the *Tessent IJTAG User’s Manual*.

Specification of Module/Instance Names

By default, the tool prepends the name of the top module in the associated netlist to the names of modules/instances in the generated EDT logic files. This ensures that internal names are unique, as long as all module names are unique.

If necessary, you can specify the prefix used for internal modules/instance names in the EDT logic with the write_edt_files *-rtl_prefix prefix_string* command. For example:

```
write_edt_files... -rtl_prefix core1
```

All internal module/instance names are prepended with *core1* instead of the top module name.

Note

The specified string must follow the standard rules for Verilog or VHDL identifiers.

The EDT Logic Files

This section describes the files generated for the EDT logic.

The structure of the logic described in the tool-generated files depends on the following:

- Location of the EDT logic (internal or external with respect to the design netlist)
- Number of external scan channels
- Number of internal scan chains and the length of the longest chain
- Clocking of the first and last scan cell in every chain (if lockup cells are inserted)
- Names of the pins

Except for the clocking of the scan chain boundaries, which affects the insertion of lockup cells, nothing in the EDT logic is dependant on the functionality of the core logic.

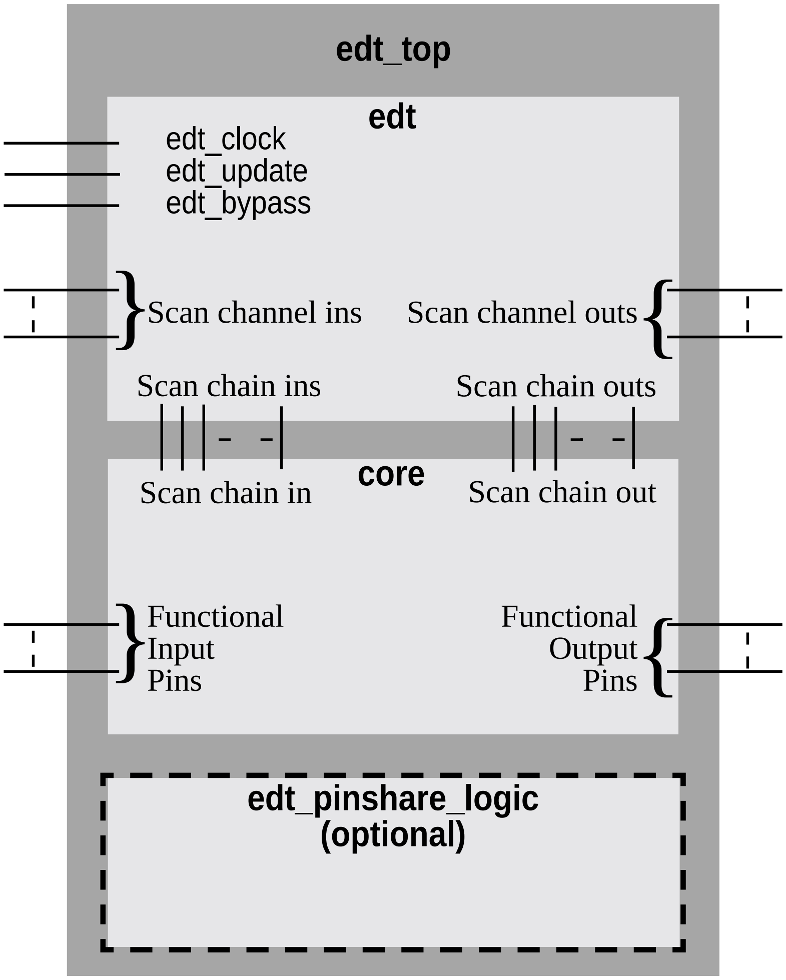
Note

Generally, you must regenerate the EDT logic if you reorder the scan chains and the clocking of the first and last scan cell or the scan chain length is affected. See “About Reordered Scan Chains” on page 42.

Top-level Wrapper

Figure 4-6 illustrates the contents of the new top-level netlist file, *created_edt_top.v*. The tool generates this file only if you are using the external flow.

Figure 4-6. Contents of the Top-Level Wrapper



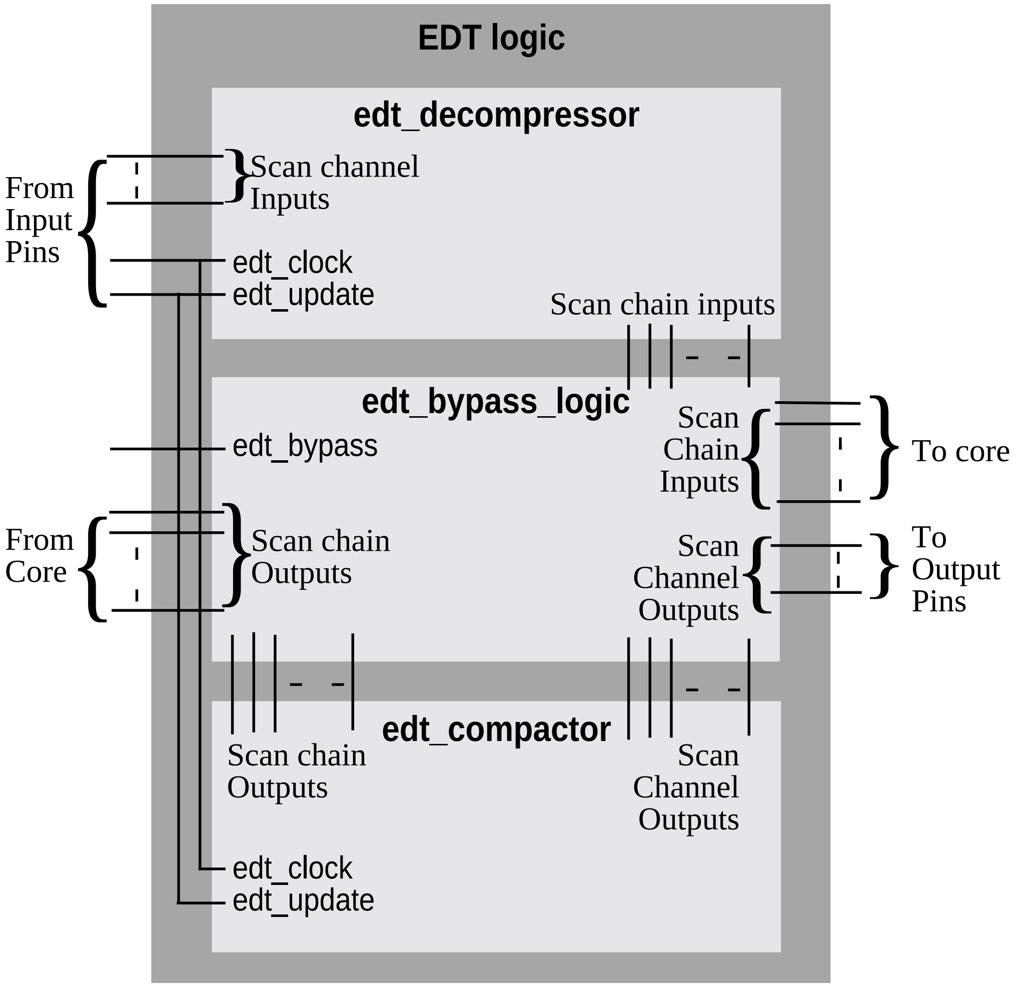
This netlist contains a module, “edt_top”, that instantiates your original core netlist and an “edt” module that instantiates the EDT logic circuitry. If any EDT pins are shared with functional pins, “edt_top” instantiates an additional module called “edt_pinshare_logic” (shown as the optional block in Figure 4-6). The EDT pins and all functional pins in the core are connected to the wrapper. Scan chain pins are not connected because they are driven and observed by the EDT block.

Because scan chain pins in the core are only connected to the “edt” block, these pins must not be shared with functional pins. For more information, refer to “Scan Chain Pins” on page 41. Scan channel pin sharing (or renaming) that you specified using the set_edt_pins command is implemented in the top-level wrapper. This is discussed in detail in “Functional/EDT Pin Sharing,” earlier in the chapter.

EDT Logic Circuitry

Figure 4-7 shows a conceptual view of the contents of the EDT logic file, *created_edt.v*.

Figure 4-7. Contents of the EDT Logic



The EDT logic file contains the top-level module and three main blocks:

- **decompressor** — connected between the scan channel input pins and the internal scan chain inputs
- **compactor** — connected between the internal scan chain outputs and the scan channel output pins
- **bypass logic** — connected between the EDT logic and the design core. Bypass logic is optional but generated by default.

Core

Generated only when the EDT logic is inserted external to the design core, the file *created_core_blackbox.v* contains a black-box description of the core netlist. This can be used when synthesizing the EDT block so the entire core netlist does not need to be loaded into the synthesis tool.

Note

Loading the entire design is advantageous in some cases as it helps optimize the timing during synthesis.

Design Compiler Synthesis Script External Flow

The tool generates a Design Compiler (DC) synthesis script, *created_dc_script.scr*. By default, the script is in Tool Command Language (TCL), but you can get the tool to write it in DC command language (dcsh) by including a “*-synthesis_script dc_shell*” argument with the write_edt_files command.

The following is an example script, in the default TCL format, for a core design that contains a top-level Verilog module named “cpu”:

```
#************************************************************************
#  Synopsys Design Compiler synthesis script for created_edt_top.v
#
#************************************************************************

# Read input design files
read_file -f verilog created_core_blackbox.v
read_file -f verilog created_edt.v
read_file -f verilog created_edt_top.v

current_design cpu_edt_top

# Check design for inconsistencies
check_design

# Timing specification
create_clock -period 10 -waveform {0 5} edt_clock

# Avoid clock buffering during synthesis. However, remember
# to perform clock tree synthesis later for edt_clock
set_clock_transition 0.0 edt_clock
set_dont_touch_network edt_clock

# Avoid assign statements in the synthesized netlist.
set_fix_multiple_port_nets -feedthroughs -outputs -buffer_constants

# Compile design
uniquify
set_dont_touch cpu
compile -map_effort medium

# Report design results for EDT logic
report_area > created_dc_script_report.out
report_constraint -all_violators -verbose >> created_dc_script_report.out
report_timing -path full -delay max >> created_dc_script_report.out
report_reference >> created_dc_script_report.out
```

```
# Remove top-level module
remove_design cpu

# Read in the original core netlist
read_file -f verilog gate_scan.v
current_design cpu_edt_top
link

# Write output netlist
write -f verilog -hierarchy -o created_edt_top_gate.v
```

Design Compiler Synthesis Script for Internal Flow

The tool generates a Design Compiler (DC) synthesis script *created_dc_script.scr* that synthesizes the EDT logic in the core netlist for the internal flow as shown in the following example.

```
#*************************************************************************
#  Synopsys Design Compiler synthesis script for config1_edt.v
#  Tessent TestKompress version: v8.2009_3.10-prerelease
#  Date:                 Thu Aug  6 01:44:15 2009
#*************************************************************************

# Bus naming style for Verilog
set bus_naming_style {%s[%d]}

# Read input design files
read_file -f verilog results/config1_edt.v

# Synthesize EDT IP
current_design circle_edt

# Check design for inconsistencies
check_design

# Timing specification
create_clock -period 10 -waveform {0 5} edt_clock

# Avoid clock buffering during synthesis. However, remember
# to perform clock tree synthesis later for edt_clock
set_clock_transition 0.0 edt_clock
set_dont_touch_network edt_clock

# Avoid assign statements in the synthesized netlist.
set_fix_multiple_port_nets -feedthroughs -outputs -buffer_constants

# Compile design
uniquify
compile -map_effort medium

# Report design results for EDT IP
report_area > results/config1_dc_script_report.out
report_constraint -all_violators -verbose >>
results/config1_dc_script_report.out
report_timing -path full -delay max >>
results/config1_dc_script_report.out
report_reference >> results/config1_dc_script_report.out
```

```
write -f verilog -hierarchy -o results/config1_circle_edt_gate.v

# Write output netlist
exec cat results/config1_circle_edt_gate.v  results/config1_edt_top_rtl.v
> results/config1_edt_top_gate.v
```

RTL Compiler Synthesis Script External Flow

The tool generates an RTL Compiler synthesis script *created_rtlc_script.scr* when the -*synthesis_script rtl_compiler* option is used with the write_edt_files command as shown:

write_edt_files created -synthesis_script rtl_compiler

This script synthesizes the EDT logic and the top-level wrapper that instantiates the core design and EDT logic for the external flow as shown in the following example.

```
#*************************************************************************
# Cadence RTL Compiler synthesis script for created_edt_top.vhd
# Tessent TestKompress version: v9.1-snapshot_2010.08.19_05.02
# Date: Thu Aug 19 14:07:25 2010
#*************************************************************************

# Set RTL Compiler attributes
set_attribute hdl_auto_async_set_reset true

# Read input design files
read_hdl -vhdl created_core_blackbox.vhd
read_hdl -vhdl created_edt.vhd
read_hdl -vhdl created_edt_top.vhd

# Elaborate design
set_attribute hdl_infer_unresolved_from_logic_abstract true /
elaborate
cd /designs/core_edt_top

# Check design for inconsistencies
check_design

# Timing specification
define_clock -period 10000 -rise 0 -fall 50 edt_clock

# Avoid clock buffering during synthesis.However, remember
# to perform clock tree synthesis later for edt_clock
# set_attribute ideal_network true edt_clock

# Avoid reset signal buffering during synthesis.However, remember
# to perform reset tree synthesis later for edt_reset
set_attribute ideal_network true edt_reset

# Avoid assign statements in the synthesized netlist.
set_attribute remove_assigns true core_edt_top
set_remove_assign_options -preserve_dangling_nets
-respect_boundary_optimization -verbose -design core_edt_top

# Compile design
edit_netlist uniquify core_edt_top
```

```
synthesize -to_mapped -effort medium
change_names -verilog

# Report design results for EDT IP
report_area > created_rtlc_script_report.out
report_design_rules >> created_rtlc_script_report.out
report_timing >> created_rtlc_script_report.out
report_gates >> created_rtlc_script_report.out

# Read in the original core netlist
read_hdl -v1995 m8051_scan.v
elaborate

# Write output netlist
write_hdl > created_edt_top_gate.v
```

RTL Compiler Synthesis Script for Internal Flow

The tool generates an RTL Compiler synthesis script *created_rtlc_script.scr* when the *-synthesis_script rtl_compiler* option is used with the write_edt_files command as shown:

write_edt_files created -synthesis_script rtl_compiler

This script synthesizes the EDT logic in the core netlist for the internal flow as shown in the following example.

```
#*************************************************************************
# Cadence RTL Compiler synthesis script for created_edt.v
# Tessent TestKompress version: v9.1-snapshot_2010.08.19_05.02
# Date: Thu Aug 19 14:10:04 2010
#*************************************************************************

# Bus naming style for Verilog
set_attribute bus_naming_style {%s[%d]}

# Read input design files
read_hdl -v1995 created_edt.v

# Elaborate design
set_attribute hdl_infer_unresolved_from_logic_abstract true /
elaborate

# Synthesize EDT IP
cd /designs/B1_edt

# Check design for inconsistencies
check_design

# Timing specification
define_clock -period 10000 -rise 0 -fall 50 edt_clock

# Avoid clock buffering during synthesis. However, remember
# to perform clock tree synthesis later for edt_clock
set_attribute ideal_network true edt_clock

# Avoid assign statements in the synthesized netlist.
set_attribute remove_assigns true B1_edt
```

```
set_remove_assign_options -preserve_dangling_nets
-respect_boundary_optimization -verbose -design B1_edt

# Compile design
edit_netlist uniquify B1_edt
synthesize -to_mapped -effort medium

# Report design results for EDT IP
report area > created_rtlc_script_report.out
report design_rules >> created_rtlc_script_report.out
report timing >> created_rtlc_script_report.out
report_gates >> created_rtlc_script_report.out
write_hdl > created_B1_edt_gate.v

cd /designs/B2_edt

# Check design for inconsistencies
check_design

# Timing specification
define_clock -period 10000 -rise 0 -fall 50 edt_clock

# Avoid clock buffering during synthesis. However, remember
# to perform clock tree synthesis later for edt_clock
set_attribute ideal_network true edt_clock

# Avoid assign statements in the synthesized netlist.
set_attribute remove_assigns true B2_edt
set_remove_assign_options -preserve_dangling_nets
-respect_boundary_optimization -verbose -design B2_edt


# Compile design
edit_netlist uniquify B2_edt
synthesize -to_mapped -effort medium

# Report design results for EDT IP
report area >> created_rtlc_script_report.out
report design_rules >> created_rtlc_script_report.out
report timing >> created_rtlc_script_report.out
report_gates >> created_rtlc_script_report.out
write_hdl > created_B2_edt_gate.v

# Synthesize EDT multiplexor
cd /designs/core_edt_mux_2_to_1

# Check design for inconsistencies
check_design

# Compile design
synthesize -to_mapped -effort medium

# Report design results for EDT mux
report area >> created_rtlc_script_report.out
report timing >> created_rtlc_script_report.out
write_hdl > created_core_edt_mux_2_to_1_gate.v

# Write output netlist
```

```
exec cat created_core_edt_mux_2_to_1_gate.v created_B2_edt_gate.v
created_B1_edt_gate.v created_edt_top_rtl.v >
created_edt_top_gate.v

# Remove all temporary files
exec rm created_core_edt_mux_2_to_1_gate.v created_B2_edt_gate.v
created_B1_edt_gate.v
```

Test Pattern Generation Files

The tool automatically writes a dofile and a test procedure file containing EDT-specific commands and test procedure steps. As with the similar files produced by Tessent Scan after scan insertion, these files perform basic setups; however, you need to add commands for any pattern generation or pattern saving steps.

- **Dofile —** The dofile includes setup commands and/or switches required to generate test patterns. An example dofile *created_edt.dofile* is shown below. The EDT-specific parts of this file are in bold font.

```
// Define the instance names of the decompressor, compactor, and the
// container module which instantiates the decompressor and
// compactor. Locating those instances in the design allows DRC to
// provide more debug information in the event of a violation. If
// multiple instances exist with the same name, subtitute the
// instance name of the container module with the instance’s
// hierarchical path name.

set_edt_instances -edt_logic_top cpu_edt_i
set_edt_instances -decompressor cpu_edt_decompressor_i
set_edt_instances -compactor cpu_edt_compactor_i

add_scan_groups grp1 created_edt.testproc
add_scan_chains -internal chain1 grp1 /cpu_i/edt_si1 /cpu_i/edt_so1
add_scan_chains -internal chain2 grp1 /cpu_i/edt_si2 /cpu_i/edt_so2
add_scan_chains -internal chain3 grp1 /cpu_i/edt_si3 /cpu_i/edt_so3
add_scan_chains -internal chain4 grp1 /cpu_i/edt_si4 /cpu_i/edt_so4
add_scan_chains -internal chain5 grp1 /cpu_i/edt_si5 /cpu_i/edt_so5
add_scan_chains -internal chain6 grp1 /cpu_i/edt_si6 /cpu_i/edt_so6
add_scan_chains -internal chain7 grp1 /cpu_i/edt_si7 /cpu_i/edt_so7
add_scan_chains -internal chain8 grp1 /cpu_i/edt_si8 /cpu_i/edt_so8

add_clocks 0 clk
add_clocks 0 edt_clock

add_write_controls 0 ramclk

add_read_controls 0 ramclk

add_pin_constraints edt_clock C0

// EDT settings.  Please do not modify.
// Inconsistency between the EDT settings and the EDT logic may
// lead to DRC violations and invalid patterns.

set_edt_options -channels 3 -ip_version 3 -decompressor_size 12
   -injectors_per_channel 2
```

Notice the -Internal switch used with the add_scan_chains command. This switch must be used for all compressed scan chains (scan chains driven by and observed through the EDT logic) when setting up to generate compressed test patterns. The reason for this requirement is explained in “Design Rule Checks,” earlier in the chapter.

Note

Be sure the scan chain input and output pin pathnames specified with the add_scan_chains -*Internal* command are kept during layout. If these pin pathnames are lost during the layout tool’s design flattening process, the generated dofile will not work anymore. If that happens, you must manually generate the add_scan_chains -*Internal* commands, substituting the original pin pathnames with new, logically equivalent, pin pathnames.

Note

If your design includes uncompressed scan chains (chains whose scan inputs and outputs are primary inputs and outputs), you must define each such scan chain using the add_scan_chains command *without* the -*Internal* switch when setting up for EDT pattern generation. You will need to add these commands to the dofile manually.

Other commands in this file add the EDT clock and constrain it to its off state, specify the number of scan channels, and specify the version of the EDT logic architecture. For information about how you can use this dofile to generate compressed test patterns, refer to “Preparation for Test Pattern Generation” on page 109.”

- **Test Procedure File** — The tool also writes a test procedure file for test pattern generation. The tool takes the test procedure file used for EDT logic creation and adds the test procedures necessary to drive the EDT logic.

  The following example is a test procedure file, *created_edt.testproc*. The EDT-specific parts of this file are shown in bold font. For complete details about the EDT-specific functionality included in this file, refer to the “Preparation for Test Pattern Generation” on page 109.

```
//
set time scale 1.000000 ns ;
set strobe_window time 100 ;

timeplate gen_tp1 =
   force_pi 0 ;
   measure_po 100 ;
   pulse clk 200 100;
   pulse edt_clock 200 100;
   pulse ramclk 200 100;
   period 400 ;
end;

procedure capture =
   timeplate gen_tp1 ;
   cycle =
      force_pi ;
      measure_po ;
      pulse_capture_clock ;
   end;
end;

procedure shift =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   cycle =
      force_sci ;
      force edt_update 0 ;
      measure_sco ;
      pulse clk ;
      pulse edt_clock ;
   end;
end;

procedure load_unload =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   cycle =
      force clk 0 ;
      force edt_bypass 0 ;
      force edt_clock 0 ;
      force edt_update 1 ;
      force ramclk 0 ;
      force scan_en 1 ;
      pulse edt_clock ;
   end ;
   apply shift 26;
end;

procedure test_setup =
   timeplate gen_tp1 ;
   cycle =
      force edt_clock 0 ;
   end;
end;
```

Bypass Mode Files

By default, the EDT logic includes bypass circuitry. To operate the bypass circuitry, an additional dofile and test procedure file is created. The additional dofile and test procedure file is then used with uncompressed ATPG to bypass the EDT logic and run regular ATPG.

For information on the options for creating files, see the write_edt_files command.

To disable the generation of bypass logic, see the set_edt_options command.

For improved design routing, the bypass logic can be inserted into the netlist instead of the EDT logic. For more information, see "Generating EDT Logic When Bypass Logic is Defined in the Netlist" on page 174.

- **Dofile —** This example dofile, *created_bypass.dofile*, enables you to run regular ATPG. The dofile specifies the scan channels as chains because in bypass mode, the channels connect directly to the input and output of the concatenated internal scan chains, bypassing the EDT circuitry.

```
//
add_scan_groups grp1 created_bypass.testproc
add_scan_chains edt_channel1 grp1 edt_channels_in1
edt_channels_out1

add_clocks 0 clk

add_write_controls 0 ramclk

add_read_controls 0 ramclk
```

- **Test Procedure File —** Notice the line (in bold font) near the end of this otherwise typical test procedure file, *created_bypass.testproc*. That line forces the EDT bypass signal, "edt_bypass" to a logic high in the load_unload procedure and activates bypass mode.

```
//
set time scale 1.000000 ns ;
set strobe_window time 100 ;

timeplate gen_tp1 =
   force_pi 0 ;
   measure_po 100 ;
   pulse clk 200 100;
   pulse ramclk 200 100;
   period 400 ;
end;

procedure capture =
   timeplate gen_tp1 ;
   cycle =
      force_pi ;
      measure_po ;
      pulse_capture_clock ;
   end;
end;
procedure shift =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   cycle =
      force_sci ;
      measure_sco ;
      pulse clk ;
   end;
end;

procedure load_unload =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   cycle =
      force clk 0 ;
      force edt_bypass 1 ;
      force ramclk 0 ;
      force scan_en 1 ;
   end ;
   apply shift 125;
end;
```

Inserting EDT Logic During Synthesis

When the EDT logic is placed internal to the design core, the tool writes a core netlist with an instance of the EDT logic connected between I/O pads and internal scan chains without the gate-level description of the EDT logic.

Alternatively, you can use this procedure to create a DC synthesis script that both inserts and synthesizes the EDT logic inside the core netlist with the write_edt_files *-insertion dc* option. This DC script is more complex because it includes additional commands that help instantiate the EDT logic within the design core and requires that the entire design be loaded into the synthesis tool so the necessary connections can be specified.

Note

The order of the gate-level EDT logic modules in the resulting design is different than that written when the tool inserts the EDT logic before synthesis.

Prerequisites

- Gate-level netlist.

Procedure

1. Invoke Tessent Shell and configure the EDT logic parameters. See "Creating EDT Logic" on page 47. You must set up the EDT logic insertion to be internal to the design core with the set_edt_options command.

2. Run DRC and correct any errors. See "Design Rule Checks" on page 75.

3. Examine test coverage and data volume estimates and adjust EDT configurations if necessary. See "Analyzing Compression" on page 48.

4. Create the EDT logic files. For example:

   ```
   write_edt_files created -insertion dc
   ```

   The following files are created:

   - **created_edt.v** — EDT logic description in RTL.
   - **created_dc_script.scr** — DC synthesis script for the EDT logic.
   - **created_edt.dofile** — Dofile for test pattern generation.
   - **created_edt.testproc** — Test procedure file for test pattern generation.

- o **created_bypass.dofile** — Dofile for uncompressed test patterns (bypass mode).
- o **created_bypass.testproc** — Test procedure file for uncompressed test patterns (bypass mode).

5. Synthesize the EDT logic. See "Synthesizing the EDT Logic" on page 99.
6. Generate test patterns. See "Generating/Verifying Test Patterns" on page 109.

Related Topics

Synthesis and Internal EDT Logic
Creation of a Reduced Netlist for Synthesis
The EDT Logic Files
Inserting EDT Logic During Synthesis

Synthesis Script that Inserts/Synthesizes EDT Logic

The following DC synthesis script is an example of the script generated when the write_edt_files -*insertion dc* option is used.

```
#************************************************************************
#  Synopsys Design Compiler script for EDT logic insertion
#
#************************************************************************

# Initialize DC variables
set bus_naming_style {%s[%d]}

# Read input design files
read_file -f verilog results/created_edt.v
read_file -f verilog netlists/gate_scan_sco.v

# Current design is the top-most level.
current_design retimetest

# Create an instantiation of EDT logic within the top-level of the design.
create_cell retimetest_edt_i [find design retimetest_edt]

# Create instantiation(s) of an EDT mux to support sharing of output
      channel and core pins.
create_cell retimetest_edt_mux_2_to_1_i1 [find design
      retimetest_edt_mux_2_to_1]

# Connect core scan inputs to EDT logic scan inputs.
set scan_inputs { "edt_si1" "edt_si2" "edt_si3" \
                  "edt_si4" "edt_si5" "edt_si6" \
                  "edt_si7" "edt_si8" "edt_si9" \
                  "edt_si10" "edt_si11" "edt_si12" \
                  "edt_si13" "edt_si14" "edt_si15" \
                  "edt_si16" }
set count 0
foreach i $scan_inputs {
   set temp_net [all_connected [find port $i]]
   disconnect_net $temp_net [find port $i]
   set temp "retimetest_edt_i/edt_scan_in[$count]"
   connect_net $temp_net [find pin $temp]
   query_objects [all_connected [find net $temp_net]]
   incr count
}

# Remove scan input ports from top-level.
set scan_inputs_to_delete { "edt_si1" "edt_si2" "edt_si3" \
                            "edt_si4" "edt_si5" "edt_si6" \
                            "edt_si7" "edt_si8" "edt_si9" \
                            "edt_si10" "edt_si11" "edt_si12" \
                            "edt_si13" "edt_si14" "edt_si15" \
                            "edt_si16" }
foreach i $scan_inputs_to_delete {
   remove_port [find port $i]
}

# Connect core scan outputs to EDT logic scan outputs.
set scan_outputs { "edt_so1" "edt_so2" "edt_so3" \
                   "edt_so4" "edt_so5" "edt_so6" \
                   "edt_so7" "edt_so8" "edt_so9" \
                   "edt_so10" "edt_so11" "edt_so12" \
                   "edt_so13" "edt_so14" "edt_so15" \
                   "edt_so16" }
set count 0
foreach i $scan_outputs {
   set temp_net [all_connected [find port $i]]
   disconnect_net $temp_net [find port $i]
   set temp "retimetest_edt_i/edt_scan_out[$count]"
   connect_net $temp_net [find pin $temp]
   query_objects [all_connected [find net $temp_net]]
   incr count
}

# Remove scan output ports from top-level.
set scan_outputs_to_delete { "edt_so1" "edt_so2" "edt_so3" \
                             "edt_so4" "edt_so5" "edt_so6" \
                             "edt_so7" "edt_so8" "edt_so9" \
                             "edt_so10" "edt_so11" "edt_so12" \
                             "edt_so13" "edt_so14" "edt_so15" \
                             "edt_so16" }
foreach i $scan_outputs_to_delete {
   remove_port [find port $i]
}

# Connect EDT control pins to the top-level.

# Route pin edt_clk_buf/Z to the EDT control pin
retimetest_edt_i/edt_clock.
set temp_net [all_connected [find pin edt_clk_buf/Z]]
connect_net $temp_net [find pin retimetest_edt_i/edt_clock]
query_objects [all_connected [find net $temp_net]]

# Route pin edt_update to the EDT control pin retimetest_edt_i/edt_update.
create_port edt_update -direction in
create_net retimetest_edt_update_net
connect_net retimetest_edt_update_net [find port edt_update]
connect_net retimetest_edt_update_net [find pin
retimetest_edt_i/edt_update]
```

```
query_objects [all_connected [find net retimetest_edt_update_net]]

# Route pin edt_bypass to the EDT control pin retimetest_edt_i/edt_bypass.
create_port edt_bypass -direction in
create_net retimetest_edt_bypass_net
connect_net retimetest_edt_bypass_net [find port edt_bypass]
connect_net retimetest_edt_bypass_net [find pin
retimetest_edt_i/edt_bypass]
query_objects [all_connected [find net retimetest_edt_bypass_net]]

# Connect EDT input channel pins to the top-level.

# Route pin edt_channels_in1 to the EDT channel input pin
retimetest_edt_i/edt_channels_in[0].
create_port edt_channels_in1 -direction in
create_net retimetest_edt_channels_in0_net
connect_net retimetest_edt_channels_in0_net [find port edt_channels_in1]
connect_net retimetest_edt_channels_in0_net [find pin
retimetest_edt_i/edt_channels_in[0]]
query_objects [all_connected [find net retimetest_edt_channels_in0_net]]

# Route pin edt_channels_in2 to the EDT channel input pin
retimetest_edt_i/edt_channels_in[1].
create_port edt_channels_in2 -direction in
create_net retimetest_edt_channels_in1_net
connect_net retimetest_edt_channels_in1_net [find port edt_channels_in2]
connect_net retimetest_edt_channels_in1_net [find pin
retimetest_edt_i/edt_channels_in[1]]
query_objects [all_connected [find net retimetest_edt_channels_in1_net]]

# Connect EDT output channel pins to the top-level.

# Route EDT channel output pin retimetest_edt_i/edt_channels_out[0] to
left/rcmd_fc_l_fromCore.
set temp_net [all_connected [find pin left/rcmd_fc_l_fromCore]]
disconnect_net $temp_net [find pin left/rcmd_fc_l_fromCore]
connect_net $temp_net [find pin retimetest_edt_mux_2_to_1_i1/a_in]
create_net retimetest_edt_channels_out0_top_net
connect_net retimetest_edt_channels_out0_top_net [find pin
retimetest_edt_mux_2_to_1_i1/b_in]
connect_net retimetest_edt_channels_out0_top_net [find pin
retimetest_edt_i/edt_channels_out[0]]

create_net retimetest_edt_mux_2_to_1_i1_out_net
connect_net retimetest_edt_mux_2_to_1_i1_out_net [find pin
retimetest_edt_mux_2_to_1_i1/z_out]
connect_net retimetest_edt_mux_2_to_1_i1_out_net [find pin
left/rcmd_fc_l_fromCore]
set temp_net [all_connected [find pin scanen_buf/scan_en_out]]
connect_net $temp_net [find pin retimetest_edt_mux_2_to_1_i1/sel]

# Route EDT channel output pin retimetest_edt_i/edt_channels_out[1] to
edt_channels_out2_buf/A.
create_net retimetest_edt_channels_out1_net
connect_net retimetest_edt_channels_out1_net [find pin
edt_channels_out2_buf/A]
connect_net retimetest_edt_channels_out1_net [find pin
retimetest_edt_i/edt_channels_out[1]]
```

```
query_objects [all_connected [find net retimetest_edt_channels_out1_net]]

# Connect system clocks to the EDT logic bypass logic.

# Route clock output clk2 to bypass logic.
set temp_net [all_connected [find port clk2]]
connect_net $temp_net [find pin retimetest_edt_i/clk2]
query_objects [all_connected [find net $temp_net]]

# Synthesize EDT logic
current_design retimetest_edt

# Check design for inconsistencies
check_design

# Timing specification
create_clock -period 10 -waveform {0 5} edt_clock

# Avoid clock buffering during synthesis. However, remember
# to perform clock tree synthesis later for edt_clock
set_clock_transition 0.0 edt_clock
set_dont_touch_network edt_clock

# Avoid assign statements in the synthesized netlist.
set_fix_multiple_port_nets -feedthroughs -outputs -buffer_constants

# Compile design
uniquify
compile -map_effort medium

# Report design results for EDT logic
report_area > results/created_dc_script_report.out
report_constraint -all_violators -verbose >>
results/created_dc_script_report.out
report_timing -path full -delay max >>
results/created_dc_script_report.out
report_reference >> results/created_dc_script_report.out

# Synthesize EDT multiplexor
current_design retimetest_edt_mux_2_to_1

# Check design for inconsistencies
check_design

# Compile design
compile -map_effort medium

# Report design results for EDT mux
report_area >> results/created_dc_script_report.out
report_timing -path full -delay max >>
results/created_dc_script_report.out

# Write output netlist
current_design retimetest
write -f verilog -hierarchy -o results/created_edt_top_gate.v
```

The preceding script performs the following EDT logic insertion and synthesis steps:

1. Fixing the bus naming style — Because bus signals can be expressed in either bus form (for example, foo[0] or foo(0)), or bit expanded form (foo_0), this command fixes the bus style to the bus form. This is particularly necessary during logic insertion because the script looks for the EDT logic bus signals to be connected to the scan chains.
2. Read input files — Next, the input gate-level netlist for the core and the RTL description of the EDT logic are read.
3. Set current design — The current design is set to the top-most level of the input netlist.
4. Instantiate the EDT logic and 2x1 multiplexer module — The EDT logic is instantiated within the top-level of the design. If there is sharing between EDT channel outputs and functional pins, a 2x1 multiplexer module (the description is included in the *created_edt.v* file) is also instantiated.
5. Connect scan chain inputs — As mentioned earlier, scan chain inputs should be connected to the top level without any I/O pads associated with them. This part of the script disconnects the nets that are connected to the scan chain input ports and connects them to the EDT logic.
6. Remove scan chain input ports — Remove the dangling scan chain input ports that are not connected to any logic after preceding step 5.
7. Connect scan chain outputs — Same as step 5, except that now the scan chain outputs are connected to the EDT logic.
8. Remove scan chain output ports — Scan chain output ports, which are left dangling after step 7, are removed from the top level.
9. Connect EDT control pins — The EDT control pins are connected to the output of pre-existing pads. This is done only if you specified an internal node pin to which to connect the EDT control signal. If not, a new port is created, and the EDT control pin is connected to the new port. The script shows the connection for only the edt_clock signal. Similar commands are necessary to connect each of the other EDT control pins.
10. Connect EDT channel input pins — The next set of commands create a new port for the input channel pin and connect the EDT input channel pin to the newly created port. This has to be repeated for each input channel pin. This is done only when no internal node name was specified for the EDT pin. If an internal node name was specified, the script would be the same as in step 9.

Note

Be aware you need to add an I/O pad later for each new port that is created.

11. Connect EDT channel output pin to a specified internal node (or a top-level pin) that is driven by functional logic — The output channel pin in this case is shared with a

functional pin. Whenever the node is shared with functional logic or is connected to TIE-X (a blackbox is assumed in such cases), a multiplexer is inserted. However, if the specified internal node is connected to a constant signal value, the net is disconnected and connected to the EDT channel output pin. This section of the script inserts a multiplexer and connects the inputs from the EDT logic and the functional net to the multiplexer inputs. The output of the multiplexer is connected to the input of the I/O pad cell.

Note

The select input of the multiplexer is connected to the existing scan enable signal, as the functional or scan chain output can be propagated through the multiplexer depending on the shift and capture modes of scan operation.

You should specify the name of the scan enable signal. If a name is not specified and a pin with a default name (scan_en) does not exist at the top level, a new scan_en pin is created.

12. Connect EDT channel output pin to a non-shared specified internal node (or a top-level pin) — in this case, the specified internal node is not shared with any functional logic. The tool removes any net that is connected to the internal node. It creates a new net and connects it to the channel output pin.
13. Connect system clocks to EDT logic — For the bypass mode, scan chains are concatenated so that the EDT logic can be bypassed and normal ATPG scan patterns can be applied to the circuit. During scan chain concatenation, lockup cells are often needed (especially if the clocks are different or the clock edges are different) to guarantee proper scan shifting. These “bypass” lockup cells are driven by the clocks that drive the source or the destination scan cells. As a result, some system clocks have to be routed to the bypass module of the EDT logic. Clock lines are tapped right before they fan out to multiple modules or scan cells and are brought up to the topmost level and connected to the EDT logic.
14. Synthesis of RTL code — At this point, the insertion of the EDT logic within the top-level of the original core is complete. The subsequent parts of the script are mainly for synthesizing the logic. This part of the script is almost the same as that of the external flow, with the following exceptions: the EDT logic is synthesized first followed by the EDT multiplexer(s). In both cases, the synthesis is local to the RTL blocks and does not affect the core, which is already at the gate level.
15. Write out final netlist — Once the synthesis step is completed, DC writes out a gate-level netlist that contains the original core, the EDT logic, and any multiplexers the tool added to facilitate sharing of output pins.

Creation of a Reduced Netlist for Synthesis

In the internal flow, the EDT logic is instantiated within an existing pad-inserted netlist and connections must be made from the pad terminals to the EDT logic pins. When using DC to insert and synthesize the EDT logic as described in "Inserting EDT Logic During Synthesis," the entire netlist must be read into the synthesis tool, so the proper connections can be specified which can cause problems.

Input netlists can be huge and it may take a lot of time or perhaps not even be possible to run the synthesis tool. If your netlist is in this category, you can use the tool to write out a reduced-size netlist especially for the synthesis run with the write_edt_files -*reduced_netlist* command. The reduced netlist includes only the modules required for the synthesis tool to make the necessary connections between the pad terminals and EDT logic pins.

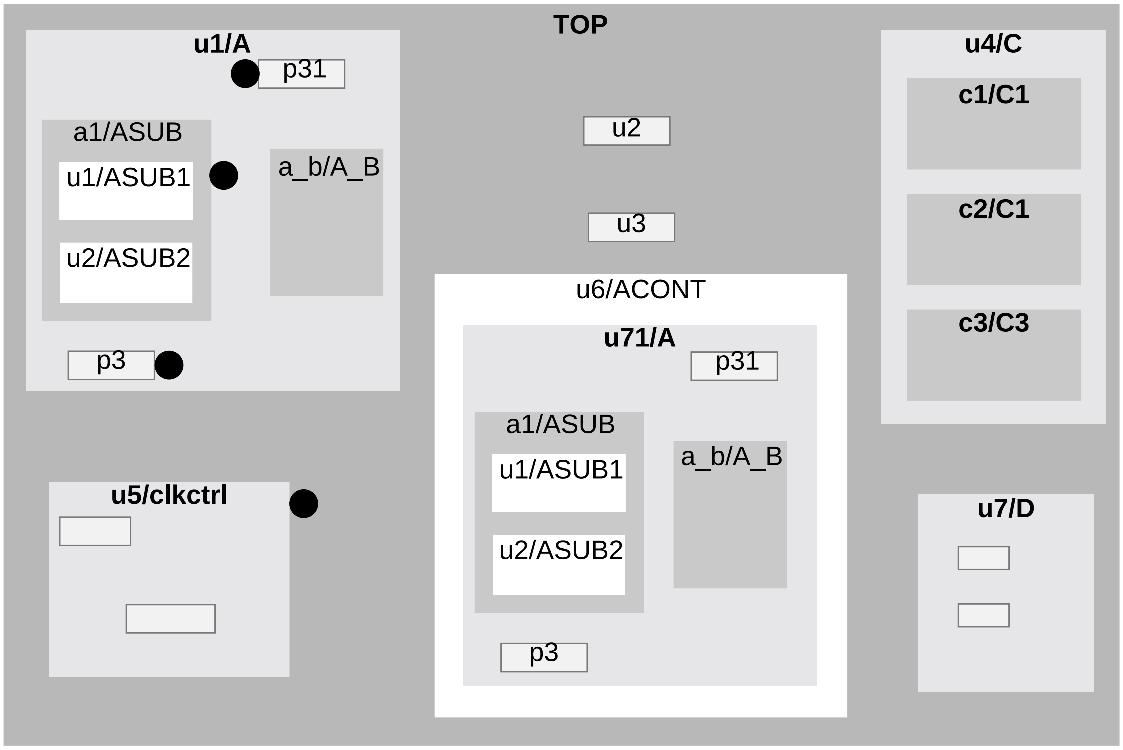
You provide this smaller file to the synthesis tool instead of providing the entire input netlist. The tool writes the rest of the input netlist into a second netlist file that excludes the modules written out in the *synthesis only* file but includes everything else. You then use the output netlist from the synthesis run along with the second netlist as inputs for ATPG.

Note

Another option is to use the tool to insert the EDT logic into the core netlist before synthesis. This is the default behavior for the internal flow.

Figure 4-8 is a conceptual example of an input netlist. Each box represents an instance with the instance_name/module_names shown. The small rectangles shaded with dots represent instances of technology library cells. The black circles represent four internal EDT logic connection nodes.

Figure 4-8. Design Netlist with Internal Connection Nodes



When writing out the compression files with the -Reduced_netlist switch, the modules are distributed between the two netlists as follows:

- **<prefix>_reduced_netlist.v (for synthesis)** — TOP, A, ASUB, clkctrl
- **<prefix>_rest_of_netlist.v (rest of the netlist for ATPG)** — ASUB1, ASUB2, A_B, C, C1, C3, ACONT, D

Note

The reduced netlist for synthesis does not include the technology library cells that have EDT logic connection nodes because the port list is sufficient for making these connections.

Chapter 5
Synthesizing the EDT Logic

After you create the EDT logic, the next step is to synthesize it. The tool creates a basic Design Compiler (DC) synthesis script, in either dcsh or TCL format, that you can use as a starting point. Running the synthesis script is a separate step in which you exit the tool and use DC to synthesize the EDT logic. You can use any synthesis tool; the generated DC script provides a template for developing a custom script for any synthesis tool.

The EDT Logic Synthesis Script

If you use DC to synthesize the netlist, you should examine the *.synopsys_dc.setup* file and verify that it points to the correct libraries. Also, examine the DC synthesis script generated by the tool and make any needed modifications. For more information, see "The EDT Logic Files" on page 78.

Note

You should preserve the pin names in the EDT logic hierarchy. Preserving pin names ensures that pins resolve when test patterns are created and increases the usefulness of the debug information returned during DRC.

Note

When using the external flow and boundary scan, you must modify this script to read in the RTL description of the boundary scan circuitry. Refer to "Preparing to Synthesize Boundary Scan and EDT Logic" on page 182 for an example DC synthesis script with modifications for boundary scan.

The following DC commands are included in the synthesis scripts created by the tool:

- set_fix_multiple_port_nets -feedthroughs -outputs -buffer_constants

  This command prevents DC from including *assign* statements in the Verilog gate-level netlist to prevent problems later in the design flow.

- set_clock_transition 0.0 edt_clock
  set_dont_touch_network edt_clock

  These commands prevent buffering of the EDT clock during synthesis and preserves the EDT clock network. However, you must perform clock tree synthesis later for the EDT clock.

After you run DC to synthesize the netlist without any errors, verify the tri-state buffers were correctly synthesized. In some cases, DC may insert incorrect references to **TSGEN**. For information on correcting these references, see "Incorrect References in Synthesized Netlist" on page 305.

Synthesis and External EDT Logic

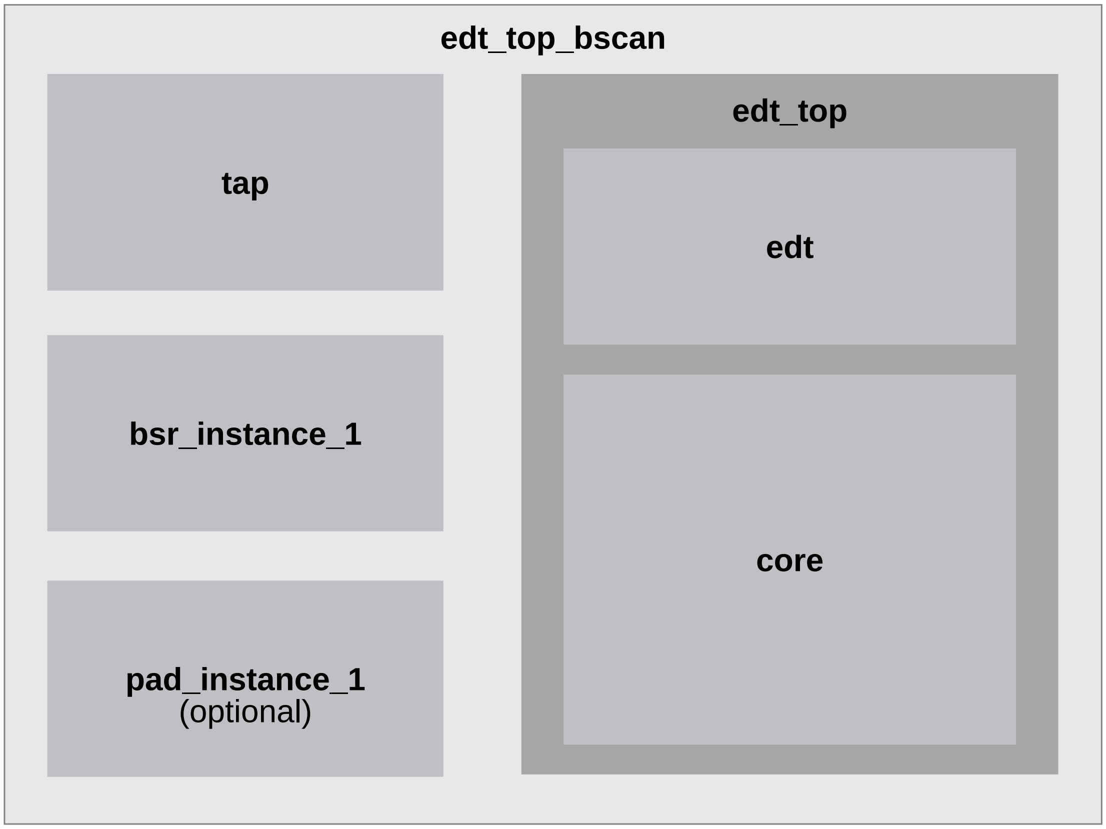
Once the EDT logic is created but before you synthesize it, you should insert I/O pads and (optionally) boundary scan. For designs that require boundary scan, you should insert the boundary scan first, followed by I/O pads. Then, synthesize the I/O pads and boundary scan together with the EDT logic.

Note
You can add boundary scan and I/O pads simultaneously with a boundary scan tool.

Boundary Scan

Boundary scan cells cannot be present in your design before the EDT logic is inserted. To include boundary scan, you perform an additional step *after* the EDT logic is created. In this step, you can use any tool to insert boundary scan. As shown in Figure 5-1, the circuitry should include the boundary scan register, TAP controller, and (optionally) I/O pads.

Figure 5-1. Contents of Boundary Scan Top-Level Wrapper



I/O Pad Insertion

You can use any method to insert I/O pads *after* scan insertion and EDT logic creation. If you need to integrate EDT logic after the I/O pads are inserted, see "Managing Pre-existing I/O Pads" on page 38.

If the core and pads are separated as described in "Managing Pre-existing I/O Pads" on page 38, you should reinsert the EDT logic-core combination into the original circuit in place of the extracted core. When you reinsert it, ensure the EDT logic-core combination is connected to the I/O pads. Add pads for any new EDT pins not shared with existing core pins.

If you need to insert I/O pads before scan insertion and you used the architecture swapping solution described in the "Managing Pre-existing I/O Pads" on page 38," then I/O pads are already included in your scan-inserted design and you can proceed to insert boundary scan.

Synthesis and Internal EDT Logic

The tool inserts and connects an instance of the EDT logic into the design netlist and creates a DC script to synthesize the EDT logic.

You may be able to run the script without modification if the following are true:

- DC is the synthesis tool.
- The default clock definitions are acceptable.
- Technology library files are set up correctly in the *.synopsys_dc.setup* file.

Note

The syntax of the *.synopsys_dc.setup* file and the DC synthesis script differ depending on which format, dcsh or TCL, they support. If the *.synopsys_dc.setup* file does not exist, you must add the library file references to the synthesis script.

Optionally, you can insert the EDT logic into the core during synthesis. See "Inserting EDT Logic During Synthesis" on page 90.

SDC Timing File Generation

You can use the tool to generate Synopsys Design Constraint (SDC) timing files for the static timing analysis of the test logic.

Separate SDC files provide timing constraints for the EDT logic and the ATPG setups as described in the following topics:

- EDT Logic/Core Interface Timing Files
- Scan Chain and ATPG Timing Files

Note

The SDC files are generated from the timing specified in the test procedure file. The generated SDC files should be used as templates and employed for static timing analysis only after appropriate values are inserted to correspond with actual timing information.

The timing files are formatted in the TCL programming language with multiple sections. This allows you to select one or all sections depending on your needs.

You can also set variables before the timing files are loaded to specify values in the timing files as described in Table 5-1.

Table 5-1. Timing File Variables

Description	Variables
Parameters for system clocks	system_clock_latency_min system_clock_latency_max system_clock_uncertainty_setup system_clock_uncertainty_hold
Parameters for EDT clocks	edt_clock_latency_min edt_clock_latency_max edt_clock_uncertainty_set_edt_clock_unce edt_clock_uncertainty_hold
I/O delay for EDT pins	edt_pins_input_delay edt_pins_output_delay

EDT Logic/Core Interface Timing Files

You can output timing files specific to the EDT logic and design core interface with the write_edt_files -*Timing_constraints* command. Depending on the application, the following timing files are written out:

- ***filename_prefix*_edt_shift_sdc.tcl** — Specifies constraints for the EDT shift mode.
- ***filename_prefix*_bypass_shift_sdc.tcl** — Specifies constraints for the EDT bypass shift mode. This file is written for applications that include a bypass configuration. By default, the tool outputs an EDT bypass configuration.
- ***filename_prefix*_slow_capture_sdc.tcl** — Specifies constraints for slow-capture mode. This file is only written when stuck-at patterns or launch-off-shift capture patterns are used.
- ***filename_prefix*_fast_capture_sdc.tcl** —Specifies constraints for fast-capture mode. This file is only written when launch-off capture transition patterns are used.

Timing files can also be generated for EDT logic with dual compression configurations. When test patterns are applied, only one of the configurations is active at any time. So, the paths originating at *edt_configuration* are declared as multi-cycle paths to avoid the need to verify each of the individual configurations separately. For more information on dual configurations, see "Dual Compression Configurations" on page 55.

Note

When the EDT logic is placed inside the design and a top-level pin name is not specified for a control pin, then the specified internal connection name is used for synthesis and in the constraints. For more information, see the set_edt_pins - command.

EDT Shift Mode Clock Constraints

During shift, the EDT logic is clocked along with all the scan cells as new data is loaded and the captured data is unloaded from the scan chains. Therefore, the *edt_clock* and all the shift clocks are declared follows:

- **create_clock** — Declares all clocks used for scan chain shifting at the very beginning of the file. For example:

  ```
  create_clock -name edt_clock -period 100 -waveform {50 90}
  [get_ports edt_clock]
  ```

- **set_clock_latency** —Describes the clock network latency for all clocks used during shift. Clock latency for both the minimum and maximum operating conditions is specified. Because the tool has no timing information, a default value of 0 is used for the latencies. These default values can be changed to reflect the actual values as necessary. For example:

  ```
  set_clock_latency -min 0 [get_clocks edt_clock]
  set_clock_latency -max 0 [get_clocks edt_clock]
  ```

- **set_clock_uncertainty** — Describes the uncertainty (skew) related to the setup and hold times for the flops driven by specified shift clocks. For example:

  ```
  set_clock_uncertainty -setup <def_value> [get_clocks edt_clock]
  set_clock_uncertainty -hold <def_value> [get_clocks edt_clocks]
  ```

EDT Shift Mode Input/Output Pin Delay Constraints

During shift mode, the input and output delays for the EDT control and channel pins are declared. For the *edt_channel* pins, the input delay is measured with respect to the *force_pi* and *measure_po* events in the test procedure. Default values can be changed to reflect the actual values as necessary. For example:

- **set_input_delay** – Specifies the arrival time of the signals relative to when the clock edge appears. For example:

  ```
  set_input_delay <def_value> -clock force_pi [get_ports edt_channel_in]
  ```

- **set_output_delay** – Specifies the departure time of the signals relative to when the clock edge appears. For example:

  ```
  set_output_delay <def_value> -clock measure_po
                                            [get_ports edt_channel_out]
  ```

EDT Shift Mode Static Constraints

During EDT shift mode, static values for certain EDT-specific signals are declared as follows:

- **EDT bypass mode signal —** *edt_bypass* signal is constrained to 0 (off), when EDT mode is enabled. For example:

  ```
  set_case_analysis 0 edt_bypass
  ```

- **EDT reset signal** — *edt_reset* signal is constrained to 0 (off). For example:

  ```
  set_case_analysis 0 edt_reset
  ```

- **Scan enable (SEN) signal** — *scan_en* signal controls the select input of the muxes when channel output pins are shared with functional pins. For example:

  ```
  set_case_analysis <on_state> scan_en
  ```

- **Dual configuration signal** — *edt_configuration* signal is set to either a 1 or 0 value by the test patterns depending on the configuration being used. Instead of constraining the *edt_configuration* signal, all paths originating from the pin are declared as multi-cycle paths. For example:

  ```
  set_multicycle_path <path_multiplier> -from edt_configuration
  ```

EDT Shift Mode Timing Exceptions

- **False and multi-cycle paths** — During shift, all paths in the EDT logic are exercised, so no false or multi-cycle paths are declared.

Bypass Shift Mode Constraints

In bypass shift mode, the EDT decompressor, compactor, and masking logic are completely bypassed and the scan chains behave as uncompressed chains that operate with regular ATPG patterns. The timing constraints for bypass shift mode are similar to those for regular scan operation except as follows:

- **EDT bypass signal** — *edt_bypass* signal is constrained to 1 (on). For example:

  ```
  set_case_analysis 1 edt_bypass
  ```

- **Scan enable (SEN) signal** — *scan_en* signal is asserted to its on state when an EDT channel output pin is shared with a functional output pin. The set_edt_pins command specifies the *scan_en* pin. For example:

  ```
  set_case_analysis <on_state> scan_en
  ```

- **Clock constraints** — All clocks used during bypass shift mode are declared using the same commands as for EDT shift mode. See "EDT Shift Mode Clock Constraints" on page 104.

- **Input and output delays** — The input and output delays should be described for all scan chain I/Os. The input and output delay constraints for bypass shift are declared with the same commands as for EDT shift. See "EDT Shift Mode Input/Output Pin Delay Constraints" on page 104.

- **EDT logic** – In the bypass mode, the EDT logic is completely bypassed, and therefore, any paths originating and ending in the EDT logic are declared as false paths as follows:

  ```
  set_false_path -from edt_clock
  set_false_path -to edt_clock
  ```

Bypass/EDT Capture Mode Constraints

In the capture mode, the primary objective is to mimic the functional operation of the design, but only timing constraints related to the test logic are written. Constraints related to the functional mode of operation should be specified by the functional timing constraints file for the design. Specifically, some of the timing constraints are as follows:

- **Clock constraints** — All clocks used during capture mode are declared using the same commands as for EDT shift. See "EDT Shift Mode Clock Constraints" on page 104.
- **Input and output delays** — The input and output delays are declared for all scan chain I/Os using the same commands as for EDT shift. See "EDT Shift Mode Input/Output Pin Delay Constraints" on page 104.
- **Static constraints** — The *edt_reset* signal is constrained to its off (0) state during capture. For example:

  ```
  set_case_analysis 0 edt_reset
  ```

Note

edt_bypass, *edt_update*, and *edt_configuration* could potentially be shared with functional pins set by ATPG, so they are not constrained. During capture, the EDT clock is not pulsed, so the values on these pins do not interfere with the EDT logic.

- **Inactive paths** — The *edt_clock* is not pulsed during capture, so the following paths are unused and need to be declared as false paths:

  - Between the *mask_shift_reg* and *mask_hold_reg*.
  - Between the *mask_hold_reg* and the output channels, pipeline cells, or lockup cells (if they exist).
  - Between the lockup cells at the output of the decompressor and the input of the scan chains.
  - Between the pipeline stages at the compactor and the EDT channel output pins.

  False paths are declared for all these cases by declaring all paths originating from state elements clocked by *edt_clock* as false paths. For example:

  ```
  set_false_path -from edt_clock
  ```

- **Paths between the scan chain outputs and the compactor** — The scan chain outputs feeding the compactor are not active during capture. Therefore, all paths from the decompressor outputs to the *edt_channel_output* or the lockup cells in front of the pipeline stages inside the compactor are declared as false paths.

  If no pipeline stages or lockup cells exist, then the following constraints declare all EDT channels as false paths. For example:

  ```
  set_false_path -to <edt_channel_output>
  ```

If pipeline stages or lockup cells do exist, then all paths originating from state elements clocked by *edt_clock* are declared as false paths. For example:

```
set_false_path -from edt_clock
```

- **Slow and fast capture modes** — The slow capture mode corresponds to stuck-at and launch-off-shift patterns, and fast capture mode corresponds to launch-off-capture (broadside) patterns.
  - **Slow capture mode** — The *scan_enable* pin is unconstrained, so the scan path could potentially be used by ATPG. For bypass patterns, the bypass chain concatenation path through *edt_bypass_logic* is unconstrained. For the EDT capture mode this path is not used, but it is unconstrained so that both bypass and EDT capture can share the same timing constraints.
  - **Fast capture mode** — The bypass chain concatenation path through the *edt_bypass_logic* is not used and is declared a false path. For example:

    ```
    create clock -period 100 { edt_i/sysclk }
    set_false_path -to edt_i/sysclk
    set_case_analysis 0 edt_i/edt_bypass
    ```

Scan Chain and ATPG Timing Files

You can output timing files specific to the scan path and ATPG setup in the core with the write_core_timing_constraints *filename_prefix* command. Depending on the application, the following timing files are written out.

- ***filename_prefix*_core_shift_sdc.tcl** — Specifies shift mode constraints.
- ***filename_prefix*_slow_capture_sdc.tcl** — Specifies slow-capture mode constraints. This file is only written when stuck-at or launch-off-shift capture patterns are used.
- ***filename_prefix*_fast_capture_sdc.tcl** — Specifies fast-capture mode constraints. This file is only written when launch-off capture transition patterns are used.

Scan Chain and ATPG Core Constraints

The scan chain and ATPG constraints associated with the core are determined as follows:

- For scan shift mode, the *scan_en* signal is constrained to its active value, so paths from scan cell outputs to the functional logic are declared as false paths. This is done by forcing the values found in the shift procedure.
- For capture mode, all shift paths between successive scan cells are declared as false paths, unless launch-off-shift (LOS) transition patterns are in effect. This is done by forcing pin constraint values.
- For at-speed testing, the *hold_pi* and *mask_po* constraints are translated into timing constraints. A warning message is issued when writing out the fast capture mode timing constraints if *hold_pi* and *mask_po* are not specified.

- Cell constraints specified for an ATPG run are declared during the capture mode.

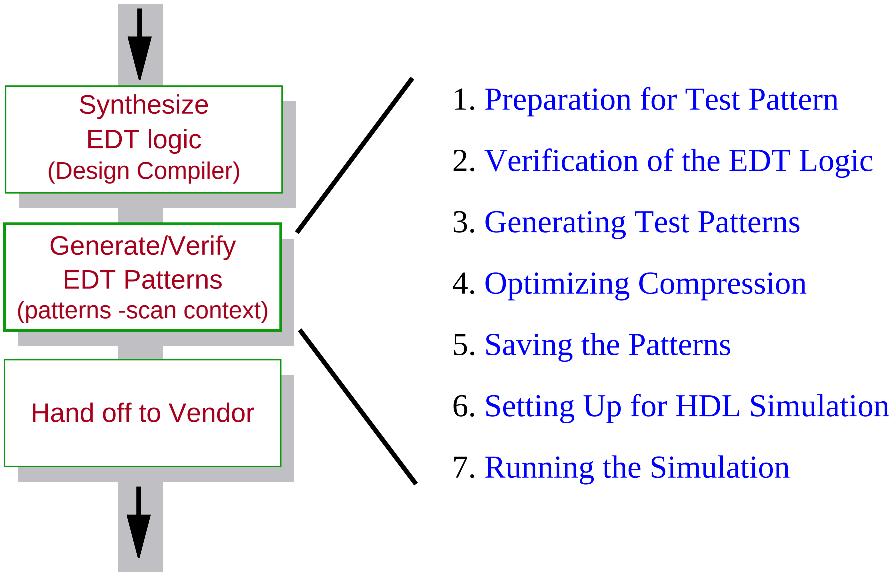
  Constraints specified using a form other than *pin_pathname* are converted into structurally reachable pins at the boundary of library cells that contain the target sequential element. This includes all non-clock input pins for *set_false_path –to* and all output pins for *set_false_path –from* and *set_case_analysis*.

  Constraints specified using –clock and –chain are translated into individual sequential elements. All constraints except TX are translated to *set_false_path –from* and *set_false_path –to.*

Chapter 6
Generating/Verifying Test Patterns

This chapter describes how to generate compressed test patterns. In this part of the flow, you generate and verify the final set of test patterns for the design. Figure 6-1 shows the layout of this chapter and the processes involved.

Figure 6-1. Test Pattern Generation and Verification Procedure



1. Preparation for Test Pattern
2. Verification of the EDT Logic
3. Generating Test Patterns
4. Optimizing Compression
5. Saving the Patterns
6. Setting Up for HDL Simulation
7. Running the Simulation

After you insert I/O pads and boundary scan and synthesize the EDT logic, invoke Tessent Shell with the synthesized top-level netlist and generate compressed test patterns.

Note
You can write test patterns in a variety of formats including Verilog and WGL.

Preparation for Test Pattern Generation

Note
You can reuse uncompressed ATPG dofiles, with the addition of some EDT-specific commands, to generate compressed patterns with the same test coverage as the original uncompressed patterns. You cannot directly reuse pre-computed, existing ATPG patterns.

To prepare for EDT pattern generation, check that EDT is on, and configure the tool the same as when you created the EDT logic. For example, if you create the EDT logic with one scan channel, you must generate test patterns for circuitry with one channel.

Note

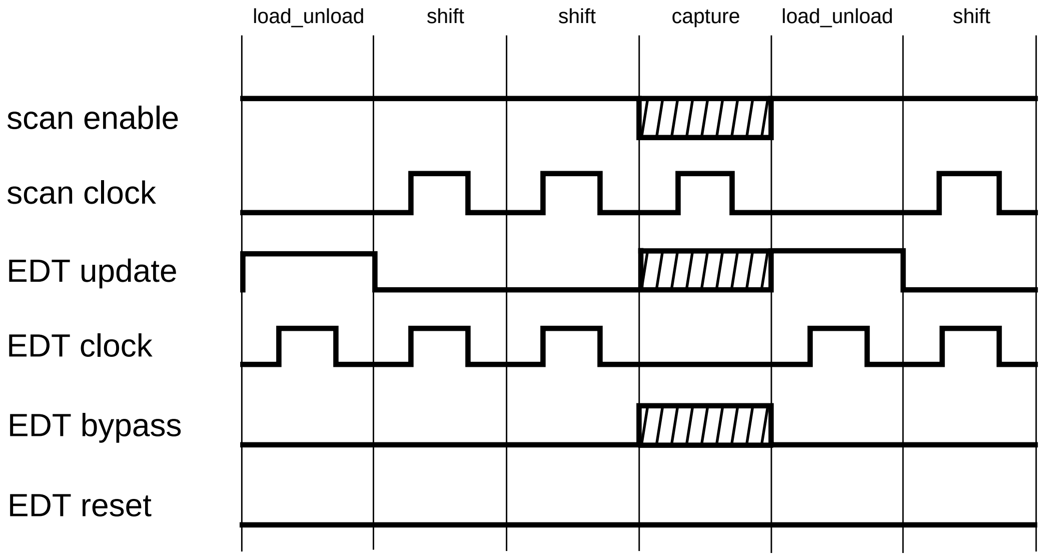
DRC violations occur if you attempt to generate patterns for a different number of scan channels than what the EDT logic is configured for.

You must also add_scan_chains in the same order they were added to the EDT logic. To reliably add the correct number of chains in the correct order in the Pattern Generation phase, you should use the setup dofile generated when the EDT logic was created. You can customize the dofile as needed.

The report_scan_chains command lists the scan chains in the same order they were added originally. For additional information, refer to the next section, "The Generated Dofile and Procedure File."

Compared to when you generated uncompressed test patterns with the scan-inserted core design (see ""ATPG Baseline Generation" on page 44"), there are certain differences in the tool setup. One of the differences arises because in the Pattern Generation phase you need to set up the patterns to operate the EDT logic. This is done by exercising the EDT clock, update and bypass (if present) control signals as illustrated in Figure 6-2.

Figure 6-2. Sample EDT Test Procedure Waveforms



Prior to each scan load, the EDT logic needs to be reset. This is done by pulsing the EDT clock once while EDT update is high.

During shifting, the EDT clock should be pulsed together with the scan clock(s). In Figure 6-2, both scan enable and EDT update are shown as 0 during the capture cycle. These two signals can have any value during capture; they do not have to be constrained. On the other hand, the EDT clock must be 0 during the capture cycle. The operation of these signals is described in the load_unload and shift procedures in the test procedure file the generated with the EDT logic. An example of this file is shown in "“Test Pattern Generation Files” on page 86."

On the command line or in a dofile, you must do the following:

- Identify the EDT clock signal as a clock and constrain it to the off-state (0) during the capture cycle. This ensures the tool does not pulse it during the capture cycle.
- Use the -*Internal* option with the add_scan_chains command to define the compressed scan chains as internal, as opposed to external channels. This definition is different from the definition you used to create the EDT logic because the scan chains are now connected to internal nodes of the design and not to primary inputs and outputs. Also, scan_in and scan_out are internal nodes, not primary inputs or outputs.
- If your design includes uncompressed scan chains. Uncompressed scan chains are chains not defined with the add_scan_chains command when setting up the EDT logic and whose scan inputs and outputs are primary inputs and outputs. You must define each uncompressed scan chain using the add_scan_chains command *without* the -*Internal* switch during test pattern generation.
- If you add levels of hierarchy (due, for example, to boundary scan or I/O pads), revise the pathnames to the internal scan pins listed in the generated dofile. An example dofile with this modification is shown “Modifying the Dofile and Procedure File for Boundary Scan” on page 183.

The Generated Dofile and Procedure File

The first two setups described in the preceding section are included in the dofile generated with the EDT logic. For an example of this dofile, see “Test Pattern Generation Files” on page 86.

The test procedure file also needs modifications to ensure the EDT update signal is active in the load_unload procedure and the EDT clock is pulsed in the load_unload and shift procedures. These modifications are implemented automatically in the test procedure file output with the EDT logic as follows:

- The timeplate used by the shift procedure is updated to include the EDT clock.
- In this timeplate, there must be a delay between the trailing edge of the clock and the end of the period. Otherwise, a P3 DRC violation will occur.
- The load_unload procedure is set up to initialize the EDT logic and apply shift a number of times corresponding to the longest “virtual” scan chain (longest scan chain plus additional shift cycles) seen by the tester. The number of additional shift cycles is reported by the report_edt_configurations command.

Note

Additional shift cycles refers to the sum of the initialization cycles, masking bits (when using Xpress), and low-power bits (when using a low-power decompressor).

- The shift procedure is updated to include pulsing of the EDT clock signal and deactivation of the EDT update signal.
- The EDT bypass signal is forced to a logic low if the EDT circuitry includes bypass logic.

For an example of this test procedure file, refer to "Test Procedure File — The tool also writes a test procedure file for test pattern generation. The tool takes the test procedure file used for EDT logic creation and adds the test procedures necessary to drive the EDT logic." in the "Test Pattern Generation Files" section of Chapter 4.

Generated Bypass Dofile and Procedure File

The tool generates a dofile and an test procedure file you can use with Tessent FastScan to activate bypass mode and run regular ATPG. Examples of these files are shown in "Bypass Mode Files" on page 88." If your design includes boundary scan and you want to run in bypass mode, you must modify the bypass dofile and procedure file to work properly with the boundary scan circuitry.

Updating Scan Pins for Test Pattern Generation

You can use the set_edt_finder command to automatically find EDT logic and get updated scan pin information for test pattern generation. Use this procedure, before test pattern generation, to get updated scan pin information when your design hierarchy changes after the EDT logic is generated.

The set_edt_finder command identifies the EDT logic contained in the gate-level netlist and updates the I/O pins associated with scan chains.

Note

If the design changes affect clock or constrained pins listed in the dofiles, you must manually correct them. This limitation will be removed in future releases.

Prerequisites

- No scan chains are defined for test pattern generation. The set_edt_finder *on* command must be used to enable this feature before setting up the scan chains for test pattern generation.
- Gate-level Verilog netlist or flat model containing EDT logic.

Note

EDT Finder must be enabled before any internal scan chains are added and saved to the flat model. Otherwise, the flat model cannot be used with the EDT Finder in subsequent sessions.

Procedure

1. Invoke Tessent Shell. For example:

   ```
   <Tessent_Tree_Path>/bin/tessent -shell
   ```

   Tessent Shell invokes in setup mode.

2. Provide Tessent Shell commands. For example:

   ```
   set_context patterns -scan
   read_verilog my_gate_scan.v
   read_cell_library my_lib.atpg
   set_current_design top
   ```

3. Enable EDT Finder. For example:

   ```
   set_edt_finder on -verbose on
   ```

   The EDT Finder feature is enabled.

4. Set up for test pattern generation as needed. For more information, see “Preparation for Test Pattern Generation” on page 109.

5. Exit setup mode. For example:

   ```
   set_system_mode analysis
   ```

   The EDT logic and internal scan chain inputs are identified, scan chains are traced, and DRC is run.

6. Correct any DRC violations.

   For information on DRCs related to the EDT Finder command, see EDT Finder (F Rules) in the *Tessent Shell Reference Manual*.

7. Report the EDT Finder results. For example:

   ```
   report_edt_finder -decompressors
   ```

   ```
   // id  #bits #inputs #chains EDT block   type
   // ------------------------------------------
   //  1     16       4      28  m1_28x16 active
   //  2     16       4       4   m2_4x32 active
   //  3     16       4      50 m3_50x187 active
   //  4     10       1       8   m4_8x16 active
   ```

   All active decompressors are reported. For more information on reporting EDT Finder results, see the report_edt_finder command.

Tip: You can also use the Test Structures Window within DFTVisualizer to browse the EDT logic after set_edt_finder is run.

8. Generate and save test patterns. For more information, see "Generating Test Patterns" on page 120.

Examples

The following example shows the output of the EDT finder command for a simple example with two channels and eight internal scan chains. The new output is highlighted in bold.

```
set_edt_finder on
dofile results/generated_edt.dofile
add_scan_groups grp1 results/generated_edt.testproc
add_scan_chains -internal chain1 grp1 /top_i/si1 /top_i/so1

//  Note: Ignoring pin names from add_scan_chains -internal command. Will
use learned internal scan inputs and scan outputs since EDT finder is
active.

add_scan_chains -internal chain2 grp1 /top_i/si2 /top_i/so2
set_edt_pins update EdtUpdate
set_edt_pins input_channel  1 ChannelInput_1
set_edt_pins output_channel 1 ChannelOutput_1
set_edt_pins input_channel  2 ChannelInput_2
set_edt_pins output_channel 2 ChannelOutput_2
set_mask_decoder connection -mode_bit 1 5
...
set_system_mode analysis

// Flattening process completed,  design_cells=221  leaf_cells=221
library_primitives=337  sim_gates=471  PIs=16  POs=11  CPU time=0.00 sec.
//  --------------------------------------------------------------------
//  Begin circuit learning analyses.
//  --------------------------------------------------------------------
...
//  --------------------------------------------------------------------
//  Begin EDT finder analyses.
//  --------------------------------------------------------------------
//  EDT Finder completed, EDT blocks=1, scan chains=8, CPU time=0.02 sec.
//  --------------------------------------------------------------------
//  Begin scan chain identification process, memory elements = 54.
//  --------------------------------------------------------------------
...
```

The following example shows the EDT Finder command output for the modular flow.

```
set_edt_mapping on
set_edt_finder on
add_edt_blocks mx8051

//  mx8051 is the current EDT block.
```

```
dofile data/mx8051_block_edt.dofile
add_scan_groups grp1 data/mx8051_block_edt.testproc
add_scan_chains -internal chain1 grp1 …

//  Note: Ignoring pin names from add_scan_chains -internal command. Will
use learned internal scan inputs and scan outputs since EDT finder is
active.
add_scan_chains -internal chain2 grp1 ...
...
set_system_mode analysis

//  Warning: Rule FN1 violation occurs 5 times
//  Warning: Rule FP13 violation occurs 7 times
//  Flattening process completed,  design_cells=13690  leaf_cells=1606
library_primitives=27093  netlist_primitive=162  sim_gates=26545  PIs=238
POs=382  CPU time=0.11 sec.
//  -------------------------------------------------------------------
//  Begin circuit learning analyses.
//  -------------------------------------------------------------------
...
//  -------------------------------------------------------------------
//  Begin EDT finder analyses.
//  -------------------------------------------------------------------
//  EDT Finder completed, EDT blocks=3, scan chains=48, CPU time=1.02 sec.
...
//  -------------------------------------------------------------------
//  Begin scan chain identification process, memory elements = 1798.
//  -------------------------------------------------------------------
...
```

Related Topics

EDT Finder (F Rules)

Verification of the EDT Logic

Two mechanisms are used to verify that the EDT logic works properly: design rules checking (DRC) and enhanced chain and EDT logic (chain+EDT logic) test.

The next two sections describe these mechanisms.

Design Rules Checking (DRC)

Several K DRCs verify the EDT logic operates correctly. F rules also verify the EDT logic when EDT Finder is on.

The tool provides the most complete information about violations of these rules when you have preserved the EDT logic structure through synthesis. Following is a brief summary of just the K rules that verify operation of the EDT logic:

- **K19** — simulates the decompressor netlist and performs diagnostic checks if a simulation-emulation mismatch occurs.

- **K20** — identifies the number of pipeline stages within the compactors, based on simulation.
- **K22** — simulates the compactor netlist and performs diagnostic checks if a simulation-emulation mismatch occurs.

For detailed descriptions of all the EDT design rules (K and F rules) checked during DRC, refer to the “Design Rules Checking” chapter of the *Tessent Shell Reference Manual*.

EDT Logic and Chain Testing

In addition to performing DRC verification of the EDT logic, the tool saves, as part of the pattern set, an EDT logic and chain test. This test consists of several scan patterns that verify correct operation of the EDT logic and the scan chains when faults are added on the core or on the entire design. This test is necessary because the EDT logic is not the standard scan-based circuitry that traditional chain test patterns are designed for. The EDT logic and chain test helps in debugging simulation mismatches and guarantees very high test coverage of the EDT logic.

You can use the following equation to predict the number of additional chain test patterns the tool generates to test the EDT logic. (In this equation, *ceil* indicates the ceiling function that rounds a fraction to the next highest integer.) Note, this equation provides a lower bound; the actual number may be higher.

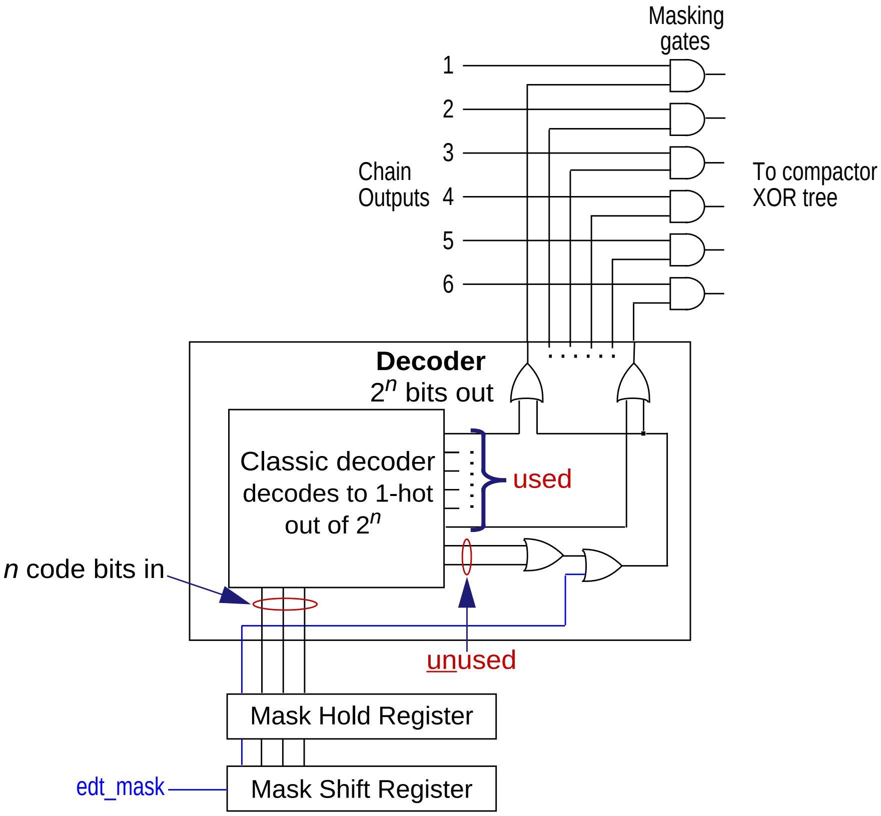
$$\text{Minimum number of chain test patterns} = 1 + 2^{ceil(\log 2(\text{number of chains}))}$$

How it Works

To better understand the enhanced chain test, you need to understand how the masking logic in the compactor works. Included in every EDT pattern are mask codes that are uncompressed and shifted into a mask shift register as the pattern data is shifted into the scan chains. Once a pattern’s mask codes are in the mask shift register, they are parallel loaded into a hold register that places the bit values on the inputs to a decoder. Figure 6-3 shows a conceptual view of the decoder circuitry for a six chains/one channel configuration.

The decoder basically has a classic binary decoder within it and some OR gates. The classic decoder decodes its n inputs to one-hot out of 2^n outputs. The 2^n outputs fall into one of two groups: the “used” group or the “unused” group. (Unless the number of scan chains exactly equals 2^n, there will always be one or more unused outputs.)

Figure 6-3. Example Decoder Circuitry for Six Scan Chains and One Channel



Each output in the used group is AND'd with one scan chain output. For a masked pattern, the decoder typically places a high on one of the used outputs, enabling one AND gate to pass its chain's output for observation.

The decoder also has a single bit control input provided by the edt_mask signal. Unused outputs of the classic decoder are OR'd together and the result is OR'd with this control bit. If any of the OR'd signals is high, the output of the OR function is high and indicates the pattern is a non-masking pattern. This OR output is OR'd with each *used* output, so that, for a non-masking pattern, all the AND gates will pass their chain's outputs for observation.

The code scanned into the mask shift register for each channel of a multiple channel design determines the chain(s) observed for each channel. If the code scanned in for a channel is a non-masking code, that channel's chains are all observed. If a channel's code is a masking code, usually only one of the chains for that channel is observed. The chain test essentially tests for all possible codes plus the edt_mask control bit.

The EDT logic and chain test for a 10X configuration has a minimum of 24 patterns. These 24 patterns can be composed of the following masking and non-masking patterns: one all chains masked pattern, ten masking patterns, seven masking patterns that observe all chains due to unused codes, two non-masking patterns that observe all chains, and four XOR patterns that observe a set of chains

The actual set of chain test patterns depends on how many chains each channel has. For example, if you have one channel and ten chains, the composition of the chain test will be:

- Pattern 0 — All masking patterns. Control bit is set to 1.
- Patterns 1 through 10 — Masking patterns. Control bit is set to 1. Only one chain will be observed per channel, due to "used" codes for each channel.
- Patterns 11 through 17 — Masking patterns. Control bit is set to 1. All chains will be observed due to "unused" codes.
- Patterns 18 and 19 — Non-masking patterns that observe all chains. Control bit is set to 0.
- Pattern 20 through 23 — XOR masking patterns that observe a set of chains. Control bit is set to 0.

You can clearly see this in the ASCII patterns. For a masking pattern, if the scanned in code corresponding to a channel is a "used" code, only one of that channel's chains will have binary expected values. All other chains in that channel will have X expected values. To see an example of a masked ASCII pattern, refer to "Understanding Scan Chain Masking in the Compactor" on page 218.

Note

When the tool generates the EDT chain patterns, in addition to non-masking and masking (1-hot and xor) patterns, the tool also generates chain test patterns to test all unused decoder values when all chain patterns are selected. Because of this, the total sum of all generated patterns may be greater than the sum of non-masking and masking patterns.

So, depending on which chain test is failing, it is possible to deduce which chain might be causing problems. In the preceding example, if a failure occurred for any of the patterns 2 through 10, you could immediately map it back to the failing chain and, based on the cycle information, to a failing cell. For pattern 11, if channel 1 had a failure, you similarly could map it back to a chain and a cell. If only a non-masking pattern or a masking pattern with "unused" codes failed, then mapping is a little bit tricky. But in this case, most likely masking patterns would fail as well.

Controlling the edt_update Signal for Load_Unload

When the tool generates chain test patterns, it adds an extra cycle to the end of the shift cycles before the load_unload procedure when neither the edt_clock or system clock is pulsed. This "dead" cycle guarantees the edt_update signal goes high during load_unload regardless of how you choose to control the edt_update signal.

For example, if you do not explicitly force edt_update high during load_unload because it has a C1 pin constraint, the STIL pattern file keeps edt_update low during load_unload unless the extra cycle is specified.

You can choose to remove this cycle from the pattern file using set_chain_test -*suppress_capture_cycle on*. However, you should use CAUTION when using this command option. If you remove the extra cycle and do not explicitly force edt_update high in the load_unload procedure, the pattern file will be incorrect and edt_update will be low during load_unload.

Coverage for EDT Logic and Chain Test

Experiments performed by Mentor Graphics engineers using sequential fault simulation demonstrate that test coverage for the EDT logic with the enhanced chain test is nearly 100% when the EDT logic does not include bypass logic (essentially multiplexers that bypass the decompressor and compactor). Test coverage declines to just above 94% when the EDT logic includes bypass logic. This is because the EDT chain test does not test the bypass mode input of each bypass multiplexer (edt_bypass is kept constant in EDT mode during the chain test).

Note

99+% coverage can be achieved in any event by including a bypass mode chain test (the standard chain test).

The size of the chain test pattern set depends on the configuration of the EDT logic and the specific design. Typically, about 18 chain test patterns are required when you approach 10X compression.

Reducing Serial EDT Chain Test Simulation Runtime

You can simulate a small subset of the chain test patterns serially. If you are not using and enabling the low-power decompressor, you can simply save one non-masking pattern as shown here:

```
set_chain_test -type nomask
write_patterns pattern_filename -pattern_sets chain -serial -end 0
```

If you are using a low-power decompressor, it is safest to run all non-masking patterns (which is still a small subset of all chain patterns) as shown here:

```
set_chain_test -type nomask
write_patterns pattern_filename  -pattern_sets chain -serial
```

For more information, see the set_chain_test command in the *Tessent Shell Reference Manual*.

Adding Faults on the Core Only is Recommended

When you generate patterns, if you add faults on the entire design, the tool tries to target faults in the EDT logic. Traditional scan patterns can probably detect most EDT logic faults. But because EDT logic fault detection cannot be serially simulated, the tool conservatively does not give credit for them. This results in a relatively high number of undetected faults in the EDT logic being included in the calculation of test coverage. You, therefore, see a lower reported test coverage than is actually the case.

The EDT logic and chain test targets faults in the EDT logic. The tool always performs the this test, so adding faults on the entire design is not necessary in order to get EDT logic test coverage. To avoid false test coverage reports, the best practice is to add faults on the core only.

Generating Test Patterns

The compression technology supports all of the pattern functionality in uncompressed ATPG, with the exception of MacroTest and random patterns. This includes combinational, clock-sequential (including patterns with multiple scan loads), and RAM sequential patterns. It also includes all the fault types. When you generate test patterns, you should use the dofile and test procedure files the tool generated during logic creation. If you added boundary scan, you will need to modify the files as explained in the section, "Modifying the Dofile and Procedure File for Boundary Scan."

To create the EDT logic, you invoked Tessent Shell with the core level of the design. To generate test patterns, you invoke Tessent Shell with the synthesized top level of the design that includes synthesized pads, boundary scan, if used, and the EDT logic. Here is an example invocation of Tessent Shell with a Verilog file named *created_edt_top.v*, assumed here to be the top-level file generated when the EDT logic was created:

Invoke Tessent Shell:

```
<Tessent_Tree_Path>/bin/tessent -shell
```

You are automatically placed in setup mode. Specify the context for generating test patterns and load the Verilog file and library:

```
set_context patterns -scan
read_verilog created_edt_top.v
read_cell_library my_atpg_lib
set_current_design top
```

For a description of how the *created_edt_top.v* file is generated, refer to the Chapter 4 section, "Creation of EDT Logic Files." Next, you need to set up for EDT pattern generation. To do this, execute the dofile. For example:

```
dofile created_edt.dofile
```

For information about the EDT-specific contents of this dofile, refer to "Test Pattern Generation Files" in Chapter 4. Enter analysis mode and verify that no DRC violations occur. Pay special attention to the EDT DRC messages.

```
set_system_mode analysis
```

Now, you can enter the commands to generate the EDT patterns. If you ran uncompressed ATPG on just the core design prior to inserting the EDT logic, it is useful to add faults on just the core now to enable you to make valid comparisons of test performance using EDT versus not using EDT.

```
add_faults /my_core   // Only target faults in core
create_patterns
report_statistics
report_scan_volume
```

Another reason to add faults on the core is to avoid incorrectly low reported test coverage, as explained earlier in "Adding Faults on the Core Only is Recommended."

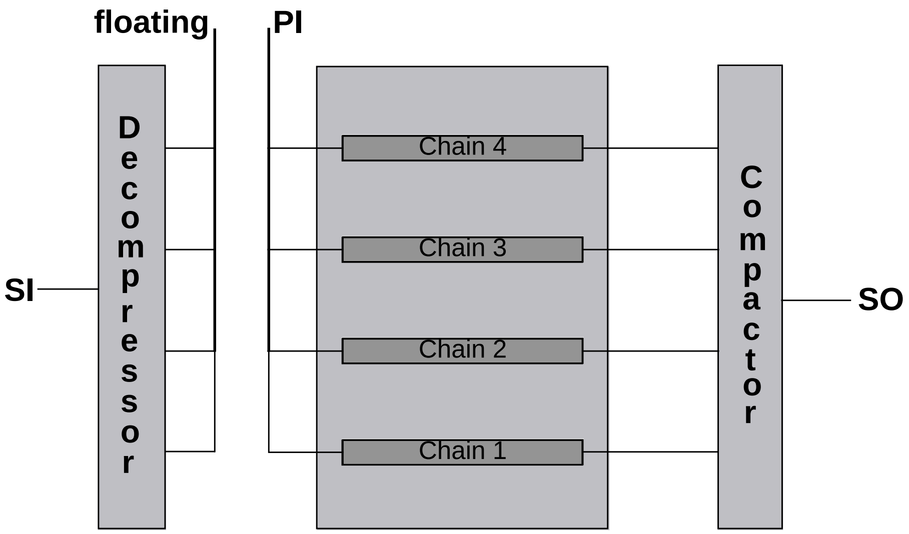
The report_scan_volume command provides reference numbers when analyzing the achieved compression.

Note

If you reorder the scan chains after you generate EDT patterns, you must regenerate the patterns. This is true even if the EDT logic has not changed. EDT patterns cannot be modified manually to accommodate the reordered scan chains.

Figure 6-4 illustrates an important characteristic of the EDT logic during test pattern generation.

Figure 6-4. Circuitry in the Pattern Generation Phase



There is no physical connection between the decompressor and the internal scan chains, as seen within the tool in the Pattern Generation phase. This modification occurs only within the tool, as a result of the "add_scan_chains -internal" command. The tool does not modify the external netlist in any way.

If EDT Finder is on, the tool traces through the compressor/decompressor logic and maintains the connections. If EDT Finder is Off, the tool breaks the connection and turns the internal scan chain inputs into PIs as a means of controlling the values the ATPG engine can place on them. The tool exercises this control by constraining these PIs to Xs at certain points during pattern generation, thereby preventing interference with values being output by the decompressor. It does this while emulating the behavior of an unbroken connection between the decompressor and the scan chains.

Note

If you report_primary_inputs, the scan chain inputs are reported in lines that begin with "USER:". This is important to remember when you are debugging simulation mismatches.

Optimizing Compression

You can do a number of things to ensure maximum compression:

- Limit observable Xs
- Use Dynamic Compaction

Using Dynamic Compaction

You should use dynamic compaction during ATPG if your primary objective is a compact pattern set. Dynamic compaction helps achieve a significantly more compact pattern set, which is the ultimate goal of using EDT. Because the two compression methods are largely independent of each other, you can use dynamic compaction and EDT concurrently. Try to use create_patterns for the smallest pattern set, as it executes a good ATPG compression flow that is optimal for most situations.

Note

For circuits where dynamic compaction is very time-consuming, you may prefer to generate patterns without dynamic compaction. The test set that is generated is not the most compact, but it is typically more compact than the test set generated by traditional ATPG with dynamic compaction. And it is usually generated in much less time.

Saving the Patterns

Save EDT test patterns in the same way you do in uncompressed ATPG. For complete information about saving patterns, refer to the write_patterns command in the *Tessent Shell Reference Manual*.

Serial Patterns

One important restriction on EDT serial patterns is that the patterns must not be reordered after they are written. Because the padding data for the shorter scan chains is derived from the scan-in data of the next pattern, reordering the patterns may invalidate the computed scan-out data. For more detailed information on pattern reordering, refer to the section, "Reordering Patterns" in Chapter 7.

Parallel Patterns

Because parallel simulation patterns force and observe the uncompressed data directly on the scan cells, they have to be written by the EDT technology which understands and emulates the EDT logic.

Some ASIC vendors write out parallel WGL patterns, and then convert them to parallel simulation patterns using their own tools. This is not possible with default EDT patterns, as they provide only scan *channel* data, not scan chain data. To convert these patterns to parallel simulation patterns, a tool must understand and emulate the EDT logic.

There is an optional switch, -Edt_internal, you can use with the write_patterns command to write parallel EDT patterns with respect to the core scan chains. You can write these patterns in tester or ASCII format and use them to produce parallel simulation patterns as described in the next section.

EDT Internal Patterns

The optional -Edt_internal switch to the write_patterns command enables you to save parallel patterns as EDT internal patterns. These are tester or ASCII formatted EDT patterns that the tool writes with respect to the core scan chains instead of with respect to the top-level scan channel PIs and POs. These patterns contain the core scan chain force and observe data with the exception that they have X expected values for cells which would not be observed on the output of the spatial compactor due to X blocking or scan chain masking. X blocking and scan chain masking are explained in the Chapter 7 section, "Understanding Scan Chain Masking in the Compactor."Also, of course, the scan chain force and observe points are internal nodes, not top-level PIs and POs. Because they provide data with respect to the core scan chains, EDT internal patterns can be converted into parallel simulation patterns.

Note

The number of scan chain inputs and outputs in EDT internal patterns corresponds to the number of scan chains in the design core, *not* the number of top-level scan channels. Also, the apparent length of the chains, as measured by the number of shifts required to load each pattern, will be shorter because the extra shift cycles that occur in normal EDT patterns for the EDT circuitry are unnecessary.

Post-Processing of EDT Patterns

Sometimes there is a need to process patterns after they are written to a file. Post-processing might be needed, for example, to control on-chip phase-locked loops (PLLs). Scan pattern post-processing requires access to the uncompressed patterns. The tool, however, writes patterns in EDT-compressed format, at which point it is too late to make any changes. Traditional post-processing, therefore, is not feasible with EDT patterns.

Note

An exception is parallel tester or ASCII patterns you write out as EDT *internal* patterns. Using your own post-processing tools, you can convert these patterns into parallel simulation patterns. See "Parallel Patterns" on page 123 for more information.

The compressed ATPG engine must set or constrain any scan cells prior to compressing the pattern. So it is essential you identify the type of post-processing you typically need and then translate it into functionality you can specify in the tool as part of your setup for pattern generation. The compressed ATPG engine can then include it when generating EDT patterns.

Simulating the Generated Test Patterns

You can verify the test patterns using parallel and serial test benches the same way you would for normal scan and ATPG. When you simulate serial simulation patterns, you can verify the correctness of the captured data for the pattern, the chain integrity, and the EDT logic (both the decompressor and the compactor blocks). When simulation mismatches occur, you can still use the parallel test bench to debug mismatches that occur during capture. You can use the serial test bench to debug mismatches related to scan chain integrity and the EDT logic.

To verify that the test patterns and the EDT circuitry operate correctly, you need to serially simulate the test patterns with full timing. Typically, you would simulate all patterns in parallel and a sample of the patterns serially. Only the serial patterns exercise the EDT circuitry. Because simulating patterns serially takes a long time for loading and unloading the scan chains, be sure to use the -*Sample* switch when you write_patterns for serial simulation. This is true even though serial patterns simulate faster with EDT than with traditional ATPG due to the fewer number of shift cycles needed for the shorter internal scan chains. The section, "Simulating the Design with Timing" in the *Tessent Scan and ATPG User's Manual* provides useful background information on the use of this switch. Refer to the write_patterns command description in the *Tessent Shell Reference Manual* for usage information.

Note

You must use Tessent Shell to generate parallel simulation patterns. You cannot use a third party tool to convert parallel WGL patterns to the required format, as you can for traditional ATPG. This is because parallel simulation patterns for EDT are uncompressed versions of the compressed EDT patterns applied by the tester to the scan channel inputs. They also contain EDT-specific modifications to emulate the effect of the compactor.

Setting Up for HDL Simulation

First, set up a work directory for ModelSim.

```
../modeltech/<platform>/vlib work
```

Then, compile the simulation library, the scan-inserted netlist, and the simulation test patterns. Notice that both the parallel and serial patterns are compiled:

```
../modeltech/<platform>/vlog my_parallel_pat.v my_serial_pat.v \
      ../created_edt_top_gate.v -y my_sim_lib
```

This will compile the netlist, all necessary library parts, and both the serial and parallel patterns. Later, if you need to recompile just the patterns, you can use the following command:

```
../modeltech/<platform>/vlog pat_p_edt.v pat_s_edt.v
```

Running the Simulation

After you have compiled the netlist and the patterns, you can simulate the patterns using the following commands:

```
../modeltech/<platform>/vsim edt_top_pat_p_edt_v_ctl -do "run -all" \
      -l sim_p_edt.log  -c

../modeltech/<platform>/vsim edt_top_pat_s_edt_v_ctl -do "run -all" \
      -l sim_s_edt.log  -c
```

The “-c” runs the ModelSim simulator in non-GUI mode.

Chapter 7
Special Topics

This chapter describes advanced features of compressed ATPG. For more information, see the following topics:

Low-Power Test

Compressed ATPG with EDT can be configured to use low power during capture and/or shift cycles. When configured for low power, both EDT mode and bypass modes are affected.

A *low-power shift* application is based on the fact that test patterns typically contain only a small fraction of test-specific bits and the remaining scan cells or *don't care bits* are randomly filled with 0s and 1s; so, there are only a few scan chains with specified bits. In a low-power application, scan chains without any specified bits are filled with a constant value (0) to minimize needless switching as the test patterns are shifted through the core. For more information, see "Low-Power Shift."

A *low-power capture* application is based on the existing clock gaters in a design. In this case, clock gaters controlling untargeted portions of the design are turned off, while clock gaters controlling targeted portions are turned on. Power is controlled most effectively in designs that employ clock gaters, especially multiple levels of clock gaters (hierarchy), to control a majority of the state elements. Configuring low-power capture affects only the test patterns and is enabled with the set_power_control command during ATPG.

Note
Low-power constraints are directly related to the number of test patterns generated in a low-power application. For example, using stricter low-power constraints results in more test patterns.

Low-Power Shift

Setting up low-power shift includes two phases:

1. **Inserting power controller logic** — The power controller logic is configured/inserted during EDT logic creation based on the -MIN_Switching_threshold_percentage *value* specified with the set_edt_power_controller *Shift* command. This *value* must fall into one of the three threshold ranges described in "Low-Power Shift and Switching Thresholds."

   For example: To enforce a 20% switching threshold for shift, (assume a worse case switching activity of 50% for scan chains driven by the decompressor), the power controller is configured to drive up to 40% of the scan chains as shown here:

   ```
   20% = 50% (max % scan chains to switch of total scan chains)
   20% = 50%(40%)
   ```

   The remaining scan chains (minimum of 60%) are loaded with a constant zero (0) value. So, in a case in which you have 300 scan chains, the maximum percentage of scan chains that will switch is 120, which is 40% of 300.

   For more information, see "Power Controller Logic" and "Low-Power Shift and Switching Thresholds."

2. **Creating low-power test patterns** — When test patterns are generated, you must enable the power controller and specify the low-power switching threshold used during scan chain shifting with the set_power_control and set_edt_power_controller *Shift* commands. The specified switching threshold should not exceed the power controller hardware capabilities; out-of-range thresholds are supported but will generate a warning.

   For example, if you configure the power controller hardware for a minimum switching threshold of 20%, you cannot set the test patterns to use a switching threshold of less than 12% or more than 24% as described in "Low-Power Shift and Switching Thresholds."

   In EDT bypass mode, the EDT logic and power controller are bypassed, and the low-power test patterns use a repeat-fill heuristic to load constant values into the *don't care bits* as they are shifted through the core. The repeat-fill heuristic minimizes needless transitions during bypass testing. This feature is only available in uncompressed ATPG or in the bypass mode of compressed ATPG.

Low-Power Shift and Switching Thresholds

The configuration/capability of the power controller hardware is determined by the *-MIN_Switching_threshold_percentage* value specified with the set_edt_power_controller command during EDT logic creation.

The *switching threshold percentage* is a percentage of the overall scan chain switching during shift. The *minimum switching threshold percentage* then represents the minimum switching threshold the power controller hardware can accommodate in a low-power application and determines the switching threshold percentage that can be used for test pattern generation.

You can use the following three threshold ranges to set up a low-power shift application. The bias value is set based on the threshold range you specify.

- < 12% (bias 2)
- >= 12% to < 25% (bias 1)
- >= 25% (bias 0)

Note
The term bias refers to “biased signal probability”, with a higher bias corresponding to an increase in the size of the power controller hardware.

If you specify a -MIN_Switching_threshold_percentage *value* that falls within one of these ranges, the tool generates a low-power controller that can generate shift patterns with low-power switching thresholds of the upper and lower bounds of the range. For example, if you specify a minimum threshold of 14, a low-power controller is generated that is capable of generating shift patterns with a low-power switching threshold of 12 to 24.

Both switching thresholds, the one for the power controller hardware and the one for low-power test patterns, must fall into the same switching threshold range. Low-power applications where power controller and test pattern thresholds fall in different ranges are not supported and may result in a higher test pattern count and decreased shift power control.

Note
If reaching the specified threshold causes a drop in test coverage, the threshold is violated to maintain coverage. You can use the set_power_control *-rejection_threshold* switch to specify a hard limit on the switching activity and disregard the test coverage impact.

Pattern Generation and Switching Thresholds

During pattern generation, you use the set_power_control command to (1) enable the low-power logic and (2) set the low-power switching threshold to be used during scan chain shifting. The switching threshold you specify cannot exceed the power controller hardware capabilities.

The switching thresholds for both the power controller hardware and the low-power test patterns must fall into the same switching threshold range. A warning message is issued when a mismatch occurs between the software switching threshold and the power controller hardware threshold. The following example is an example warning message that reports a mismatch between the switching percentage threshold specified for shift and that specified for pattern generation:

```
//  command: set_power_control shift on -switching_threshold_percentage 7
//  Warning: Specified software switching threshold [7] is not consistent
//  with the switching threshold used to generate the shift power control
//  hardware [30] in block odd.
//  The software and hardware thresholds should be in the same bias range
//  (except for the full control case). The following are the valid bias
//  ranges: [0-11], [12-24], [25-50].
```

Note

Low-power applications where power controller and test pattern thresholds fall into different ranges are not supported and may result in a higher test pattern count and decreased shift power control.

Low-Power Shift and Test Patterns

An additional test pattern (edt_setup) is added before every test pattern set. This test pattern sets up the low-power mask registers before the load of the very first real test pattern. Similarly, the first real test pattern carries the low-power mask setup for the second pattern and so on. The unload values of the edt_setup pattern are not observed.

Power Controller Logic

The power controller loads constant values into the *don't care bits* within scan chains as the test patterns are uncompressed and shifted into the core.

The power controller must be enabled in both the EDT logic hardware and the test pattern generation software to use low-power ATPG. By default, the power controller is enabled. For more information, see set_edt_power_controller. If you are not sure whether you need to use the low-power feature, you can insert a disabled controller and then, enable it if you need to lower power consumption. If you change the controller setting during pattern generation, remember to modify the generated test procedure file to force the shift_const_en signal to the appropriate value.

Note

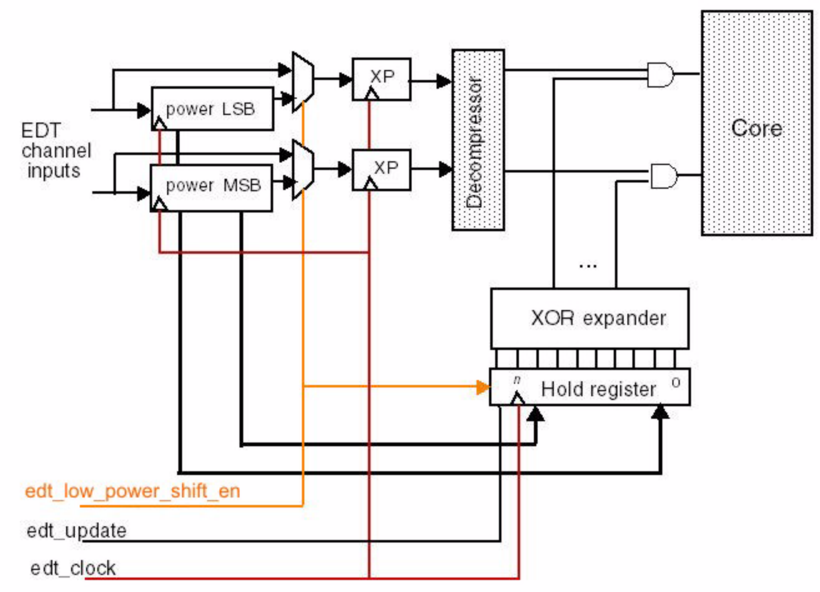
Low-power ATPG adds additional shift cycles to each test pattern, so the power controller should be disabled when it is not needed to prevent unnecessary cycles.

The *edt_low_power_shift_en* signal shown in Figure 7-1 controls the low-power controller as follows:

- When *edt_low_power_shift_en* is asserted, the power controller is enabled and a control code is generated by pipeline stages at the channel inputs. The control code is loaded into a hold register and applied to the XOR expander to control whether the biasing AND gates are enabled. If the control code is 1, the AND gate is enabled and the decompressor drives the scan chain; if the control code is a 0, the AND gate is disabled and the 0 logic source drives the scan chain.
- When the *edt_low_power_shift_en* signal is forced off, the power controller is disabled, the input pipeline stages are bypassed, the hold register is filled with 1s, and the decompressor drives all the scan chains. For information on disabling the power controller, see set_edt_power_controller.

For information on defining a signal for the power controller, see set_edt_pins.

Figure 7-1. Low Power Controller Logic



Static Timing Analysis and Hold Violations From Low-Power Hold Registers

Lockup cells are inserted on paths between the EDT decompressor and the scan cells to avoid clock skew issues. However, lockup cells are not required in the path between the low-power hold register and the first scan cell of each chain. This is because this path does not operate as a shift register due to the following:

- The low-power hold register only updates in the load_unload cycle, and the scan cells are not clocked in the load_unload cycle.
- The low-power hold register does not change during the shift cycle when the scan cells are clocked.

When you verify shift mode timing, both the edt_clock and the scan cell clocks are pulsed; this means that a static timing analysis (STA) tool will look for and report violations in the paths between low-power hold registers and the scan cells. You can prevent these violations from being reported by adding timing exceptions to your STA tool, directing it to ignore violations on these paths. An example of setting a timing exception is shown here:

```
set_multicycle_path -hold 1 \
-from [get_cells edt_i/edt_contr_i/low_power_shift_contr_i/low_power_hold_reg_*_reg*]
```

Related Topics

EDT Logic with Power Controller

Setting Up Low-Power Test

Setting Up Low-Power Test

This procedure creates EDT logic configured with an enabled power controller, and programs the power controller for the desired level of shift control during test pattern generation. This procedure also enables the low-power capture feature of the test patterns.

Prerequisites

- RTL or a gate-level netlist with scan chains inserted.
- DFT compression strategy for your design. A compression strategy helps define the most effective testing process for your design.

Procedure

1. Invoke Tessent Shell to perform EDT logic creation. For example:

   ```
   <Tessent_Tree_Path>/bin/tessent -shell
   ```

   Tessent Shell invokes in setup mode.

2. Set up for EDT logic creation. For example:

```
set_context dft -edt
read_verilog my_gate_scan.v
read_cell_library my_lib.atpg
set_current_design top
dofile edt_ip_creation.do
```

3. Define a power controller with a minimum switching threshold. For example:

```
set_edt_options power_controller shift \
-min_switching_threshold_percentage 20
```

   An enabled power controller with 20% minimum switching threshold is set up.

   If no minimum threshold is specified, 15% is used. For more information, see "Low-Power Shift and Switching Thresholds."

   You can use the set_edt_pins command to define a signal for the power controller.

4. Define the remaining EDT logic parameters. For more information, see "Parameter Specification for the EDT Logic."

5. Exit setup mode and run DRC. For example:

```
set_system_mode analysis
```

6. Correct any DRC violations.

7. Create the EDT logic. For example:

```
write_edt_files ../generated/low_power_enabled_edt -replace
```

8. Exit Tessent Shell. For example:

```
exit
```

9. Synthesize the EDT logic. For more information, see "Synthesizing the EDT Logic."

10. Invoke Tessent Shell in setup mode and then set context to perform test pattern generation.

```
set_context patterns -scan
```

11. Program the power controller switching threshold. For example:

```
set_power_control shift on -switching_threshold_percentage 20 \
        -rejection_threshold_percentage 25
```

   The switching during scan chain loading is minimized to 20% and any test patterns that exceed a 25% rejection threshold are discarded. For information on switching threshold constraints, see "Low-Power Shift and Switching Thresholds."

   By default, the switching threshold for ATPG is set to match the threshold used for the power controller hardware. For modular applications, the highest individual switching threshold is used.

12. Report the power controller and switching threshold status. For example:

    report_edt_configurations -all

    ```
    //  IP version:                  4
    //  External scan channels:      2
    //  Compactor type:              Xpress
    //  Bypass logic:               On
    //  Lockup cells:               On
    //  Clocking:                   edge-sensitive
    //  Low power shift controller: Enabled and active
    //  Min switching threshold:    20%
    ```

    Bold text indicates the output relevant to the power controller.

13. Turn on low-power capture. For example:

    **set_power_control capture on -switching_threshold_percentage 30 \
    -rejection_threshold_percentage 35**

    Switching during the capture cycle is minimized to 30% and any test patterns that exceed a 35% rejection threshold are discarded.

14. Exit setup mode and run DRC. For example:

    set_system_mode analysis

15. Correct any DRC violations.

16. Create test patterns. For example:

    create_patterns

    Test patterns are generated and the test pattern statistics and power metrics display.

17. Analyze reports, and adjust power and test pattern settings until power and test coverage goals are met. You can use the report_power_metrics command to report the capture and shift power usage associated with a specific instance or set of modules.

18. Save test patterns. For example:

    write_patterns ../generated/patterns_edt_p.stil -stil -replace

Related Topics

Low-Power Test

set_edt_power_controller

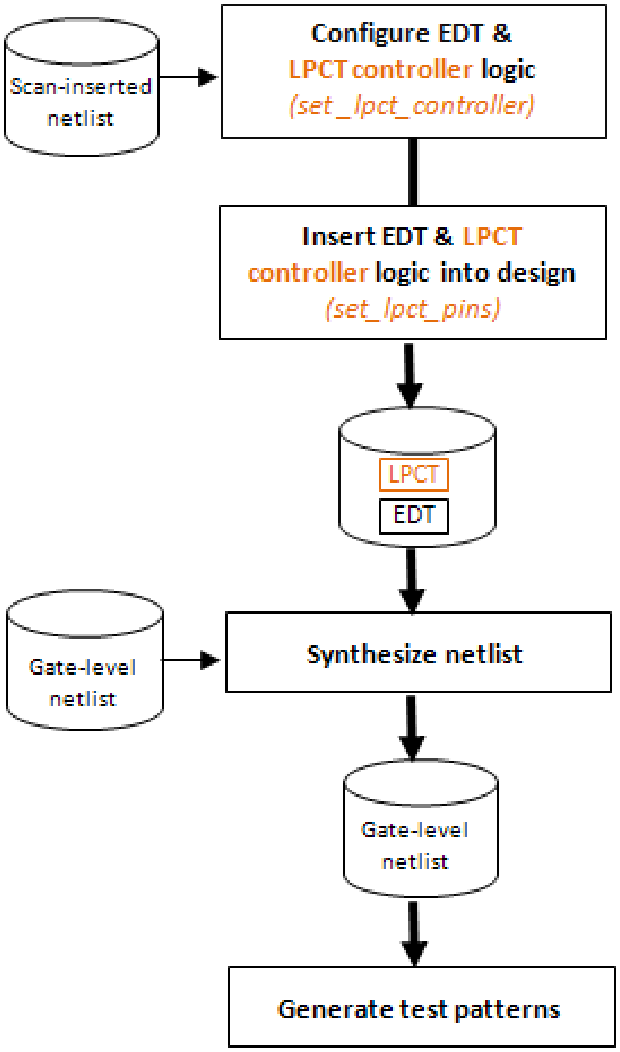
set_power_control

Low Pin Count Test Controller

The Low Pin Count Test (LPCT) controller minimizes the top-level pins required for the EDT application. The LPCT controller generates the control signals needed to operate the EDT logic and test the design core thereby making control signals from top-level pins unnecessary.

The LPCT controller is configured and embedded in the design along with the EDT logic according to the design process shown in Figure 7-2.

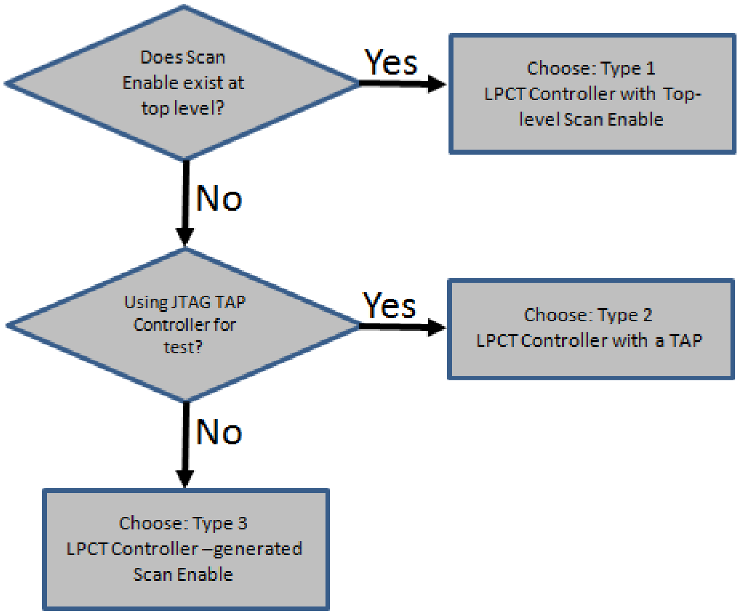
Figure 7-2. LPCT Design Process



LPCT Controller Decision Tree

You can implement an LPCT controller using one of three configuration types: Type 1, Type 2, or Type 3. These three LPCT configuration types are described in this section. Figure 7-3 illustrates the decision-making process for choosing which LPCT controller configuration fits your design constraints.

Figure 7-3. LPCT Controller Decision Tree


Does Scan Enable exist at top level?
Yes
Choose: Type 1 LPCT Controller with Top-level Scan Enable
No
Using JTAG TAP Controller for test?
Yes
Choose: Type 2 LPCT Controller with a TAP
No
Choose: Type 3 LPCT Controller –generated Scan Enable

Note

Be aware that the Type 3 LPCT controller requires on-chip controller (OCC) logic in the design. Additionally, the OCC logic must be capable of turning off the clocks for capture which is necessary for ATPG to detect reset faults. OCC logic is supported for Type 1 and Type 2 LPCT controllers as well, but is not a requirement.

Test Mode Clock Multiplexer Requirement

Internal capture clocks — If you are using an internal capture clock such as the output of a programmable clock controller with any of the LPCT controller types, you need a multiplexer to choose between the clock controller output used during test and the original functional clock source such as the output of a PLL. The tool does not add this multiplexer because the tool only knows the test clock source (i.e. clock controller output) and not the functional clock source (i.e. PLL output). Therefore, you must add a test mode clock multiplexer for all internal capture clocks for all three types of LPCT controller.

Shared LPCT and design clock — The test mode clock multiplexer is also needed when the LPCT clock is shared with the scan shift clock. In this case, you must add the test mode clock mux and provide the tool the connection point to connect the shift and capture clock that is generated from LPCT clock,

Figure 7-4 shows an example of how the test mode clock multiplexer can be inserted and connected in the design prior to LPCT logic generation and insertion.

Sharing of the LPCT Clock and a Top-Level Scan Clock

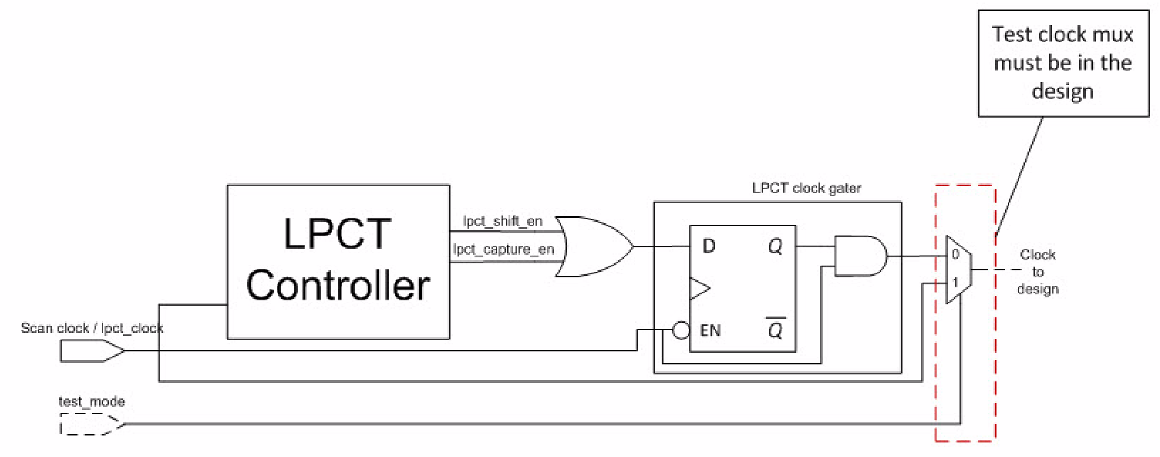
The Type 3 LPCT controller requires a dedicated LPCT clock that is different from other top-level scan clocks. The Type 1 and Type 2 LPCT controllers can use the top-level scan clock that is used for both shift and capture cycles as the LPCT clock also.

When an internal scan clock is used, the tool inserts a clock mux to choose between the controller-generated shift clock during shift and the original internal clock during capture. For top-level clocks, the tool does not insert a mux because the clock can be controlled as needed during both shift and capture.

When a top-level scan clock is used as the LPCT clock and the -shift_control option is set to “clock”, the tool adds a clock gater for Type 1 and Type 2 controllers as shown in Figure 7-4. The clock gater is not added when it generates a shift enable signal. This clock gater is enabled for all shift and capture cycles, but disabled during the pre-shift and post-shift cycles.

The clock input of the clock gater is connected to the top-level clock, so ATPG has full control of the scan clock during capture. This allows ATPG to turn off the capture clock, for example when detecting asynchronous reset faults. Reusing a top-level scan clock as the LPCT clock is inferred when the defined LPCT clock is pulsed during shift in the incoming logic creation test procedure file.

Figure 7-4. Clock Gater for Sharing LPCT Clock with Top-Level Scan Clock



Shift Clock Control for LPCT Controller

All LPCT controller configuration types have the ability to generate and control the shift clocks. You should specify the option of shift clock control that is compatible with your design. To specify how the LPCT controller generates the shift clock control signal, use following command and options:

set_lpct_controller -shift_control {<u>Enable</u> | Clock | None}

The three options for specifying shift control are defined as follows:

- **Enable** — The LPCT controller generates the lpct_shift_en enable signal to generate the shift clock. You can use this enable signal to create the shift clock for the design by gating it with a free-running clock. In this case, you must define the connections from the enable signal to the clock control logic. This is the default setting.
- **Clock** — The LPCT controller generates the shift clock. All necessary connections and gating are added so that shift clocks are controlled and driven from the LPCT controller. In the case of internal capture clocks, when the LPCT clock is shared with the scan clock and this option is used, the tool adds a clock gater in the clock path. In this case, the LPCT clock is used as both a shift clock and a capture clock.
- **None** — No signal is generated. You should use this option when shift clocks are available at the top level.

Type 1 LPCT Controller

Configuration: Uses a top-level scan enable pin to generate the dynamic EDT signals edt_update and edt_clock.

Requirements: A top-level scan_en signal and a top-level lpct_clock signal.

LPCT Controller Configuration	Required Inputs	Generated Outputs
Type 1	scan_en lpct_clock	edt_clock edt_update lpct_capture_en lpct_shift_clock OR lpct_shift_en lpct_clock_mux_select

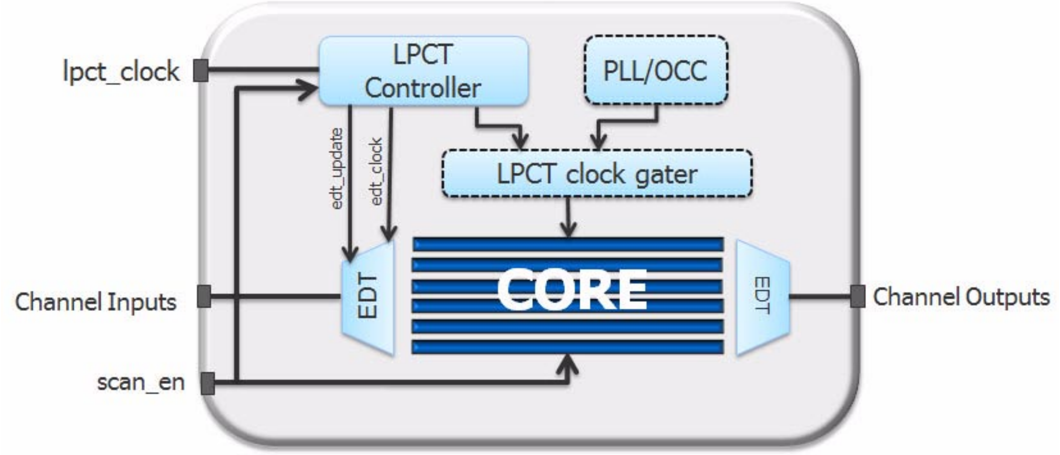
Description: If your design has a top-level scan enable pin, you can implement the Type 1 LPCT controller to generate the dynamic test signals edt_clock, edt_update, and shift clock from the scan_en and lpct_clock signals. All other static EDT-specific test signals (edt_bypass, edt_low_power_shift_en, and so on) are assumed to be available either from the top level or through user-provided test logic. You can choose to control them by some internal test data register. This LPCT controller does not generate any hardware to control any of these static signals.

Note

OCC logic is optional for the Type 1 LPCT controller.

Figure 7-5 shows the configuration of the Type 1 LPCT controller. For an in-depth description, see "Type 1 - LPCT Controller with Top-level Scan Enable".

Figure 7-5. Type 1 LPCT Controller Configuration



Hardware area: The LPCT controller logic is approximately equal to 14 NAND gates and is independent of design size or test application.

Command: To generate a Type 1 controller use the following command:

```
set_lpct_controller on -generate_scan_enable off -tap_controller_interface off
```

Table contains additional commands and switches that apply to the Type 1 controller.

LPCT Controller Type 1 Commands and Switches

To generate a Type 1 LPCT Controller, use:	set_lpct_controller	set_lpct_pins	set_lpct_condition_bits
set_lpct_controller -generate_scan_enable Off -tap_controller_interface Off	-shift_control	clock input_scan_en (input) clock_mux_select (output) capture_en (output) shift_en (output) shift_clock (output) test_clock_connection (output)	None

Note

When EDT channel outputs are shared with functional output pins, the tool adds an output channel sharing mux. The select signal of this mux is the scan enable signal specified using the “set_edt_pins scan_en” command. If you do not specify the scan enable signal using the set_edt_pins command for the Type-1 LPCT controller, the tool uses the specified LPCT input scan enable pin as the select signal for the mux.

Type 2 LPCT Controller

Configuration: Uses a TAP state machine to generate scan_enable and the dynamic EDT signals edt_update and edt_clock.

Requirement: 1149.1 TAP controller that is P1687 compliant:

LPCT Controller Configuration	Required Inputs	Generated Outputs
Type 2	tck test_mode test_logic_reset update_dr shift_dr capture_dr	scan_en edt_clock edt_update lpct_capture_en lpct_shift_clock OR lpct_shift_en lpct_clock_mux_select

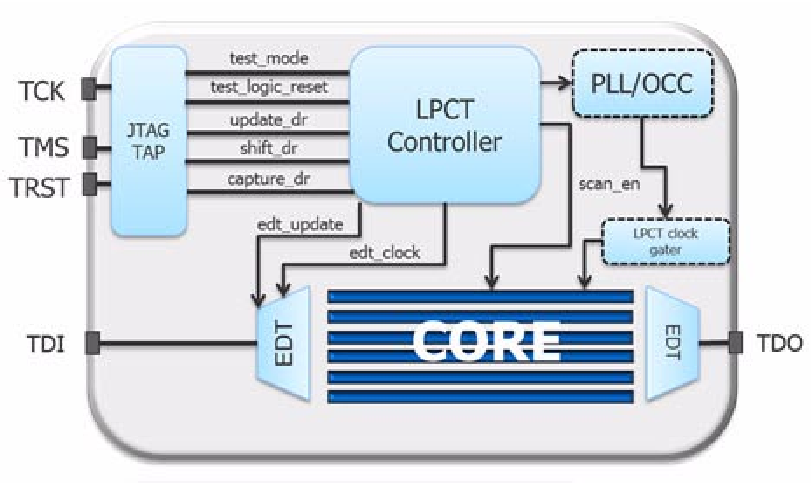
Note

For the Type 2 LPCT controller, the top-level scan enable pin is removed and the internally-generated scan enable pin is used.

Description: If your design uses a 1149.1 JTAG TAP controller at the top level to run compression, you can implement the Type 2 controller to generate scan_en, edt_update and edt_clock on chip. All other static test signals can be controlled by the TAP controller. The LPCT controller only uses the shift_dr, capture_dr, update_dr, test_logic_reset and test_mode signals from the TAP controller. All other EDT-specific static signals (edt_bypass, edt_low_power_shift_en, and so on) are assumed to be available at the top level or are part of a user-defined data register in the JTAG TAP controller. This LPCT controller does not generate any hardware to control any of these static signals.

Figure 7-6 shows the configuration of the Type 2 LPCT controller. For an in-depth description, see "Type 2 - LPCT Controller with a TAP".

Figure 7-6. Type 2 LPCT Controller Configuration



Hardware area: The LPCT controller logic is approximately equal to 20 NAND gates and is independent of design size or test application.

Command: To generate a Type 2 controller use the following command:

set_lpct_controller on -generate_scan_enable on -tap_controller_interface on

Table 7-1 contains additional commands and switches that apply to the Type 2 controller.

Table 7-1. LPCT Controller Type 2 Commands and Switches

To generate a Type 2 LPCT Controller, use:	set_lpct_controller	set_lpct_pins	set_lpct_condition_bits
set_lpct_controller -generate_scan_enable On -tap_controller_interface On	-shift_control	clock (input) test_mode (input) capture_dr (input) shift_dr (input) update_dr (input) tms (input) reset (input) clock_mux_select (output) capture_en (output) shift_en (output) shift_clock (output) output_scan_en (output) test_clock_connection (output)	None

Type 3 LPCT Controller

Configuration: Scan enable signal and all other EDT-specific static and dynamic signals are generated by the LPCT controller.

Requirements: Generate all EDT-specific signals on chip including scan_en.

LPCT Controller Configuration	Required Inputs	Generated Outputs
Type 3	lpct_clock lpct_data_in (edt_channels_in1)	edt_update edt_clock scan_en edt_bypass edt_low_power_shift_en lpct_shift_clock OR lpct_shift_en lpct_capture_en edt_configuration

Note

For Type 3 controllers, the top-level scan enable pin is removed and the internally-generated scan enable pin is used.

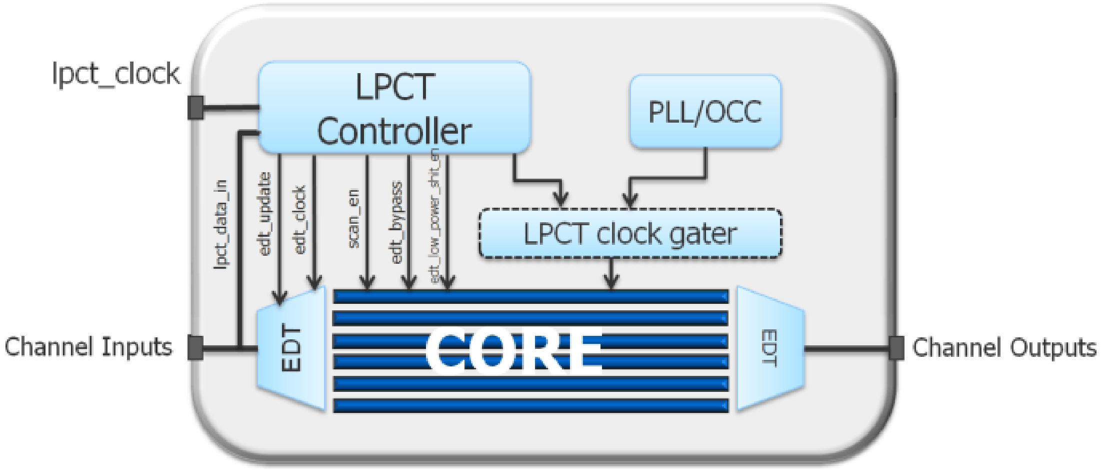
Note

OCC logic is required to detect reset faults for the design with a Type 3 LPCT controller.

Description: The LPCT controller will internally generate the scan enable signal and all EDT-specific control signals; this includes the dynamic signals edt_update, edt_clock, and scan_en and the static signals edt_bypass, edt_low_power_shift_en, and edt_configuration. If a design shift clock is not available at the top level, the LPCT controller can generate the shift clock from the LPCT clock.

Figure 7-7 shows the configuration of the Type 3 LPCT controller. For an in-depth description, see “Type 3 - LPCT Controller-generated Scan Enable”.

Figure 7-7. Type 3 LPCT Controller Configuration



Hardware area: The LPCT controller logic is approximately equal to 1200 NAND gate equivalent and is independent of design size or test application.

Command:

```
set_lpct_controller on -generate_scan_enable on -tap_controller_interface off
```

Table 7-2 contains additional commands and switches that apply to the Type 3 controller.

Table 7-2. LPCT Controller Type 3 Commands and Switches

To generate a Type 3 LPCT Controller, use:	set_lpct_controller	set_lpct_pins	set_lpct_condition_bits
set_lpct_controller -generate_scan_enable On -tap_controller_interface Off	-max_shift_cycles -max_capture_cycles -max_scan_patterns -max_chain_patterns -test_mode_detect -shift_control -load_unload_cycles	clock (input) reset (input) data_in(input) test_mode (input) clock_mux_select (output) capture_en (output) shift_en (output) shift_clock (output) output_scan_en (output) reset_out (output) test_end (output)	-condition reset -condition scan_en

For an in-depth description of this configuration, see “Type 3 - LPCT Controller-generated Scan Enable”.

Limitations

The limitations described in the following sections apply to this release.

Design Flow/Hardware Limitations

The LPCT controller has the following design flow limitations:

- Compression logic inserted external to the design core within a top-level wrapper is not supported.
- LSSD architecture is not supported.
- The LPCT controller is primarily targeted at generating test control signals internally to reduce the number of ATE pins required during test. Currently, the Type 2 LPCT controller does not support using the boundary scan register cells to further reduce the number of ATE pins required to connect with the design functional pins.
- The scan_en signal must be constrained to "0" during capture for the Type 1 LPCT controller.
- The number of pre-shift and post-shift cycles cannot be changed for any type controller during pattern generation. However the Type 3 controller allows changing of these cycles during IP creation.
- Pulsing edt_clock before shift is not supported because edt_clock and shift_clock are derived from the same clock source.
- When scan_en is available at the top level, the EDT static control signals such as *edt_bypass* and *edt_low_power_shift_en* are implemented as top-level pins.You are responsible for connecting these pins to some internal test logic to avoid having them assigned as top-level pins.

Test Pattern Limitations When Using a Type 3 Controller

When using an LPCT controller-generated scan enable configuration, the controller has the following test pattern limitation:

- Multiple load type test patterns are not supported.

Single Shared LPCT Controller for All EDT Design Blocks

If you define all EDT blocks at the same time during IP creation (top-down), the tool correctly generates only one LPCT controller and drives all EDT blocks from this controller. In a design with multiple power domains, you should ensure that the LPCT controller is placed in an always-ON power domain. The EDT blocks can still be placed on the same power domain as the block level logic. You can use the set_lpct_instances command to control where the LPCT controller is instantiated.

If you use the integration flow (bottom-up), do not create LPCT logic along with block-level EDT logic. During the top-level integration, the tool can generate the LPCT controller while also making the connections for the block level EDT signals. In a design with multiple power domains, you should also ensure that the LPCT controller is placed in an always-ON power domain using the set_lpct_instances command.

LPCT Controller Types

The following sections fully describe the three LPCT controller types and their differences:

- Type 1 - LPCT Controller with Top-level Scan Enable
- Type 2 - LPCT Controller with a TAP
- Type 3 - LPCT Controller-generated Scan Enable

Note

All three LPCT controller types apply to scan and EDT control pins only and do not limit the number of EDT channels that can be used.

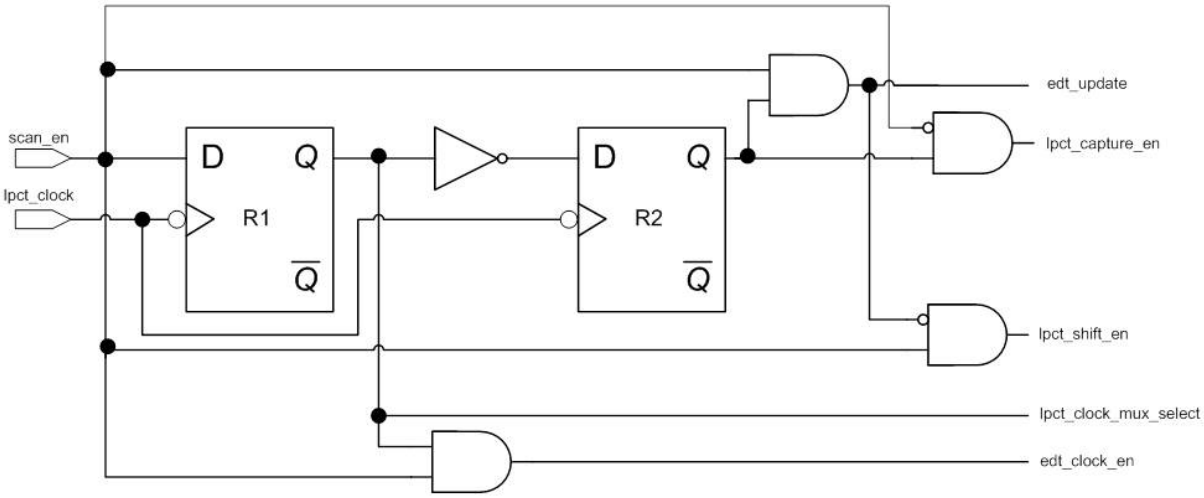
Type 1 - LPCT Controller with Top-level Scan Enable

When you implement an LPCT controller using a top-level scan_en pin, the LPCT controller generates the edt_update and edt_clock signals. However, it does not generate any of the EDT static control signals such as edt_bypass, edt_low_power_shift_en, and so on. To avoid having these signals assigned as top-level pins, you must connect them to some internal test logic.

When implementing the LPCT controller with a top-level scan enable signal (Type 1), the design must have a top-level clock. The clock can be a free running reference clock or a tester-controllable top-level clock pin as shown in Figure 7-8. If this clock is also a shift clock, and the shift control is set to “clock”, a clock gater is automatically inserted in this clock path to enable this clock to be used during shift and capture

Figure 7-8 shows the controller logic. In this configuration, the scan_en signal is constrained to 0 in the capture cycle and set to 1 during the shift cycle, but is set to 0 during the post-shift cycles.

Figure 7-8. Type 1 LPCT Controller Operation



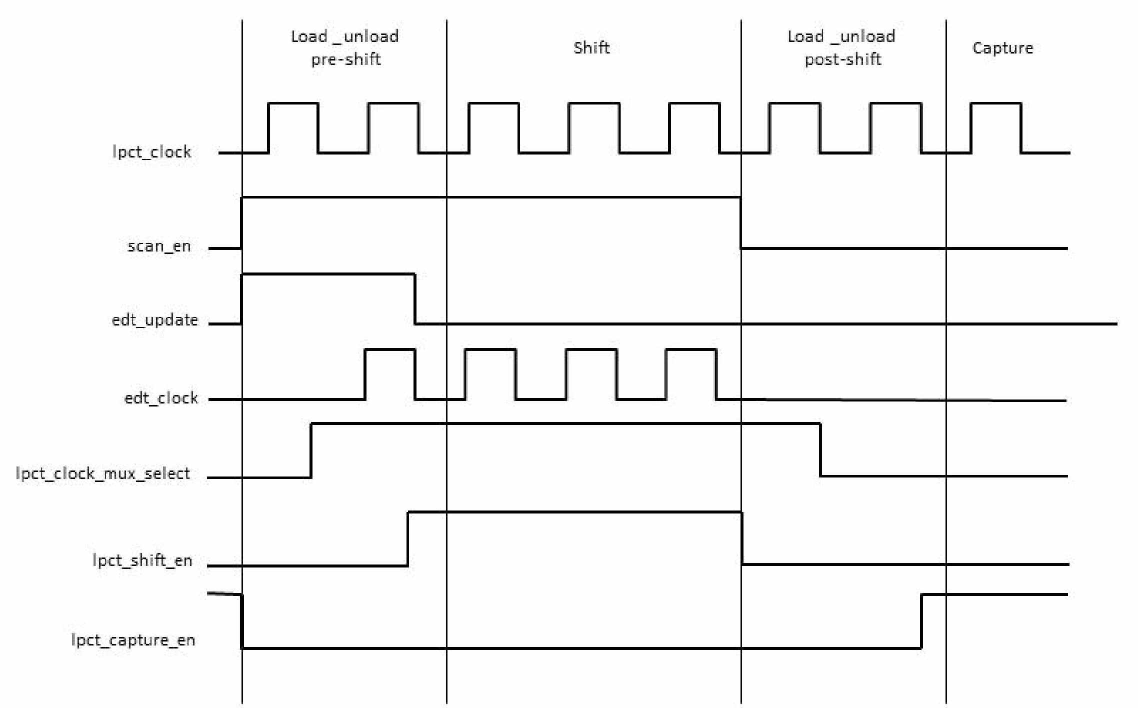
edt_clock	= scan_en & R1/Q
edt_update	= scan_en & R2/Q
lpct_shift_en	= scan_en & ~edt_update
lpct_capture_en	= R2/Q & ~scan_en
lpct_clock_mux_select	= R1/Q

The Type 1 LPCT controller does not have a finite state machine or counters to track the test procedure states. The start of the load_unload procedure is inferred when the scan_en signal transitions from 0 to 1.

- For scan test patterns, the pin constraint on scan_en provides the initial 0 value for the transition.
- For chain test patterns, there are no capture cycles when using free running clocks; the post-shift cycles in load_unload provide the initial 0 value for the transition.

Figure 7-9 shows the waveforms generated by Figure 7-8.

Figure 7-9. Signal Waveforms for Type 1 LPCT Controller



Two pre-shift and post-shift cycles are added to the load_unload procedures when using the Type 1 LPCT controller. These two cycles separate the transition between the shift clock, the capture clock, lpct_capture_en, and the lpct_clock_mux_select signals when transitioning from *capture to shift* and from *shift to capture*.

- **Pre-Shift Cycles**: At the beginning of the pre-shift cycles, the scan enable signal transitions from 0 (during capture) to 1 which allows the edt_update output from the LPCT controller to be asserted immediately. The edt_clock signal is generated one cycle later. However, the shift clock begins to pulse two cycles after the transition of the scan_en signal.
- **Post-Shift Cycles**: At the end of scan chain shifting, the scan_en signal is deasserted (transitions from 1 to 0). The signal scan_en is deasserted for both post-shift cycles and the clocks to the design are expected to be turned off as shown in the waveforms in Figure 7-9. The lpct_clock_mux_select signal is deasserted at the negedge of the clock in the first post-shift cycle. The lpct_capture_en signal transitions one cycle later—on the negedge of the second post-shift cycle. The capture pulses can only be generated in the cycle after the lpct_capture_en signal is asserted (after the 2nd post-shift cycle). During each of these cycles, only one of the signals, lpct_clk_mux_select and lpct_capture_en, transition at a time.

Type 2 - LPCT Controller with a TAP

When you implement an LPCT controller with a TAP, the LPCT controller generates the scan enable, edt_update, and edt_clock signals based on the output of the TAP controller. However, it does not generate any of the EDT static control signals such as edt_bypass and edt_low_power_shift_en. To avoid having these signals assigned as top-level pins, you must connect them to some internal test logic.

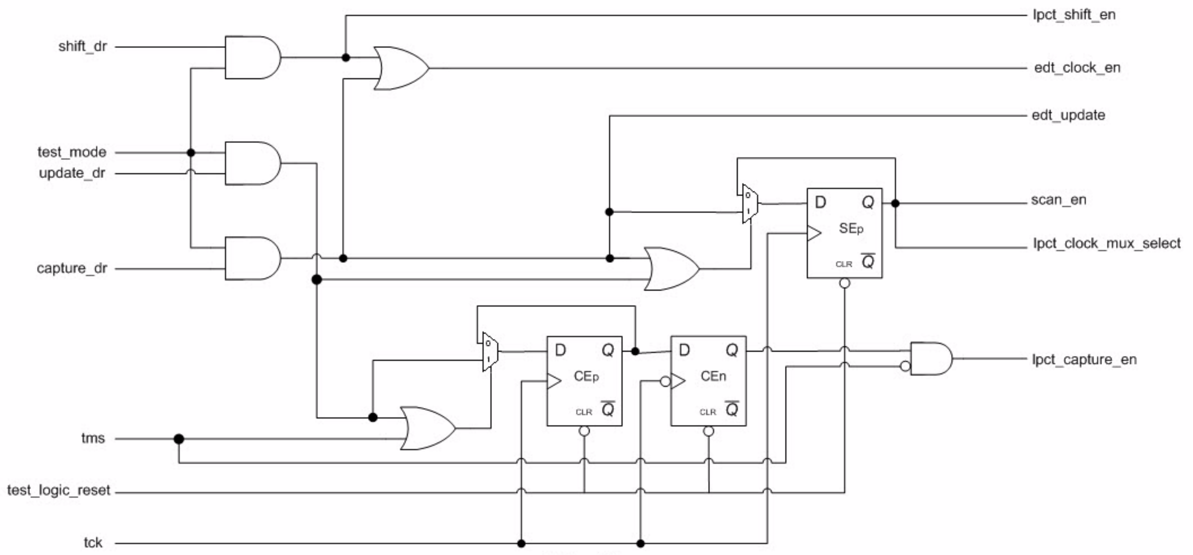
When implementing a Type 2 LPCT controller, the dynamic test control signals are generated based on the TAP controller state machine. In addition to the clock (tck) and test mode signal (tms), the enable signals corresponding to capture_dr, test_logic_reset, shift_dr, and update_dr are used as inputs to the controller. The shift_dr and capture_dr signals are assumed to change at the rising edge of the tck signal. These signals can be connected either to combinational tap output pins, or registered at the negedge of TCK in the TAP controller.

The update_dr signal is assumed to change at the falling edge of the tck signal. This signal can be connected either to a combinational tap output pin, or registered early at the prior posedge of TCK in the TAP controller.

These signal change edges are consistent with the IEEE 1149.1 standard.

Figure 7-10 shows the controller logic of the LPCT controller when using a TAP.

Figure 7-10. LPCT Controller with TAP



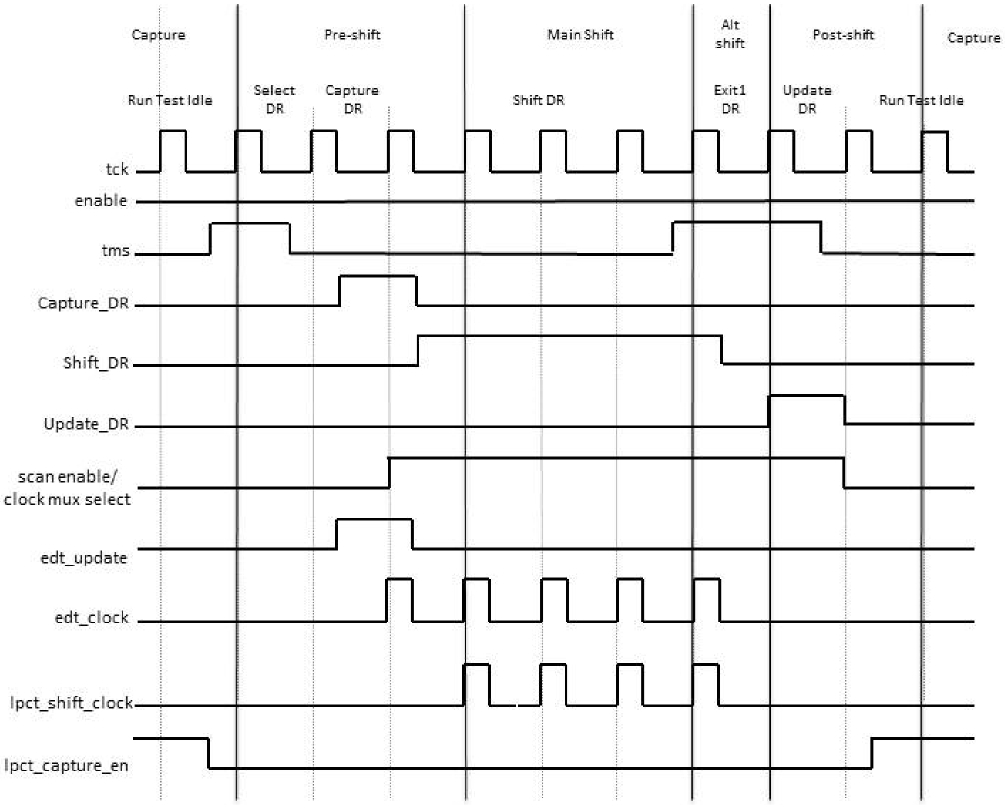
scan_en	= SEp/Q
edt_clock_en	= capture_dr | shift_dr
edt_update	= capture_dr
lpct_shift_en	= shift_dr
lpct_capture_en	= CEn/Q & ~tms
lpct_clock_mux_select	= SEp/Q

The TAP controller must provide an enable signal (test_mode) that signals the LPCT controller to enter test mode. This test_mode signal can be generated using a JTAG user-defined instruction.

The test_mode signal indicates whether the instruction corresponding to scan test is currently loaded in the instruction register. The signal test_logic_reset is used to asynchronously reset the flip-flops driving scan_en and capture_en because the design is required to go to functional mode of operation immediately on reset of the TAP controller. The scan_en signal has a re-circulating mux to hold its ON value between capture_dr and update_dr; this allows the scan_en to not change unnecessarily when long shift sequences are broken using the pause_dr state.

Figure 7-11 illustrates the waveforms of the input signals from the TAP controller and the generated output signals. Capture is performed during the run_test_idle state; this allows for an arbitrary number of tck capture cycles by constraining tms to 0.

Figure 7-11. Signal Waveforms for TAP-based LPCT Controller



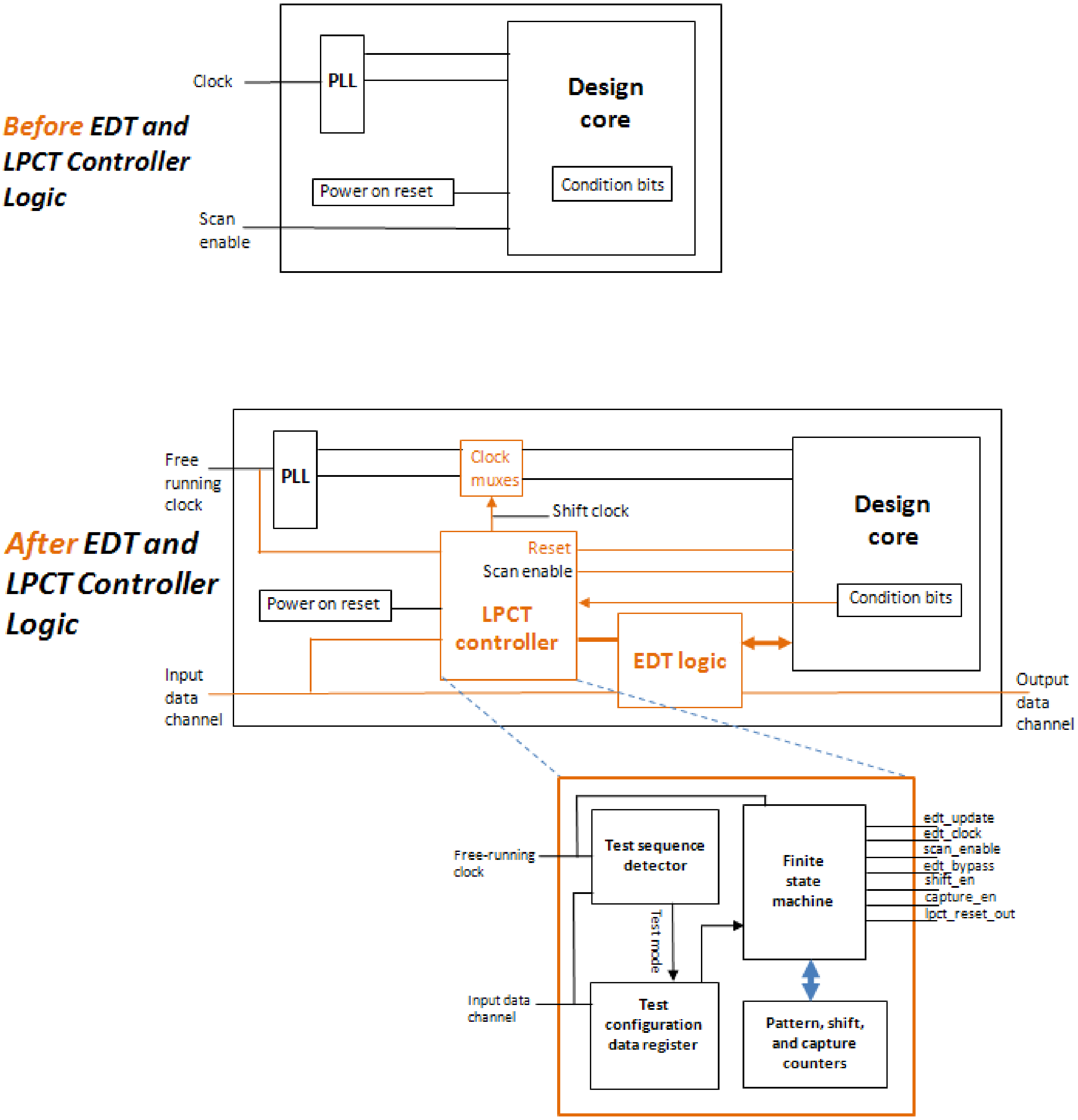
Type 3 - LPCT Controller-generated Scan Enable

A Type 3 LPCT controller requires a minimum of three top-level pins including a free-running clock, an input data channel, and an output data channel as shown in Figure 7-12. The free-running clock source can be either the reference clock from an on-chip PLL or the output of a

PLL that is always running. The LPCT controller logic operates based on this clock; the EDT and capture clocks are derived from this clock.

Note
For Type 2 and Type 3 controllers, the top-level scan enable pin is removed and the internally-generated scan enable pin is used.

Figure 7-12. After EDT and LPCT Controller Logic



In the Type 3 configuration, the LPCT controller contains the following components as shown in Figure 7-12:

- **Test sequence detector** — Detects a specified input sequence and produces a signal to enable test mode. This is optional depending on how you configure the test mode enable.
- **Test configuration data register** — Contains information about the test pattern set such as the number of chain/scan tests, shift/capture cycles, and the EDT logic mode of operation including low-power, bypass, and dual configurations. The size of the test configuration register is approximately 50 bits and is directly related to the size of the shift, capture, and pattern counters. The test configuration data is read once during the test_setup procedure.
- **Finite state machine** — Generates the scan control signals during test pattern application and controls pattern shift and capture counters.
- **Pattern, shift, and capture counters** — Track test pattern data for the finite state machine.

Test Mode Enable

Depending on the application, a signal from the LPCT controller to enable test mode must be configured using one of the following methods:

- **Test mode signal** — Test mode is enabled after the test mode signal is asserted for one cycle, and the test session end is determined by the test pattern counters. When this signal is a top-level pin, the correct test_setup procedure is automatically generated. When this signal is an internal pin, you must modify the test_setup procedure to ensure that the internal test mode signal is asserted as necessary and that the controller logic is reset before entering test mode.
- **Test mode sequence** — Test mode is enabled when a specific input sequence is detected within a specific number of cycles after the LPCT controller is reset. This is required when no top-level pin is used. You can specify the sequence/cycles with the set_lpct_controller command when setting up the LPCT controller. The generated test procedure file contains all the initialization cycles necessary to enter test mode when using sequence detection.

Note

Only one of these methods can be used to enable test mode for any single application.

Test Patterns and the Type 3 LPCT Controller

When generating test patterns for a Type 3 LPCT controller, you must take into account the following test pattern setups:

- **NCPs or clock control definitions can be used for capture cycles** — The LPCT controller hardware is configured for a fixed number of capture cycles as determined by the set_lpct_controller *-max_capture_cycles* command. Consequently, you must use NCPs to specify all possible clocking sequences and add additional cycles so all test patterns use the fixed number of capture cycles.
    - If NCPs are used, each one must have the same number of capture cycles.
    - If clock control definitions are used, the tool will automatically ensure that all patterns in the pattern set have the same number of capture cycles.
    - The value of the capture cycle width portion of the test configuration data is automatically stored in the test patterns as part of the *test_setup* procedure.
- **Chain test patterns** — The LPCT controller includes separate counters for chain test and scan test patterns. The chain tests do not include a capture cycle, so the controller does not enter capture state for chain test patterns.
- **Iddq test patterns** — Iddq tests do not have a capture cycle, but there is a quiescent (dead) cycle between pattern loads. During this time, you must ensure that no functional/design clocks pulse. NCPs are not supported for iddq test patterns.
- **Parallel test patterns** — The LPCT controller includes internally-added primary input pins, so you must use the *-mode_internal* switch when saving parallel test patterns. For more information, see the write_patterns command.

Figure 7-13 and Figure 7-14 show the waveforms for signals generated by the Type 3 LPCT controller configuration.

Note

The *edt_clock* signal is a gated version of the free-running clock that is always generated by the LPCT controller.

Figure 7-13. Scan Test Pattern Timing

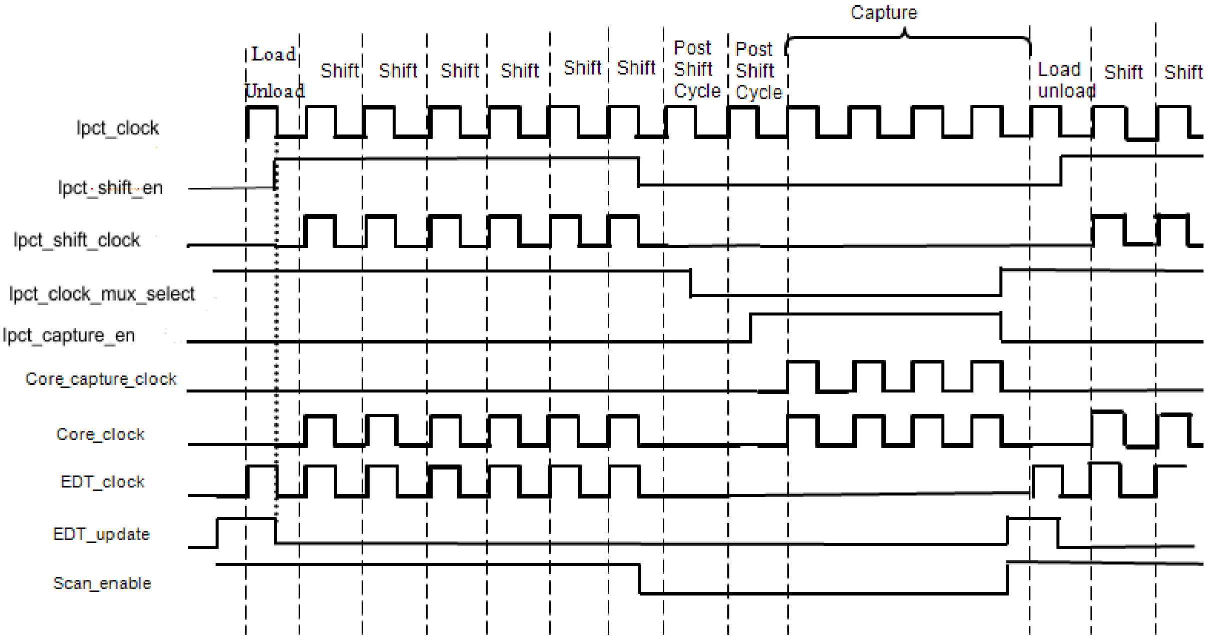
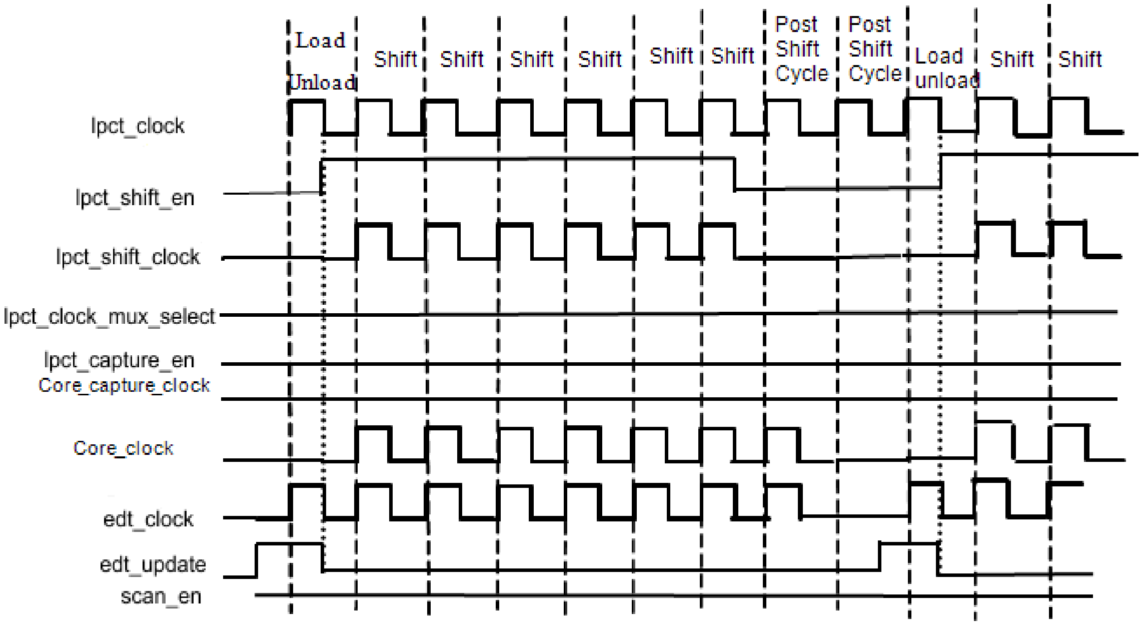


Figure 7-14. Chain Test Pattern Timing



Detecting Faults on Reset Lines. When using the Type 3 LPCT controller, the functional reset for the design is also used to reset the LPCT controller. Therefore, the faults along the reset lines are not detected by ATPG because the reset is in the deasserted state during the entire ATPG session. You can use one of the following two methods to recover the coverage along the reset lines:

- **Assert reset for the entire pattern set** — With this method, the patterns to detect faults along the reset lines must be in a different pattern set with their own test_setup procedure. To use this method, make the following change in the dofile generated during the logic creation phase of the LPCT controller:

  In the Pattern Generation dofile, specify pin constraints and register values as follows:

  Change: add_pin_constraints *reset_control C0*

  To: add_pin_constraints *reset_control C1*

  Change: add_register_value *lpct_config_reset_control 0*

  To: add_register_value *lpct_config_reset_control 1*

Note

When specifying an active low reset, using the set_lpct_pins -reset -active low command, you should reverse the pin constraint and register values. That is, you should flip the pin constraint from C1 to C0 and the register value from 1 to 0.

  The add_register_value command holds the reset to the design in the asserted state while the reset to the controller is deasserted. Although this method requires two separate sets of patterns each with their own test_setup procedure, no additional design requirements are needed to create these patterns.

- **Use LPCT condition bits** — With this method, you use a scan flop in the design as a control (condition) bit which allows ATPG to automatically justify the appropriate value to assert or deassert the reset signal to the design. With this method, only one pattern set is created for each fault model. To use this method, make the following changes:

  In the IP Creation dofile, specify the condition scan cell as follows:

  Add: set_lpct_condition_bits *-condition reset -from scancell_name*.

  In the Pattern Generation dofile, specify pin constraints and register values as follows:

  Change: add_pin_constraints *reset_control C0*

  To: add_pin_constraints *reset_control C1*

  Change: add_register_value *lpct_config_reset_control 0*

  To: add_register_value *lpct_config_reset_control 1*

Note

When specifying an active low reset, using the set_lpct_pins -reset -active low command, you should reverse the pin constraint and register values. That is, you should flip the pin constraint from C1 to C0 and the register value from 1 to 0.

ATPG justifies the value in this scan cell in the last load cycle before capture to ensure the reset to the design is asserted or deasserted as needed. Using this method allows you to generate a single test pattern set to detect reset line faults as well as other design faults.

Toggling Scan Enable. When using the Type 3 LPCT controller, the scan enable signal generated by the LPCT controller (lpct_scan_en) is always driven to 0 during capture. For stuck-at patterns, if the scan enable signal to the design is required to toggle during capture, you must use dedicated scan cells (condition bits); these scan cells must be part of the scan chain. To toggle the scan enable signal, do the following:

In the IP Creation dofile, specify the condition scan cell:

Add: set_lpct_condition_bits *-condition scan_en -from scancell_name*.

In the Pattern Generation dofile, specify pin constraints and register values:

Change: add_pin_constraints *scan_en_control C0*

To: add_pin_constraints *scan_en_control C1*

Change: add_register_value *lpct_config_scan_en_control 0*

To: add_register_value *lpct_config_scan_en_control 1*

LPCT Configuration Examples

This section provides an example for creating each of the three LPCT configuration types and examples of the dofile and test procedure files generated for each configuration.

For more information on the differences between configuration types, see "LPCT Controller Types."

Example 1

This example generates a Type 1 LPCT controller and displays the associated pattern generation dofile and test procedure file generated by the tool.

Sample dofile:

```
// Group definition

add_scan_groups grp1 scan_setup.testproc

// Clock definitions

add_clocks 0 /occ/NX2 -internal -pin_name NX2
add_clocks 0 /occ/NX1 -internal -pin_name NX1

// Scan chain definitions

add_scan_chains chain1 grp1 scan_in1 scan_out1
add_scan_chains chain2 grp1 scan_in2 scan_out2
add_scan_chains chain3 grp1 scan_in3 scan_out3
add_scan_chains chain4 grp1 scan_in4 scan_out4
add_scan_chains chain5 grp1 scan_in5 scan_out5
add_scan_chains chain6 grp1 scan_in6 scan_out6
add_scan_chains chain7 grp1 scan_in7 scan_out7
add_scan_chains chain8 grp1 scan_in8 scan_out8
add_scan_chains chain9 grp1 scan_in9 scan_out9
add_scan_chains chain10 grp1 scan_in10 scan_out10
add_scan_chains chain11 grp1 scan_in11 scan_out11
add_scan_chains chain12 grp1 scan_in12 scan_out12
add_scan_chains chain13 grp1 scan_in13 scan_out13
add_scan_chains chain14 grp1 scan_in14 scan_out14
add_scan_chains chain15 grp1 scan_in15 scan_out15
add_scan_chains chain16 grp1 scan_in16 scan_out16

// EDT configuration
set_edt_options -channels 2 -location internal

// LPCT configuration
set_lpct_controller -generate_scan_enable off -tap_controller_interface
off -shift_control clock

// LPCT Pin connections
set_lpct_pins clock refclk
set_lpct_pins input_scan_en scan_en
```

```
// Run DRC
set_system_mode analysis

// Insert EDT and LPCT controller logic in design
write_edt_files created -verilog -replace
```

Sample pattern generation dofile:

```
set_edt_instances -edt_logic_top m8051_edt_i
set_edt_instances -decompressor  m8051_edt_decompressor_i
set_edt_instances -compactor     m8051_edt_compactor_i

add_scan_groups grp1 created_edt.testproc
add_scan_chains -internal chain1 grp1 /m8051_edt_i/edt_scan_in[0]
    /m8051_edt_i/edt_scan_out[0]
add_scan_chains -internal chain2 grp1 /m8051_edt_i/edt_scan_in[1]
    /m8051_edt_i/edt_scan_out[1]
add_scan_chains -internal chain3 grp1 /m8051_edt_i/edt_scan_in[2]
    /m8051_edt_i/edt_scan_out[2]
add_scan_chains -internal chain4 grp1 /m8051_edt_i/edt_scan_in[3]
    /m8051_edt_i/edt_scan_out[3]
add_scan_chains -internal chain5 grp1 /m8051_edt_i/edt_scan_in[4]
    /m8051_edt_i/edt_scan_out[4]
add_scan_chains -internal chain6 grp1 /m8051_edt_i/edt_scan_in[5]
    /m8051_edt_i/edt_scan_out[5]
add_scan_chains -internal chain7 grp1 /m8051_edt_i/edt_scan_in[6]
    /m8051_edt_i/edt_scan_out[6]
add_scan_chains -internal chain8 grp1 /m8051_edt_i/edt_scan_in[7]
    /m8051_edt_i/edt_scan_out[7]
add_scan_chains -internal chain9 grp1 /m8051_edt_i/edt_scan_in[8]
    /m8051_edt_i/edt_scan_out[8]
add_scan_chains -internal chain10 grp1 /m8051_edt_i/edt_scan_in[9]
    /m8051_edt_i/edt_scan_out[9]
add_scan_chains -internal chain11 grp1 /m8051_edt_i/edt_scan_in[10]
    /m8051_edt_i/edt_scan_out[10]
add_scan_chains -internal chain12 grp1 /m8051_edt_i/edt_scan_in[11]
    /m8051_edt_i/edt_scan_out[11]
add_scan_chains -internal chain13 grp1 /m8051_edt_i/edt_scan_in[12]
    /m8051_edt_i/edt_scan_out[12]
add_scan_chains -internal chain14 grp1 /m8051_edt_i/edt_scan_in[13]
    /m8051_edt_i/edt_scan_out[13]
add_scan_chains -internal chain15 grp1 /m8051_edt_i/edt_scan_in[14]
    /m8051_edt_i/edt_scan_out[14]
add_scan_chains -internal chain16 grp1 /m8051_edt_i/edt_scan_in[15]
    /m8051_edt_i/edt_scan_out[15]

add_primary_inputs /occ/NX2 -internal -pin_name NX2
add_primary_inputs /occ/NX1 -internal -pin_name NX1
add_clocks 0 refclk
add_clocks 0 NX1
add_clocks 0 NX2

add_pin_constraints scan_en C0


// EDT settings.  Please do not modify.
// Inconsistency between the EDT settings and the EDT logic may
// lead to DRC violations and invalid patterns.

set_edt_options -channels 2 -longest_chain_range 2 32 -ip_version 5
   -decompressor_size 12 -injectors_per_channel 3 -scan_chains 16
   -compactor_type xpress

set_edt_pins update -
set_edt_pins clock -

set_mask_register -input_channel_mask_register_sizes   1 7   2 6

set_mask_decoder_connection -mode_bit              1 7
set_mask_decoder_connection -1hot_decoder 1        1 6   1 5   1 4   1 3
set_mask_decoder_connection -xor_decoder chain1    1 6   1 5   1 4
set_mask_decoder_connection -xor_decoder chain2    1 6   1 5   1 3
set_mask_decoder_connection -xor_decoder chain3    1 6   1 5   1 2
set_mask_decoder_connection -xor_decoder chain4    1 6   1 5   1 1
set_mask_decoder_connection -xor_decoder chain5    1 6   1 4   1 3
set_mask_decoder_connection -xor_decoder chain6    1 5   1 4   1 3
set_mask_decoder_connection -xor_decoder chain7    1 4   1 2   1 1
set_mask_decoder_connection -xor_decoder chain8    1 3   1 2   1 1

set_mask_decoder_connection -1hot_decoder 2        2 6   2 5   2 4   2 3
set_mask_decoder_connection -xor_decoder chain9    2 6   2 5   2 4
set_mask_decoder_connection -xor_decoder chain10   2 6   2 5   2 3
set_mask_decoder_connection -xor_decoder chain11   2 6   2 5   2 2
set_mask_decoder_connection -xor_decoder chain12   2 6   2 5   2 1
set_mask_decoder_connection -xor_decoder chain13   2 6   2 4   2 3
set_mask_decoder_connection -xor_decoder chain14   2 5   2 4   2 3
set_mask_decoder_connection -xor_decoder chain15   2 4   2 2   2 1
set_mask_decoder_connection -xor_decoder chain16   2 3   2 2   2 1

// LPCT configuration settings.  Please do not modify.
// Inconsistency between the LPCT configuration settings and the LPCT
// logic may lead to DRC violations and invalid patterns.

set_lpct_controller on -generate_scan_enable off
   -tap_controller_interface off -shift_control clock
   -load_unload_cycles 2 2
```

Sample pattern generation test procedure file:

```
set time scale 1.000000 ns ;
set strobe_window time 10 ;

 timeplate gen_tp1 =
    force_pi 0 ;
    measure_po 10 ;
    pulse /NX1 20 10;
    pulse /NX2 20 10;
    pulse refclk 20 10;
    period 40 ;
 end;

 procedure shift =
    scan_group grp1 ;
    timeplate gen_tp1 ;
```

```
   // cycle 1 starts at time 0
   cycle =
       force_sci ;
       measure_sco ;
       pulse /NX1 ;
       pulse /NX2 ;
       pulse refclk ;
   end;
end;

procedure load_unload =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   // cycle 1 starts at time 0
   cycle =
       force /NX1 0 ;
       force /NX2 0 ;
       force RST 0 ;
       force edt_bypass 0 ;
       force scan_en 1 ;
       pulse refclk ;
   end ;
   // cycle 2 starts at time 40
   cycle =
       force scan_en 1 ;
       pulse refclk ;
   end ;
   apply shift 45;
   // cycle 3 starts at time 120
   cycle =
       force scan_en 0 ;
       pulse refclk ;
   end ;
   // cycle 4 starts at time 160
   cycle =
       force scan_en 0 ;
       pulse refclk ;
   end;
end;

procedure test_setup =
   timeplate gen_tp1 ;
   // cycle 1 starts at time 0
   cycle =
       force scan_en 0 ;
       pulse refclk ;
   end ;
   // cycle 2 starts at time 40
   cycle =
       force scan_en 0 ;
       pulse refclk ;
   end;
end;
```

Example 2

This example generates a Type 2 LPCT controller and displays the associated pattern generation dofile and test procedure file generated by the tool.

Sample dofile:

```
// Group definition

add_scan_groups grp1 scan_setup.testproc

// Clock definitions

add_clocks 0 /occ/NX2 -internal -pin_name NX2
add_clocks 0 /occ/NX1 -internal -pin_name NX1
add_clocks 0 tck

// Pin constraints

add_pin_constraints trst C1

// Scan chain definitions

add_scan_chains chain1 grp1 scan_in1 scan_out1
add_scan_chains chain2 grp1 scan_in2 scan_out2
add_scan_chains chain3 grp1 scan_in3 scan_out3
add_scan_chains chain4 grp1 scan_in4 scan_out4
add_scan_chains chain5 grp1 scan_in5 scan_out5
add_scan_chains chain6 grp1 scan_in6 scan_out6
add_scan_chains chain7 grp1 scan_in7 scan_out7
add_scan_chains chain8 grp1 scan_in8 scan_out8
add_scan_chains chain9 grp1 scan_in9 scan_out9
add_scan_chains chain10 grp1 scan_in10 scan_out10
add_scan_chains chain11 grp1 scan_in11 scan_out11
add_scan_chains chain12 grp1 scan_in12 scan_out12
add_scan_chains chain13 grp1 scan_in13 scan_out13
add_scan_chains chain14 grp1 scan_in14 scan_out14
add_scan_chains chain15 grp1 scan_in15 scan_out15
add_scan_chains chain16 grp1 scan_in16 scan_out16

// EDT configuration

set_edt_options -channels 1
set_edt_options -location internal

set_edt_pins input_channel 1 tdi m8051_i/edt_channels_in1
set_edt_pins output_channel 1 tdo tap_i/tap_edt_channel_reg_in

// LPCT configuration

set_lpct_controller -generate_scan_enable on -tap_controller_interface on
-shift_control clock

// LPCT Pin connections to LPCT controller pins

set_lpct_pins clock tck pad_instance_1_i/po_pad_tck
set_lpct_pins reset - tap_i/U2/Z
set_lpct_pins capture_dr  - tap_i/tap_ctrl_i/capturedr
set_lpct_pins shift_dr - tap_i/tap_ctrl_i/shiftdr
set_lpct_pins update_dr - tap_i/tap_ctrl_i/updatedr
set_lpct_pins test_mode - tap_i/instruction_decoder_i/edt_scan_inst
set_lpct_pins tms tms pad_instance_1_i/po_pad_tms
set_lpct_pins output_scan_en scan_en

// Run DRC

set_system_mode analysis

// Insert EDT and LPCT controller logic in design

write_edt_files created -replace
```

Sample pattern generation dofile:

```
set_edt_instances -edt_logic_top m8051_bscan_edt_i
set_edt_instances -decompressor  m8051_bscan_edt_decompressor_i
set_edt_instances -compactor     m8051_bscan_edt_compactor_i

add_scan_groups grp1 created_edt.testproc
add_scan_chains -internal chain1 grp1 /m8051_bscan_edt_i/edt_scan_in[0]
   /m8051_bscan_edt_i/edt_scan_out[0]
add_scan_chains -internal chain2 grp1 /m8051_bscan_edt_i/edt_scan_in[1]
   /m8051_bscan_edt_i/edt_scan_out[1]
add_scan_chains -internal chain3 grp1 /m8051_bscan_edt_i/edt_scan_in[2]
   /m8051_bscan_edt_i/edt_scan_out[2]
add_scan_chains -internal chain4 grp1 /m8051_bscan_edt_i/edt_scan_in[3]
   /m8051_bscan_edt_i/edt_scan_out[3]
add_scan_chains -internal chain5 grp1 /m8051_bscan_edt_i/edt_scan_in[4]
   /m8051_bscan_edt_i/edt_scan_out[4]
add_scan_chains -internal chain6 grp1 /m8051_bscan_edt_i/edt_scan_in[5]
   /m8051_bscan_edt_i/edt_scan_out[5]
add_scan_chains -internal chain7 grp1 /m8051_bscan_edt_i/edt_scan_in[6]
   /m8051_bscan_edt_i/edt_scan_out[6]
add_scan_chains -internal chain8 grp1 /m8051_bscan_edt_i/edt_scan_in[7]
   /m8051_bscan_edt_i/edt_scan_out[7]
add_scan_chains -internal chain9 grp1 /m8051_bscan_edt_i/edt_scan_in[8]
   /m8051_bscan_edt_i/edt_scan_out[8]
add_scan_chains -internal chain10 grp1 /m8051_bscan_edt_i/edt_scan_in[9]
   /m8051_bscan_edt_i/edt_scan_out[9]
add_scan_chains -internal chain11 grp1 /m8051_bscan_edt_i/edt_scan_in[10]
   /m8051_bscan_edt_i/edt_scan_out[10]
add_scan_chains -internal chain12 grp1 /m8051_bscan_edt_i/edt_scan_in[11]
   /m8051_bscan_edt_i/edt_scan_out[11]
add_scan_chains -internal chain13 grp1 /m8051_bscan_edt_i/edt_scan_in[12]
   /m8051_bscan_edt_i/edt_scan_out[12]
add_scan_chains -internal chain14 grp1 /m8051_bscan_edt_i/edt_scan_in[13]
   /m8051_bscan_edt_i/edt_scan_out[13]
add_scan_chains -internal chain15 grp1 /m8051_bscan_edt_i/edt_scan_in[14]
   /m8051_bscan_edt_i/edt_scan_out[14]
add_scan_chains -internal chain16 grp1 /m8051_bscan_edt_i/edt_scan_in[15]
   /m8051_bscan_edt_i/edt_scan_out[15]

add_primary_inputs /occ/NX2 -internal -pin_name NX2
add_primary_inputs /occ/NX1 -internal -pin_name NX1
add_clocks 0 tck -always_capture
add_clocks 0 NX1
add_clocks 0 NX2

add_pin_constraints trst C1
add_pin_constraints tms C0


// EDT settings.  Please do not modify.
// Inconsistency between the EDT settings and the EDT logic may
// lead to DRC violations and invalid patterns.

set_edt_options -channels 1 -longest_chain_range 2 32 -ip_version 5
   -decompressor_size 12 -injectors_per_channel 6 -scan_chains 16
   -compactor_type xpress

set_edt_pins update -
set_edt_pins clock -
set_edt_pins input_channel  1 tdi
set_edt_pins output_channel 1 tdo


set_mask_register -input_channel_mask_register_sizes   1 8

set_mask_decoder_connection -mode_bit 1 8
set_mask_decoder_connection -1hot_decoder 1 1 7   1 6   1 5   1 4   1 3
set_mask_decoder_connection -xor_decoder chain1  1 7  1 6  1 5
set_mask_decoder_connection -xor_decoder chain2  1 7  1 6  1 4
set_mask_decoder_connection -xor_decoder chain3  1 7  1 6  1 3
set_mask_decoder_connection -xor_decoder chain4  1 7  1 6  1 2
set_mask_decoder_connection -xor_decoder chain5  1 7  1 6  1 1
set_mask_decoder_connection -xor_decoder chain6  1 7  1 5  1 4
set_mask_decoder_connection -xor_decoder chain7  1 6  1 5  1 4
set_mask_decoder_connection -xor_decoder chain8  1 3  1 2  1 1
set_mask_decoder_connection -xor_decoder chain9  1 5  1 4  1 3
set_mask_decoder_connection -xor_decoder chain10 1 6  1 5  1 2
set_mask_decoder_connection -xor_decoder chain11 1 7  1 2  1 1
set_mask_decoder_connection -xor_decoder chain12 1 6  1 5  1 1
set_mask_decoder_connection -xor_decoder chain13 1 6  1 3  1 1
set_mask_decoder_connection -xor_decoder chain14 1 6  1 4  1 2
set_mask_decoder_connection -xor_decoder chain15 1 6  1 3  1 2
set_mask_decoder_connection -xor_decoder chain16 1 4  1 3  1 2

// LPCT configuration settings.  Please do not modify.
// Inconsistency between the LPCT configuration settings and the LPCT
// logic may lead to DRC violations and invalid patterns.

set_lpct_controller on -generate_scan_enable on -tap_controller_interface
   on -shift_control clock -load_unload_cycles 3 2
```

Sample pattern generation test procedure file:

Note

The following test_setup procedure is not generated by the tool but copied from a user-provided test procedure file as an example.

```
set time scale 1.000000 ns ;
set strobe_window time 10 ;

 timeplate gen_tp1 =
    force_pi 0 ;
    measure_po 10 ;
    pulse /NX1 20 10;
    pulse /NX2 20 10;
    pulse tck 20 10;
    period 40 ;
 end;

 procedure shift lpct_tap_last_shift =
    scan_group grp1 ;
    timeplate gen_tp1 ;
    // cycle 1 starts at time 0
    cycle =
        force_sci ;
        force tms 1 ;
        measure_sco ;
        pulse /NX1 ;
        pulse /NX2 ;
        pulse tck ;
    end;
 end;

 procedure test_setup =
    timeplate gen_tp1 ;

    // cycle 1 starts at time 0
    cycle =
        force tck 0 ;
        force tms 1 ;
        force trst 0 ;
    end ;
    // cycle 2 starts at time 40
    cycle =
        force trst 1 ;
    end ;
    // cycle 3 starts at time 80
    cycle =
        force tms 0 ;
        pulse tck ;
    end ;
    // cycle 4 starts at time 120
    cycle =
        force tms 1 ;
        pulse tck ;
    end ;
    // cycle 5 starts at time 160
```

```
        cycle =
            force tms 1 ;
            pulse tck ;
        end ;
        // cycle 6 starts at time 200
        cycle =
            force tms 0 ;
            pulse tck ;
        end ;
        // cycle 7 starts at time 240
        cycle =
            force tms 0 ;
            pulse tck ;
        end ;
        // cycle 8 starts at time 280
        cycle =
            force tdi 0 ;
            force tms 0 ;
            pulse tck ;
        end ;
        // cycle 9 starts at time 320
        cycle =
            force tdi 1 ;
            force tms 0 ;
            pulse tck ;
        end ;
        // cycle 10 starts at time 360
        cycle =
            force tdi 0 ;
            force tms 0 ;
            pulse tck ;
        end ;
        // cycle 11 starts at time 400
        cycle =
            force tdi 0 ;
            force tms 1 ;
            pulse tck ;
        end ;
        // cycle 12 starts at time 440
        cycle =
            force tms 1 ;
            pulse tck ;
        end ;
        // cycle 13 starts at time 480
        cycle =
            force tms 0 ;
            pulse tck ;
        end;
     end;
    procedure shift =
        scan_group grp1 ;
        timeplate gen_tp1 ;
        // cycle 1 starts at time 0
        cycle =
            force_sci ;
            force tms 0 ;
            measure_sco ;
            pulse /NX1 ;
```

```
       pulse /NX2 ;
       pulse tck ;
   end;
end;

procedure load_unload =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   // cycle 1 starts at time 0
   cycle =
       force /NX1 0 ;
       force /NX2 0 ;
       force RST 0 ;
       force edt_bypass 0 ;
       force tck 0 ;
       force tdi 0 ;
       force tms 1 ;
       force trst 1 ;
       pulse tck ;
   end ;
   // cycle 2 starts at time 40
   cycle =
       force tms 0 ;
       pulse tck ;
   end ;
   // cycle 3 starts at time 80
   cycle =
       force tms 0 ;
       pulse tck ;
   end ;
   apply shift 51;
   apply lpct_tap_last_shift 1;
   // cycle 4 starts at time 200
   cycle =
       force tms 1 ;
       pulse tck ;
   end ;
   // cycle 5 starts at time 240
   cycle =
       force tms 0 ;
       pulse tck ;
   end;
end;
```

Example 3

This example generates a Type 3 LPCT controller and displays the associated pattern generation dofile and test procedure file generated by the tool.

Sample dofile:

```
// Group definition
add_scan_groups grp1 scan_setup.testproc

// Clock definitions
add_clocks 0 /occ/NX2 -internal -pin_name NX2
add_clocks 0 /occ/NX1 -internal -pin_name NX1

// Scan chain definitions
add_scan_chains chain1 grp1 scan_in1 scan_out1
add_scan_chains chain2 grp1 scan_in2 scan_out2
add_scan_chains chain3 grp1 scan_in3 scan_out3
add_scan_chains chain4 grp1 scan_in4 scan_out4
add_scan_chains chain5 grp1 scan_in5 scan_out5
add_scan_chains chain6 grp1 scan_in6 scan_out6
add_scan_chains chain7 grp1 scan_in7 scan_out7
add_scan_chains chain8 grp1 scan_in8 scan_out8
add_scan_chains chain9 grp1 scan_in9 scan_out9
add_scan_chains chain10 grp1 scan_in10 scan_out10
add_scan_chains chain11 grp1 scan_in11 scan_out11
add_scan_chains chain12 grp1 scan_in12 scan_out12
add_scan_chains chain13 grp1 scan_in13 scan_out13
add_scan_chains chain14 grp1 scan_in14 scan_out14
add_scan_chains chain15 grp1 scan_in15 scan_out15
add_scan_chains chain16 grp1 scan_in16 scan_out16

// EDT configuration
set_edt_options -channels 2 -location internal

// LPCT configuration
set_lpct_controller -generate_scan_enable on -tap_controller_interface
    off -shift_control clock
set_lpct_controller -max_shift 1000 -max_capture 3

// LPCT Pin connections
set_lpct_pins clock refclk
set_lpct_pins output_scan_en scan_en

// Run DRC
set_system_mode analysis

// Insert EDT and LPCT controller logic in design
write_edt_files created -verilog -replace
```

Sample pattern generation dofile:

```
set_edt_instances -edt_logic_top m8051_edt_i
set_edt_instances -decompressor  m8051_edt_decompressor_i
set_edt_instances -compactor     m8051_edt_compactor_i

add_scan_chains -internal chain1 grp1 /m8051_edt_i/edt_scan_in[0]
    /m8051_edt_i/edt_scan_out[0]
add_scan_chains -internal chain2 grp1 /m8051_edt_i/edt_scan_in[1]
     /m8051_edt_i/edt_scan_out[1]
add_scan_chains -internal chain3 grp1 /m8051_edt_i/edt_scan_in[2]
     /m8051_edt_i/edt_scan_out[2]
add_scan_chains -internal chain4 grp1 /m8051_edt_i/edt_scan_in[3]
     /m8051_edt_i/edt_scan_out[3]
add_scan_chains -internal chain5 grp1 /m8051_edt_i/edt_scan_in[4]
     /m8051_edt_i/edt_scan_out[4]
add_scan_chains -internal chain6 grp1 /m8051_edt_i/edt_scan_in[5]
     /m8051_edt_i/edt_scan_out[5]
add_scan_chains -internal chain7 grp1 /m8051_edt_i/edt_scan_in[6]
```

```
     /m8051_edt_i/edt_scan_out[6]
add_scan_chains -internal chain8 grp1 /m8051_edt_i/edt_scan_in[7]
     /m8051_edt_i/edt_scan_out[7]
add_scan_chains -internal chain9 grp1 /m8051_edt_i/edt_scan_in[8]
     /m8051_edt_i/edt_scan_out[8]
add_scan_chains -internal chain10 grp1 /m8051_edt_i/edt_scan_in[9]
     /m8051_edt_i/edt_scan_out[9]
add_scan_chains -internal chain11 grp1 /m8051_edt_i/edt_scan_in[10]
     /m8051_edt_i/edt_scan_out[10]
add_scan_chains -internal chain12 grp1 /m8051_edt_i/edt_scan_in[11]
     /m8051_edt_i/edt_scan_out[11]
add_scan_chains -internal chain13 grp1 /m8051_edt_i/edt_scan_in[12]
     /m8051_edt_i/edt_scan_out[12]
add_scan_chains -internal chain14 grp1 /m8051_edt_i/edt_scan_in[13]
     /m8051_edt_i/edt_scan_out[13]
add_scan_chains -internal chain15 grp1 /m8051_edt_i/edt_scan_in[14]
     /m8051_edt_i/edt_scan_out[14]
add_scan_chains -internal chain16 grp1 /m8051_edt_i/edt_scan_in[15]
     /m8051_edt_i/edt_scan_out[15]

add_primary_inputs /occ/NX2 -internal -pin_name NX2
add_primary_inputs /occ/NX1 -internal -pin_name NX1
add_primary_input -internal
/m8051_lpct_clock_gater_i/m8051_lpct_edt_clock_gater_i/clk_out
   -pin_name edt_clock
add_primary_input -internal
/m8051_lpct_i/m8051_lpct_fsm_i/m8051_lpct_control_signal_generator_i/edt_
update -pin_name edt_update
add_primary_input -internal
/m8051_lpct_i/m8051_lpct_interface_i/edt_bypass -pin_name edt_bypass
add_primary_input -internal
/m8051_lpct_i/m8051_lpct_fsm_i/m8051_lpct_control_signal_generator_i/scan
_en -pin_name lpct_scan_en
add_primary_input -internal
/m8051_lpct_i/m8051_lpct_fsm_i/m8051_lpct_control_signal_generator_i/lpct
_capture_en -pin_name lpct_capture_en
add_primary_input -internal
/m8051_lpct_i/m8051_lpct_fsm_i/m8051_lpct_control_signal_generator_i/lpct
_clock_mux_select -pin_name lpct_clock_mux_select
add_primary_input -internal
/m8051_lpct_i/m8051_lpct_fsm_i/m8051_lpct_control_signal_generator_i/lpct
_shift_en -pin_name lpct_shift_en
add_primary_input -internal
/m8051_lpct_i/m8051_lpct_fsm_i/m8051_lpct_control_signal_generator_i/lpct
_test_active -pin_name lpct_test_active
add_primary_input -internal
/m8051_lpct_i/m8051_lpct_interface_i/reset_control -pin_name
reset_control
add_primary_input -internal
/m8051_lpct_i/m8051_lpct_interface_i/scan_en_control
-pin_name scan_en_control

add_clocks 0 refclk -free_running
add_clocks 0 NX1
add_clocks 0 NX2
add_clocks 0 edt_clock

add_pin_constraints edt_clock C0
add_pin_constraints edt_update C0
add_pin_constraints edt_bypass CX
add_pin_constraints lpct_capture_en C1
add_pin_constraints lpct_clock_mux_select C0
add_pin_constraints lpct_shift_en C0
add_pin_constraints lpct_test_active C1
add_pin_constraints lpct_reset C0
add_pin_constraints reset_control C0
add_pin_constraints scan_en_control C0

// EDT settings.  Please do not modify.
// Inconsistency between the EDT settings and the EDT logic may
// lead to DRC violations and invalid patterns.

set_edt_options -channels 2 -longest_chain_range 2 32 -ip_version 5
   -decompressor_size 12 -injectors_per_channel 3 -scan_chains 16
   -compactor_type xpress

set_edt_pins update edt_update
set_edt_pins clock edt_clock
set_edt_pins bypass edt_bypass

set_mask_register -input_channel_mask_register_sizes   1 7   2 6

set_mask_decoder_connection -mode_bit             1 7
set_mask_decoder_connection -1hot_decoder 1       1 6   1 5   1 4   1 3
set_mask_decoder_connection -xor_decoder chain1   1 6   1 5   1 4
set_mask_decoder_connection -xor_decoder chain2   1 6   1 5   1 3
set_mask_decoder_connection -xor_decoder chain3   1 6   1 5   1 2
set_mask_decoder_connection -xor_decoder chain4   1 6   1 5   1 1
set_mask_decoder_connection -xor_decoder chain5   1 6   1 4   1 3
set_mask_decoder_connection -xor_decoder chain6   1 5   1 4   1 3
set_mask_decoder_connection -xor_decoder chain7   1 4   1 2   1 1
set_mask_decoder_connection -xor_decoder chain8   1 3   1 2   1 1

set_mask_decoder_connection -1hot_decoder 2       2 6   2 5   2 4   2 3
set_mask_decoder_connection -xor_decoder chain9   2 6   2 5   2 4
set_mask_decoder_connection -xor_decoder chain10  2 6   2 5   2 3
set_mask_decoder_connection -xor_decoder chain11  2 6   2 5   2 2
set_mask_decoder_connection -xor_decoder chain12  2 6   2 5   2 1
set_mask_decoder_connection -xor_decoder chain13  2 6   2 4   2 3
set_mask_decoder_connection -xor_decoder chain14  2 5   2 4   2 3
set_mask_decoder_connection -xor_decoder chain15  2 4   2 2   2 1
set_mask_decoder_connection -xor_decoder chain16  2 3   2 2   2 1

// LPCT configuration settings.  Please do not modify.
// Inconsistency between the LPCT configuration settings and the LPCT
// logic may lead to DRC violations and invalid patterns.

set_lpct_controller on -generate_scan_enable on -tap_controller_interface
    off -shift_cycles_reg_width 10 -capture_cycles_reg_width 2
   -scan_patterns_reg_width 20 -chain_patterns_reg_width 10
   -test_mode_detect signal -shift_control clock -load_unload_cycles 0 2
   -bypass_controller off -reset_condition off

set_pattern_type -max_sequential 3

add_register_value lpct_config_edt_bypass 0
add_register_value lpct_config_reset_control 0
add_register_value lpct_config_scan_en_control 0
add_register_value lpct_config_chain_pattern_load_count -width 10
   -load_count chain_patterns -lsb_shifted_first
add_register_value lpct_config_scan_pattern_load_count -width 20
   -load_count scan_patterns -lsb_shifted_first
add_register_value lpct_config_capture_depth -width 2 -capture_cycles_max
    -lsb_shifted_first
add_register_value lpct_config_shift_length -width 10 -shift_length
   -lsb_shifted_first

set_chain_test -suppress_capture on
```

Sample pattern generation test procedure file:

```
set time scale 1.000000 ns ;
set strobe_window time 10 ;

 timeplate gen_tp1 =
    force_pi 0 ;
    measure_po 10 ;
    pulse /NX1 20 10;
    pulse /NX2 20 10;
    pulse edt_clock 20 10;
    pulse refclk 20 10;
    period 40 ;
 end;

 procedure load_unload_register lpct_shift_data =
    timeplate gen_tp1 ;
    shift =
    // cycle 1 starts at time 0
      cycle =
          force lpct_data_in # ;
          pulse refclk ;
      end;
    end;
 end;

 procedure shift =
    scan_group grp1 ;
    timeplate gen_tp1 ;
    // cycle 1 starts at time 0
    cycle =
        force_sci ;
        force edt_update 0 ;
        force lpct_shift_en 1 ;
        measure_sco ;
        pulse /NX1 ;
        pulse /NX2 ;
        pulse edt_clock ;
    end;
 end;

 procedure load_unload =
    scan_group grp1 ;
    timeplate gen_tp1 ;
    // cycle 1 starts at time 0
    cycle =
        force /NX1 0 ;
        force /NX2 0 ;
        force RST 0 ;
        force edt_bypass 0 ;
        force edt_clock 0 ;
        force edt_update 1 ;
        force lpct_capture_en 0 ;
        force lpct_clock_mux_select 1 ;
        force lpct_scan_en 1 ;
        force lpct_shift_en 0 ;
        force lpct_test_active 1 ;
        pulse edt_clock ;
    end ;
    apply shift 45;
    // cycle 2 starts at time 80
    cycle =
        force lpct_clock_mux_select 1 ;
        force lpct_scan_en 0 ;
        force lpct_shift_en 0 ;
    end ;
    // cycle 3 starts at time 120
    cycle =
        force lpct_clock_mux_select 0 ;
        force lpct_shift_en 0 ;
    end;
 end;


 procedure test_setup =
    timeplate gen_tp1 ;
    // cycle 1 starts at time 0
    cycle =
        force edt_clock 0 ;
        force lpct_data_in 0 ;
        force lpct_reset 1 ;
        force lpct_test_mode 0 ;
    end ;
    // cycle 2 starts at time 40
    cycle =
        force lpct_reset 0 ;
    end ;
    // cycle 3 starts at time 80
    cycle =
        force lpct_test_mode 1 ;
    end ;
    apply lpct_shift_data  lpct_data_in = 1 ;
    apply lpct_shift_data  lpct_data_in = lpct_config_edt_bypass ;
    apply lpct_shift_data  lpct_data_in = lpct_config_reset_control ;
    apply lpct_shift_data  lpct_data_in = lpct_config_scan_en_control ;
    apply lpct_shift_data  lpct_data_in =
       lpct_config_chain_pattern_load_count ;
    apply lpct_shift_data  lpct_data_in =
       lpct_config_scan_pattern_load_count ;
    apply lpct_shift_data  lpct_data_in = lpct_config_capture_depth ;
    apply lpct_shift_data  lpct_data_in = lpct_config_shift_length ;
    apply lpct_shift_data  lpct_data_in = 0 ;
 end;
```

```
procedure test_end =
   timeplate gen_tp1 ;
   // cycle 1 starts at time 0
   cycle =
       force lpct_test_active 1 ;
   end ;
   // cycle 2 starts at time 40

   cycle =
       force lpct_test_active 1 ;
   end ;
   // cycle 3 starts at time 80
   cycle =
       force lpct_test_active 1 ;
   end;
end;
```

Example 4

This example generates a simple Type 1 controller that specifies a top-level scan clock as the LPCT clock.

```
set_context dft -edt
add_clock 0 clk //Note - there is a single clock in the design
add_scan_chains ...
set_lpct_controller on -shift_control clock
set_lpct_pin clock clk //LPCT clock is shared with scan clock
set_lpct_pin input_scan_enable scan_en
set_lpct_pin test_clock_connection test_mode_mux/B
set_system_mode analysis
report_lpct_pins
write_edt_files created -replace
```

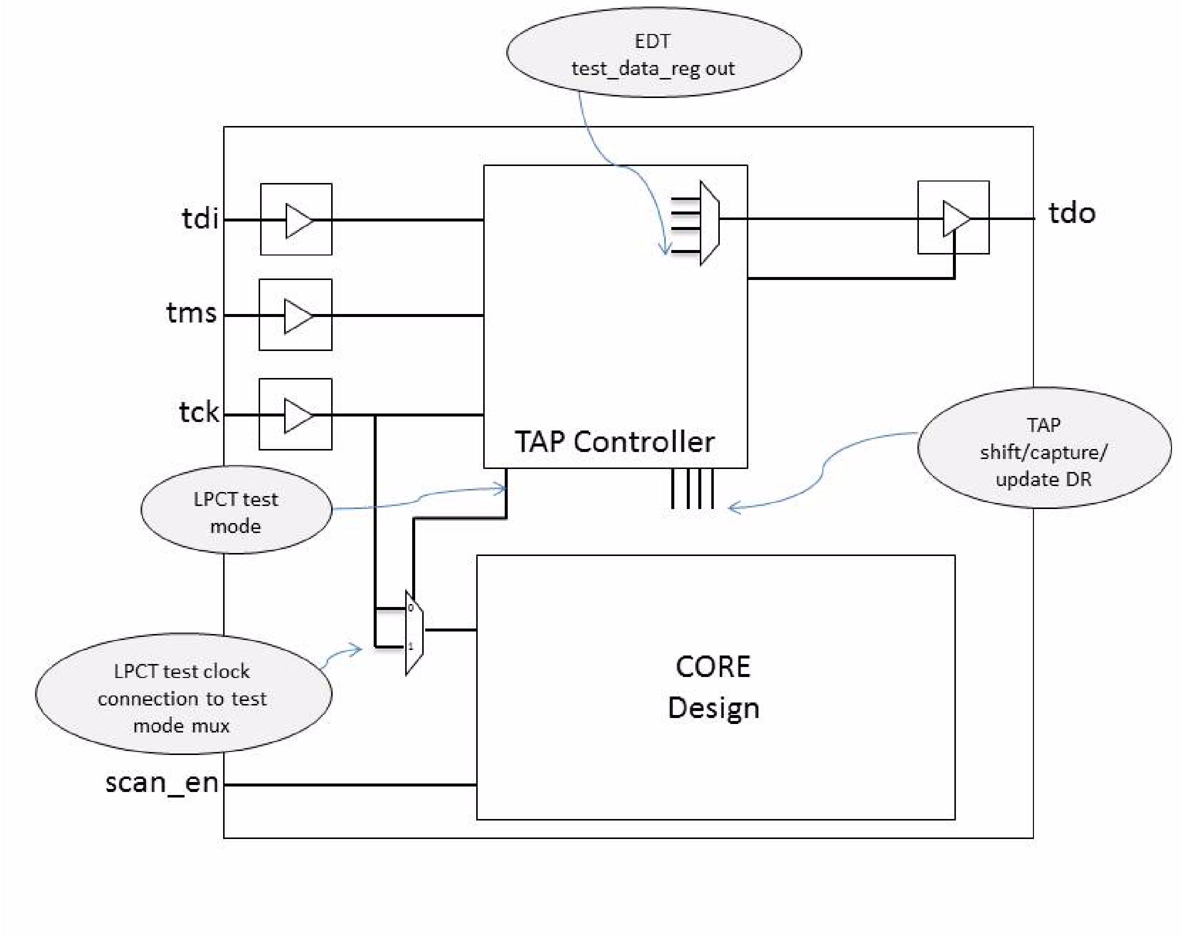
Example 5

This example generates a Type 2 LPCT controller that uses tck as a scan shift clock. The test mode multiplexer, which chooses between the LPCT-generated scan clock and the functional clock, already exists in the design. The test_clock_connection pin on the mux is specified with the test_clock_connection pin type. This example is illustrated in Figure 7-15.

```
set_context dft -edt
add_clock 0 tck
add_scan_chains ...
set_lpct_controller -tap_controller_interface on
set_lpct_controller -shift_control clock
set_lpct_pins output_scan_enable scan_en
set_lpct_pins tck tck pad_tck/Z //LPCT clock is tck
set_lpct_pins tms tms pad_tms/Z
set_lpct_pins reset - tap_i/tlr
set_lpct_pins capture_dr - tap_i/capturedr
set_lpct_pins shift_dr - tap_i/shiftdr
set_lpct_pins update_dr - tap_i/updatedr
set_lpct_pins test_mode - tap_i/edt_scan_inst
set_lpct_pins test_clock_connection test_mode_mux/B
```

```
set_edt_options -channel 1
set_edt_pins input 1 tdi pad_tdi/Z
set_edt_pins output 1 tdo tap_i/tap_edt_channel_reg_in
set_system_mode analysis
report_lpct_pins
report_lpct_configuration
write_edt_files created -replace
```

Figure 7-15. Type 2 LPCT Design Example



Compression Bypass Logic

By default, bypass circuitry is included in the EDT logic. The bypass circuitry allows you to bypass the EDT logic and access uncompressed scan chains in the design core.

Bypassing the EDT logic enables you to apply uncompressed test patterns to the design to:

- Debug compressed test patterns.
- Apply additional custom uncompressed scan chains.
- Apply test patterns from other ATPG tools.

Bypass logic can also be inserted in the core netlist at scan insertion time. This allows you to place the multiplexers and lockup cells required to operate the bypass mode inside the core netlist instead of the EDT logic. This option allows more effective design routing. For more information, see "Insertion of Bypass Chains in the Netlist" on page 40.

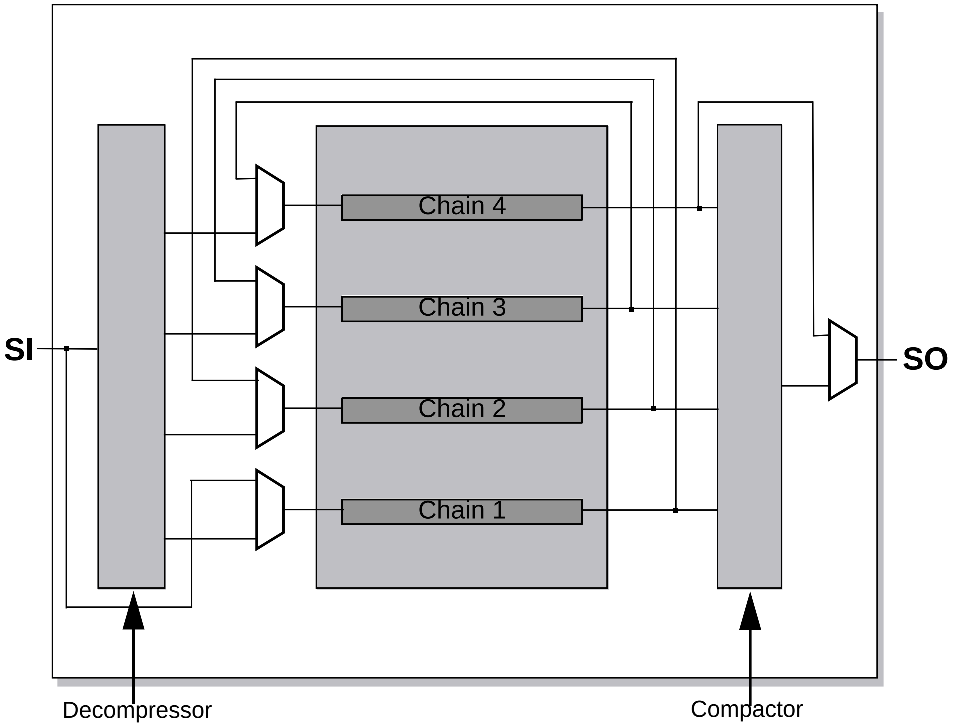
You can also set up two bypass scan chain configurations. In addition to the default configuration, you can create a second bypass configuration that concatenates all scan chains together into one bypass chain for use when hardware test channels are limited. For more information, see "Dual Bypass Configurations" on page 175.

Structure of the Bypass Logic

Because the number of core scan chains is relatively large, they are reconfigured into fewer, longer scan chains for bypass mode. For example, in a design with 100 core scan chains and four external channels, every 25 scan chains are concatenated to form one bypass chain. This bypass chain is then connected between the input and output pins of a given channel.

Figure 7-16 illustrates conceptually how the bypass mode is implemented.

Figure 7-16. Bypass Mode Circuitry



Notice that the bypass logic is implemented with multiplexers. The tool includes the multiplexers and any lockup cells needed to concatenate scan chains in the EDT logic.

Note

When lockup cells are inserted as part of the bypass logic, the EDT logic requires a system clock. If the same bypass logic is placed in the netlist, the EDT logic does not require a system clock.

You can also set up the EDT clock to pulse before the scan chain shift clocks to avoid using a system clock. For more information, see the *-pulse_edt_before_shift_clocks* switch of the set_edt_options command.

The bypass circuitry is run from bypass mode in Tessent FastScan.

Generating EDT Logic When Bypass Logic is Defined in the Netlist

EDT technology supports netlists that contain two sets of pre-defined scan chains. Predefining two sets of scan chains allows you to insert both the bypass chains and the core chains into the core design with a scan-insertion tool.

Note

Design blocks that contain bypass chains in the EDT logic and design blocks that contain bypass chains in the core can coexist in a design.

Limitations

- Bypass patterns cannot be created from compressed test patterns. You must generate bypass patterns from Tessent Shell. See "Creating Bypass Test Patterns in Uncompressed ATPG" on page 180.

Prerequisites

- Both bypass and core scan chains must be inserted in the design netlist. For more information, see "Insertion of Bypass Chains in the Netlist" on page 40.

Procedure

1. Invoke Tessent Shell. For example:

```
<Tessent_Tree_Path>/bin/tessent -shell
```

2. Load the design and library and set the context for EDT logic generation.

```
set_context dft -edt
read_verilog my_gate_scan.v
read_cell_library my_lib.aptg
set_current_design top
```

3. Set up parameters for the EDT logic generation.

For more information, see "Preparation for EDT Logic Creation" on page 52.

4. Enable the tool to use existing bypass chains. For example:

   ```
   set_edt_options -bypass_logic use_existing_bypass_chains
   ```

   For more information, see the set_edt_options command.

5. Specify the number of bypass chains. For example:

   ```
   set_edt_options -bypass_chain 2
   ```

   For more information, see the set_bypass_chains command.

6. Specify the input and output pins for the bypass chains. For example:

   ```
   set_bypass_chains 2 -pins scan_in2 scan_out2
   ```

   For more information, see the set_bypass_chains command.

7. Generate the EDT logic. For more information, see "Creation of EDT Logic Files" on page 77.

Related Topics

Synthesizing the EDT Logic
Creating Bypass Test Patterns in Uncompressed ATPG

Dual Bypass Configurations

You can use the set_edt_options *-single_bypass_chain* command to output EDT logic with two bypass configurations as follows:

- **Default scan chain configuration** — All scan chains are evenly distributed and concatenated into scan chains equal to the number of input/output channels in the EDT logic. This configuration can also be specified with the *set_edt_options -bypass_chains* command. For more information, see "Compression Bypass Logic" on page 172.
- **Single bypass scan chain configuration** — All scan chains are concatenated together to form one scan chain for bypass mode. A single bypass chain configuration can be used in test environments with hardware limitations.

When dual configurations are specified, an additional primary input *edt_single_bypass_chain* pin is created to enable and disable the single chain configuration. For more information, see "Single Chain Bypass Logic" on page 277.

An additional dofile <*design*>*_single_bypass_chain.dofile* is also produced to define the single top-level scan chain and force the edt_single_bypass_chain pin to 1.

Additional lockup cells are inserted as needed. For more information, see "Lockups in the Bypass Circuitry" on page 196.

By default only test patterns for the default configuration are saved. To save the test patterns for the single chain bypass configuration, you must use the write_patterns *edt_single_bypass_chain* command.

Note

Single bypass chain configuration is associated with one compression block. To use this feature in a block-level architecture, you must manually integrate all single bypass chains together at the top-level.

Note

The single bypass configuration is not included in reported test pattern statistics and scan chains. Only information about the default bypass configuration is reported.

Related Topics

Structure of the Bypass Logic	Lockups in the Bypass Circuitry
Bypass Pattern Flow Example	Structure of the Bypass Chains
Bypass Mode Files	

Generating Identical EDT and Bypass Test Patterns

The EDT technology supports the creation of uncompressed versions of each EDT pattern. The availability of uncompressed EDT patterns enables you to use uncompressed ATPG in bypass mode to directly load the scan cells with the same values that compressed ATPG loads. For debugging simulation mismatches in the core logic, it is sometimes helpful if you can apply the exact same patterns with uncompressed ATPG in bypass mode that you applied with compressed ATPG.

Note

You can only convert EDT test patterns to uncompressed test patterns for bypass mode if the bypass scan chains are created with compressed ATPG. Otherwise, you must use uncompressed ATPG to generate bypass test patterns. See "Creating Bypass Test Patterns in Uncompressed ATPG" on page 180.

After you generate EDT patterns in the Pattern Generation phase, you can direct the tool to translate the EDT patterns into bypass mode uncompressed ATPG patterns and write the translated patterns to a file. The file format is the same as the regular uncompressed ATPG binary file format. You accomplish the translation and create the binary file by issuing the write_patterns command with the -*EDT_Bypass* and -*Binary* switches. For example:

```
write_patterns my_bypass_patterns.bin -binary -edt_bypass
```

You can then read the binary file into uncompressed ATPG, optionally simulate the patterns in the analysis system mode to verify that the expected values computed in compressed ATPG are still valid in bypass mode, and save the patterns in any of the tool's supported formats; WGL or Verilog for example. An example of this tool flow is provided in the section, "Using Bypass Patterns in Uncompressed ATPG."

There are several reasons you cannot use EDT technology alone to create the EDT bypass patterns:

- The bypass operation requires a different set of test procedures. These are only loaded when running uncompressed ATPG and are unknown to EDT in the Pattern Generation phase.

  If the bypass test procedures produce different tied cell values than the EDT test procedures, simulation mismatches can result if the EDT patterns are simply reformatted for bypass mode. An example of this would be if a boundary scan TAP controller were used to drive the EDT bypass signal. The two sets of test procedures would cause the register driving the signal to be forced to different values and the expected values computed for EDT would therefore not be correct for bypass mode.
- EDT would not have run any DRCs to ensure that the scan chains can be traced in bypass mode.
- You may need to verify that captured values do not change in bypass mode.

When it translates EDT patterns into bypass patterns, EDT changes the captured values on some scan cells to Xs to emulate effects of EDT compaction and scan chain masking. For example, if two scan cells are XOR'd together in the compactor and one of them had captured an X, the tool sets the captured value of the other to X so no fault can be detected on those cells, incorrectly credited, then lost during compaction.

Similarly, if a scan chain is masked for a given pattern, the tool sets captured values on all scan cells in that chain to X. When translating the EDT patterns, the tool preserves those Xs so the two pattern sets are identical. While this can lower the "observability" possible with the bypass patterns, it emulates EDT test conditions. For more information on how EDT uses masking, refer to "Understanding Scan Chain Masking in the Compactor."

Chain Test Pattern Handling for Bypass Operation

The EDT technology saves only the translated EDT scan patterns in the binary file. The enhanced *chain + EDT logic* test patterns are not saved. The purpose of the enhanced test patterns is to verify the operation of the EDT logic as well as the scan chains. Because no shifting occurs through the EDT logic when it is bypassed, regular chain test patterns are sufficient to verify the scan chains work in bypass configuration; The regular chain test patterns are appended to the compressed test pattern set when you write out the bypass patterns.

Note

Because the EDT pattern set contains the enhanced test patterns and the bypass pattern set does not, the number of patterns in the EDT and bypass pattern sets are different.

You can use the bypass test patterns with uncompressed ATPG to debug problems in the core design and scan chains but not in the EDT logic. If the enhanced tests fail in compressed ATPG and the bypass chain test passes in uncompressed ATPG, the problem is probably in the EDT logic or the interface between the EDT logic and the scan chains.

Using Bypass Patterns in Uncompressed ATPG

After you save the bypass patterns, invoke Tessent Shell, read the design, and use the dofile and test procedure file generated when the EDT logic is created. You then read into Tessent Shell the binary pattern file you previously saved from compressed ATPG. You can optionally simulate the patterns in the analysis system mode to verify that the expected values computed with compressed ATPG are still valid in bypass mode. Then save the patterns in any of the tool's supported formats, WGL or Verilog for example.

Bypass Pattern Flow Example

Note

The following steps assume that, as part of a normal flow, you already have run Tessent Shell to create the EDT logic, followed by Design Compiler to synthesize it. You must complete both steps in order to run Tessent Shell with uncompressed ATPG in bypass mode. The bypass dofile and the bypass test procedure file generated by compressed ATPG are required by uncompressed ATPG in order to correctly apply a bypass pattern set.

In the compressed ATPG Pattern Generation phase, issue a "write_patterns -binary -edt_bypass" command to write bypass patterns. For example:

```
write_patterns my_bypass_patterns.bin -binary -edt_bypass
```

Notice that the *-Binary* and the *-Edt_bypass* switches are both required in order to write bypass patterns.

Setting Up Tessent Shell in Uncompressed ATPG

Invoke Tessent Shell in setup mode and invoke the bypass dofile generated by compressed ATPG. Place the design in the same state in uncompressed ATPG that you used in compressed ATPG, then run DRC.

Note

Placing the design in the same state in uncompressed ATPG as in compression ATPG ensures the expected test values in the bypass patterns remain valid when the design is configured for bypass operation.

The following example uses the bypass dofile, *created_bypass.dofile,* described in section "Creation of EDT Logic Files":

```
dofile created_bypass.dofile
set_system_mode analysis
```

Verify that no DRC violations occurred.

Processing the Bypass Patterns

To simulate the bypass patterns and verify the expected values, enter commands similar to the following:

```
read_patterns my_bypass_patterns.bin
report_failures -pdet
```

Note

The expected values in the binary pattern file mirror those with which compressed ATPG observes EDT patterns. Therefore, if compressed ATPG cannot observe a scan cell (for example, due to scan chain masking or compaction with a scan cell capturing an X), the expected value of the cell is set to X even if it can be observed by uncompressed ATPG in bypass mode.

Saving the Patterns with Compressed ATPG Observability

To save the patterns in another format using the expected values in the binary pattern file, issue the write_patterns command with the -*External* switch. For example, to save ASCII patterns:

```
read_patterns my_bypass_patterns.bin
write_patterns my_bypass_patterns.ascii -external
```

Saving the Patterns with Uncompressed ATPG Observability

Alternatively, you can save expected values based on what is observable by uncompressed ATPG when the design is in bypass operation. Some scan cells which had X expected values in compressed ATPG, due to scan chain masking or compaction with an X in another scan cell, may be observed by uncompressed ATPG. To write_patterns where the expected values reflect uncompressed ATPG observability, first simulate the patterns as follows:

```
set_system_mode analysis
read_patterns my_bypass_patterns.bin
simulate_patterns -store_patterns all
```

Note

The preceding command sequence will cause the Xs that emulate the effect of compaction in EDT to disappear from the expected values. The resultant bypass patterns will no longer be equivalent to the EDT patterns; only the stimuli will be identical in the two pattern sets. For a given EDT pattern, therefore, the corresponding bypass pattern will no longer provide test conditions identical to what the EDT pattern provided in compressed ATPG.

Using the -*Store_patterns* switch in analysis system mode when specifying the external file as the pattern source causes uncompressed ATPG to place the simulated patterns in the tool's internal pattern set. The simulated patterns include the load values read from the external pattern source and the expected values based on simulation.

Note

If you fault simulate the patterns loaded into uncompressed ATPG, the test coverage reported may be slightly higher than it actually is in compressed ATPG. This is because uncompressed ATPG recomputes the expected values during fault simulation rather than using the values in the external pattern file. The recomputed values do not reflect the effect of the compactors and scan chain masking that are unique to EDT. Therefore, there likely will be fewer Xs in the recomputed values, resulting in the higher coverage number.

When you subsequently save these patterns, take care <u>not</u> to use the -*External* switch with the write_patterns command. The -*External* switch saves the current external pattern set rather than the internal pattern set containing the simulated expected values. The following example saves the simulated expected values in the internal pattern set to the file, *my_bypass_patterns.ascii*:

```
write_patterns my_bypass_patterns.ascii
```

Creating Bypass Test Patterns in Uncompressed ATPG

Use this procedure to generate test patterns for the bypass chains located in your netlist or in the EDT logic.

Prerequisites

- If a signal other than the edt_bypass signal is used for the mux select that enables the bypass chains, the test procedure file for the bypass chains must be modified to allow bypass chains to be traced.
- EDT logic must be created and synthesized into your netlist, and the bypass dofile and test procedure files generated by compressed ATPG are available.

Procedure

1. Invoke Tessent Shell. The setup prompt displays.
2. Set the context, read in the design library.
3. Run the bypass dofile. For example:

   ```
   dofile created_bypass.dofile
   ```

4. Change to analysis system mode to run DRC. For example:

   ```
   set_system_mode analysis
   ```

5. Check for and debug any DRC violations.
6. Create uncompressed ATPG patterns as you would for a design without EDT. For example:

   ```
   add_faults /my_core
   create_patterns
   report_statistics
   report_scan_volume
   ```

   This example creates patterns with dynamic compression. Be sure to add faults only on the core of the design (assumed to be "/my_core" in this example) and disregard the EDT logic.

   The report_scan_volume command provides information for analyzing pattern data and achieved compression.

   Uncompressed ATPG patterns that utilize the bypass circuitry are generated.

Related Topics

Using Bypass Patterns in Uncompressed ATPG

Preparation for Test Pattern Generation

Generating Test Patterns

Simulating the Generated Test Patterns

Uncompressed ATPG (External Flow) and Boundary Scan

The information in this section applies to the external compressed pattern flow. For more information on this flow, see "Compressed Pattern External Flow" on page 33.

Flow overview

Note

As mentioned previously, boundary scan cells must not be present in your design before you add the EDT logic. This is the same requirement that applies to I/O pads and is for the same reason; to enable compressed ATPG to create the EDT logic as a wrapper around your core design.

Once the EDT logic is created, you can use any tool to insert boundary scan.

When you insert boundary scan, you typically configure the TAP controller in one of two ways:

- Drive the minimal amount of the EDT control circuitry with the TAP controller, so the boundary scan simply coexists with EDT. This is described in the next section, "Boundary Scan Coexisting with EDT Logic."
- Drive the EDT logic clock, update, and bypass signals with the TAP controller as described in the section, "Driving Compressed ATPG with the TAP Controller."

These two approaches are described in the following sections.

Boundary Scan Coexisting with EDT Logic

This section describes how EDT logic can coexist with boundary scan and provides a flow reference for this methodology. This approach enables the EDT logic to be controlled by primary input pins and not by the boundary scan circuitry. In test mode, the boundary scan circuitry just needs to be reset. Also, all PIs and POs are directly accessible.

Preparing to Synthesize Boundary Scan and EDT Logic

Prior to synthesizing the EDT logic and boundary scan circuitry, you should ensure any scripts used for synthesis include the boundary scan circuitry. For example, the Design Compiler synthesis script that compressed ATPG generates needs the following modifications (shown in bold font) to ensure the boundary scan circuitry is synthesized along with the EDT logic:

Note

The modifications are to the example script shown in the "Design Compiler Synthesis Script External Flow" section of Chapter 4.

```
/************************************************************************
**  Synopsys Design Compiler synthesis script for created_edt_bs_top.v
**
************************************************************************/

/* Read input design files */
read -f verilog created_core_blackbox.v
read -f verilog created_edt.v
read -f verilog created_edt_top.v
read -f verilog edt_top_bscan.v      /*ADDED*/

current_design edt_top_bscan         /*MODIFIED*/

/* Check design for inconsistencies */
check_design

/* Timing specification */
create_clock -period 10 -waveform {0,5} edt_clock
create_clock -period 10 -waveform {0,5} tck  /*ADDED*/

/* Avoid clock buffering during synthesis. However, remember */
/* to perform clock tree synthesis later for edt_clock */
set_clock_transition 0.0 edt_clock
set_dont_touch_network edt_clock
set_clock_transition 0.0 tck         /*ADDED*/
set_dont_touch_network tck           /*ADDED*/

/* Avoid assign statements in the synthesized netlist.
set_fix_multiple_port_nets -feedthroughs -outputs -buffer_constants

/* Compile design */
uniquify
set_dont_touch cpu
compile -map_effort medium

/* Report design results for EDT logic */
report_area > created_dc_script_report.out
report_constraint -all_violators -verbose >>
      created_dc_script_report.out
report_timing -path full -delay max >> created_dc_script_report.out
report_reference >> created_dc_script_report.out

/* Remove top-level module */
remove_design cpu

/* Read in the original core netlist */
read -f verilog gate_scan.v
current_design edt_top_bscan         /*MODIFIED*/
link

/* Write output netlist using a new file name*/
write -f verilog -hierarchy -o created_edt_bs_top_gate.v   /*MODIFIED*/
```

After you have made any required modifications to the synthesis script to support boundary scan, you are ready to synthesize the design. This is described in the section, "The EDT Logic Synthesis Script."

Modifying the Dofile and Procedure File for Boundary Scan

Note

The information in this section applies only when the design includes boundary scan.

To correctly operate boundary scan circuitry, you need to edit the dofile and test procedure file created by compressed ATPG. Typical changes include:

- The internal scan chains are one level deeper in the hierarchy because of the additional level added by the boundary scan wrapper. This needs to be taken into consideration for the add_scan_chains command.
- The boundary scan circuitry needs to be initialized. This typically requires you to revise both the dofile and test procedure file.
- You may need to make additional changes if you drive compressed ATPG signals with the TAP controller.

In the simplest configuration, the EDT logic is controlled by primary input pins, not by the boundary scan circuitry. In test mode, the boundary scan circuitry just needs to be reset.

Following is the same dofile shown in the Chapter 4 section, "Test Pattern Generation Files," except now it includes the changes (shown in bold font) necessary to support boundary scan when configured simply to coexist with EDT logic. The boundary scan circuitry is assumed to include a TRST asynchronous reset for the TAP controller.

```
add_scan_groups grp1 modified_edt.testproc

add_scan_chains -internal chain1 grp1 /core_i/cpu_i/edt_si1
   /core_i/cpu_i/edt_so1
add_scan_chains -internal chain2 grp1 /core_i/cpu_i/edt_si2
   /core_i/cpu_i/edt_so2
add_scan_chains -internal chain3 grp1 /core_i/cpu_i/edt_si3
   /core_i/cpu_i/edt_so3
add_scan_chains -internal chain4 grp1 /core_i/cpu_i/edt_si4
   /core_i/cpu_i/edt_so4
add_scan_chains -internal chain5 grp1 /core_i/cpu_i/edt_si5
   /core_i/cpu_i/edt_so5
add_scan_chains -internal chain6 grp1 /core_i/cpu_i/edt_si6
   /core_i/cpu_i/edt_so6
add_scan_chains -internal chain7 grp1 /core_i/cpu_i/edt_si7
   /core_i/cpu_i/edt_so7
add_scan_chains -internal chain8 grp1 /core_i/cpu_i/edt_si8
   /core_i/cpu_i/edt_so8

add_clocks 0 clk
add_clocks 0 edt_clock

add_pin_constraints tms C1

add_write_controls 0 ramclk

add_read_controls 0 ramclk

add_pin_constraints edt_clock C0

set_edt_options -channels 1 -ip_version 1
```

The test procedure file, *created_edt.testproc*, shown in the Chapter 4 section, "Test Pattern Generation Files," must also be changed to accommodate boundary scan circuitry that you configure to simply coexist with EDT logic. Here is that file again, but with example changes for boundary scan added (in bold font). This modified file was saved with the new name *modified_edt.testproc*, the name referenced in the fifth line of the preceding dofile.

```
set time scale 1.000000 ns ;
set strobe_window time 100 ;

timeplate gen_tp1 =
   force_pi 0 ;
   measure_po 100 ;
   pulse clk 200 100;
   pulse edt_clock 200 100;
   pulse ramclk 200 100;
   period 400 ;
end;

procedure capture =
   timeplate gen_tp1 ;
   cycle =
      force_pi ;
      measure_po ;
      pulse_capture_clock ;
   end;
end;

procedure shift =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   cycle =
      force_sci ;
      force edt_update 0 ;
      measure_sco ;
      pulse clk ;
      pulse edt_clock ;
   end;
end;

procedure load_unload =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   cycle =
      force clk 0 ;
      force edt_bypass 0 ;
      force edt_clock 0 ;
      force edt_update 1 ;
      force ramclk 0 ;
      force scan_en 1 ;
      pulse edt_clock ;
   end ;
   apply shift 26;
end;

procedure test_setup =
   timeplate gen_tp1 ;
```

```
      cycle =
         force edt_clock 0 ;
         ...
         force tms 1;
         force tck 0;
         force trst 0;
      end;
      cycle =
         force trst 1;
      end;
   end;
```

Driving Compressed ATPG with the TAP Controller

You can drive one or more compressed ATPG signals from the TAP controller; however, there are a few more requirements and restrictions than in the simplest case where the boundary scan just coexists with EDT logic. Some of these apply when you set up the boundary scan circuitry, others when you generate patterns:

- If you want to completely drive the EDT logic from the TAP controller, you first should decide on an instruction to drive the EDT channels.
- To ensure the TAP controller stays in the proper state for shift as well as capture during EDT pattern generation, you should specify TCK as the capture clock. This requires a "set_capture_clock *TCK -atpg*" command in the EDT dofile that causes the capture clock TCK to be pulsed only once during the capture cycle.
- Also, the TAP controller must step through the Exit1-DR, Update-DR, and Select-DR-Scan states to go from the Shift-DR state to the Capture-DR state. This requires three intervening TCK pulses between the pulse corresponding to the last shift and the capture. These three pulses need to be suppressed for the clock supplied to the core.
- The EDT update signal is usually asserted during the first cycle of the load/unload procedure, so as not to restrict clocking in the capture window. Typically, the EDT clock must be in its off state in the capture window. Because there is already a restriction in the capture window due to the "set_capture_clock *TCK -atpg*" command, you can supply the EDT clock from the same waveform as the core clock without adding any more constraints. To update the EDT logic, the EDT update signal must now be asserted in the capture window. You can use the Capture-DR signal from the TAP controller to drive the EDT update signal.
- You should also modify any synthesis scripts to include the boundary scan circuitry. For an example of a Design Compiler script with the necessary changes, see "Preparing to Synthesize Boundary Scan and EDT Logic" on page 182.

Using Pipeline Stages in the Compactor

Pipeline stages can sometimes improve the overall rate of data transfer through the logic in the compactor by increasing the scan shift frequencies. Pipeline stages are flip-flops that hold intermediate values output by a logic level so that values entering that logic level can be updated earlier in a clock cycle. Because the EDT logic is relatively shallow, most designs do need compactor pipeline stages to attain the desired shift frequency. The limiting factors on shift frequencies are usually the performance of the scan chains and power considerations.

You can enable the addition of pipeline stages in the compactor with the set_edt_options -Pipeline_logic_levels_in_compactor command when creating the EDT logic. Pipeline stages added to the compactor use the EDT clock and lockup cells as described in "Lockups Between Scan Chain Outputs and Compactor" on page 194.

Note

The -*Pipeline_logic_levels_in_compactor* switch specifies the maximum number of combinational logic levels (XOR gates) between compactor pipeline stages, not the number of pipeline stages. The number of logic levels between any two pipeline stages controls the propagation delay between pipeline stages.

Using Pipeline Stages Between Pads and Channel Inputs or Outputs

When the signal propagation delay between a pad and the corresponding channel input or output is excessive, you may want to add pipeline stages. Use the guidelines provided in this section to add pipeline stages between a top-level channel input pin/pad and the corresponding decompressor input, or between a compactor output and the corresponding channel output pin/pad. The number of pipeline stages on each input/output channel can vary.

Typically, pipeline stages are inserted throughout the design during top-level design integration. Pipeline stages are generally not placed within the EDT logic.

Note

You must use the set_edt_pins -Pipeline_stages command during test pattern generation to enable channel pipeline stages. You must also modify the associated test procedure file as described in the following subsections.

Channel Output Pipelining

To support channel output pipelines, the tool ensures there are enough shift cycles per pattern to flush out the pipeline and observe all scan chains. Without pipelining, the number of additional shift cycles per pattern, compared to the length of the longest scan chain, is typically four. As long as the total number of output pipeline stages (including both compactor and channel output pipelining) is less than or equal to four, no additional shift cycles are added. If the total number of output pipeline stages is more than four, the number of additional shift cycles is increased to equal the number of pipeline stages.

Channel Input Pipelining

While the contents of the channel output pipeline stages at the beginning of shifting each pattern are irrelevant since they will be flushed out, the contents of the channel input pipeline stages do matter because they will go to the decompressor when shifting begins (just after the decompressor is initialized in the load_unload procedure).

The tool adds an additional test pattern before every test pattern set. This test pattern initializes the channel input pipelining stages before the load of the very first real test pattern.

The number of additional shift cycles is typically incremented by the number of channel input pipeline stages. If the number of additional shift cycles is four without input pipelining, and the channel input with the most pipeline stages has two stages, the number of additional shift cycles in each test pattern is incremented to six.

If you have a choice between using either input or output pipeline stages, you should choose output stages for the following reasons:

- The number of shift cycles for the same number of pipeline stages is higher when the pipeline stages are on the input side.
- You must ensure that input channel pipelines hold their value during the capture cycle. For information on how to do this, see "Ensuring Input Channel Pipelines Hold Their Value During Capture" on page 189.

Clocking of Channel Input Pipeline Stages

If you use channel input pipelining, you must ensure there is no clock skew between the channel input pipeline and the decompressor. If you use channel output pipelining, you must ensure there is no clock skew between the compactor (if you also use compactor pipelining) and the channel output pipeline, or between the scan chain outputs (if no compactor pipelining is used) and the channel output pipeline.

On the input side, the pipeline stages are connected to the decompressor, which is clocked by the leading edge of the EDT clock. If the channel input pipeline is not clocked by the EDT clock, a lockup cell must be inserted between the pipeline and the decompressor.

Note

EDT patterns saved for application through bypass mode (write_patterns *-EDT_Bypass*) may not work correctly if the first cell of a chain, driven by channel input pipeline stages in bypass mode, captures on the trailing edge of the clock. This is because that first cell of the chain, which is normally a master, becomes a copy of the last input pipeline stage in bypass mode. To resolve this, you must add a lockup cell that is clocked on the trailing edge of a shift clock at the end of the pipeline stages for a particular channel input. This ensures that the first cell in the scan chain remains a master.

Clocking of Channel Output Pipeline Stages

On the output side, the last state element driving the channel output is either a compactor pipeline stage clocked by the EDT clock or the last elements of the scan chains when the compactor has no pipelining. In addition to ensuring no clock skew between the chains/compactor and the pipeline stages, you must ensure that the first pipeline stages capture on the leading edge (LE) when no compactor pipelining is used. This is because if the last scan cell in a chain captures on the LE and the path from the last scan cell to the channel pipeline is combinational, and the channel pipeline stage captures on the trailing edge (TE), the pipeline stage is essentially a copy during shift and the last scan cell no longer gets observed.

To ensure there is no clock skew between the pipeline stages and the compactor outputs, you can use the set_edt_pins -*CHange_edge_on_compactor_output* command to specify whether compactor output data changes on the LE or TE of the EDT clock. For example, specify the compactor output changes at the trailing edge of the clock before feeding LE pipeline stages. Depending on your application, compressed ATPG automatically inserts lockup cells and output channel pipeline stages as needed. For more information, see set_edt_pins in the *Tessent Shell Reference Manual*.

If you use pipeline stages clocked with the rising edge of the edt_clock, the tool inserts lockup cells in the IP Creation phase to balance clock skew on the output side pipeline registers. For more information, see “Lockups in the Bypass Circuitry”.

Note
If the clock used for the pipeline stages is not a shift clock, it must be pulsed in the shift procedure.

Ensuring Input Channel Pipelines Hold Their Value During Capture

As mentioned earlier in the “Channel Input Pipelining” section, the tool adds an additional test pattern before every test pattern set to initialize channel input pipelining stages before the load of the first test pattern. Following the initialization pattern, the tool ensures that every generated pattern has sufficient trailing zeros (ones for channels with pad inversion) to set the pipeline stages to zeros/ones after every pattern is shifted in.

You must ensure that the values that get shifted into the input pipeline stages at the *end* of shift (for every pattern) are not changed during capture. You can ensure this in one of the following ways:

- Constrain the clock used for the pipeline stages off.
- Constrain the channel input pin to 0 (or 1 in case of channel inversion).

Note

During scan pattern retargeting or when EDT Mapping or EDT Finder is enabled, TestKompress automatically adds proper constraints to input channels if pipelines are detected and their clocks are not constrained off during capture. For more information on EDT mapping and EDT Finder, see set_edt_mapping and set_edt_finder in the *Tessent Shell Reference Manual*. For more information on scan pattern retargeting, see "Scan Pattern Retargeting" in the *Tessent Scan and ATPG User's Manual*.

Since the EDT clock is already constrained during the capture cycle, and drives the decompressor (no clock skew), using the EDT clock to control the input pipeline stages is recommended.

Note

If the pipeline stages use the EDT clock, the channel pins must be forced to zero (or one if there is channel inversion) in load_unload as well, since the EDT clock is pulsed there as well (to reset the decompressor and update the mask logic). TestKompress will automatically add the needed force statements in the load_unload procedure if they are not already added by the user.

DRC for Channel Input Pipelining

The K19 and K22 design rules detect errors in initializing the channel input pipeline stages. If the pipeline is not correctly initialized for the first pattern, K19 reports mismatches on the EDT block channel inputs - assuming the hierarchy is not dissolved and the EDT logic is identified. If the EDT logic channel inputs cannot be located, for example because the design hierarchy was dissolved, K19 reports that Xs are shifted out of the decompressor. On the EDT logic channel inputs, the simulated values would mismatch within the first values shifted out, while the rest of the bits subsequently applied would match.

If the pipeline is correctly initialized for the first pattern and K19 passes, but the pipeline contents change (during capture or the following load_unload prior to shift) such that it no longer contains zeros, K22 fails. K19 and K22 detect these cases if input channel pipelining is defined and issue warnings about the possible problems related to channel pipelining.

DRC for Channel Output Pipelining

The K20 rule check considers channel output pipelining, in addition to any compactor pipelining that may exist. K20 reports any discrepancy between the number of identified and specified pipeline stages between the scan chains and pins (including compactor and channel output pipelines).

If the first stage of the channel output pipeline is TE instead of LE, this will result in one less cycle of delay than expected, which will also trigger a K20 violation. If the first stage is TE, and the user specifies one less pipeline stage, those 2 errors may mask each other and no violation may be reported. However, this may result in mismatches during serial pattern simulation.

Input/Output Pipeline Examples

The following command defines two pipeline stages for input channel 1:

```
set_edt_pins input_channel 1 -pipeline_stages 2
```

This example sets the EDT context to core1 (EDT context is specific to modular compressed ATPG and is explained in the Modular Compressed ATPG chapter), and then specifies that all output channels of the core1 block have one pipeline stage:

```
set_current_edt_block core1
set_edt_pins output_channel -pipeline_stages 1
```

Following is the modified load_unload procedure for a design with two channels having input pipelining; edt_channel1 has inversion and edt_channel2 does not. The input pipeline stages are clocked by the EDT clock, edt_clock. The user-added events that support pipelining are shown in bold and comments are shown in italics.

```
procedure load_unload =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   cycle =
      // To ensure the values shifted into the input pipeline stages at
      // the end of shift are not changed during capture, you must force
      // channel pins with pipelines to zero (or one if there is channel
      // inversion) since edt_clock is pulsed in load_unload and is also
      // used for the pipeline stages.
      force edt_channel1 1 ;
      force edt_channel2 0 ;
      force system_clk 0 ;
      force edt_bypass 0 ;
      force edt_clock 0 ;
      force edt_update 1 ;
      force ramclk 0 ;
      force scan_en 1 ;
      pulse edt_clock ;
   end;
   apply shift 21 ;
end;
```

Understanding Lockup Cells

The tool analyzes the timing relationships of the clocks that control the sequential elements between the scan chains and the EDT logic and inserts edge-triggered flip-flops (lockup cells) when necessary to synchronize the clocks and ensure data integrity.

For more information about lockup cells, see “Merging Scan Chains with Different Shift Clocks” in the *Tessent Scan and ATPG User’s Manual*. You can use the report_edt_lockup_cells command to display a detailed report of the lockup cells the tool has inserted.

Lockup Cell Insertion

The tool analyzes the relationship between the clock that controls each sequential element sourcing data (source clock) and the clock that controls the sequential element receiving the data (destination clock) and inserts a lockup cell when the source and destination clocks overlap as follows:

- Both clocks have identical waveform timing within a tester cycle; clocks are *on* at the same time and their edges are aligned.
- The active edge of the destination clock occurs later in the cycle than the active edge of the source clock.

When clocks are non-overlapping, data is protected by the timing sequence and no lockup cells are inserted.

Note

Partially overlapping clocks are not supported.

You can set up the EDT logic clock and scan chain shift clocks to be non-overlapping by pulsing the EDT clock before the shift clock of each scan chain. When the EDT logic is set up in this manner, there is no need for lockup cells between the EDT logic and scan chains. However, a lockup cell driven by the EDT clock is still inserted between all bypass scan chains. For more information, see “Pulse EDT Clock Before Scan Shift Clocks” on page 64.

If your design contains a mix of overlapping and non-overlapping clocking, or the shift clocks are pulsed before the EDT logic clock, you must let the tool analyze the design and insert lockup cells (default behavior) as described in the following sections.

Lockup Cell Analysis For Bypass Lockup Cells Not Included as Part of the EDT Chains

This section includes the following sections:

- Lockups Between Decompressor and Scan Chain Inputs
- Lockups Between Scan Chain Outputs and Compactor
- Lockups in the Bypass Circuitry

Lockups Between Decompressor and Scan Chain Inputs

The decompressor is located between the scan channel input pins and the scan chain inputs. It contains sequential circuitry clocked by the EDT clock. As the off state of the EDT clock (at the EDT logic module port) is always 0, leading edge triggered (LE) flip-flops are used in this sequential circuitry. Scan chain clocking does not utilize the EDT clock. Therefore, there is a possibility of clock skew between the decompressor and the scan chain inputs.

For each scan chain, the tool analyzes the clock timing of the last sequential element in the decompressor stage (source) and the first active sequential element in the scan chain (destination).

Note

The first sequential element in the scan chain could be an existing lockup cell (a transparent latch for example) and may not be part of the first scan cell in the chain.

The tool analyzes the need for lockup cells on the basis of the waveform edge timings (change edge and capture edge, respectively) of the source and destination clocks. The change edge is typically the first time at which the data on the source scan cell's output may update. The capture edge is the capturing transition at which data is latched on the destination scan cell's output. The tool inserts lockup cells between the decompressor and scan chains based on the following rules:

- A lockup cell is inserted when a source cell's change edge coincides with the destination cell's capture edge.
- A lockup cell is inserted when the change edge of the source cell precedes the capture edge of the destination cell.

In addition, the tool attempts to place lockup cells in a way that introduces no additional delay between the decompressor and the scan chains and tries to minimize the number of lockup cells at the input side of the scan chains. The lockup cells are driven by the EDT clock to reduce routing of the system clocks from the core to the EDT logic.

Table 7-3 summarizes the relationships and the lockup cells the tool inserts on the basis of the preceding rules, assuming there is no pre-existing lockup cell (transparent latch) between the decompressor and the first scan cell in each chain.

Table 7-3. Lockup Cells Between Decompressor and Scan Chain Inputs

Clock Waveforms	Source clock	Dest. clock	Source[1] change edge	Dest.[1, 2] capture edge	# Lockups inserted	Lockup[3] edge(s)
Overlapping	EDT clock	Scan clock	LE	LE	1	TE
	EDT clock	Scan clock	LE	TE	2	TE, LE
	EDT clock	Scan clock	LE	active high (TE)	2	TE, LE
	EDT clock	Scan clock	LE	active low (LE)	2	TE, LE
Non-Overlapping[4]	EDT clock	Scan clock	LE	LE	2	TE, LE
	EDT clock	Scan clock	LE	TE	2	TE, LE
	EDT clock	Scan clock	LE	active high (TE)	2	TE, LE
	EDT clock	Scan clock	LE	active low (LE)	2	TE, LE

1. LE = Leading edge, TE = Trailing edge.
2. Active high/low = Active clock level when destination is a latch. Active high means the latch is active when the primary input (PI) clock is on. Active low means the latch is active when the PI clock is off. (LE) or (TE) indicates the clock edge corresponding to the latch’s capture edge.
3. Lockup cells are driven by the EDT clock.
4. These are cases for which the tool determines the source edge precedes the destination edge. (Lockups are unnecessary if the destination edge precedes the source edge).

To minimize the number of lockup cells added, the tool always adds a trailing edge triggered (TE) lockup cell at the output of the LFSM in the decompressor. The tool adds a second LE lockup cell at the input of the scan chain only when necessary, as shown in Table 7-3.

Note

If there is a pre-existing transparent latch between the decompressor and the first scan cell, a single lockup cell (LE) is added between the decompressor and the latch. This ensures the correct value is captured into the first scan cell from the decompressor.

Lockups Between Scan Chain Outputs and Compactor

When compactor pipeline stages are inserted, lockup cells are inserted as needed in front of the first pipeline stage. Pipeline stages are LE flip-flops clocked by the EDT clock, similar to the sequential elements in the decompressor.

The clock timing between the last active sequential element in the scan chain (source) and the first sequential element (first pipeline stage) that it feeds in the compactor (destination) is analyzed. Similar to the input side of the scan chains, the tool analyzes the need for lockup cells on the basis of the waveform edge timings (change edge and capture edge, respectively, of the source and destination clocks). The change edge is typically the first time at which the data on the source scan cell's output may update. The capture edge is the capturing transition at which data is latched on the destination scan cell's output.

Lockup cells driven by the EDT clock are added according to the following rules:

- A lockup cell is inserted when a source cell's change edge coincides with the destination cell's capture edge.
- A lockup cell is inserted when the change edge of the source cell precedes the capture edge of the destination cell.

In addition, the tool attempts to place lockup cells in a way that introduces no additional delay between the scan chains and the compactor pipeline stages. It also tries to minimize the number of lockup cells at the output side of the scan chains. The lockup cells are driven by the EDT clock so as to reduce routing of the system clocks from the core to the EDT logic.

Table 7-4 shows how the tool inserts lockup cells in the compactor.

Table 7-4. Lockup Cells Between Scan Chain Outputs and Compactor

Clock Waveforms	**Source clock**	**Dest. clock**	**Source[1, 2] change edge**	**Dest.[1] capture edge**	**# Lockups inserted**	**Lockup[3] edge(s)**
Overlapping	Scan clock	EDT clock	LE	LE	1	TE
	Scan clock	EDT clock	TE	LE	none	-
	Scan clock	EDT clock	active high (LE)	LE	1	TE
	Scan clock	EDT clock	active low (TE)	LE	none	-
Non-Overlapping[4]	Scan clock	EDT clock	LE	LE	1	TE
	Scan clock	EDT clock	TE	LE	1	TE
	Scan clock	EDT clock	active high (LE)	LE	1	TE
	Scan clock	EDT clock	active low (TE)	LE	1	TE

[1] LE = Leading edge, TE = Trailing edge.
[2] Active high/low = Active clock level when source is a latch. Active high means the latch is active when the primary input (PI) clock is on. Active low means the latch is active when the PI clock is off. (LE) or (TE) indicates the clock edge corresponding to the latch's change edge.

3. Lockup cells are driven by the EDT clock.
4. These are cases for which the tool determines the source edge precedes the destination edge. (Lockups are unnecessary if the destination edge precedes the source edge).

Lockups in the Bypass Circuitry

The number and location of lockup cells the tool inserts in the bypass logic depend on the active edges (change edge and capture edge, respectively) of the source and destination clocks. The change edge is typically the first time at which the data on the source scan cell's output may update. The capture edge is the capturing transition at which data is latched on the destination scan cell's output.

The number and location of lockup cells also depend on whether the first and last active sequential elements in the scan chain are clocked by the same clock. The first and last active sequential elements in a scan chain could be existing lockup cells and may not be part of a scan cell. The tool inserts the lockup cells between source and destination scan cells according to the following rules:

- A lockup cell is inserted when a source cell's change edge coincides with the destination cell's capture edge and the cells are clocked by different clocks.
- A lockup cell is inserted when the change edge of the source cell precedes the capture edge of the destination cell.
- If multiple lockup cells are inserted, the tool ensures that:
    - A master/copy scan cell combination is always driven by the same clock. This prevents the situation where captured data in the master cell is lost because a different clock drives the copy cell and is not pulsed in a particular test pattern.
    - The earliest data capture edge of the last lockup cell is not before the latest time when the destination cell can capture new data. This makes the first scan cell of every chain a master and prevents D2 DRC violations.
    - If the earliest time when data is available at the output of the source is before the earliest data capture edge of the first lockup, the first lockup cell is driven with the same clock that drives the source.
- If there is an active lockup cell at the beginning of a scan chain, the tool identifies it and treats it as the source cell.
- If a lockup cell already exists at the end of a scan chain, the tool learns its behavior and treats it as the source cell.

Table 7-5 summarizes how the tool inserts lockup cells in the bypass circuitry.

Table 7-5. Bypass Lockup Cells

Clock Waveforms	Source[1] clock	Dest.[1] clock	Source[2, 3] change edge	Dest.[2, 3] capture edge	# Lockups inserted	Lockup edge(s)
Overlapping	clk1	clk1	LE	LE	none	-
	clk1	clk1	LE	TE	1	TE clk1
	clk1	clk1	TE	TE	none	-
	clk1	clk1	TE	LE	none	-
Overlapping	clk1	clk2	LE	LE	1	TE clk1
	clk1	clk2	LE	TE	2	LE clk1, TE clk2
	clk1	clk2	TE	TE	2	LE clk1, TE clk2
	clk1	clk2	TE	LE	none	-
Non-Overlapping[4]	clk1	clk2	LE	LE	2	LE clk1, TE clk2
	clk1	clk2	LE	TE	2	LE clk1, TE clk2
	clk1	clk2	TE	TE	2	LE clk1, TE clk2
	clk1	clk2	TE	LE	2	LE clk1, TE clk2
Overlapping	clk1	clk1	active high (LE)	active high (TE)	1	TE clk1
	clk1	clk1	active high (LE)	active low (LE)	1	TE clk1
	clk1	clk1	active low (TE)	active low (LE)	none	-
	clk1	clk1	active low (TE)	active high (TE)	none	-
Overlapping	clk1	clk2	active high (LE)	active high (TE)	2	LE clk1, TE clk2
	clk1	clk2	active high (LE)	active low (LE)	2	LE clk1, TE clk2
	clk1	clk2	active low (TE)	active low (LE)	none	-
	clk1	clk2	active low (TE)	active high (TE)	2	LE clk1, TE clk2

Table 7-5. Bypass Lockup Cells

Clock Waveforms	Source[1] clock	Dest.[1] clock	Source[2, 3] change edge	Dest.[2, 3] capture edge	# Lockups inserted	Lockup edge(s)
Non-Overlapping[4]	clk1	clk2	active high (LE)	active high (TE)	2	LE clk1, TE clk2
	clk1	clk2	active high (LE)	active low (LE)	2	LE clk1, TE clk2
	clk1	clk2	active low (TE)	active low (LE)	2	LE clk1, TE clk2
	clk1	clk2	active low (TE)	active high (TE)	2	LE clk1, TE clk2
Overlapping	clk1	clk1	LE	active high (TE)	1	TE clk1
	clk1	clk1	LE	active low (LE)	none	-
	clk1	clk1	active high (LE)	LE	none	-
	clk1	clk1	active low (TE)	LE	none	-
Overlapping	clk1	clk2	LE	active high (TE)	2	LE clk1, TE clk2
	clk1	clk2	LE	active low (LE)	2	LE clk1, TE clk2
	clk1	clk2	active high (LE)	LE	1	TE clk1
	clk1	clk2	active low (TE)	LE	none	-
Non-Overlapping[4]	clk1	clk2	LE	active high (TE)	2	LE clk1, TE clk2
	clk1	clk2	LE	active low (LE)	2	LE clk1, TE clk2
	clk1	clk2	active high (LE)	LE	2	LE clk1, TE clk2
	clk1	clk2	active low (TE)	LE	2	LE clk1, TE clk2

Table 7-5. Bypass Lockup Cells

Clock Waveforms	Source[1] clock	Dest.[1] clock	Source[2, 3] change edge	Dest.[2, 3] capture edge	# Lockups inserted	Lockup edge(s)
Overlapping	clk1	clk1	TE	active high (TE)	none	-
	clk1	clk1	TE	active low (LE)	none	-
	clk1	clk1	active high (LE)	TE	1	TE clk1
	clk1	clk1	active low (TE)	TE	none	-
Overlapping	clk1	clk2	TE	active high (TE)	2	LE clk1, TE clk2
	clk1	clk2	TE	active low (LE)	2	LE clk1, TE clk2
	clk1	clk2	active high (LE)	TE	2	LE clk1, TE clk2
	clk1	clk2	active low (TE)	TE	2	LE clk1, TE clk2
Non-Overlapping[4]	clk1	clk2	TE	active high (TE)	2	LE clk1, TE clk2
	clk1	clk2	TE	active low (LE)	2	LE clk1, TE clk2
	clk1	clk2	active high (LE)	TE	2	LE clk1, TE clk2
	clk1	clk2	active low (TE)	TE	2	LE clk1, TE clk2

[1] clk1 & clk2 are the functional (scan) clocks.

[2] LE = Leading edge, TE = Trailing edge.

[3] Active high/low = Active clock level when source or destination is a latch. Active high means the latch is active when the primary input (PI) clock is on. Active low means the latch is active when the PI clock is off. (LE) or (TE) indicates the clock edge corresponding to the latch’s change/capture edge.

[4] These are cases for which the tool determines the source edge precedes the destination edge. (Lockups are unnecessary if the destination edge precedes the source edge).

Lockup Cell Analysis For Bypass Lockup Cells Included as Part of the EDT Chains

This section describes how the tool adds lockup cells at the scan chain boundary to eliminate bypass only lockup cells. Sub-section "Differences Based on Inclusion/Exclusion of Bypass Lockup Cells in EDT Chains" provides a thorough explanation of the differences that result when bypass lockup cells are included in the EDT chain as opposed to when they are not.

This section is organized as follows:

- EDT Lockup and Scan Chain Boundary Lockup Cells
- Differences Based on Inclusion/Exclusion of Bypass Lockup Cells in EDT Chains
- Limitations
- Comparison of Bypass Lockup Cell Insertion Results

The tool analyzes the clocking of first and last active scan elements and adds lockup cells at scan chain inputs and outputs as required. These cells are added to ensure each scan chain starts with a LE register and ends with a TE register. These lockup cells are included as part of both EDT and EDT-bypass scan chains. They avoid clock skew problems between the decompressor and scan chains, scan chains and compactor, as well as when concatenating EDT scan chains into bypass chains. They also provide the ability to map EDT mode patterns into bypass mode.

As an exception, when all of the first and last scan elements are driven by the LE of the same clock and the compactor has no sequential registers, scan chain output lockup cells are added only for the last internal chain grouped into bypass chains.

EDT Lockup and Scan Chain Boundary Lockup Cells

When lockup cells at chain boundaries are inserted, the tool combines the analysis of decompressor and compactor lockup cells along with the scan chain input/output bypass lockup cells. Table 7-6 summarizes how the tool adds lockup cells for different clocking configurations.

Table 7-6. EDT Lockup and Scan Chain Boundary Lockup Cells

Clock (source → destination) (last cell → first cell)	**EDT Decompressor Lockup Cells**[1]	**Scan Chain Input Lockup**	**Scan Chain Output Lockup**	**Compactor Lockup Cell**[1]
Same source and destination clocks				
LE clk → LE clk	TE edt_clock	-	TE clk	-
LE clk → LE clk[2]	TE edt_clock	-	-	TE edt_clock
LE clk → TE clk	TE edt_clock	LE clk	TE clk	-
TE clk → LE clk	TE edt_clock	-	-	-

Table 7-6. EDT Lockup and Scan Chain Boundary Lockup Cells

Clock (source → destination) (last cell → first cell)	EDT Decompressor Lockup Cells[1]	Scan Chain Input Lockup	Scan Chain Output Lockup	Compactor Lockup Cell[1]
TE clk → TE clk	TE edt_clock	LE clk	-	-
Overlapping clocks, clkS and clkD				
LE clkS → LE clkD	TE edt_clock	-	TE clkS	-
LE clkS → TE clkD	TE edt_clock	LE clkD	TE clkS	-
TE clkS → LE clkD	TE edt_clock	-	-	-
TE clkS → TE clkD	TE edt_clock	LE clkD	-	-
Non-overlapping clocks, clkS overlaps with edt_clock, clkD later than edt_clock & clkS				
LE clkS → LE clkD	TE edt_clock	LE clkS	TE clkS	-
LE clkS → TE clkD	TE edt_clock	LE clkS	TE clkS	-
TE clkS → LE clkD	TE edt_clock	LE clkS	-	-
TE clkS → TE clkD	TE edt_clock	LE clkS	-	-
Non-overlapping clocks, clkS and clkD (either same or different clocks) later than edt_clock				
LE clkS → LE clkD	TE, LE edt_clock	-	TE clkS	-
LE clkS → TE clkD	TE, LE edt_clock	LE clkD	TE clkS	-
TE clkS → LE clkD	TE, LE edt_clock	-	-	-
TE clkS → TE clkD	TE, LE edt_clock	LE clkD	-	-
Overlapping clocks, same or different				
active high clkS (LE) → active high clkD (TE)	TE edt_clock	LE clkD	TE clkS	-
active high clkS (LE) → active low clkD (LE)	-	-	TE clkS	-
active low clkS (TE) → active high clkD (TE)	TE edt_clock	LE clkD	-	-
active low clkS (TE) → active low clkD (LE)	TE edt_clock	-	-	-
LE clkS → active high clkD (TE)	TE edt_clock	LE clkD	TE clkS	-
LE clkS → active low clkD (LE)	TE edt_clock	-	TE clkS	-

Table 7-6. EDT Lockup and Scan Chain Boundary Lockup Cells

Clock (source → destination) (last cell → first cell)	EDT Decompressor Lockup Cells[1]	Scan Chain Input Lockup	Scan Chain Output Lockup	Compactor Lockup Cell[1]
TE clkS → active high clkD (TE)	TE edt_clock	LE clkD	-	-
TE clkS → active low clkD (LE)	TE edt_clock	-	-	-
active high clkS (LE) → LE clkD	TE edt_clock	-	TE clkS	-
active high clkS (LE) → TE clkD	TE edt_clock	LE clkD	TE clkS	-
active low clkS (TE) → LE clkD	TE edt_clock	-	-	-
active low clkS (TE) → TE clkD	TE edt_clock	LE clkD	-	-

1. Decompressor and compactor lockup cells are not included as part of the EDT scan chains.
2. Special case where all scan cells are clocked by a single LE clock.

Differences Based on Inclusion/Exclusion of Bypass Lockup Cells in EDT Chains

The tool adds decompressor/compactor lockup cells and scan chain lockup cells according to the rules described in the following sections:

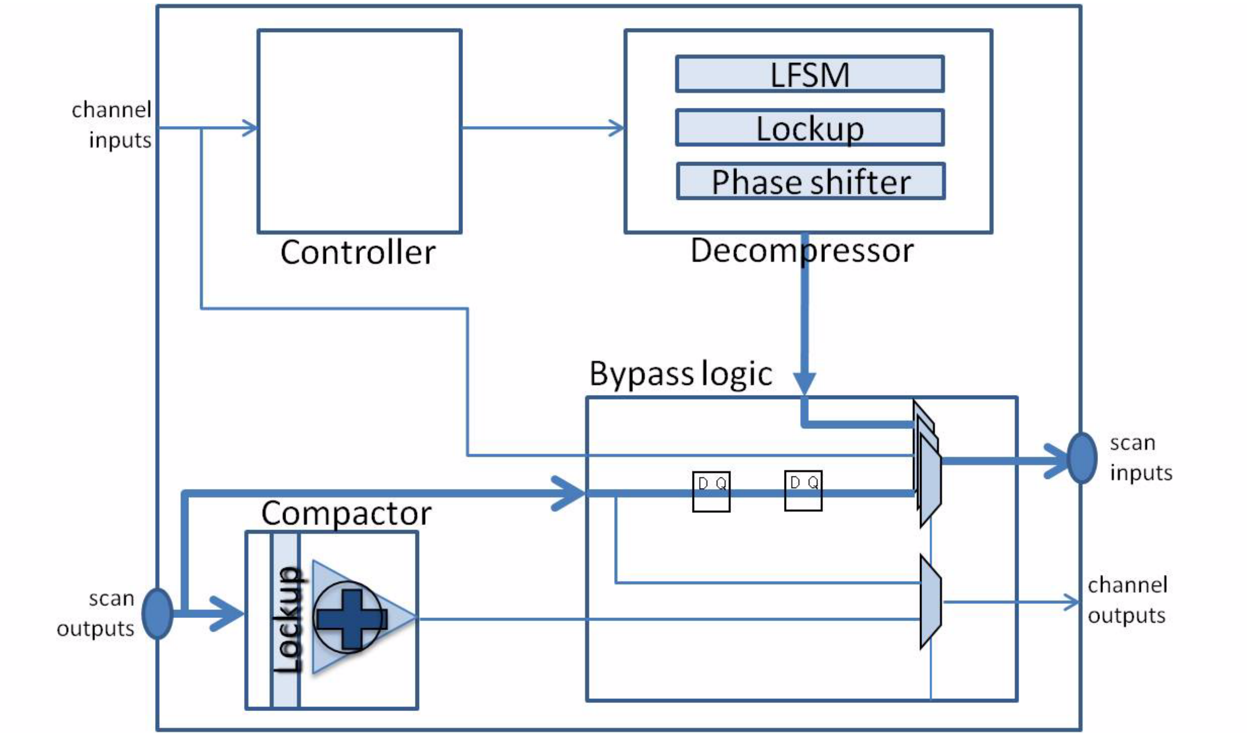
- Internal Scan Chain Definition
- Insertion Algorithm When Bypass Lockup Cells are Included at the Boundary of the EDT Chains
- Single Bypass Chain

Complete information on how the tool adds lockup cells when they are not included in EDT chains is presented in “Lockup Cell Analysis For Bypass Lockup Cells Not Included as Part of the EDT Chains.”

Internal Scan Chain Definition

Figure 7-17 illustrates the internal scan chain definition anchor points (scan inputs and scan outputs) during pattern generation when bypass lockup cells are not included as part of the EDT scan chains.

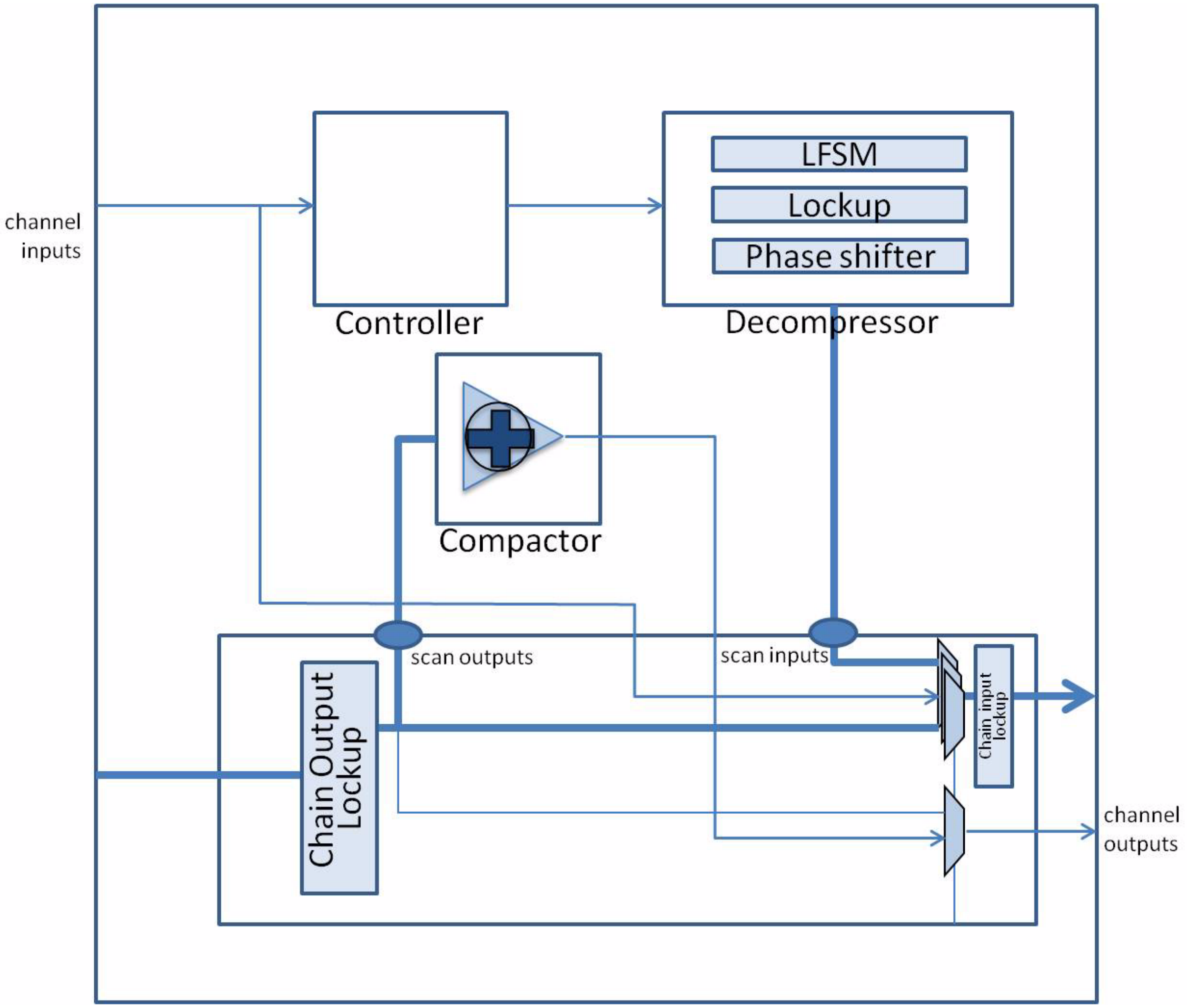
Figure 7-17. Scan Chain and Bypass Lockup Cells Not in the EDT Scan Chain



You can insert bypass lockup cells such that they are included as part of the EDT scan chains. This allows the tool to see the actual bypass lockup cells and account for them correctly.

Figure 7-18 illustrates the internal scan chain definition anchor points (scan inputs and scan outputs) when bypass lockup cells are included as part of the EDT scan chains.

Figure 7-18. Scan Chain and Bypass Lockup Cells in the EDT Scan Chain



Note

The lockup cells inside bypass logic are now included as part of the EDT scan chains as well. The first level lockup cells for the decompressor are still excluded from the scan chain definition as before.

Insertion Algorithm When Bypass Lockup Cells are Included at the Boundary of the EDT Chains

As shown in Figure 7-18, when bypass lockup cells are included in the EDT scan chain, TestKompress does the following:

- If the last scan cell is a LE scan cell, the tool adds a TE lockup cell clocked by the last scan cell clock to the scan chain output.
- If the first scan cell is a TE scan cell, the tool adds a LE lockup cell to the scan chain input.
- If all of the scan chains are clocked by the LE of the same clock, the tool makes an exception and adds lockup cells only for the last internal scan chain of each bypass chain. This facilitates the concatenation of the bypass chains of an EDT block at a higher level.
- The LE lockup cell at the scan chain input is clocked by the source scan clock if it has an early waveform compared with the destination scan clock; otherwise, the lockup cell is clocked by the destination scan clock.
- The second decompressor lockup cell is not required when the destination scan cell is a TE with the same waveform as the EDT clock. When the first scan cell has a late clock, the second decompressor lockup cell is included only if the lockup cell at the last scan chain output is pulsed with a late clock.
- The new lockup cell can influence EDT and compactor lockup cells because these new lockup cells are visible in the EDT path and are cumulative with dedicated EDT-only lockup cells in the decompressor and compactor.
- Compactor lockup cell analysis includes the source lockup cell at the scan chain output. In particular, if a TE source lockup cell is needed for a bypass lockup cell, it will also be used for a compactor lockup cell.

Single Bypass Chain

When using this functionality, bypass lockup cells are also added at the input of the first and output of the last internal chains grouped into a bypass chain. This enables the regular bypass chains to be easily concatenated to form the single bypass chain for the entire EDT block.

The lockup cells for bypass mode concatenation also allow concatenating the single bypass chain of all EDT blocks declared in the tool during IP creation. TestKompress does not actually concatenate the single bypass chains of the EDT blocks; rather TestKompress facilitates the process for some other tool to make such a concatenation.

You can concatenate the single bypass chains of all the EDT blocks in a design to construct a system-wide single bypass chain, even across blocks not declared in IP creation. In such cases, if the source clock from the preceding EDT block is pulsed earlier than the destination clock from the succeeding EDT block in the system-wide single chain concatenation order, these scan cells will become a master-copy pair. This should be properly accounted for when translating EDT mode patterns into the single system-wide bypass chain patterns

Limitations

This functionality cannot be used with the following features:

- EDT integration flow using the add_edt_connections command is not supported.
- Generating a blackbox for the EDT logic using set_edt_options -blackbox on. When including bypass lockup cells in EDT scan chains, the scan chains are defined on the EDT decompressor and compactor instance pins in the EDT logic which are not available in a blackbox description of the EDT module.
- Pulsing EDT clock before shift clock — Since the bypass lockup cells are clocked by edt_clock in this case, including them as part of the scan chains will result in D1 violations on all the lockup cells at scan chain inputs.

Comparison of Bypass Lockup Cell Insertion Results

This section compares the circuitry created by bypass lockup cell insertion depending upon whether the bypass lockup cell is included or excluded from the EDT chain. The following three cases are illustrated:

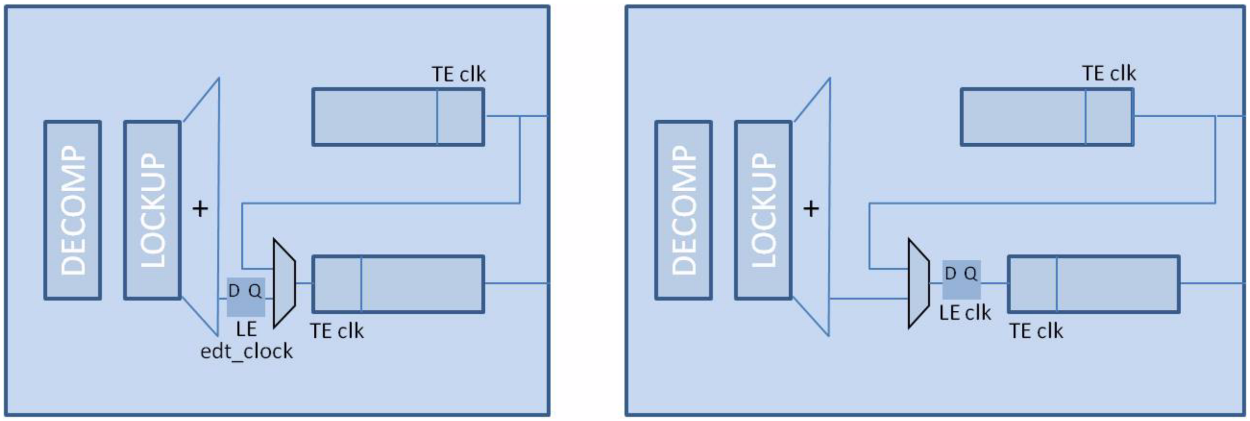
- Case 1: No Bypass Lockup Cell
- Case 2: One Bypass Lockup Cell
- Case 3: Two Bypass Lockup Cells

Case 1: No Bypass Lockup Cell

Figure 7-19 illustrates the circuitry when the bypass lockup cell insertion algorithm does not insert any lockup cells: on the left is the circuitry when bypass lockup cells are excluded from EDT chains, and on the right is the circuitry when they are included in EDT chains.

When both the last and first scan cells are TE and clocked by the same clock, a LE lockup cell is added to the destination scan chain input. In this case, the second decompressor lockup cell in the EDT decompressor is not added. This is shown in Figure 7-19. This is an example that demonstrates the case when the bypass lockup cell affects the EDT decompressor lockup cell.

Figure 7-19. TE CLK to TE CLK



Bypass Lockup Cell Excluded from EDT Chain

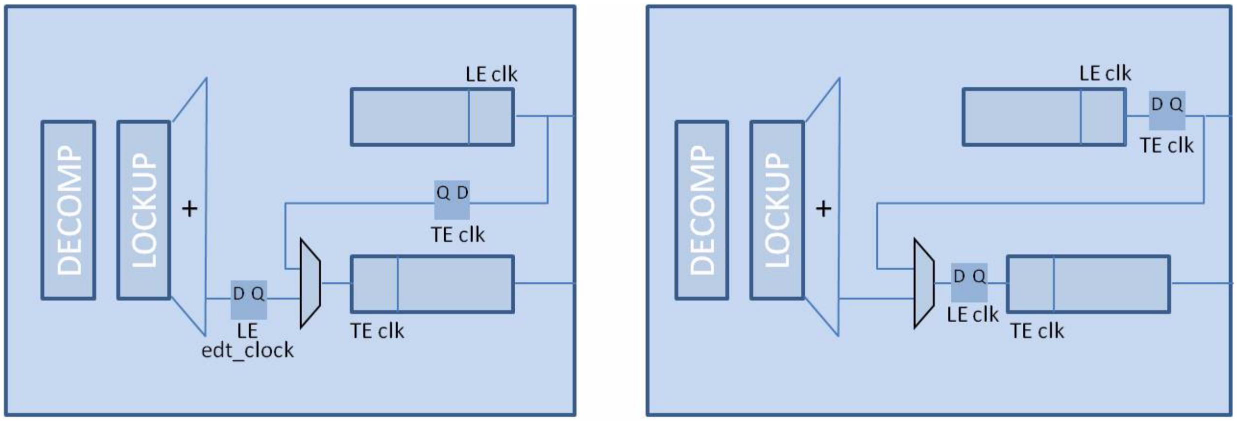
Bypass Lockup Cell Included in EDT Chain

Case 2: One Bypass Lockup Cell

When bypass lockup cells are excluded from the EDT chain, the tool inserts one bypass lockup cell in the following cases:

- If LE clk to TE clk, the tool inserts a TE clk lockup as illustrated, on the left, in Figure 7-20. Note, the absence of the second decompressor lockup cell, on the right, in Figure 7-20.

Figure 7-20. LE Clk to TE Clk

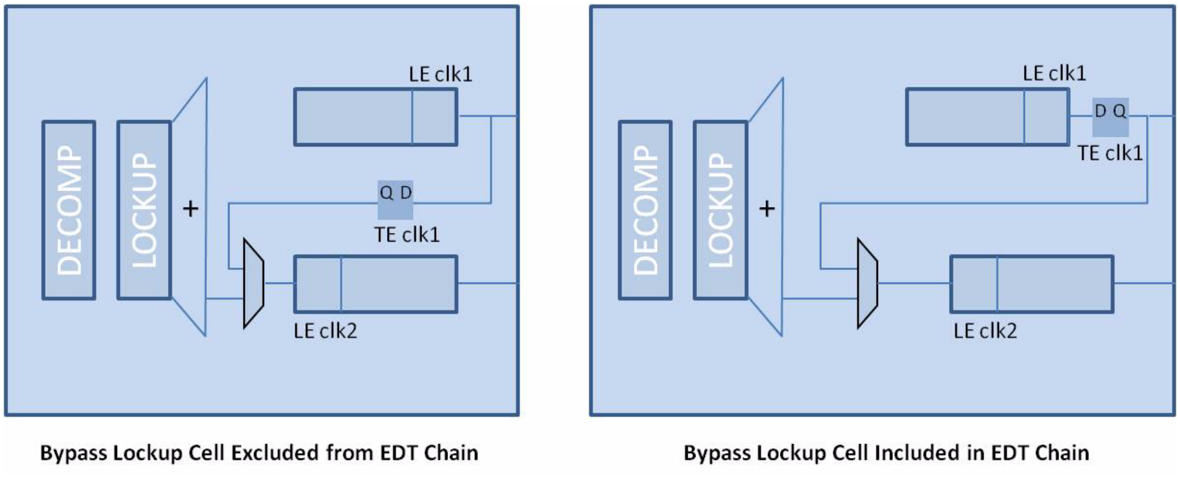


Bypass Lockup Cell Excluded from EDT Chain

Bypass Lockup Cell Included in EDT Chain

- If LE clk1 to LE clk2, the tool inserts a TE clk1 lockup as illustrated, on the left, in Figure 7-21.

Figure 7-21. LE Clk1 to LE Clk2 Overlapping



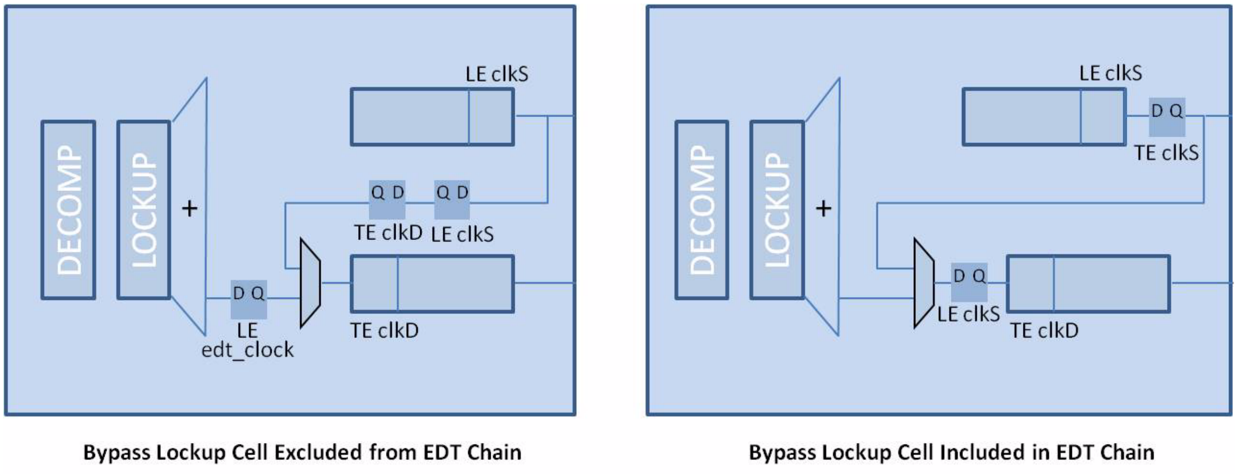
Bypass Lockup Cell Excluded from EDT Chain

Bypass Lockup Cell Included in EDT Chain

Case 3: Two Bypass Lockup Cells

Figure 7-22 illustrates the case where the tool infers two lockup cells in both cases, but the clock edges of the lockup cells are different. This case applies when both clkS and clkD are overlapping with the EDT clock, and when clkS overlaps with the EDT clock but clkD has a late waveform.

Figure 7-22. LE ClkS to TE ClkD

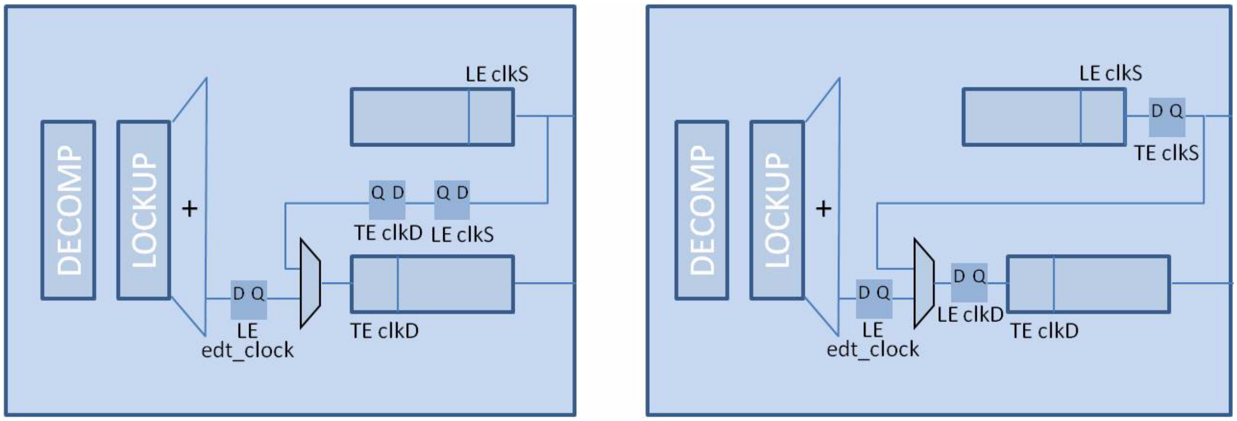


Bypass Lockup Cell Excluded from EDT Chain

Bypass Lockup Cell Included in EDT Chain

Figure 7-23 illustrates the case when the destination cell is TE, but the same situation applies when the destination cell is LE.

Figure 7-23. ClkS to ClkD, Both Clocks Later Than EDT Clock



Bypass Lockup Cell Excluded from EDT Chain

Bypass Lockup Cell Included in EDT Chain

Lockups Between Channel Outputs and Output Pipeline Stages

During the top-level design integration process, clocking requirements may require you to insert lockup cells between the EDT logic and pad terminals. If the clocking of the last scan cells compacted into an output channel and the clocking of the output pipeline stage (outside the EDT logic) overlap, you must add a lockup cell (outside the EDT logic). Tessent Shell in the EDT IP Creation phase does not insert these lockup cells.

However, if internal compactor pipelining is enabled in the EDT logic, and the output pipeline stages are active on the leading edge (LE) of the EDT clock, no lockup cells are necessary because the internal compactor pipeline stages also use the leading edge (LE) of the EDT clock.

For more information on pipeline stages, see "Using Pipeline Stages Between Pads and Channel Inputs or Outputs" on page 187".

If the output pipeline stages use a different edge or clock, the existing lockup cells may be insufficient, and you must specify the change edge for the compactor outputs or insert lockup cells manually. When you specify the change edge for the compactor outputs, the tool inserts pipeline stages and lockup cells as needed to ensure the compactor outputs change as specified.

You use the set_edt_pins command with the -*CHange_edge_at_compactor_output* option to specify the change edge for the compactor outputs. Depending on the change edge specified for the compactor outputs, the tool inserts lockup cells between the compactor output and output channels as described in Table 7-7.

Table 7-7. Lockup Insertion Between Channel Outputs and Output Pipeline

-CHange_edge_at_compactor_output	**Compactor pipeline stages**	**Lockup between scan chain and compactor**	**Last scan cell**	**Lockup inserted between compactor output & output channels**
LEading_edge_of_edt_clock	LE[1]	NA[2]	NA	none[3]
	none	LE	NA	none
	none	TE	NA	LE
	none	none	LE	none
	none	none	TE[4]	LE
TRailing_edge_of_edt_clock	LE	NA	NA	TE
	none	LE	NA	TE
	none	TE	NA	none
	none	none	LE	TE
	none	none	TE	none

1. LE indicates the leading edge of the clock pulse.
2. NA indicates the state of that column has no effect on the resulting action described in the right-most column (Lockup inserted between compactor output and output channels).
3. None indicates the object does not exist or is not inserted.
4. TE indicates the trailing edge of the clock pulse.

Related Topics

Merging Scan Chains with Different Shift Clocks in the *Tessent Scan and ATPG User's Manual*

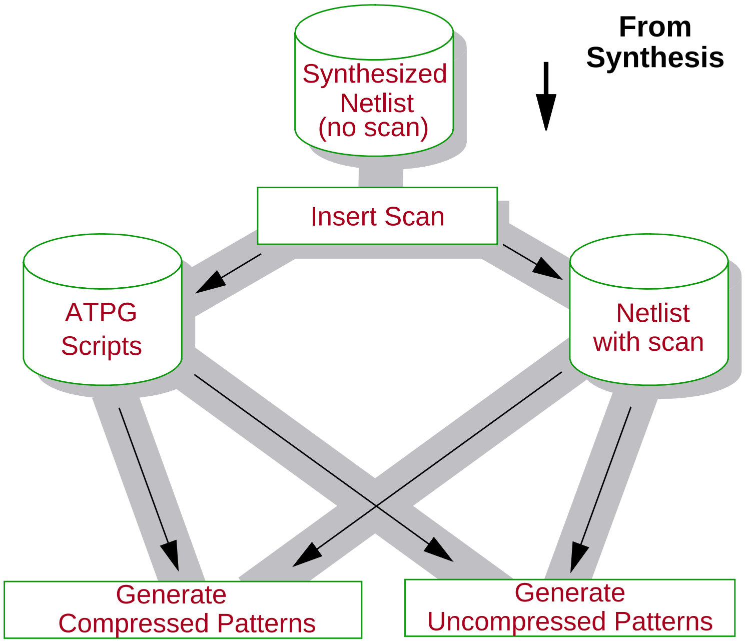
report_edt_lockup_cells
set_edt_options -retime_chain_boundaries

Evaluating Performance

The purpose of this section is to focus on the parts of the compressed ATPG flow that are necessary to perform experiments on compression rates and performance so you can make informed choices about how to fine-tune performance.

Figure 7-24 illustrates the typical evaluation flow.

Figure 7-24. Evaluation Flow


From Synthesis
Synthesized Netlist (no scan)
Insert Scan
ATPG Scripts
Netlist with scan
Generate Compressed Patterns
Generate Uncompressed Patterns

The complete Tessent TestKompress flow is described in section “Top-Down Design Flows.”

In an experimentation flow, where your intention is to verify how well EDT works in a design, you generate compressed patterns and use these patterns to verify coverage and pattern count, but not to perform final testing. Consequently, you do not need to write out the hardware description files. The first thing you should do, though, to make the data you obtain from running compressed ATPG meaningful, is establish a point of reference using uncompressed ATPG.

Establishing a Point of Reference

To illustrate how you establish a point of reference using uncompressed ATPG, assume as a starting point, that you have both a non-scan netlist and a netlist with eight scan chains. You would calculate the test data volume for measuring compression performance in the following way:

$$\begin{aligned}\text{Test Data Volume} &= (\text{\#scan loads}) \times (\text{volume per scan load}) \\ &= (\text{\#scan loads}) \times (\text{\#shifts per patterns}) \times (\text{\#scan channels})\end{aligned}$$

Note

#patterns may provide a reasonable approximation for #scan loads, but be aware that some patterns require multiple scan loads.

For a regular scan-based design without EDT, the volume per scan load will remain fairly constant for any number of scan chains because the number of shifts decreases when the number of chains increases. Therefore, it does not matter much which scan chain configuration you use when you establish the reference point.

The required steps to establish a point of reference are described briefly here. A design configured with eight scan chains is assumed.

1. Invoke Tessent Shell.

   ```
   <Tessent_Tree_Path>/bin/tessent -shell
   ```

2. Set the context, read in the netlist with eight scan chains and a library, and set the current design.

   ```
   set_context patterns -scan
   read_verilog mydesign_scan_8.v
   read_cell_library my_lib.atpg
   set_current_design top
   ```

3. Execute the dofile that performs basic setup.

   ```
   dofile atpg_8.dofile
   ```

4. Run DRC and verify that no DRC violations occur.

   ```
   set_system_mode analysis
   ```

5. Generate patterns. Assuming the design does not have RAMs, you can just generate basic patterns. To speed up the process, use fault sampling. It is important to use the same fault sample size in both the uncompressed and compressed runs.

   ```
   add_faults /cpu_i
   set_fault_sampling 10
   create_patterns
   report_statistics
   report_scan_volume
   ```

6. Note the test coverage and the total data volume as reported by the report_scan_volume command.

Measuring Performance

In these two runs (compressed and uncompressed), the numbers you will want to examine are:

- Test coverage (report_statistics)
- CPU time report_statistics)
- Scan data volume (report_scan_volume)

Another interesting number is the number of observable X sources (E5) violations which can explain lower compression performance.

Also, you can do a run that compares the results with and without fault sampling.

Improving Performance

Table 7-8 suggests some analyses you can do if the measured performance is not as expected:

Table 7-8. Summary of Performance Issues

Unsatisfactory Result	Suggested Analysis
Compression	- Many observable X sources. Examine E5 violations. - Too short scan chain vs. # of additional shift cycles.[1] Verify the # of additional shift cycles, and scan chain length using the report_edt_configurations command.
Run time	- Untestable/hard to compress patterns. If they cause a high runtime for uncompressed ATPG, they will also cause a high runtime for compressed ATPG. - If compressed ATPG has a much larger runtime than uncompressed ATPG, examine X sources, E5 violations.
Coverage	- Shared scan chain I/Os. Scan pins are masked by default. These pins should be dedicated. - Too aggressive compression (chain-to-channel ratio too high), leading to incompressible patterns. Use Report Aborted Faults command to debug. Look for EDT aborted faults.

[1] Additional shift cycles refers to the sum of the initialization cycles, masking bits (when using Xpress), and low-power bits (when using a low-power decompressor).

Varying the Number of Scan Chains

The effective compression depends primarily on the ratio between the number of internal scan chains and the number of external scan channels. In most cases, it is sufficient to just do an approximate configuration. For example, if the number of scan channels is eight and you need 4X compression, you can configure the design with 38 chains. This will typically result in 3.5X to 4.5X compression.

In certain cases, such a rough estimate is not enough. Usually, the number of scan channels is fixed because it depends on characteristics of the tester. Therefore, to experiment with different compression outcomes, different versions of the netlist (each with a different number of scan chains) are necessary.

Varying the Number of Scan Channels

Another alternative is to first use a design with a relatively high number of scan chains, and experiment with different numbers of channels. You can do these experiments, varying the chain-to-channel ratio. Then, when you find the optimum ratio, reconfigure the scan chains to match the number of scan channels you want. You can achieve similar test data volume reduction for a 100:10 configuration as for a 50:5 configuration.

For example, assume you have a design with 350,000 gates and 27,000 scan cells. If a certain tester requires the chip to have 16 scan channels, and your compression goal is to have no less than 4X compression, you might proceed as follows:

1. Determine the approximate number of scan chains you need. This example assumes a reasonable estimate is 60 scan chains.
2. Use Tessent Scan to configure the design with many more scan chains than you estimated, say, 100 scan chains.
3. Run the tool for 30, 26, 22, and 18 scan channels. Notice that these numbers are all between 1-2X the 16 channels you need.

Note

Use the same commands with compressed ATPG that you used with uncompressed ATPG when you established a point of reference, with one exception: with compressed ATPG, you must use the set_edt_options command to reconfigure the number of scan channels.

Suppose the results show that you achieve 4X compression of the test data volume using 22 scan channels. This is a chain-to-channel ratio of 100:22 or 4.55. For the final design, where you want to have 16 scan channels, you would expect approximately a 4X reduction with 16 x 4.55 = 73 scan chains.

Determining the Limits of Compression

You will find that the maximum amount of compression you can attain is limited by the ratio of scan chains to channels. If the number of scan channels is fixed, the number of scan chains in your design becomes the limiting factor. For example, if your design has eight scan chains, the most compression you can achieve under optimum conditions will be less than 8X compression. To exceed this maximum, you would need to reconfigure the design with a higher number of scan chains.

Speeding up the Process

If you need to perform multiple iterations, either by changing the number of scan chains or the number of scan channels, you can speed up the process by using fault sampling. When you use fault sampling, first perform uncompressed ATPG with fault sampling. Then, use the same fault sample when generating compressed patterns.

Note

You should always use the entire fault list when you do the final test pattern generation. Use fault sampling only in preliminary runs to obtain an estimate of test coverage with a relatively short test runtime. Be aware that sampling has the potential to produce a skewed result and is a means of estimation only.

Understanding Compactor Options

There are two compactors available in compressed ATPG:

- Xpress

  The Xpress compactor is the second generation compactor generated by default. The Xpress compactor optimizes compression for all designs but is especially effective for designs that generate X values. The Xpress compactor observes all chains with known values and masks out scan chains that contain X values. This X handling results in fewer test patterns being required for designs that generate X values.

  Depending on the application, the EDT logic generated with the Xpress compactor requires additional clocking cycles. The additional clocking cycles are determined by the ratio of scan chains to output channels and are relatively few when compared with the total shift cycles.

- Basic

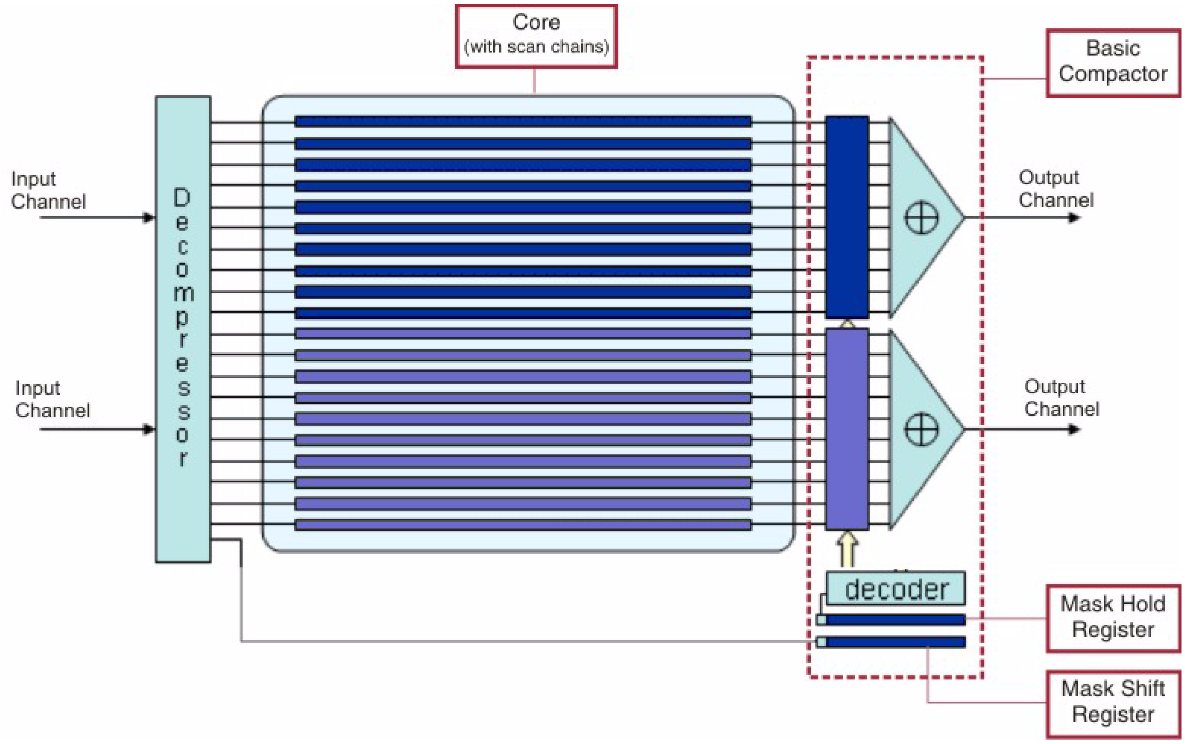
  The basic compactor is the first generation compactor enabled with the -COMpactor_type BAsic switch with the set_edt_options command.

  The basic compactor should be used for designs that do not generate many unknown (X) values. Due to scan cell masking, the basic compactor is significantly less effective on designs that generate unknown (X) values in scan cells when a test pattern is applied.

  The EDT logic generated when the basic compactor is used may be up to 30% smaller than EDT logic generated when the Xpress compactor is used. However, when X values are present, more test patterns may be required.

Basic Compactor Architecture

Figure 7-25. Basic Compactor

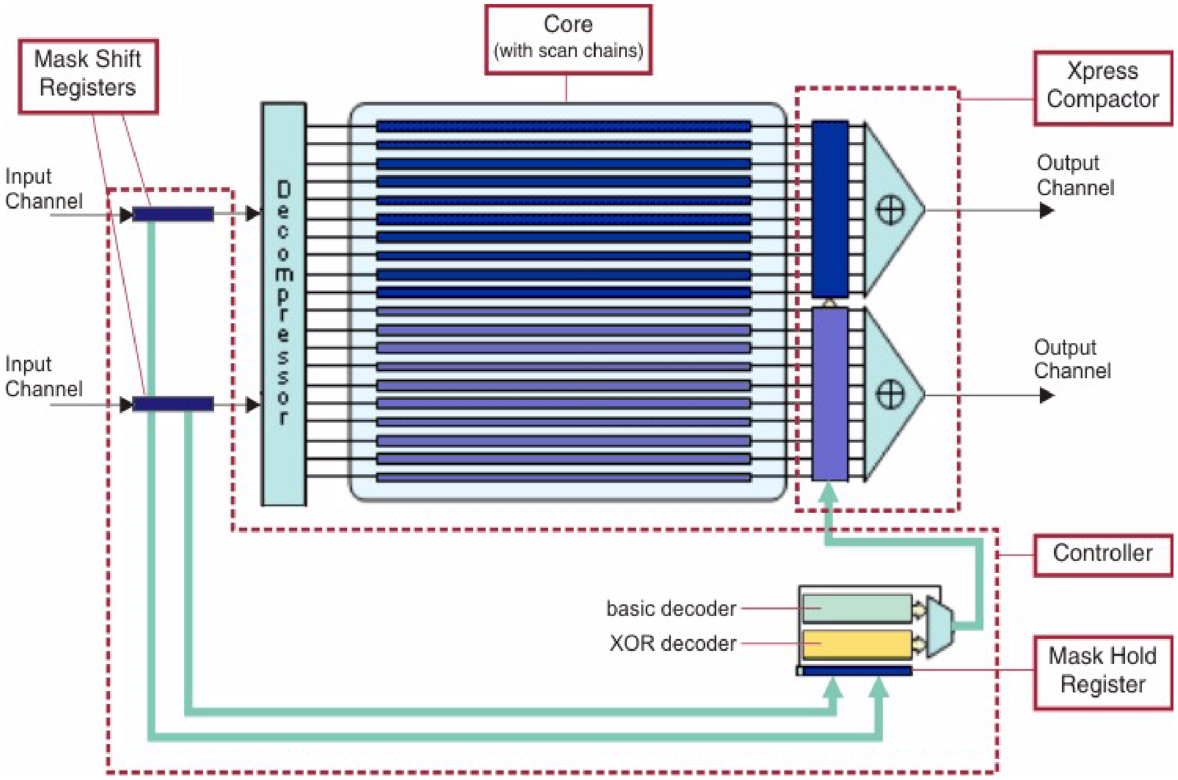


A mask code (prepended with a decoder mode bit) is generated with each test pattern to determine which scan chains are masked or observed. The basic compactor determines which chains to observe or mask using the mask code as follows:

1. The decompressor loads the mask code into the mask shift register.
2. The mask code is parallel-loaded into the mask hold register, where the decoder mode bit determines the observe mode: either one scan chain or all scan chains.
3. The mask code in the mask hold register is decoded and each bit drives one input of a masking AND gate in the compactor. Depending on the observe mode, the output of these AND gates is either enabled or disabled.

Xpress Compactor Architecture

Figure 7-26. Xpress Compactor



A mask code (prepended with a decoder mode bit) is generated with each test pattern to determine which scan chains are masked or observed. The Xpress compactor determines which chains to observe or mask using the mask code as follows:

1. Each test pattern is loaded into the decompressor through a mask shift register on the input channel.
2. The mask code is appended to each test pattern and remains in the mask shift register once the test pattern is completely loaded into the decompressor.
3. The mask code is then parallel-loaded into the mask hold register, where the decoder mode bit determines whether the basic decoder or the XOR decoder is used on the mask code.
   - The basic decoder selects only one scan chain per compactor. The basic decoder is selected when there is a very high rate of X values during scan testing or during chain test to allow failing chains to be fully observed and easy to diagnose.
   - The XOR decoder masks or observes multiple scan chains per compactor, depending on the mask code. For example, if the mask code is all 1s, then all the scan chains are observed.

4. The decoder output is shifted through a multiplexer, and each bit drives one input on the masking AND gates in the compactor to either disable or enable the output, depending on the decoder mode and bit value.

Understanding Scan Chain Masking in the Compactor

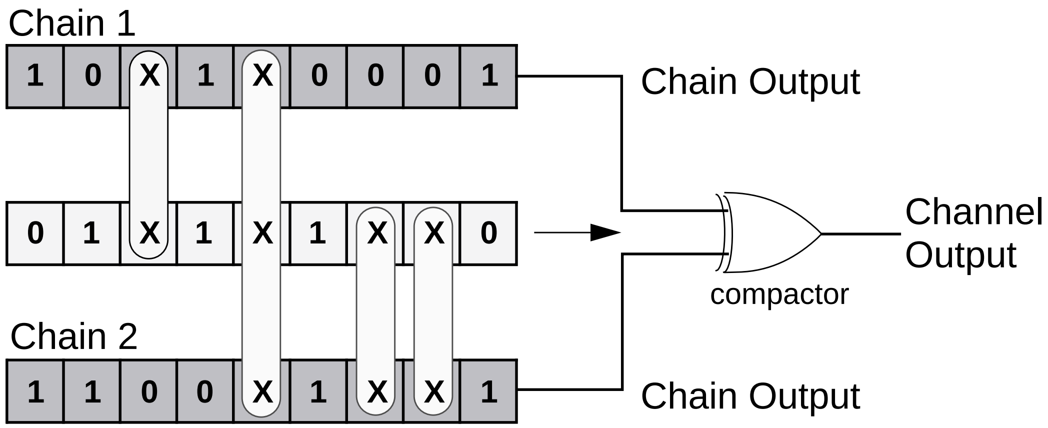
This section describes how and why scan chain masking is used in the compactor to ensure accurate scan chain observations.

Why Masking is Needed

To facilitate compression, the tool inserts a compactor between the scan chain outputs and the scan channel outputs. In this circuitry, one or more stages of XOR gates compact the response from several chains into each channel output. Scan chains compacted into the same scan channel are said to be in the same compactor group.

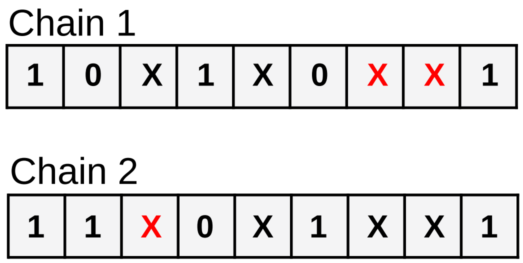
One common problem with different compactor strategies is handling of Xs (unknown values). Scan cells can capture X values from unmodeled blocks, memories, non-scan cells, and so forth. Assume two scan chains are compacted into one channel. An X captured in Chain 1 will then block the corresponding cell in Chain 2. If this X occurs in Chain 1 for all patterns, the value in the corresponding cell in Chain 2 will never be measured. This is illustrated in Figure 7-27, where the row in the middle shows the values measured on the channel output.

Figure 7-27. X-Blocking in the Compactor



The tool records an X in the pattern file in every position made unmeasurable as a result of the actual occurrence of an X in the corresponding cell of a different scan chain in the same compactor group. This is referred to as X blocking. The capture data for Chain 1 and Chain 2 that you would see in the ASCII pattern file for this example would look similar to Figure 7-28. The Xs substituted by the tool for actual values, unmeasurable because of the compactor, are shown in red.

Figure 7-28. X Substitution for Unmeasurable Values



Chain 1	1	0	X	1	X	0	X	X	1
Chain 2	1	1	X	0	X	1	X	X	1

Resolving X Blocking with Scan Chain Masking

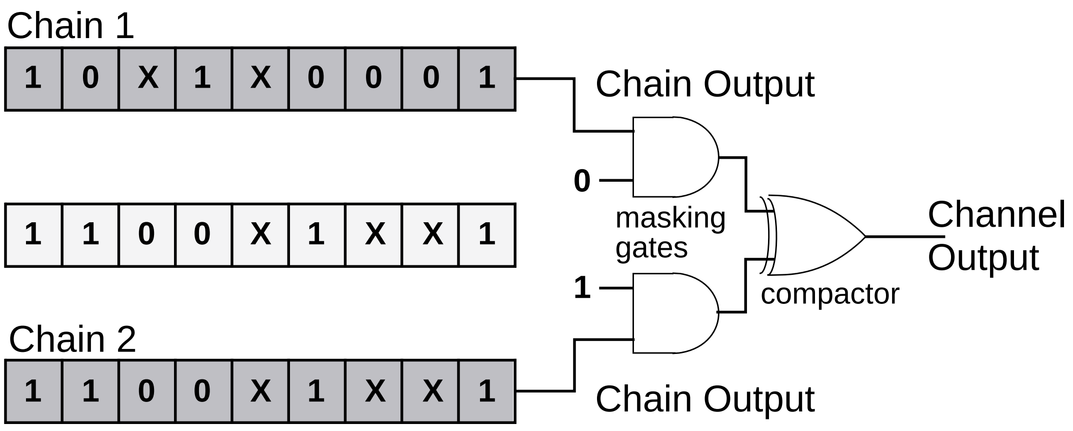
The solution to this problem is a mechanism utilized in the EDT logic called “scan chain masking.” This mechanism allows selection of individual scan chains on a per-pattern basis. Two types of scan chain masking are used: 1-hot masking and flexible masking.

- With 1-hot masking, only one chain is observed via each scan channel's compaction network. All the other chains in that compactor are masked so they produce a constant 0 to the input of the compactor. This allows observation of fault effects for the observed chains even if there are Xs in the observation cycles for the other chains. 1-hot masking patterns are only generated for a few ATPG cycles at points when the non-masking and flexible masking algorithms fail to detect any significant number of faults.

- Flexible masking patterns allow multiple chains to be observed via each scan channel's compaction network. Flexible masking is not fully non-masking; with fully non-masking patterns, none of the chains are masked so Xs in some cycles of some chains can block the observation of the fault effects in some other chain. The Xpress compactor observes all chains with known values and masks out those scan chains that contain X values so they do not block observation of other chains. With Xpress flexible masking, only a subset of the chains is masked to maximize the fault detection profile while reducing the impact on pattern count. When a fault effect cannot be observed at the channel output under any of the flexible masking configurations, the tool uses 1-hot masking to guarantee the detection of such faults.

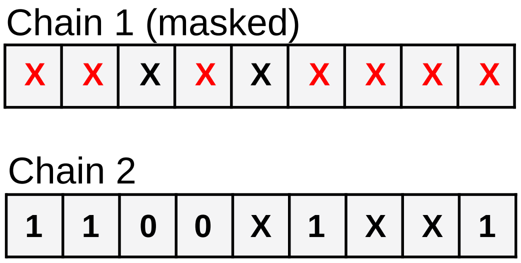
Figure 7-29 shows how scan chain masking would work for the example of the preceding section. For one pattern, only the values of Chain 2 are measured on the scan channel output. This way, the Xs in Chain 1 will not block values in Chain 2. Similar patterns would then also be produced where Chain 2 is disabled while the values of Chain 1 are observed on the scan channel output.

Figure 7-29. Example of Scan Chain Masking



When using scan chain masking, the tool records the actual measured value for each cell in the unmasked, selected scan chain in a compactor group. The tool masks the rest of the scan chains in the group, which means the tool changes the values to all Xs. With masking, the capture data for Chain 1 and Chain 2 that you would see in the ASCII pattern file would look similar to Figure 7-30, assuming Chain 2 is to be observed and Chain 1 is masked. The values the tool changed to X for the masked chain are shown in red.

Figure 7-30. Handling of Scan Chain Masking



Following is part of the transcript from a pattern generation run for a simple design where masked patterns were used to improve test coverage. The design has three scan chains, each containing three scan cells. One of the scan chain pins is shared with a functional pin, contrary to recommended practice, in order to illustrate the negative impact such sharing has on test coverage.

```
// ------------------------------------------------------
// Simulation performed for #gates = 134  #faults = 68
// system mode = analysis  pattern source = internal patterns
// ---------------------------------------------------------
// #patterns  test   #faults      #faults     #eff.    #test
// simulated  cvrg   in list      detected  patterns patterns
// deterministic ATPG invoked with abort limit = 30
// EDT without scan masking. Dynamic compaction disabled.
// ---       ------      ---          ---      ---      ---
//  32       82.51%       16           47        6        6
// ---       ------      ---          ---      ---      ---
//  Warning: Unsuccessful test for 10 faults.
//  deterministic ATPG invoked with abort limit = 30
//  EDT with scan masking. Dynamic compaction disabled.
// ---      ------      ---          ---      ---      ---
//  96      91.26%        0           16        6       12
// ---      ------      ---          ---      ---      ---
```

The transcript shows six non-masked and six masked patterns were required to detect all faults. Here's an excerpt from the ASCII pattern file for the run showing the last unmasked pattern and the first masked pattern:

```
pattern = 5;
apply "edt_grp1_load" 0 =
   chain "edt_channel1" = "00011000000";
end;
force   "PI" "100XXX0" 1;
measure "PO" "1XXX" 2;
pulse "/CLOCK" 3;
apply "grp1_unload" 4 =
   chain "chain1" = "1X1";
   chain "chain2" = "1X1";
   chain "chain3" = "0X1";
end;

pattern = 6;
apply "edt_grp1_load" 0 =
   chain "edt_channel1" = "11000000000";
end;
force   "PI" "110XXX0" 1;
measure "PO" "0XXX" 2;
pulse "/CLOCK" 3;
apply "grp1_unload" 4 =
   chain "chain1" = "XXX";
   chain "chain2" = "111";
   chain "chain3" = "XXX";
end;
```

The capture data for Pattern 6, the first masked pattern, shows that this pattern masks chain1 and chain3 and observes only chain2.

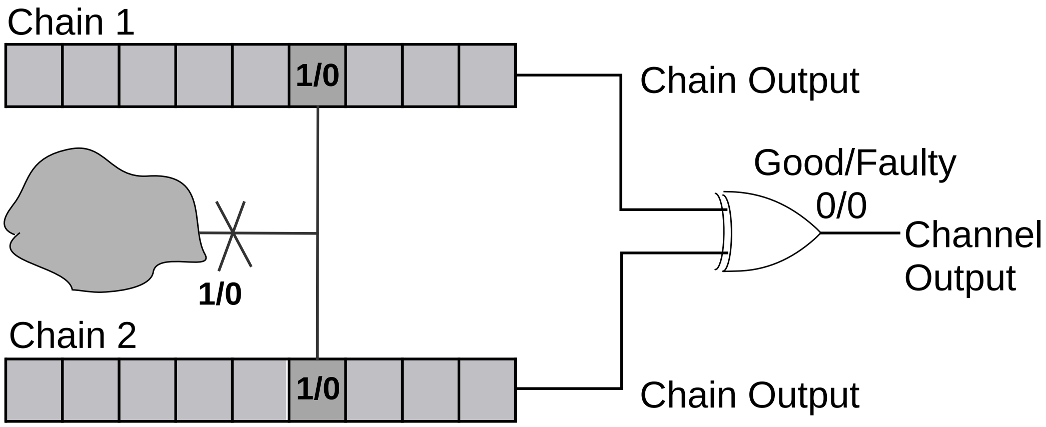
Fault Aliasing

Another potential issue with the compactor used in the EDT logic is called fault aliasing. Assume one fault is observed by two scan cells, and that these scan cells are located in two scan chains that are compacted to the same scan channel. Further, assume that these cells are in the same locations (columns) in the two chains and neither chain is masked. Figure 7-31 illustrates this.

Assume that the good value for a certain pattern is a 1 in the two scan cells. This corresponds to a 0 measured on the scan channel output, due to the XOR in the compactor. If a fault occurs on this site, 0s are measured in the scan cells, which also result in a 0 on the scan channel output.

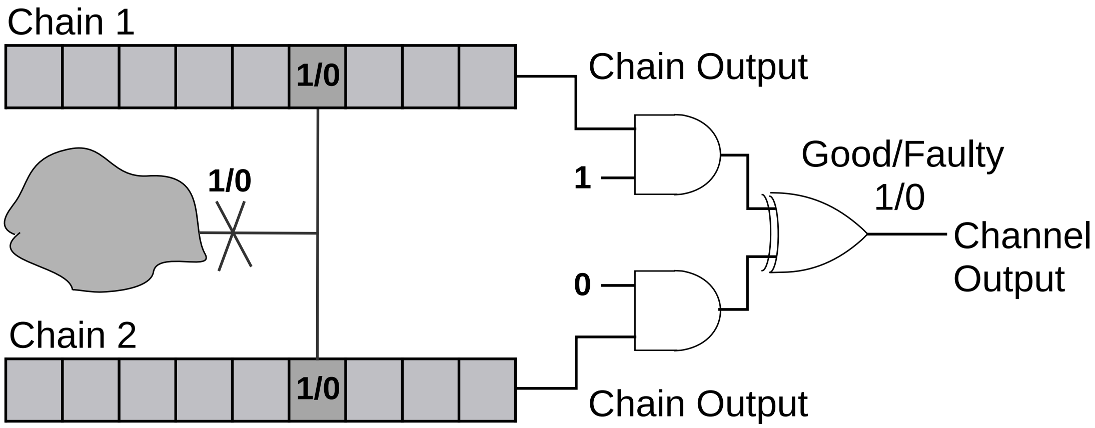
For this unique scenario, it is not possible to see the difference between a good and a faulty circuit.

Figure 7-31. Example of Fault Aliasing



The solution to this problem is to utilize scan chain masking. The tool does this automatically. In compressed ATPG, a fault that is aliased will not be marked detected for the unmasked pattern (Figure 7-31). Instead, the tool uses a masked pattern as shown in Figure 7-32. This mechanism guarantees that all potentially aliased faults are securely detected. Cases in which a fault is always aliased and requires a masking pattern to detect it are rare.

Figure 7-32. Using Masked Patterns to Detect Aliased Faults



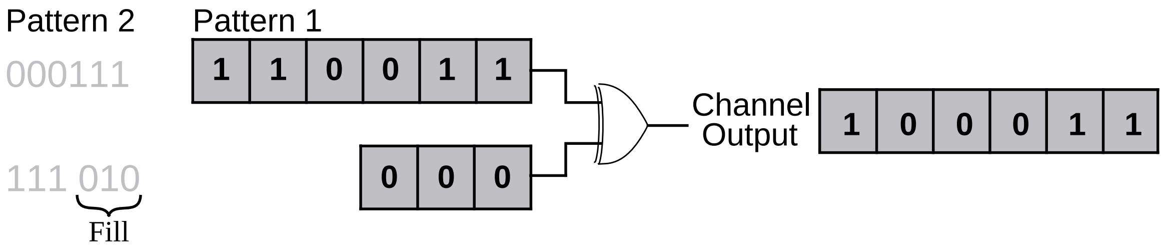
Reordering Patterns

In Tessent Shell, you can reorder patterns using static compaction with the compress_patterns command, and pattern optimization with the order_patterns command. You can also use split pattern sets by, for example, reading a binary or ASCII pattern file back into the tool, and then saving a portion of it using the -Begin and -End options to the write_patterns command.

The tool does not support reordering of serial EDT patterns by a third-party tool, after the compressed patterns are saved.

This has to do with what happens in the compactor when two scan chains have different lengths. Suppose two scan chains are compacted into one channel, as illustrated in Figure 7-33. Chain 1 is six cells long and Chain 2 is three cells long. The captured values of the last three bits of Chain 1 are going to be XOR'd with the first three values of the next pattern being loaded into Chain 2. For regular ATPG, this problem does not occur because the expected values on Chain 2, after you shift three positions, are all Xs. So you never observe the values being loaded as part of the next pattern. But, if that is done with EDT, the last three positions of Chain 1 are XOR'd with X and faults observed on these last cells are lost. Because the padding data for the shorter scan chains is derived from the scan-in data of the next pattern, avoid reordering serial patterns to ensure valid computed scan-out data.

Figure 7-33. Handling Scan Chains of Different Length



Handling of Last Patterns

In order to completely shift out the values contained in the final capture cycle, the tool shifts in the last pattern one additional time so that the output matches the calculated value.

When the design contains chains of different lengths, the tool pads the shorter chains using the values generated by the decompressor during next pattern load. The calculated expected values on the last pattern unload are based on loading the last pattern one more time. Changing the last pattern load will result in simulation mismatches.

Note

If the last pattern load is modified, mismatches will occur.

Chapter 8
Integrating Compression at the RTL Stage

You can create EDT logic during the RTL design phase, rather than waiting for the complete synthesized gate-level design netlist. Creating the EDT logic early allows you to consider the EDT logic earlier in the floor-planning, placement, and routing phases.

To create EDT logic during the RTL stage, you must know the following parameters for your design:

- Number of external scan channels
- Number of internal scan chains
- Clocking of the first and last scan cell of each chain
  This scan chain clocking information is not necessary if you set up the EDT clock to pulse before the scan chain shift clocks. For more information, see "Pulse EDT Clock Before Scan Shift Clocks" on page 64.
- Longest scan chain length range (an estimate of the minimum number of scan cells and maximum number of scan cells the tool can expect in the longest scan chain)

You should also have knowledge about the design interface if you are creating/inserting the EDT logic external to the design core.

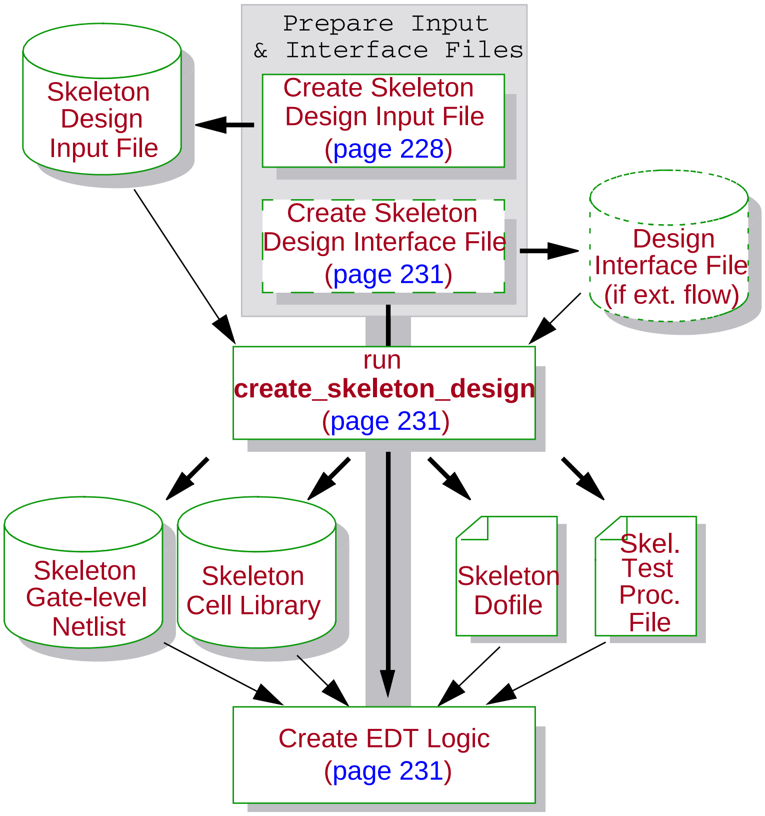
About the RTL Stage Flow

Figure 8-1 shows the IP Creation RTL stage flow. The utility, create_skeleton_design is used to create a skeleton design. This utility writes out a gate-level *skeleton* Verilog design and several related files required to create EDT logic.

To use the create_skeleton_design utility, you must create a Skeleton Design Input File. The Skeleton Design Input File contains the requisite number of scan chains with the first and last cell of each of these chains driven by the appropriate clocks. For more information, see "Skeleton Design Input File" on page 228.

If you are creating/inserting the EDT logic external to the design core, you must also create a Skeleton Design Interface File. For more information, see "Skeleton Design Interface File" on page 231.

Figure 8-1. EDT IP Creation RTL Stage Flow


Prepare Input & Interface Files
Create Skeleton Design Input File (page 228)
Skeleton Design Input File
Create Skeleton Design Interface File (page 231)
Design Interface File (if ext. flow)
run create_skeleton_design (page 231)
Skeleton Gate-level Netlist
Skeleton Cell Library
Skeleton Dofile
Skel. Test Proc. File
Create EDT Logic (page 231)

Use the following steps to create EDT logic for an RTL design:

1. Create a Skeleton Design Input File. For more information, see "Skeleton Design Input File" on page 228.

2. If you are inserting the EDT logic external to the core design (Compressed Pattern External Flow), create a Design Interface File to provide the interface description of the core design in Verilog format. For more information, see "Skeleton Design Interface File" on page 231.

3. Run the create_skeleton_design utility. For example:

   - Internal Flow:

     ```
     create_skeleton_design -o output_file_prefix  \
        -i skeleton_design_input_file
     ```

   - External Flow:

     ```
     create_skeleton_design -o output_file_prefix  \
        -i skeleton_design_input_file -design_interface file_name
     ```

The utility writes out the following four files:

<*output_file_prefix*>.v — Skeleton design netlist

<*output_file_prefix*>.dofile — Dofile

<*output_file_prefix*>.testproc — Test procedure file

<*output_file_prefix*>.atpglib — Tessent cell library

For a complete example showing create_skeleton_design input files and the resultant output files, see the "Skeleton Flow Example" on page 234.

4. Invoke Tessent Shell, set the correct context, and read the skeleton design netlist and the Tessent cell library.

5. Provide compression setup commands.

   - Run the dofile and test procedure file to set up the scan chains for the EDT logic.
   - Issue the set_edt_options command to specify the number of scan channels. You should use the -Longest_chain_range switch with this command to specify an estimated length range (*min_number_cells* and *max_number_cells*) for the longest scan chain in the design. For additional information, refer to "Longest Scan Chain Range Estimate" on page 232.

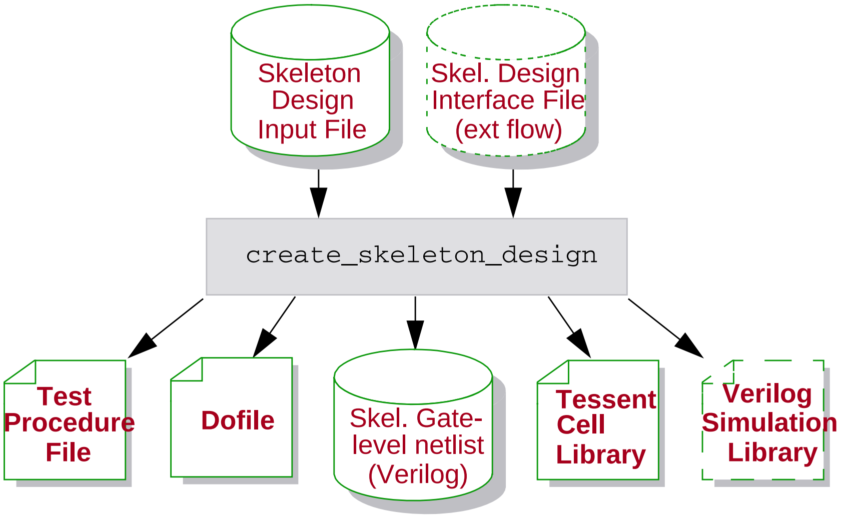
6. Provide EDT DRC, configuration, and logic creation commands.

   - Use the set_system_mode analysis command to flatten the design and run DRCs.
   - Issue other configuration commands as needed.
   - Write out the RTL description of the EDT logic with the write_edt_files command.

Skeleton Design Input and Interface Files

Figure 8-2 shows the inputs and outputs to the **create_skeleton_design** utility. The Skeleton Design Input File is always required. The Skeleton Design Interface File is needed only if you will be creating the EDT logic external to the core design (Compressed Pattern External Flow). You must create both files using the format and syntax described in the following subsections.

Figure 8-2. Create_skeleton_design Inputs and Outputs


Skeleton Design Input File
Skel. Design Interface File (ext flow)
create_skeleton_design
Test Procedure File
Dofile
Skel. Gate-level netlist (Verilog)
Tessent Cell Library
Verilog Simulation Library

Skeleton Design Input File

Create the skeleton design input file using the rules described in the next section.

Input File Format

Example 8-1 shows the format of the skeleton design input file. Required keywords are highlighted in bold. This file contains distinct sections that are described after the example. Example 8-2 on page 231 shows a small working example.

Example 8-1. Skeleton Design Input File Format

```
// Description of scan pins and LSSD system clock with design interface
//   (required)
scan_chain_input <prefix> <bused|indexed> [<starting_index_if_indexed>]
scan_chain_output <prefix> <bused|indexed> [<starting_index_if_indexed>]
lssd_system_clock <clock_name>  // Any system clock for LSSD designs
scan_enable <scan_enable_name>  // Any scan_enable pin name

// Clock definitions (required)
begin_clocks                    // Keyword to begin clock definitions
   <clock_name> <off_state>     // Clock name and off state
   <clock_name> <off_state>     // Clock name and off state
end_clocks                      // Keyword to end clock definitions

// Scan chain specification (required)
begin_chains                    // Keyword to begin chain definitions
// first_chain_number and last_chain_number specify range of chains
// MUXD chain
<first_chain_number> <last_chain_number> <chain_length> \
   <TE|LE> <first_cell_clock> <TE|LE> <last_cell_clock>

//LSSD chain
<first_chain_number> <last_chain_number> <chain_length> \
   LA <first_cell_master_clock> <first_cell_slave_clock> \
   LA <last_cell_master_clock> <last_cell_slave_clock>
end_chains                      // Keyword to end chain definitions
```

Scan Pins and LSSD System Clock Specification Section

Note

This section is required when you use the -Design_interface switch with create_skeleton_design to enable the tool to create a correct instantiation of the core in the top-level EDT wrapper (Compressed Pattern External Flow). If the scan pins specified in this section are not present in the design interface, the utility automatically adds them to the skeleton design. You can omit this section if you are not using the -Design_interface switch.

In this section, specify the scan chain pin name prefix and the type, bused or indexed, using the keywords, “scan_chain_input” and “scan_chain_output”. The bused option will result in scan chain pins being declared as vectors, i.e., <prefix>[Max-1:0]. The indexed option will result in scan chain pins being declared as scalars, numbered consecutively beginning with the specified starting index, and named in “<prefix><index>” format.

If you intend to share channel outputs, you can specify the name of a scan enable pin using the “scan_enable” keyword. If you do not specify a scan enable pin, the tool will automatically add a default pin named “scan_en” to the output skeleton design.

If the design contains LSSD scan cells, you can optionally use the lssd_system_clock keyword to specify the name of any one LSSD system clock. If you do not specify a name, the tool will use the default name, “lssd_system_clock”.

Clock Definition Section

In this section, specify clock names and their corresponding off states. The utility uses these off states to create a correct skeleton dofile and skeleton test procedure file. (See the add_clocks command for additional details about the meaning of clock off states.)

Scan Chain Specification Section

The scan chain specification section is the key section. Here, you specify the number of scan chains, length of the chains, and clocking of the first and last scan cell.

Note

If the EDT logic clock is pulsed before the scan chain shift clock, you do not need to account for the clocking of the first and last cell in each scan chain as this information will not be evaluated. For more information, see "Pulse EDT Clock Before Scan Shift Clocks" on page 64.

To simplify and shorten this section, you can list, on one line, a range of chains that have the same specifications. Each line should contain the chain number of the first chain in the range, the chain number of the last chain in the range, length of the chains, and the edge and clock information of the first and last scan cell. The length of the scan chains can be any value not less than 2, but typically 2 suffices for the purpose of creating appropriate EDT logic. In the created skeleton design, all chains in this range will be the same length and contain a first and last scan cell with the same clocking.

The edge specification must be one of the following:

- LE for a scan cell whose output changes on the leading edge of the specified clock
- TE for a scan cell whose output changes on the trailing edge of the specified clock
- LA for an LSSD scan cell

When you specify the clock edge of the last scan cell, it is critical to include the lockup cell timing as well. For example, if a leading edge (LE) triggered scan memory element is followed by a lockup cell, the edge specification of the scan *cell* must be TE (not LE) since the cell contains a scan memory element followed by a lockup cell and the scan cell output changes on the trailing edge (TE) of the clock. Specifying incorrect edges will result in the tool inserting improper lockup cells and you may need to regenerate the EDT logic later.

Note

When the scan chain specification indicates the first and last scan cell have master/slave or master/copy clocking (for example, an LE first scan cell and a TE last scan cell), the create_skeleton_design utility will increase that chain's length by one cell in the skeleton netlist it writes out. This is done to satisfy a requirement of lockup cell analysis and will not alter the EDT logic; the length of the scan chains seen by the tool after it reads in the skeleton netlist will be as specified in the skeleton design input file.

Comment Lines

You can place comments in the file by beginning them with a double slash (//). Everything after a double slash on a line is treated as a comment and ignored.

Input File Example

The following example utilizes bused scan chain input and output pins. It also defines two clocks, clk1 and clk2, with off-states 0 and 1, respectively. A total of eight scan chains are specified. Chains 1 through 4 are of length 2, with the first cell being LE clk1 triggered and the last cell being TE clk1 triggered. Chains 5 and 6 are of length 3, with the first cell being LE clk2 triggered and the last cell being TE clk2 triggered. Chains 7 and 8 are also of length 3, with the first and last cells being of LSSD type, clocked by master and slave clocks, mclk and sclk, respectively.

Example 8-2. Skeleton Design Input File Example

```
// Double slashes (//) mean everything following on the line is a comment.
//
// edt_si[7:0] and edt_so[7:0] pins are created for scan chains.
scan_chain_input edt_si bused
scan_chain_output edt_so bused
begin_clocks
   clk1 0
   clk2 1
   mclk 0
   sclk 0
end_clocks
begin_chains
// chains 1 to 4 have the following characteristics (Mux scan)
   1 4 2 LE clk1 TE clk1
// chains 5 and 6 have the following characteristics (Mux scan)
   5 6 3 LE clk2 TE clk2
// chains 7 and 8 have the following characteristics (LSSD)
   7 8 3 LA mclk sclk LA mclk sclk
end_chains
```

Skeleton Design Interface File

You should create a skeleton design interface file if you are creating EDT logic that is inserted external to the design core. It should contain only the interface description of the core design in Verilog format; that is, only the module port list and declarations of these ports as input, output, or inout. For an example of this file, see "Interface File" on page 235.

Tip: The interface file ensures the files written out by the **create_skeleton_design** utility contains the information the tool needs to write out valid core blackbox (**_core_blackbox.v*) and top-level wrapper (**_edt_top.v*) files.

Creating EDT Logic for a Skeleton Design

After invoking Tessent Shell and reading the skeleton design, you must set up the following parameters with the set_edt_options command:

- Number of external scan channels

- Estimate of the longest scan chain length (optional). This value allows flexibility when configuring scan chains. For more information, see "Longest Scan Chain Range Estimate" on page 232.

For example:

```
set_edt_options -channels 2
set_edt_options -longest_chain_range 75 125
```

For more information on setting up and creating the EDT logic, see "Creation of EDT Logic Files" on page 77.

Longest Scan Chain Range Estimate

The longest scan chain range estimate defines a range for the length of the longest scan chain in the design. The EDT logic is then configured to allow the longest scan chain in the design to fall within this range without requiring the EDT logic to be regenerated. This builds in flexibility in cases, such as the RTL flow, where the scan chains may change after the EDT logic is created as follows:

- *min_number_cells* — specifies the lower bound of the longest scan chain range. You should avoid specifying an artificially low value for the set_edt_options "min_number_cells" command option. Specifying an artificially low value results in the creation of an EDT logic configuration that can result in incompressible patterns.

  Note that the set_edt_options "longest_chain_range" switch defines a range for the length of the longest scan chain in your design — this does *not* mean the range of lengths of all the scan chains in your design. Setting the min_number_cells option based on these considerations enables the tool to configure the EDT logic to assure robust pattern compression.

  For more information on compactors, see "Understanding Compactor Options" on page 215.

- *max_number_cells* — specifies the higher bound of the longest scan chain range and is used to configure the phase shifter in the decompressor. The phase shifter is configured to separate the bit streams provided to the scan chains by at least as many cycles as specified by the *max_number_cells* value. This reduces linear dependencies among the bit streams supplied to the internal scan chains.

  The flexibility of this restriction is determined by the linear dependencies present in a design and the number of scan cells specified for the longest scan chain. Some designs tolerate up to a 25% increase in scan chain length before the EDT logic is affected.

Integrating the EDT Logic into the Design

After you create the EDT logic, integrating it into the design is a manual process.

- For EDT logic created external to the design core (Compressed Pattern External Flow):

  If you provided the **create_skeleton_design** utility with the recommended interface file when it generated the skeleton design, you can continue with the compressed pattern external flow (optionally insert I/O pads and boundary scan, then synthesize the I/O pads, boundary scan, and EDT logic).

  If you did not use an interface file, you will need to manually provide the interface and all related interconnects needed for the functional design before synthesizing the EDT logic.

- For EDT logic created within the design core (Compressed Pattern Internal Flow):

  Integrating the EDT logic into the design is a manual process you perform using your own tools and infrastructure to stitch together different blocks of the design to create a top level design.

Note

The Design Compiler synthesis script that the tool writes out does not contain information for connecting the EDT logic to design I/O pads, as the tool did not have access to the complete netlist when it created the EDT logic.

Knowing When to Regenerate the EDT Logic

By the time the gate-level netlist is available, there may be changes to the design that affect the EDT logic as described in the following list. When one of these changes occurs in the design, the safest approach is to always regenerate the EDT logic and compare the new RTL with the previous RTL to determine if the EDT logic is changed.

- **Number of channels or chains has changed** — In this case, the EDT logic must be regenerated.
- **Clocking of a first or last scan cell has changed** — Whether the EDT logic actually needs to be regenerated depends on whether the clock edge that triggers the first or last scan cell has changed and whether lockup cells are inserted for bypass mode scan chains. You should regenerate the EDT logic any time the clocking of the first or last scan cell changes. Note, this scan chain clocking information is not relevant (not a cause for regenerating EDT logic) if you set up the EDT clock to pulse before the scan chain shift clocks. For more information, see “Pulse EDT Clock Before Scan Shift Clocks” on page 64.
- **Length of the longest scan chain is less than the *min_number_cells* specified with the set_edt_options -Longest_chain_range switch** — If the EDT logic uses the Xpress

compactor (default), this value does not affect the architecture and the EDT logic does not need to be regenerated.

However, if the EDT logic uses the Basic compactor, this parameter is used to configure the length of the mask register in the compactor. In this case, you should regenerate the EDT logic. For more information, see "Longest Scan Chain Range Estimate" on page 232".

- **Length of the longest scan chain is greater than the *max_number_cells* specified with the set_edt_options -Longest_chain_range switch** — Whether the EDT logic actually changes or not depends on whether the phase shifter in the decompressor needs to be redesigned or not. The flexibility of this restriction is determined by the linear dependencies present in a design and the number of scan cells specified for the longest scan chain. Some designs tolerate up to a 25% increase in scan chain length before the EDT logic is affected. For more information, see "Longest Scan Chain Range Estimate" on page 232".

Skeleton Flow Example

This section shows example skeleton design input and interface files and the output files the **create_skeleton_design** utility generated from them.

Input File

Note
If you will be creating the EDT logic within the core design (Compressed Pattern Internal Flow), this file is the only input the utility needs.

The following example skeleton design input file, *my_skel_des.in*, utilizes indexed scan chain input and output pins. The file defines two clocks, NX1 and NX2, with off-states 0, and specifies a total of 16 scan chains, most of which are 31 scan cells long. Notice the clocking of the first and last scan cell in each chain is specified, but no other scan cell definition is required. This is because the utility has built-in ATPG models of simple mux-DFF and LSSD scan cells that are sufficient for it to write out a skeleton design (and for the tool to use later to create the EDT logic).

```
scan_chain_input scan_in indexed 1
scan_chain_output scan_out indexed 1

begin_clocks
    NX1 0
    NX2 0
   end_clocks

begin_chains
    1 1 31 TE NX1 TE NX1
    2 2 30 TE NX1 TE NX1
    3 3 30 TE NX1 TE NX1
    4 4 31 TE NX1 TE NX1
    5 5 31 TE NX1 TE NX1
    6 6 32 LE NX2 LE NX2
    7 7 31 LE NX2 LE NX2
    8 8 31 LE NX2 LE NX2
    9 9 31 LE NX2 LE NX2
    10 10 31 LE NX2 LE NX2
    11 11 31 LE NX2 LE NX2
    12 12 31 LE NX2 LE NX2
    13 13 31 LE NX2 LE NX2
    14 14 31 LE NX2 LE NX2
    15 15 31 LE NX2 LE NX2
    16 16 31 LE NX2 LE NX2
end_chains
```

Interface File

The following shows an example interface file *nemo6_blackbox.v* for the design described in the preceding input file. Use of an interface file is recommended if you intend to create the EDT logic as a wrapper external to the core design (Compressed Pattern External Flow).

```
module nemo6 ( NMOE , NMWE , DLM , ALE , NPSEN , NALEN , NFWE , NFOE ,
               NSFRWE , NSFROE , IDLE , XOFF , OA , OB , OC , OD , AE ,
               BE , CE , DE , FA , FO , M , NX1 , NX2 , RST , NEA ,
               NESFR , ALEI , PSEI , AI , BI , CI , DI , FI , MD ,
               scan_in1 , scan_out1 , scan_in2 , scan_out2 , scan_in3 ,
               scan_out3 , scan_in4 , scan_out4 , scan_in5 , scan_out5 ,
               scan_in6 , scan_out6 , scan_in7 , scan_out7 , scan_in8 ,
               scan_out8 , scan_in9 , scan_out9 , scan_in10 , scan_out10 ,
               scan_in11 , scan_out11 , scan_in12 , scan_out12 ,
               scan_in13 , scan_out13 , scan_in14 , scan_out14 ,
               scan_in15 , scan_out15 , scan_in16 , scan_out16 , scan_en);
input   NX1 , NX2 , RST , NEA , NESFR , ALEI , PSEI , scan_in1 , scan_in2 ,
        scan_in3 , scan_in4 , scan_in5 , scan_in6 , scan_in7 , scan_in8 ,
        scan_in9 , scan_in10 , scan_in11 , scan_in12 , scan_in13 ,
        scan_in14 , scan_in15 , scan_in16 , scan_en ;
input   [7:0] AI ;
input   [7:0] BI ;
input   [7:0] CI ;
input   [7:0] DI ;
input   [7:0] FI ;
input   [7:0] MD ;
output  NMOE , NMWE , DLM , ALE , NPSEN , NALEN , NFWE , NFOE , NSFRWE ,
        NSFROE , IDLE , XOFF , scan_out1 , scan_out2 , scan_out3 ,
        scan_out4 , scan_out5 , scan_out6 , scan_out7 , scan_out8 ,
        scan_out9 , scan_out10 , scan_out11 , scan_out12 , scan_out13 ,
        scan_out14 , scan_out15 , scan_out16 ;
output  [7:0] OA ;
output  [7:0] OB ;
output  [7:0] OC ;
output  [7:0] OD ;
output  [7:0] AE ;
output  [7:0] BE ;
output  [7:0] CE ;
output  [7:0] DE ;
```

```
    output   [7:0] FA ;
    output   [7:0] FO ;
    output   [15:0] M ;
    endmodule
```

Outputs

This section shows examples of the four ASCII files written out by the **create_skeleton_design** utility when run on the preceding input and interface files using the following shell command:

```
    create_skeleton_design -o bb1 -design_interface nemo6_blackbox.v \
                   -i my_skel_des.in
```

The utility wrote out the following files:

bb1.v
bb1.dofile
bb1.testproc
bb1.atpglib

Skeleton Design

Following is the gate-level skeleton netlist that resulted from the example input and interface files of the preceding section. For brevity, lines are not shown when content is readily apparent from the structure of the netlist. Parts attributable to the interface file are highlighted in bold; the utility would not have included them if there had not been an interface file.

Note

The utility obtains the module name from the interface file, if available. If you do not use an interface file, the utility names the module “skeleton_design_top”.

```
    module nemo6 (NMOE, NMWE, DLM, ALE, NPSEN, NALEN, NFWE, NFOE, NSFRWE,
    NSFROE, IDLE, XOFF, OA, OB, OC, OD, AE, BE, CE, DE, FA, FO, M, NX1, NX2,
    RST, NEA, NESFR, ALEI, PSEI, AI, BI, CI, DI, FI, MD, scan_in1,
    scan_in2, ..., scan_in16, scan_out1, scan_out2, ..., scan_out16, scan_en);

    output NMOE;
    output NMWE;
    output DLM;
    output ALE;
    output NPSEN;
    output NALEN;
    output NFWE;
    output NFOE;
    output NSFRWE;
    output NSFROE;
    output IDLE;
    output XOFF;
    output [7:0] OA;
    output [7:0] OB;
    output [7:0] OC;
```

```
output [7:0] OD;
output [7:0] AE;
output [7:0] BE;
output [7:0] CE;
output [7:0] DE;
output [7:0] FA;
output [7:0] FO;
output [15:0] M;
input NX1;
input NX2;
input RST;
input NEA;
input NESFR;
input ALEI;
input PSEI;
input [7:0] AI;
input [7:0] BI;
input [7:0] CI;
input [7:0] DI;
input [7:0] FI;
input [7:0] MD;

input scan_in1;
input scan_in2;
...
input scan_in16;
output scan_out1;
output scan_out2;
...
output scan_out16;
input scan_en;

   wire NX1_inv;

   wire chain1_cell1_out;
   wire chain1_cell2_out;
   ...
   wire chain1_cell31_out;
   wire chain2_cell1_out;
   wire chain2_cell2_out;
   ...
   wire chain2_cell30_out;
     .
     .
     .
   wire chain16_cell1_out;
   wire chain16_cell2_out;
   ...
   wire chain16_cell31_out;

   inv01 NX1_inv_inst ( .Y(NX1_inv), .A(NX1));

   muxd_cell chain1_cell0 ( .Q(scan_out1), .SI(chain1_cell1_out),
      .D(1'b0), .CLK(NX1_inv), .SE(scan_en) );
   muxd_cell chain1_cell1 ( .Q(chain1_cell1_out), .SI(chain1_cell2_out),
      .D(1'b0), .CLK(NX1_inv), .SE(scan_en) );
      ...
   muxd_cell chain1_cell30 ( .Q(chain1_cell30_out), .SI(scan_in1),
      .D(1’b0),.CLK(NX1_inv), .SE(scan_en) );

   muxd_cell chain2_cell0 ( .Q(scan_out2), .SI(chain2_cell1_out),
      .D(1’b0), .CLK(NX1_inv), .SE(scan_en) );
   muxd_cell chain2_cell1 ( .Q(chain2_cell1_out), .SI(chain2_cell2_out),
      .D(1’b0), .CLK(NX1_inv), .SE(scan_en) );
   ...
   muxd_cell chain2_cell29 ( .Q(chain2_cell29_out), .SI(scan_in2),
      .D(1’b0), .CLK(NX1_inv), .SE(scan_en) );
     .
     .
     .

   muxd_cell chain16_cell0 ( .Q(scan_out16), .SI(chain16_cell1_out),
      .D(1’b0), .CLK(NX2), .SE(scan_en) );
   muxd_cell chain16_cell1 ( .Q(chain16_cell1_out),
      .SI(chain16_cell2_out), .D(1’b0), .CLK(NX2), .SE(scan_en) );
   ...
   muxd_cell chain16_cell30 ( .Q(chain16_cell30_out), .SI(scan_in16),
      .D(1’b0), .CLK(NX2), .SE(scan_en) );

endmodule
```

Skeleton Design Dofile

The generated dofile includes most setup commands required to create the EDT logic. Following is the example dofile *bb1.dofile* the utility wrote out based on the previously described inputs:

```
add_scan_groups grp1 bb1.testproc
add_scan_chains chain1 grp1    scan_in1   scan_out1
add_scan_chains chain2 grp1    scan_in2   scan_out2
add_scan_chains chain3 grp1    scan_in3   scan_out3
add_scan_chains chain4 grp1    scan_in4   scan_out4
add_scan_chains chain5 grp1    scan_in5   scan_out5
add_scan_chains chain6 grp1    scan_in6   scan_out6
add_scan_chains chain7 grp1    scan_in7   scan_out7
add_scan_chains chain8 grp1    scan_in8   scan_out8
add_scan_chains chain9 grp1    scan_in9   scan_out9
add_scan_chains chain10 grp1    scan_in10   scan_out10
add_scan_chains chain11 grp1    scan_in11   scan_out11
add_scan_chains chain12 grp1    scan_in12   scan_out12
add_scan_chains chain13 grp1    scan_in13   scan_out13
add_scan_chains chain14 grp1    scan_in14   scan_out14
add_scan_chains chain15 grp1    scan_in15   scan_out15
add_scan_chains chain16 grp1    scan_in16   scan_out16
add_clocks 0 NX1
add_clocks 0 NX2
```

Skeleton Design Test Procedure File

The utility also writes out a test procedure file that has the test procedure steps needed to create EDT logic. Following is the example test procedure file *bb1.testproc* the utility wrote out using the previously described inputs:

```
set time scale 1.000000 ns ;
 timeplate gen_tp1 =
    force_pi 0 ;
    measure_po 10 ;
    pulse NX1 40 10;
    pulse NX2 40 10;
    period 100 ;
 end;

 procedure shift =
    scan_group grp1 ;
    timeplate gen_tp1 ;
    cycle =
        force_sci ;
        measure_sco ;
        pulse NX1 ;
        pulse NX2 ;
    end;
 end;

 procedure load_unload =
    scan_group grp1 ;
    timeplate gen_tp1 ;
    cycle =
        force NX1 0 ;
        force NX2 0 ;
        force scan_en 1 ;
    end ;
    apply shift 2 ;
 end ;
```

Skeleton Design Tessent Cell Library

The Tessent cell library written out by the utility contains the models used to create the skeleton design. **You must use this library when you perform EDT IP Creation on the skeleton design in Tessent Shell.**

```
model inv01(A, Y) (
   input (A) ()
   output(Y) (primitive = _inv(A, Y); )
)

// muxd_scan_cell is the same as sff in adk library.
model muxd_scan_cell (D, SI, SE, CLK, Q, QB) (
   scan_definition (
      type = mux_scan;
      data_in = D;
      scan_in = SI;
      scan_enable = SE;
      scan_out = Q, QB;
   )
   input (D, SI, SE, CLK) ()
   intern(_D) (primitive = _mux (D, SI, SE, _D);)
   output(Q, QB) (primitive = _dff(, , CLK, _D, Q, QB); )
)

model lssd_scan_cell (D, SYS_CLK, SI, MCLK, SCLK, Q, QB) (
   scan_definition (
      type = lssd;
      data_in = D;
      scan_in = SI;
      scan_master_clock = MCLK;
      scan_slave_clock = SCLK;
      scan_out = Q;
   )
   input (D, SYS_CLK, SI, MCLK, SCLK) ()
   intern(MOUT) (primitive = _dlat master ( , , SYS_CLK, D, MCLK, SI,
MOUT, );)
   output(Q, QB) (primitive = _dlat slave ( , , SCLK, MOUT, Q, QB );)
)
```

Note

You can get the utility to write out a Verilog simulation library that matches the Tessent cell library by including the optional -Simulation_library switch in the shell command.

Chapter 9
Modular Compressed ATPG

About the Modular Flow

Modular Compressed ATPG is the process used to integrate compression into the block-level design flow. Integrating compression at the block-level is similar to integrating compression at the top-level, except you create/insert EDT logic into each design block and then, integrate the blocks into a top-level design and generate test patterns.

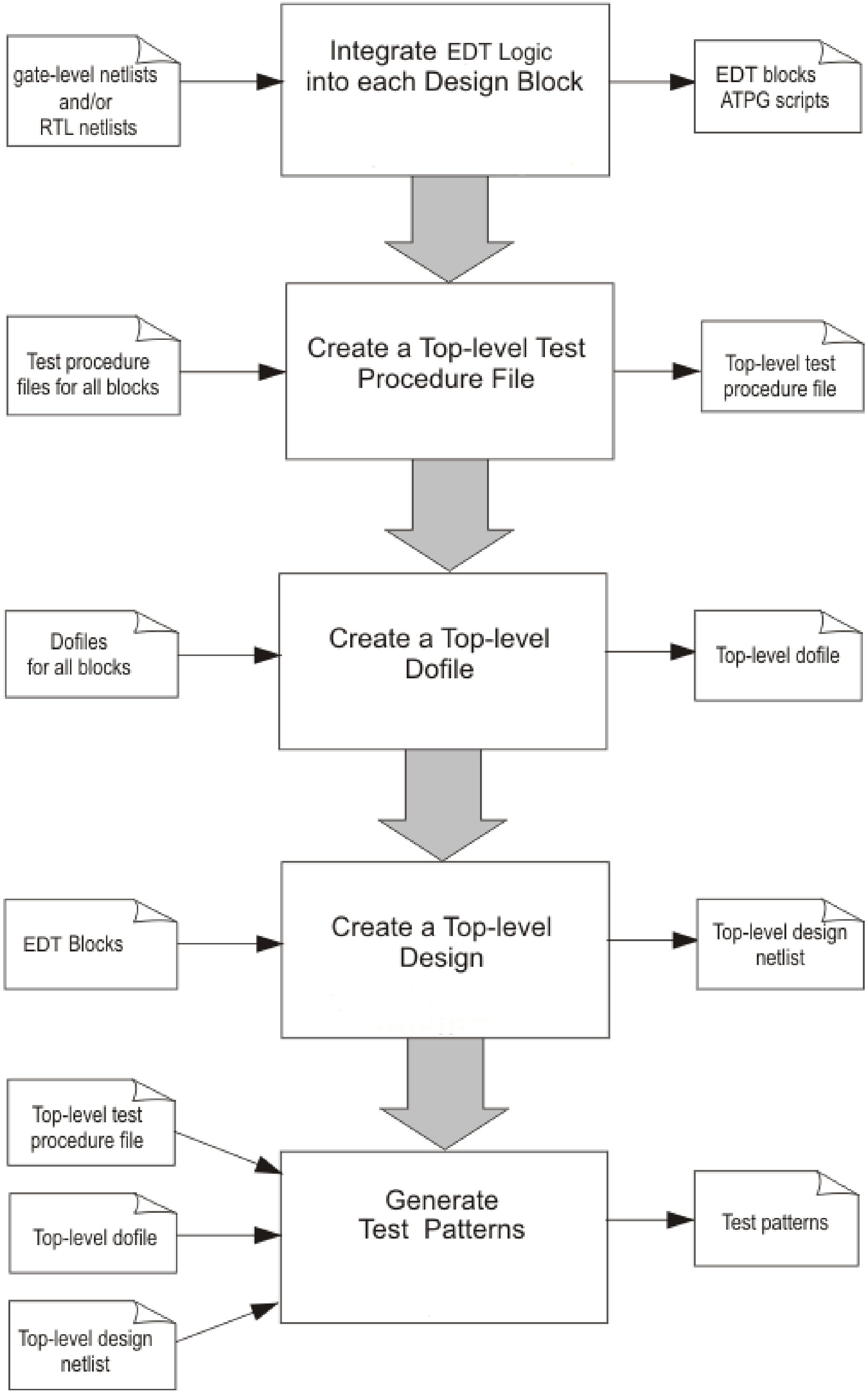
Note
In this chapter, an EDT block refers to a design block that contains a full complement of EDT logic controlling all the scan chains associated with the block.

The modular flow includes one or more of the top-level compressed pattern flows. For information on these top-level flows, see, "The Compressed Pattern Flows" on page 27 of this manual.

Requirements

- Block-level compression strategy
- Gate-level or RTL netlist for each block in the design
- Tessent Scan or other scan insertion tool (optional)
- Tessent cell library
- Design Compiler or other synthesis tool
- ModelSim or other timing simulator

Modular Flow Diagram


gate-level netlists and/or RTL netlists
Integrate EDT Logic into each Design Block
EDT blocks ATPG scripts
Test procedure files for all blocks
Create a Top-level Test Procedure File
Top-level test procedure file
Dofiles for all blocks
Create a Top-level Dofile
Top-level dofile
EDT Blocks
Create a Top-level Design
Top-level design netlist
Top-level test procedure file
Top-level dofile
Top-level design netlist
Generate Test Patterns
Test patterns

Flow Stage Descriptions

Table 9-1. Modular Flow Stage Descriptions

Stage	Description
Integrate EDT logic into each Design Block	EDT logic can be integrated into each design block using any of the top-level methods described in this document. For more information, see the following sections of this document: • Integrating Compression at the RTL Stage • The Compressed Pattern Flows The first step to using compression in your design flow is developing a compression strategy. For more information, see "Creating a Block-level Compression Strategy" on page 245.
Create a Top-level Test Procedure File	The test procedure files generated during EDT IP creation for each block must be merged to form a top-level test procedure file. For more information, see "Creating a Top-level Test Procedure File" on page 249.
Create a Top-level Design	Design blocks must be integrated to form a single top-level design netlist. For more information, see "Creating the Top-level Netlist" on page 257.
Create a Top-level Dofile	The dofiles generated during EDT IP creation for each block must be combined to create a single top-level dofile. For more information, see "Creating the Top-level Dofile" on page 262.
Generate Test Patterns	Test patterns are set up and generated using the top-level netlist, test procedure file, and dofile. For more information, see "Generating Top-level Test Patterns" on page 265. You should also create bypass test patterns for the top-level netlist at this point. For more information, see "Compression Bypass Logic" on page 172.

Related Topics

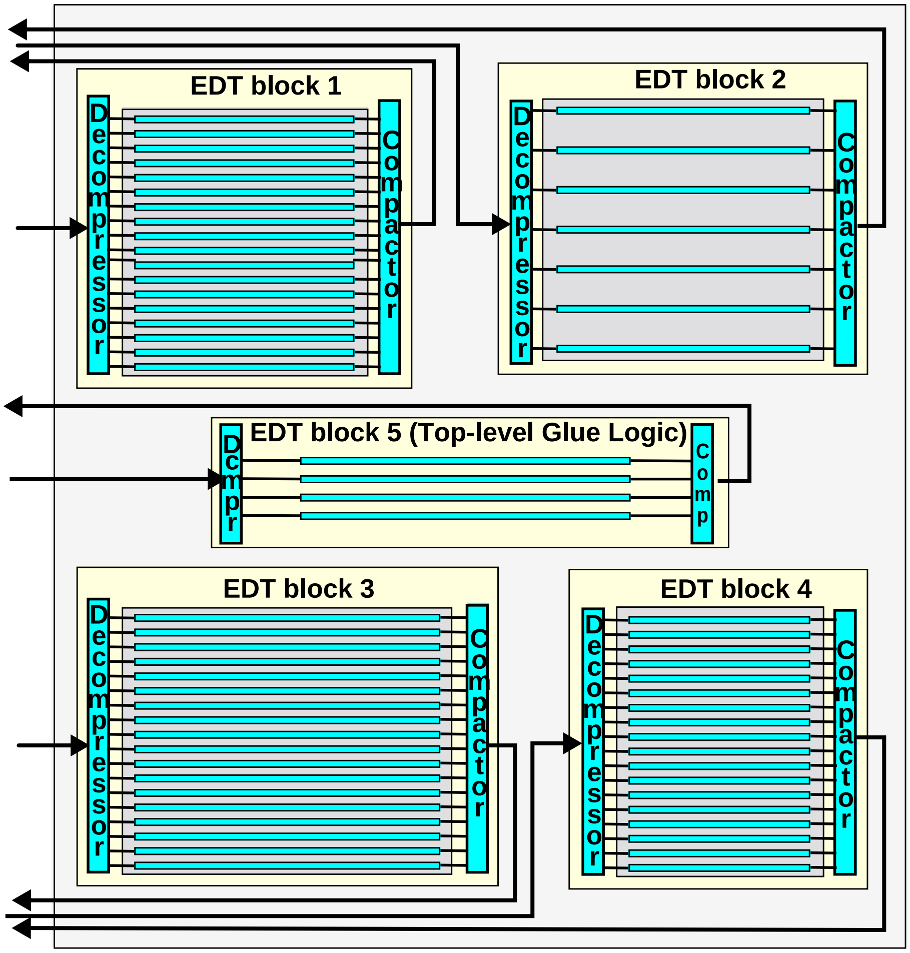
Generating Modular EDT Logic for a Fully Integrated Design

Understanding Modular Compressed ATPG

The EDT logic inserted in a design block controls all scan chains within the block.

Figure 9-1 shows an example of a modular design with four EDT blocks. Each EDT block consists of a design block with integrated EDT logic. The design also contains a separate EDT block for the top-level glue logic. The top-level glue logic can be tested with EDT logic as shown or with bypass logic as described in "Compression Bypass Logic" on page 172.

Figure 9-1. Modular Design with Five EDT blocks



Each EDT block has a discrete netlist, dofile, and test procedure file that are integrated together to form top-level files for test pattern generation.

Creating a Block-level Compression Strategy

You can create and insert EDT logic into design blocks with any of the methods outlined in Chapters 2 through 5 of this manual. You can also mix and match methods between blocks. Reference the following rules and guidelines while developing your compression strategy for the modular flow:

- **Scan chain lengths should be balanced** — Balanced scan chains yield optimal compression. Plan the lengths of scan chains inside all blocks in advance so that top-level (inter-block) scan chain lengths are relatively equal. See "Balancing Scan Chains Between Blocks" on page 246.
- **EDT logic names must be unique** — When multiple EDT blocks are integrated into a top-level netlist, all of the EDT logic file names and internal module/instance names must be unique. See "Creation of EDT Logic Files" on page 77.
- **Each EDT block must have a discrete set of scan chains —** Scan chains cannot be shared between blocks.
- **Uncompressed scan chains must be connected to top-level pins** — Uncompressed scan chains are scan chains not driven by or observed through the EDT logic. Uncompressed scan chains are supported if the inputs and outputs are connected directly to top-level pins. Uncompressed scan chains can also share top-level pins. See "Inclusion of Uncompressed Scan Chains" on page 40.
- **Only certain control pins can be shared with functional pins** — These pins can be shared within the same EDT block. See "Functional/EDT Pin Sharing" on page 67.
- **Control signals can be shared by EDT blocks** — Control signals such as edt_update, edt_clock, edt_reset, scan_enable and test_en may be shared between EDT blocks; for example, the edt_update signals from different blocks could be connected to the same top-level pin. See "Creating the Top-level Netlist" on page 257.
- **Scan channels must have dedicated top-level pins** — Only input scan channels between identical EDT blocks can share top-level pins. See "Sharing Input Scan Channels on Identical EDT blocks" on page 246.
- **Block-level signals must be connected in the top-level netlist** — This includes connecting EDT logic signals to I/O pads and inserting any multiplexers needed for channel output signals shared with functional signals. See "Creating the Top-level Netlist" on page 257.
- **EDT logic must be synthesized and verified for each block** — See "Synthesizing the EDT Logic" on page 99 and "Generating/Verifying Test Patterns" on page 109.

Balancing Scan Chains Between Blocks

Design blocks may contain a large amount of hardware with many internal blocks and many scan chains, so scan chain balance is very important for generating efficient test patterns. You should carefully plan the lengths of scan chains inside each design block so that all blocks have approximately the same scan chain lengths. The following sections provide information on scan chain planning at the block level:

- "Determining How Many Scan Chains to Use" on page 41
- "Varying the Number of Scan Chains" on page 213
- "Varying the Number of Scan Channels" on page 213

You should target the same compression for every block and apportion available tester channels according to the relative share of the overall design gate count contained in each block. Use the following two equations to calculate balanced scan chain lengths across multiple blocks:

$$\text{Scan Chain Length} \approx \frac{\text{\# of Scan Cells in block}}{(\text{\# of Channels for block}) \times (\text{Chain-to-channel ratio})}$$

$$\text{\# of Channels for block} \approx \frac{\text{\# of Scan Cells in block}}{\text{\# of Scan Cells in chip}} \times \text{\# of top-level Channels}$$

Tip: Since different designers may perform scan insertion for different design blocks, it is important to work together to select a scan chain length target that works for all blocks.

Sharing Input Scan Channels on Identical EDT blocks

You can set up identical EDT blocks to share input scan channels and top-level pins when integrating modular design blocks into a top-level netlist.

When EDT blocks share input scan channels, test patterns are broadcast via shared top-level pins to all the identical EDT blocks simultaneously. This functionality reduces top-level pin requirements and increases the compression ratio for the input side of the EDT logic.

Requirements

- EDT blocks must be identical as follows:
    - Number of input channels and output channels must match
    - Input, output, and compactor pipeline stages must match
    - Order of scan chains and the number of scan cells in each must match
    - Input channel/top-level pin inversions must match

- All corresponding input channels on identical EDT blocks must be shared in the corresponding order. For example the following channels can be shared:
    - input channel 1 of block1
    - input channel 1 of block2
    - input channel 1 of block3 and so on

Top-Level Dofile Modifications

You need to set up the input channel sharing when the block-level dofiles are integrated into a top-level dofile. Depending on the application, you can set up the input channel sharing in one of two ways:

- **Make top-level pins equivalent**

  Use this method when a top-level pin exists for each input channel by defining the pins for the corresponding input channels on each block as equivalent. For example:

  ```
  add_edt_blocks core1
  set_edt_pins input 1 core1_edt_channels_in1
  set_edt_pins input 2 core1_edt_channels_in2
  add_edt_blocks core2
  set_edt_pins input 1 core2_edt_channels_in1
  set_edt_pins input 2 core2_edt_channels_in2
  add_input_constraints -eq core1_edt_channels_in1
     core2_edt_channels_in1
  add_input_constraints -eq core1_edt_channels_in2
     core2_edt_channels_in2
  ```

- **Physically share top-level pins**

  Use this method when top-level pins need to be shared between input channels by explicitly specifying the top-level pins to be same. For example:

  ```
  add_edt_blocks core1
  set_edt_pins input 1 edt_channels_in1
  set_edt_pins input 2 edt_channels_in2
  add_edt_blocks core2
  set_edt_pins input 1 edt_channels_in1
  set_edt_pins input 2 edt_channels_in2
  ```

During DRC, the blocks that share input channels are reported. As long as the EDT blocks are identical and the channel sharing is set up properly, EDT DRCs should pass.

Use the report_edt_configurations -All command to display information on the EDT blocks set up to share input channels.

Generating Modular EDT Logic for a Fully Integrated Design

Use this procedure to simultaneously generate modular EDT logic for all blocks within a fully integrated design. The resulting EDT logic can be set up as multiple instances within the design. If the integrated design shares top-level channels or requires any form of test scheduling, you must generate modular EDT logic one block at a time.

The files generated by this procedure support the same capabilities as the block by block modular flow.

Prerequisites

- The integrated design must be complete and fully functional.
- Each block must have dedicated input and output channels.

Procedure

1. Add each EDT block, one at a time, using the add_edt_blocks command.
2. Once a EDT block is added, set up the EDT logic for it with a set_edt_options command. The set_edt_options command only applies to the current EDT block. EDT control signals can be shared among blocks.
3. Once all the design blocks are added and set up, enter analysis mode. For more information, see the set_system_mode command.
4. Enter a write_edt_files command. A composite set of files is created including an RTL file, a synthesis script, a dofile/testproc file, and a bypass dofile/testproc file. All block-level EDT pins are automatically connected to the top level.
5. Use this composite set of files to synthesize EDT logic and generate test patterns.

Estimating Test Coverage/Pattern Count for EDT Blocks

After you create EDT logic for a block, you should create test patterns and estimate test coverage and pattern count before synthesis. See “Analyzing Compression” on page 48.

Test coverage reported may be higher than when the EDT block is embedded in the design because the tool has direct access to the block-level inputs and outputs at this point.

To get a more realistic coverage estimate, you should:

1. Constrain all functional inputs to X. For example:

   ```
   add_pin_constraints my_func_in cx
   ```

   Where the functional input *my_func_in* is constrained to X.

2. Mask all functional outputs. For example:

   add_output_masks my_func_out1 my_func_out2

   Where the two primary outputs *my_func_out1* and *my_func_out2* are masked.

Note

Constraining inputs to X and masking the outputs produces very conservative estimates that negatively affect compression because all inputs become X sources when the CX constraints are added to the pins.

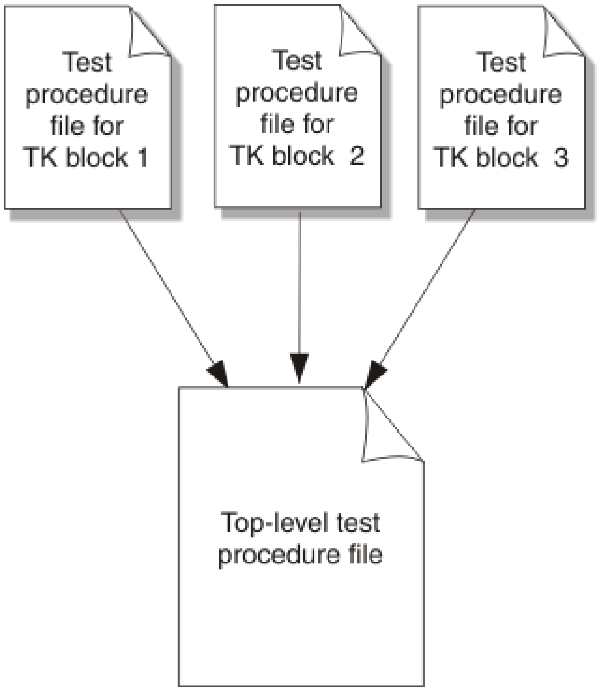
Note

Because final test patterns are generated at the top-level of the design and are affected by all cores, the final test coverage and pattern count may vary.

Creating a Top-level Test Procedure File

Prior to running top-level ATPG, you must integrate all block-level test procedure files into a single top-level test procedure file as illustrated in Figure 9-2.

Figure 9-2. Creating the Top-level Test Procedure File


Test procedure file for TK block 1
Test procedure file for TK block 2
Test procedure file for TK block 3
Top-level test procedure file

When EDT logic is created for each block, a block-level test procedure file is created. See "Test Procedure File — The tool also writes a test procedure file for test pattern generation. The tool takes the test procedure file used for EDT logic creation and adds the test procedures necessary to drive the EDT logic." on page 87. In a normal compressed ATPG flow, this test procedure file is used to create final test patterns for the design. However, in a *modular* compressed ATPG flow, you manually combine the relevant content from each block-level test procedure file into a single top-level test procedure file that is used for top-level pattern generation.

To create the top-level test procedure file, you take the block-level test procedure files and aggregate test-setup, load-unload, shift and capture procedure information into one file. The top-level test procedure file is a superset of the block-level test procedure files and typically consists of clock definitions and force or pulse statements for edt_update and edt_clock during the load_unload, shift and capture procedures. Pin names may also need to be changed. This process is illustrated in the following procedure.

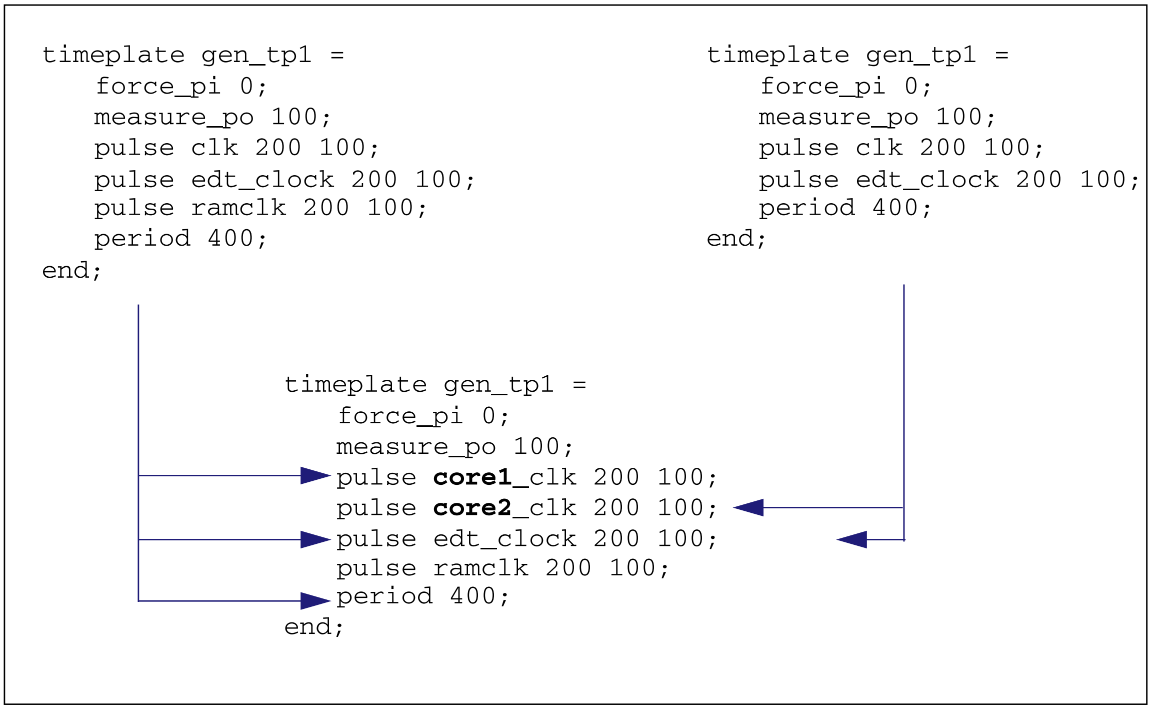
Prerequisites

- All EDT blocks are created and verified.

Procedure

1. Copy and use one of the block-level test procedure files as a template for the top-level file. You should use the block-level test procedure file with the most test procedures.
2. Using a text editor, copy and paste test procedures from the other block-level files into the top-level file.
3. Update the timeplate or timeplates to include all statements present in each block-specific timeplate and customize the top-level timeplate as needed. As you add statements, update pin names to match the corresponding name changes in the top-level netlist. See Figure 9-3.

Figure 9-3. Creating a Top-level Timeplate



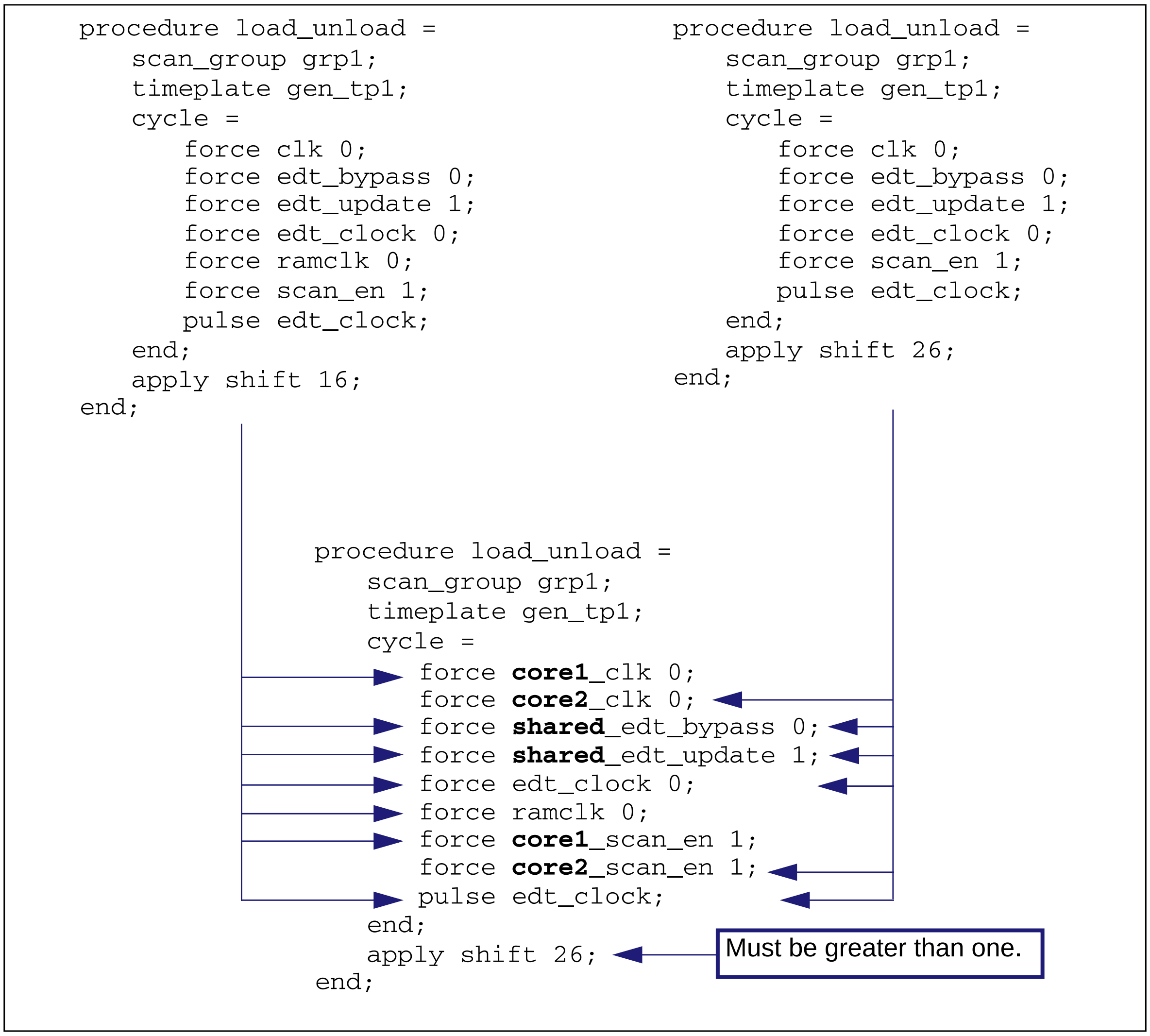
```
timeplate gen_tp1 =
   force_pi 0;
   measure_po 100;
   pulse clk 200 100;
   pulse edt_clock 200 100;
   pulse ramclk 200 100;
   period 400;
end;
```

```
timeplate gen_tp1 =
   force_pi 0;
   measure_po 100;
   pulse clk 200 100;
   pulse edt_clock 200 100;
   period 400;
end;
```

```
timeplate gen_tp1 =
   force_pi 0;
   measure_po 100;
   pulse core1_clk 200 100;
   pulse core2_clk 200 100;
   pulse edt_clock 200 100;
   pulse ramclk 200 100;
   period 400;
end;
```

4. Update the load_unload procedure to include all statements present in each block-level test procedure. As you add statements, update the pin names. See Figure 9-4.

Figure 9-4. Creating a Top-level Load_unload Procedure



```
procedure load_unload =
   scan_group grp1;
   timeplate gen_tp1;
   cycle =
      force clk 0;
      force edt_bypass 0;
      force edt_update 1;
      force edt_clock 0;
      force ramclk 0;
      force scan_en 1;
      pulse edt_clock;
   end;
   apply shift 16;
end;
```

```
procedure load_unload =
   scan_group grp1;
   timeplate gen_tp1;
   cycle =
      force clk 0;
      force edt_bypass 0;
      force edt_update 1;
      force edt_clock 0;
      force scan_en 1;
      pulse edt_clock;
   end;
   apply shift 26;
end;
```

```
procedure load_unload =
   scan_group grp1;
   timeplate gen_tp1;
   cycle =
      force core1_clk 0;
      force core2_clk 0;
      force shared_edt_bypass 0;
      force shared_edt_update 1;
      force edt_clock 0;
      force ramclk 0;
      force core1_scan_en 1;
      force core2_scan_en 1;
      pulse edt_clock;
   end;
   apply shift 26;
end;
```

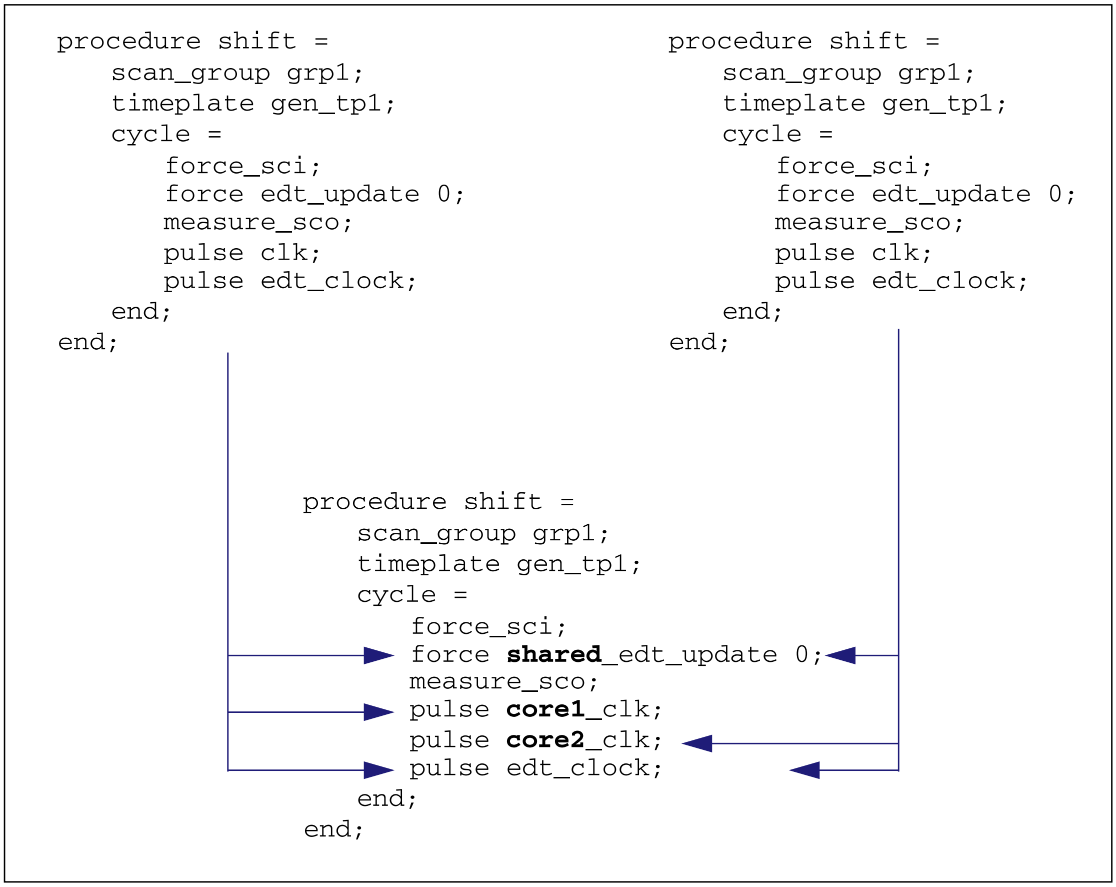
Must be greater than one.

Note

The number specified in the apply shift statement should be greater than one but is otherwise irrelevant; actual shifts are determined by the actual traced length of the scan chains.

5. Update the shift procedure to include statements present in each block-specific shift procedure. As you add statements, make updates to the pin names similar to those in the top-level timeplate and load_unload procedure. See Figure 9-5.

Figure 9-5. Creating a Top-level Shift Procedure



```
procedure shift =
   scan_group grp1;
   timeplate gen_tp1;
   cycle =
      force_sci;
      force edt_update 0;
      measure_sco;
      pulse clk;
      pulse edt_clock;
   end;
end;
```

```
procedure shift =
   scan_group grp1;
   timeplate gen_tp1;
   cycle =
      force_sci;
      force edt_update 0;
      measure_sco;
      pulse clk;
      pulse edt_clock;
   end;
end;
```

```
procedure shift =
   scan_group grp1;
   timeplate gen_tp1;
   cycle =
      force_sci;
      force shared_edt_update 0;
      measure_sco;
      pulse core1_clk;
      pulse core2_clk;
      pulse edt_clock;
   end;
end;
```

If the same procedure occurs in multiple block-level test procedure files, you need to include it just once in the top-level file. But you must include all the pin-specific statements (force, pulse, and so on) from each block-level version.

Block-level Test Procedure Files Example

The following example illustrates two test procedure files written when the EDT logic was created for the same two sub-blocks used in the netlist and dofile examples. Following these block-level examples is the top-level test procedure file created from them. Notice the identical pin names in the force and pulse statements in each block-level file. To force/pulse the correct pins at the top-level, you need to change the block-level pin names at the top level when you merge the statements from the block-level files into the top-level procedure file.

```
// created_edt.testproc
//
set time scale 1.000000 ns ;
set strobe_window time 100 ;

timeplate gen_tp1 =
   force_pi 0 ;
   measure_po 100 ;
```

```
// created2_edt.testproc
//
set time scale 1.000000 ns ;
set strobe_window time 100 ;

timeplate gen_tp1 =
   force_pi 0 ;
   measure_po 100 ;
   pulse clk 200 100;
   pulse edt_clock 200 100;
   pulse ramclk 200 100;
   period 400 ;
end;

procedure capture =
   timeplate gen_tp1 ;
   cycle =
      force_pi ;
      measure_po ;
      pulse_capture_clock ;
   end;
end;

procedure ram_passthru =
   timeplate gen_tp1 ;
   cycle =
      force_pi ;
      pulse_write_clock ;
   end ;
   cycle =
      measure_po ;
      pulse_capture_clock ;
   end;
end;

procedure ram_sequential =
   timeplate gen_tp1 ;
   cycle =
      force_pi ;
      pulse_read_clock ;
      pulse_write_clock ;
   end ;
end ;

procedure clock_sequential =
   timeplate gen_tp1 ;
   cycle =
      force_pi ;
      pulse_capture_clock ;
      pulse_read_clock ;
      pulse_write_clock ;
   end ;
end ;

procedure shift =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   cycle =
      force_sci ;
      force edt_update 0 ;
      measure_sco ;
      pulse clk ;
      pulse edt_clock ;
   end;
end;
```

```
   pulse clk 200 100;
   pulse edt_clock 200 100;
   period 400;
end;

procedure capture =
   timeplate gen_tp1 ;
   cycle =
      force_pi ;
      measure_po ;
      pulse_capture_clock ;
   end;
end;

procedure shift =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   cycle =
      force_sci ;
      force edt_update 0 ;
      measure_sco ;
      pulse clk ;
      pulse edt_clock ;
   end;
end;

procedure load_unload =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   cycle =
      force clk 0 ;
      force edt_bypass 0;
      force edt_clock 0;
      force edt_update 1;
      force scan_en 1 ;
      pulse edt_clock ;
   end ;
   apply shift 26;
end;

procedure test_setup =
   timeplate gen_tp1 ;
   cycle =
      force edt_clock 0 ;
   end;
end;
```

```
procedure load_unload =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   cycle =
      force clk 0 ;
      force edt_bypass 0;
      force edt_clock 0;
      force edt_update 1;
      force ramclk 0 ;
      force scan_en 1 ;
      pulse edt_clock ;
   end ;
   apply shift 16;
end;

procedure test_setup =
   timeplate gen_tp1 ;
   cycle =
      force edt_clock 0 ;
   end;
end;
```

Top-level Test Procedure File Example

The following example illustrates a top-level test procedure file that aggregates all the procedures from the preceding block-level files. Notice that, with the exception of all the shared EDT control pins, pins with the same name from different blocks were given unique names by addition of a prefix (shown in bold font) indicating the block-level design where they occur. Be sure the top-level pin names you use in the top-level test procedure file match the names you gave these pins in the netlist.

```
// all_cores_edt.testproc
//
// Manually created from created1_edt.testproc & created2_edt.testproc
//
set time scale 1.000000 ns ;
set strobe_window time 100 ;

timeplate gen_tp1 =
   force_pi 0 ;
   measure_po 100 ;
   pulse core1_clk 200 100 ;
   pulse core2_clk 200 100 ;
   pulse edt_clock 200 100 ;
   pulse ramclk 200 100 ;
   period 400 ;
end;

procedure capture =
   timeplate gen_tp1 ;
   cycle =
      force_pi ;
      measure_po ;
      pulse_capture_clock ;
   end;
```

```
end;

procedure ram_passthru =
   timeplate gen_tp1 ;
   cycle =
      force_pi ;
      pulse_write_clock ;
   end ;
   cycle =
      measure_po ;
      pulse_capture_clock ;
   end;
end;

procedure ram_sequential =
   timeplate gen_tp1 ;
   cycle =
      force_pi ;
      pulse_read_clock ;
      pulse_write_clock ;
   end ;
end ;

procedure clock_sequential =
   timeplate gen_tp1 ;
   cycle =
      force_pi ;
      pulse_capture_clock ;
      pulse_read_clock ;
      pulse_write_clock ;
   end ;
end ;

procedure shift =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   cycle =
      force_sci ;
      force shared_edt_update 0 ;
      measure_sco ;
      pulse core1_clk ;
      pulse core2_clk ;
      pulse edt_clock ;
   end;
end;

procedure load_unload =
   scan_group grp1 ;
   timeplate gen_tp1 ;
   cycle =
      force core1_clk 0 ;
      force core2_clk 0 ;
      force shared_edt_bypass 0 ;
      force shared_edt_update 1 ;
      force shared_edt_clock 0 ;
      force ramclk 0 ;
      force core1_scan_en 1 ;
      force core2_scan_en 1 ;
```

```
         pulse edt_clock ;
      end;
      apply shift 26;
   end;

   procedure test_setup =
      timeplate gen_tp1 ;
      cycle =
         force edt_clock 0 ;
      end;
   end;
```

Creating the Top-level Netlist

Once all EDT blocks are created and verified, you must merge them into a single top-level netlist for final test pattern generation. Merging the EDT blocks into a single design is the process of building a netlist that instantiates all of the EDT blocks including all of the interconnects needed for the functional design. This process is referred to as the *Integration phase.*

This step-by-step process invokes Tessent Shell in an integration session to make the connections required to integrate the EDT blocks into the netlist. EDT control signals, such as edt_update, edt_clock, edt_bypass, edt_reset, scan_enable and test_en, can be shared between blocks, but all scan channels must have dedicated top-level pins. You must use the add_edt_connections command to actually create the connections between the blocks and the top level of the design as shown in Figure 9-6 on page 259 and Figure 9-7 on page 261.

Note

Alternatively, you can skip the integration process and make these connections manually or with another netlist editing tool.

Prerequisites

- All EDT blocks are created and verified. All design blocks that include EDT logic are fully defined; no blackboxes can be used.
- Prior to beginning this step-by-step process, a top-level test procedure file must be defined. The top-level test procedure file can only use top-level pins that already exist in the netlist before integration.

Procedure

The following commands are typically assembled in a dofile and used to drive the session automatically from the command line. For more information, see “Batch Mode” on page 24.

1. Invoke Tessent Shell. The tool invokes in setup mode.
2. Read in the top-level design netlist and perform the necessary setup.

3. Add the top-level scan chains and test procedure file. For example:

   ```
   add_scan_groups grp1 top.testproc
   ```

4. Define top-level clocks and pin constraints. For example:

   ```
   add_clocks 0 clk tk_clk
   add_pin_constraints tk_clk C0
   ```

5. Enable EDT mapping. For example:

   ```
   set_edt_mapping on -verbose on
   ```

   EDT mapping enables the tool to obtain EDT pin information from the block-level dofiles.

6. Add each EDT block and specify its location and dofile. For example:

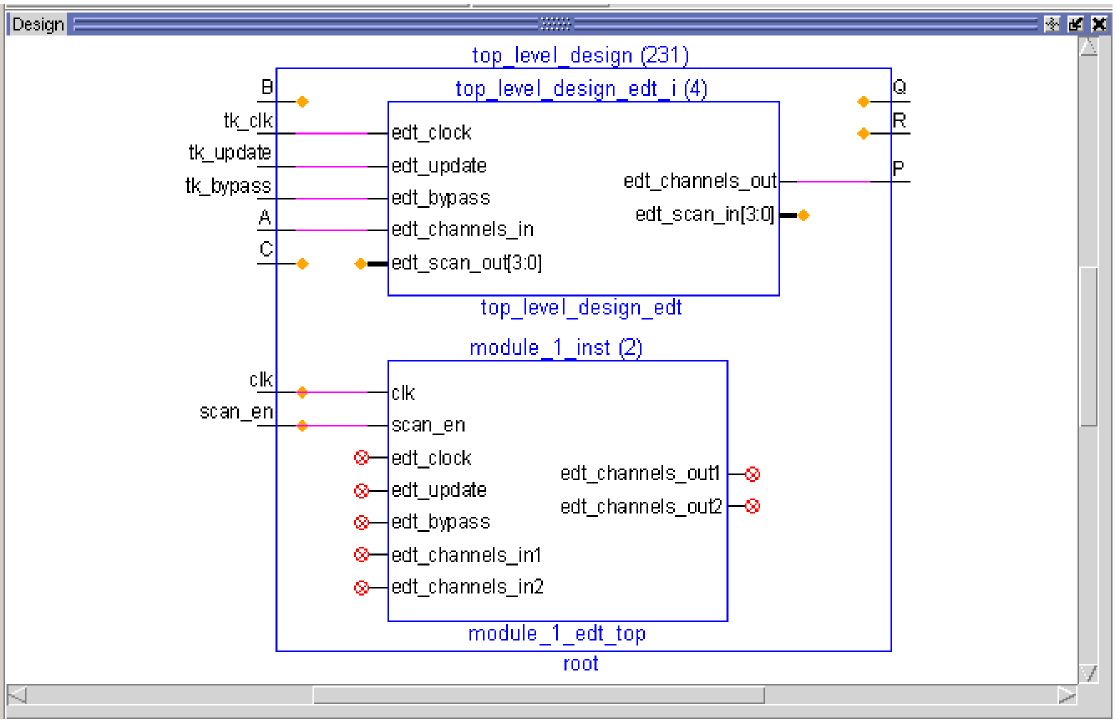
   ```
   add_edt_block top
   set_edt_instances -block_location /
   dofile edt_ip_top/created_edt.dofile

   add_edt_block block1
   set_edt_instances -block_location /module_1_inst
   Sdofile ../module_1/1_edt_ip/created_edt.dofile
   ```

   Where:

   - *top* and *block1* are the names of the EDT blocks
   - "*set_edt_instances -block_location /*" specifies the EDT block location of *top* as the top-level of the design
   - *"set_edt_instances -block_location /module_1_inst"* specifies the EDT block location of *block1* as */module_1_inst*
   - *edt_ip_top/created_edt.dofile* and *../module_1/1_edt_ip/created_edt.dofile* are the dofiles for each block

Figure 9-6 shows the design at this point, before you make any connections to the top level, in the DFTVisualizer Design window.

Figure 9-6. Before Top-level Connections are Created



7. Specify the connections necessary to connect the EDT blocks to the top-level pins.

```
add_edt_connections -signal clock -to tk_clk -all_blocks
add_edt_connections -signal update -to tk_update -all_blocks
add_edt_connections -signal bypass -to tk_bypass -all_blocks

add_edt_connections -signal scan_en -to scan_en -block block1
add_edt_connections -signal input 1 -to B -block block1
add_edt_connections -signal input 2 -to C -block block1

add_edt_connections -signal output 1 -to R -block block1
add_edt_connections -signal output 2 -to Q -block block1
```

8. Specify any additional non-EDT connections needed.

9. Change to analysis mode and run DRC. For example:

```
set_system_mode analysis
```

10. Report and verify that the EDT connections are setup as specified. For example:

    report_edt_connections -all_blocks

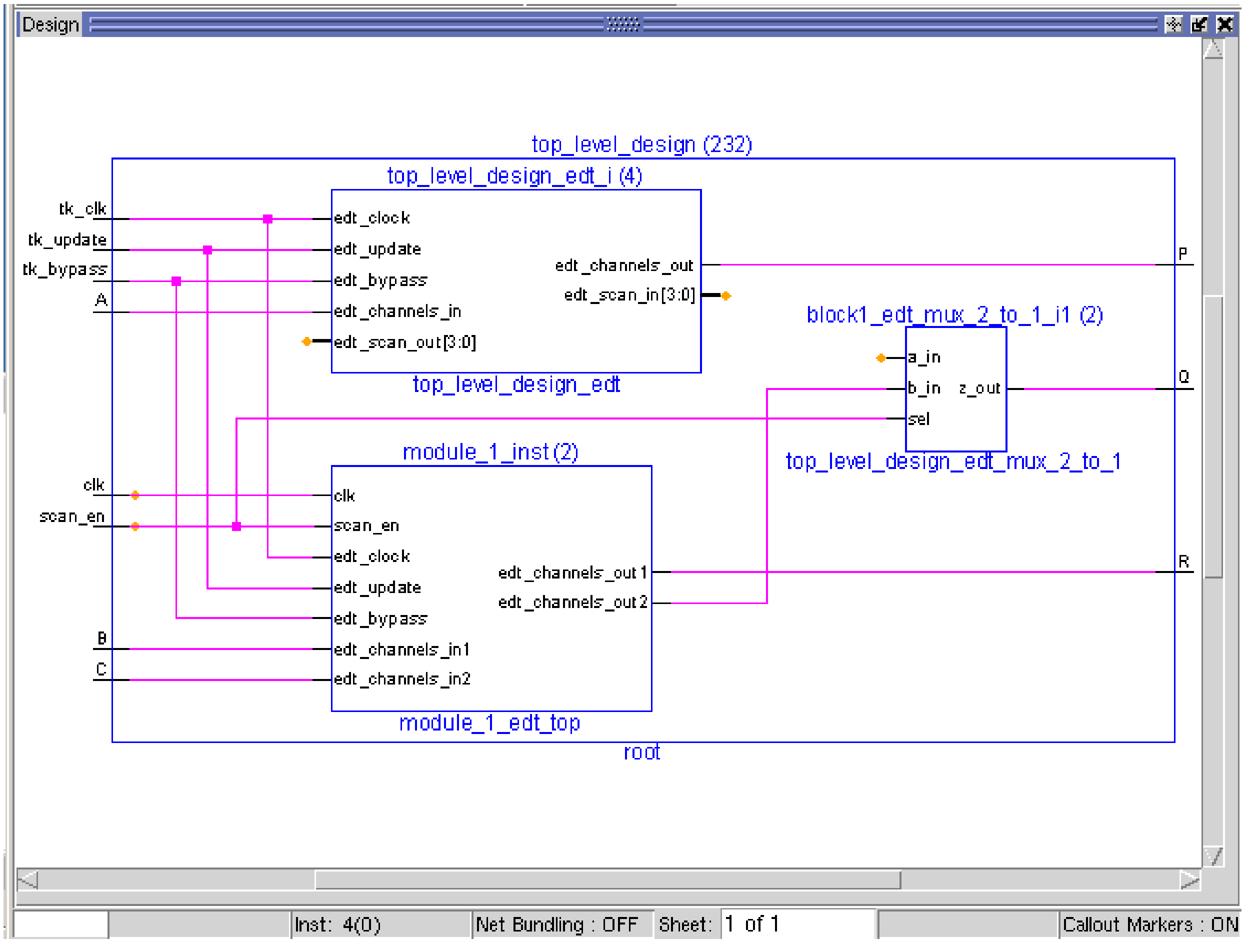
```
//  ---------------------------------------------------------
//                    EDT Signal Connections
//  ---------------------------------------------------------
//  Block   Pin description        Pin name   Connection name
//  -----   ---------------        --------   ---------------
//  top     Clock                  tk_clk
//  top     Update                 tk_update
//  top     Scan channel 1 input   A
//  top      "      "    " output  P
//
//  block1  Clock                  tk_clk
//  block1  Update                 tk_update
//  block1  Scan channel 1 input   B
//  block1   "      "    " output  R
//  block1  Scan channel 2 input   C
//  block1   "      "    " output  Q
//
```

11. Write out the integrated design and DC synthesis script as needed if channel sharing is used. For example:

    write_edt_files integration/top_integrated -replace

Figure 9-7 displays the design after the write_edt_files command is executed.

Figure 9-7. After Top-level Connections are Created



Inst: 4(0) | Net Bundling : OFF | Sheet: 1 of 1 | Callout Markers : ON

Limitations

- The integration process only supports two levels of EDT hierarchy as shown in Figure 9-1. If your design contains additional levels of nested hierarchy within any of the EDT blocks, you must manually update the dofiles for the nested blocks with the correct block-level EDT pins.
- EDT blocks can only be instantiated in a top-level design. The integration session does not support the creation of an external wrapper like the one used for the external flow. For more information on the external flow, see "Compressed Pattern External Flow" on page 33.
- Uncompressed scan chains cannot be added during the integration session. For more information on using uncompressed scan chains, see "Inclusion of Uncompressed Scan Chains" on page 40.

- EDT mapping must be enabled for the integration session. See "Creating the Top-level Dofile" on page 262.

Related Topics

Creating a Top-level Test Procedure File

delete_edt_connections

Creating the Top-level Dofile

This procedure creates a top-level dofile that combines the block-level dofiles and specifies clock pins and pin constraints required for top-level test pattern generation.

Prerequisites

- Top-level test procedure file. For more information, see "Creating a Top-level Test Procedure File" on page 249.

Procedure

1. Open a text editor to create a dofile.
2. At the beginning of the dofile, reference the top-level test-procedure file to use for generating test patterns. For example:

   ```
   add_scan_groups grp1 ./all_top_level_edt.testproc
   ```

   For more information, see the add_scan_groups command and "Creating a Top-level Test Procedure File" on page 249.
3. Define the top-level clocks and pin constraints common to all blocks. For example:

   ```
   add_clocks 0 edt_clock
   add_pin_constraints edt_clock C0
   ```

4. Enable the tool to get the setup information for each block from the block-level dofile created when the EDT logic was created. For example:

   ```
   set_edt_options mapping on -verbose on
   ```

   The setup information is automatically obtained from dofiles specified for each block and mapped to the top-level pins. If EDT mapping is not enabled, you must manually enter and map the set information from each dofile in the top-level dofile.
5. Specify pin constraints and clock signals for the first EDT block (core1) using pin names from the top-level netlist. For example:

   ```
   add_clocks 0 clk_core1
   add_clocks 0 ramclk_core1
   add_write_controls 0 ramclk_core1
   ```

```
add_read_controls 0 ramclk_core1
```

6. Instantiate the first EDT block (core1) into the dofile. For example:

   ```
   add_edt_blocks Core1
   ```

   Precede each block-specific set of commands with an add_edt_blocks command to set the context. This command also defines an arbitrary identifier of your choice, referred to as the block tag, for block being defined. The block tag, which provides a way to refer to this block later in the session, should be unique.

7. Specify the path to the dofile the tool created for core1. For example:

   ```
   dofile ../../Core1/generated/created_core1_edt.dofile
   ```

8. Repeat steps 5, 6, and 7 for each EDT block, including the top-level glue logic block. To instantiate the same EDT block multiple times at the top-level, see "Instantiating an EDT Block Multiple Times" on page 263.

9. After all EDT blocks are defined, run DRC and top-level pattern creation. For example:

   ```
   set_system_mode analysis
   report_drc_rules
   create test patterns
   ```

Related Topics

Instantiating an EDT Block Multiple Times	Top-level Dofile Example

Instantiating an EDT Block Multiple Times

To instantiate a EDT block multiple times in the top-level dofile, you must specify a unique location within the top-level design for each instance using the set_edt_instances command.

The set_edt_instances command specifies the name of the Verilog module in the top-level netlist where the EDT logic is located.

For example:

```
add_edt_blocks a1
set_edt_instances -block_location
   /piccpu_top_level_i/piccpu_core1_i
dofile created2_edt.dofile

add_edt_blocks a2
set_edt_instances -block_location
   /piccpu_top_level_i/piccpu_core2_i
dofile created2_edt.dofile

add_edt_blocks a3
set_edt_instances -block_location
   /piccpu_top_level_i/piccpu_core3_i
```

```
dofile created2_edt.dofile
```

Block a1, block a2, and block a3 are instantiations of the same EDT block and use the same dofile but are located in separate modules of the design netlist.

Top-level Dofile Example

The following dofile specifies setups for creating top-level test patterns for a modular design with three unique EDT blocks.

```
// all_cores_edt.do
//
// Dofile to create top-level patterns for design with 3 unique EDT blocks
add_scan_groups grp1 all_cores_edt.testproc

// Define top-level clocks and pin constraints for all blocks.
add_clocks 0 core1_clk
add_write_controls 0 core1_ramclk
add_read_controls 0 core1_ramclk
add_clocks 0 core2_clk
add_clocks 1 core3_clk
add_clocks 0 shared_edt_clock
add_pin_constraints shared_edt_clock C0

set_edt_options mapping on

// DEFINE BLOCK 1
add_edt_blocks a1 //Name this block “a1” (block’s tag for later cmds)
dofile created1_edt.dofile

// DEFINE BLOCK 2
add_edt_blocks a2 //Name this block “a2”
dofile created2_edt.dofile

// DEFINE BLOCK 3
add_edt_blocks a3 //Name this block “a3”
dofile created3_edt.dofile

// Report what’s been set.
report_scan_chains -all_blocks
report_edt_blocks
report_edt_configurations -all_blocks

// Once all EDT blocks are defined, perform DRC and verify there are
// no DRC violations.
set_system_mode analysis
report_drc_rules

// create_patterns that use all blocks simultaneously and generate
// patterns that target faults in the entire design.
create_patterns
...
```

Generating Top-level Test Patterns

Note
To generate top-level patterns, you must have a top-level design netlist, dofile and test procedure file, prepared as described earlier in this chapter.

Generating test patterns for the top-level of a modular design is similar to creating test patterns in the standard flow except that you set one block up at a time with the following commands:

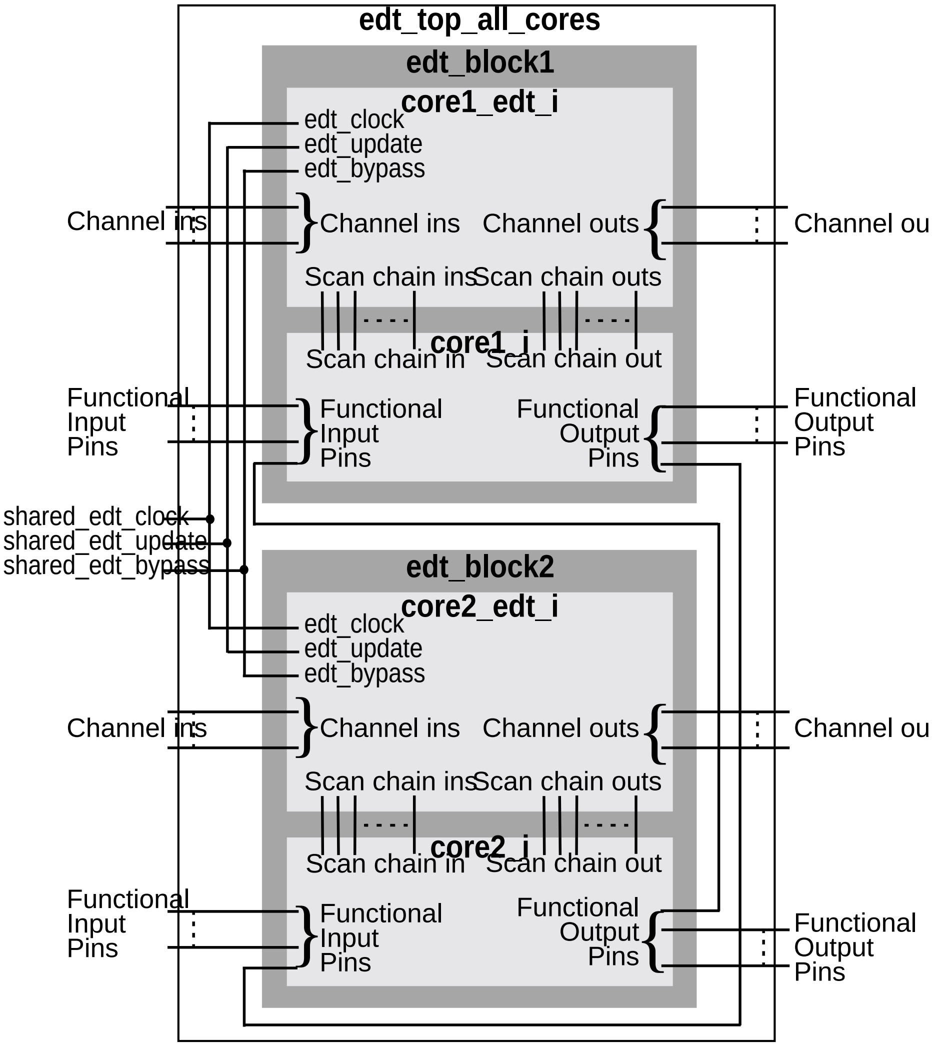
- set_current_edt_block — Applies EDT-specific commands and the add_scan_chains command to a particular EDT block. Restricting commands in this way enables you to re-specify the characteristics of an individual block without affecting other parts of the design.
- report_edt_blocks — Reports on EDT blocks currently defined in Tessent Shell memory.
- delete_edt_blocks — Deletes EDT blocks from Tessent Shell memory.

A few reporting commands also operate on the current EDT block by default, but provide an -All_blocks switch that enables you to report on the entire design. All other commands (set_system_mode, create_patterns and report_statistics for example) operate only on the entire design.

Modular Compressed ATPG Flow Example

Figure 9-8 shows an example of the modular compressed ATPG commands used to integrate EDT blocks and generate test patterns. In this example, EDT control signals are shared at the top level; each EDT block is created with the EDT logic and the scan-inserted core inside of a wrapper.

Figure 9-8. Netlist with Two Cores Sharing EDT Control Signals



1. Invoke Tessent Shell, set the context, and read in the design and library.
2. Perform necessary setup and then define scan chains, clocks and EDT logic for the first block. For example:

```
// Perform setup.
set_current_design edt_block1
...

// Define scan chains, clocks, and EDT hardware.
add_scan_groups grp1 group1.testproc
add_scan_chains chain1 grp1 edt_si1 edt_so1
add_scan_chains chain2 grp1 edt_si2 edt_so2
...
add_clocks clk1 0
set_edt_options -channels 6
set_system_mode analysis
```

3. Create EDT logic with unique module names based on the core module name for the first block. For example:

   ```
   // Create EDT hardware with unique module names.
   write_edt_files created1 -replace
   ```

4. Delete the design using the delete_design command.
5. Return to setup mode using the set_system_mode command.
6. Read in the second block and repeat steps 2 and 3.
7. Using the DC script output during the EDT logic creation, synthesize the EDT logic for each block.
8. Verify that the EDT logic is instantiated properly by generating and simulating test patterns for each of the resultant gate-level netlists. This is done using the test bench created during test pattern generation and a timing-based simulator.
9. Verify that the block-level scan chains are balanced.
10. Create the top-level netlist, dofile, and test procedure files. The following example shows the top-level dofile. For more information, see "Creating the Top-level Dofile" on page 262.

    Commands and options specific to *modular* compressed ATPG are shown in bold font.

```
// Define the top-level test procedure file to be used by all blocks.
add_scan_groups grp1 top_level.testproc

// Define top-level clocks and pin constraints here.
add_clocks...
add_read_controls...
add_write_controls...
add_pin_constraints...
...

// Activate automatic mapping of commands from the block-level dofiles.
set_edt_options mapping on

// Define the block tag (this is an arbitrary name) for an EDT block
//    and automatically set it as the current EDT block.
add_edt_blocks cpu1

// Define the block by executing the commands in its block-level dofile.
dofile cpu1_edt.dofile

// Repeat the preceding procedure for another block.
add_edt_blocks cpu2
dofile cpu2_edt.dofile

// Once all EDT blocks are defined, create_patterns that use all the
// blocks simultaneously and generate patterns that target faults in
// the entire design.

// Flatten the design, run DRCs.
```

```
set_system_mode analysis

// Verify the EDT configuration.
report_edt_configurations -all_blocks

// Generate patterns.
create_patterns

// Create reports.
report_statistics
report_scan_volume

write_patterns...
exit
```

Modular Flow Command Reference

Table 9-2 describes commands used for the modular design flow.

Table 9-2. Modular Compressed ATPG Command Summary

Command	Description
add_edt_blocks	Creates a name identifier for an EDT block instantiated in a netlist.
add_edt_connections	Specifies EDT logic connections during the top-level integration step of the modular design flow.
delete_edt_blocks	Removes the specified EDT block(s) from the internal database.
delete_edt_connections	Deletes connections previously specified with the add_edt_connections command
report_edt_blocks	Displays current user-defined EDT block names.
report_edt_configurations	Displays the configuration of the EDT logic.
report_edt_connections	Reports pin connections made during the top-level integration session of the modular flow.
report_edt_instances	Displays the instance pathnames of the top-level EDT logic, decompressor, and compactor.
set_current_edt_block	Directs the tool to apply subsequent commands only to a particular EDT block, not globally.
set_edt_instances	Specifies the instance name or instance pathname of the design block that contains the EDT logic for DRC.
set_edt_mapping	Enables the automatic mapping necessary for block-level dofiles to be reused for top-level pattern creation.

Table 9-2. Modular Compressed ATPG Command Summary

Command	Description
write_design	If the design has been modified after executing the write_edt_files, you must update the netlist using this command.
write_edt_files	Writes all the EDT logic files required to implement the EDT technology in a design.

Appendix A
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 Mentor Graphics Support.

Documentation

A comprehensive set of reference manuals, user guides, and release notes is available in two formats:

- HTML for searching and viewing online
- PDF for searching, viewing online, and printing

The documentation is available from each software tool and online at:

`http://supportnet.mentor.com`

For more information on setting up and using Tessent documentation, see the "Using Tessent Documentation" chapter in the *Managing Mentor Graphics Tessent Software* manual.

Mentor Graphics Support

Mentor Graphics software support includes software enhancements, access to comprehensive online services with SupportNet, and the optional On-Site Mentoring service. For details, see:

`http://supportnet.mentor.com/about/`

If you have questions about a software release, you can log in to SupportNet and search thousands of technical solutions, view documentation, or open a Service Request online:

`http://supportnet.mentor.com`

If your site is under current support and you do not have a SupportNet login, you can register for SupportNet by filling out a short form here:

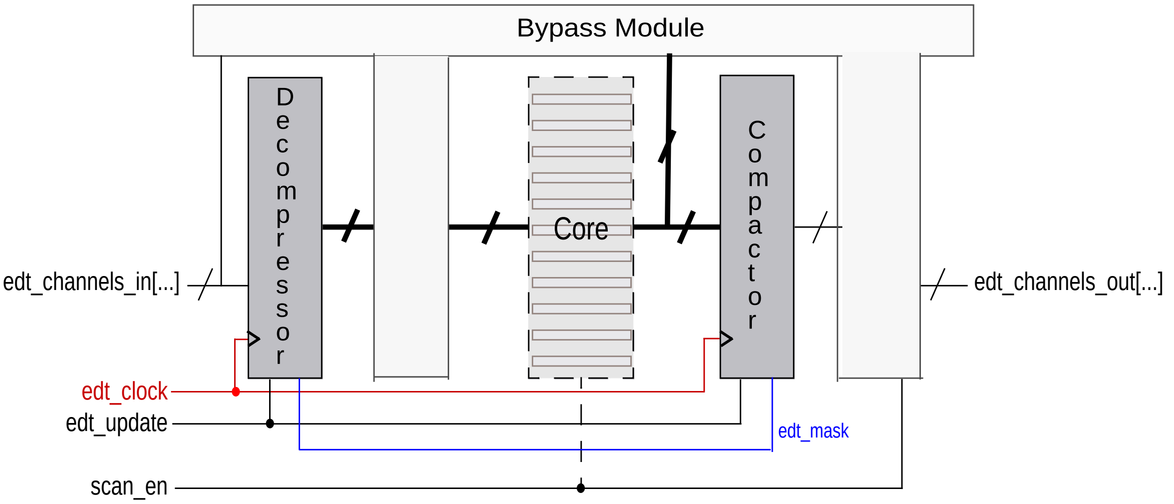
`http://supportnet.mentor.com/user/register.cfm`

All customer support contact information is available here:

`http://supportnet.mentor.com/contacts/supportcenters/index.cfm`

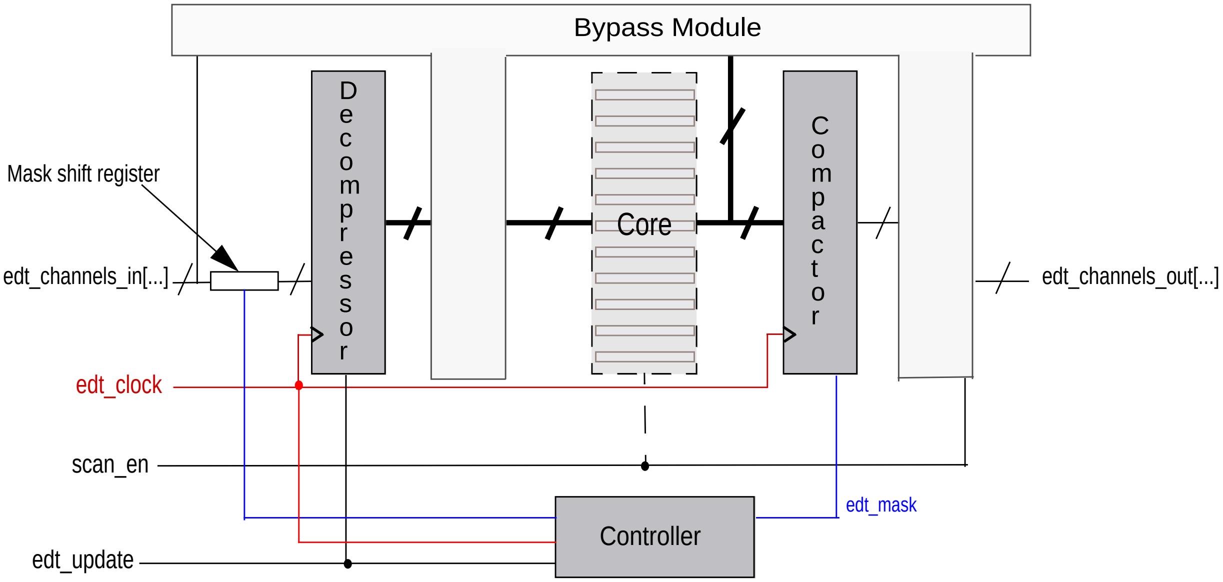
Appendix B
EDT Logic Specifications

EDT Logic with Basic Compactor and Bypass Module



NOTE: Functional pins not shown

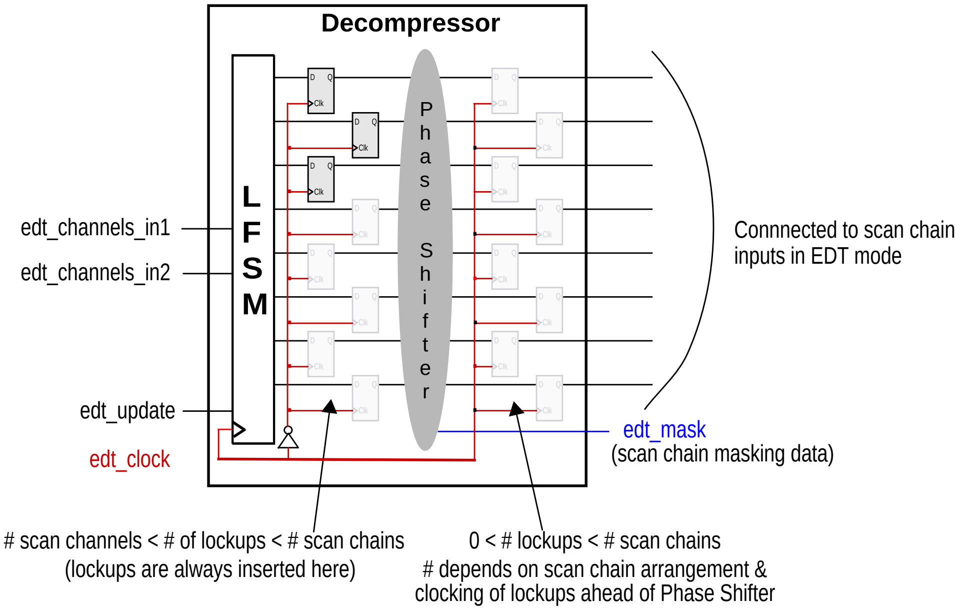
EDT Logic with Xpress Compactor and Bypass Module



NOTE: Functional pins not shown

Decompressor Module with Basic Compactor

The following illustration shows details for a decompressor used with a basic compactor, eight scan chains, and two scan channels.



Connnected to scan chain inputs in EDT mode

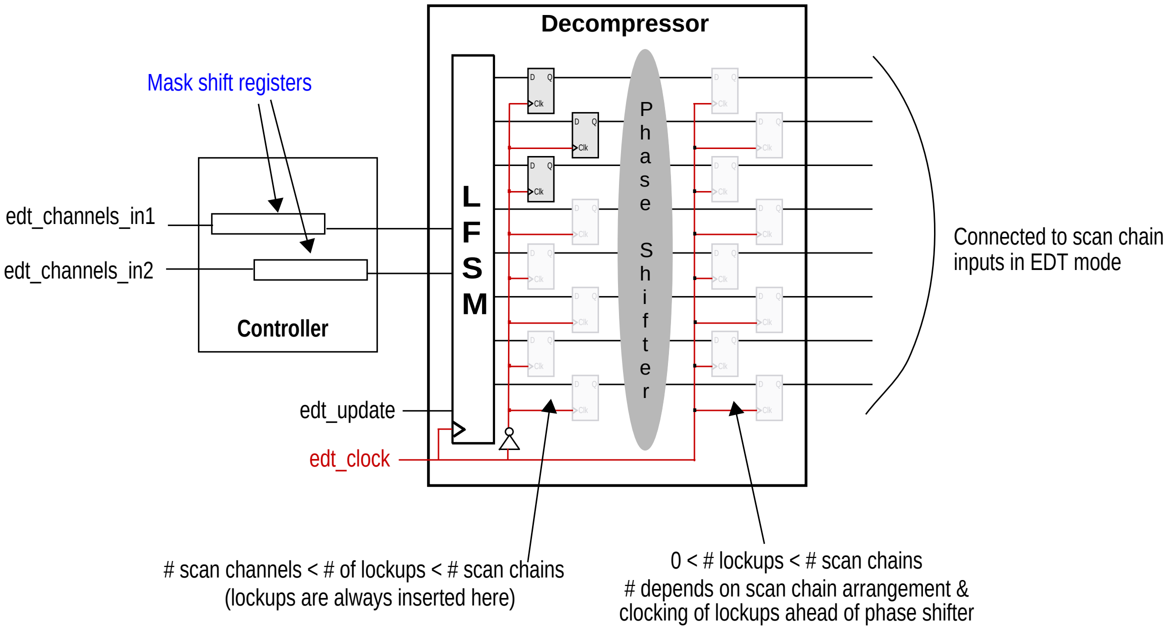
(scan chain masking data)

# scan channels < # of lockups < # scan chains (lockups are always inserted here)

0 < # lockups < # scan chains
# depends on scan chain arrangement & clocking of lockups ahead of Phase Shifter

Decompressor Module with Xpress Compactor

The following illustration shows details for a decompressor used with an Xpress compactor, eight scan chains, and two scan channels.

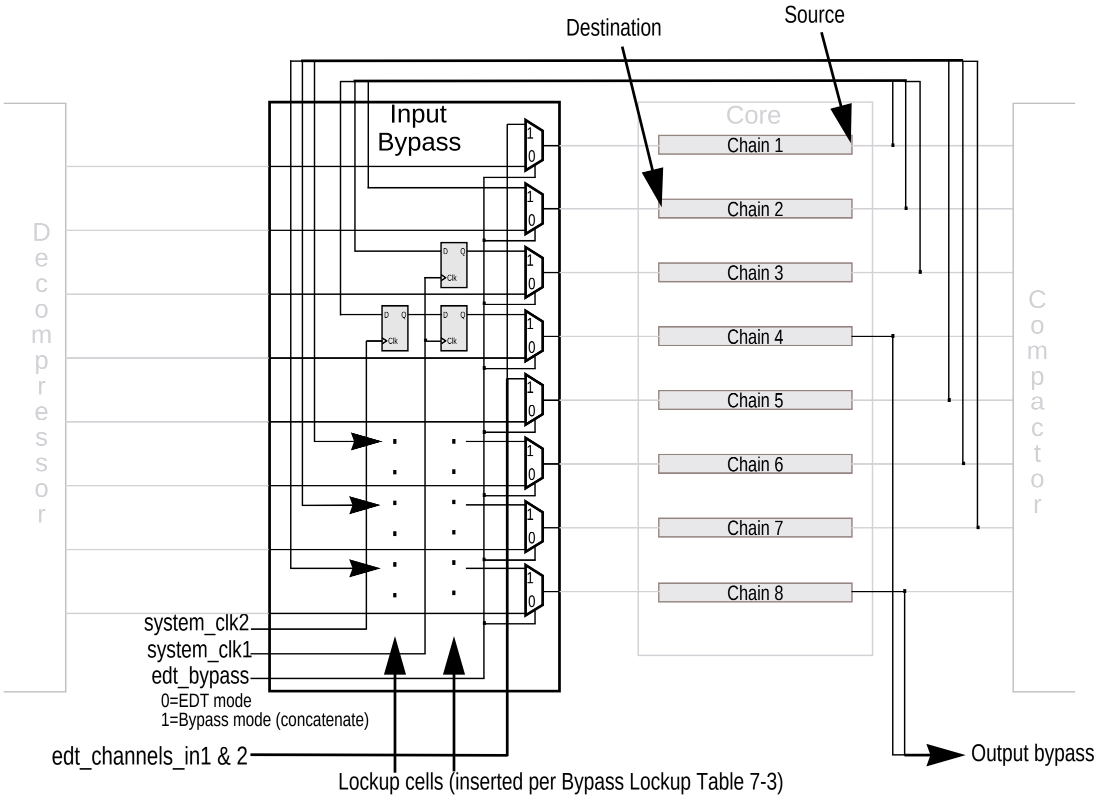


Connected to scan chain inputs in EDT mode

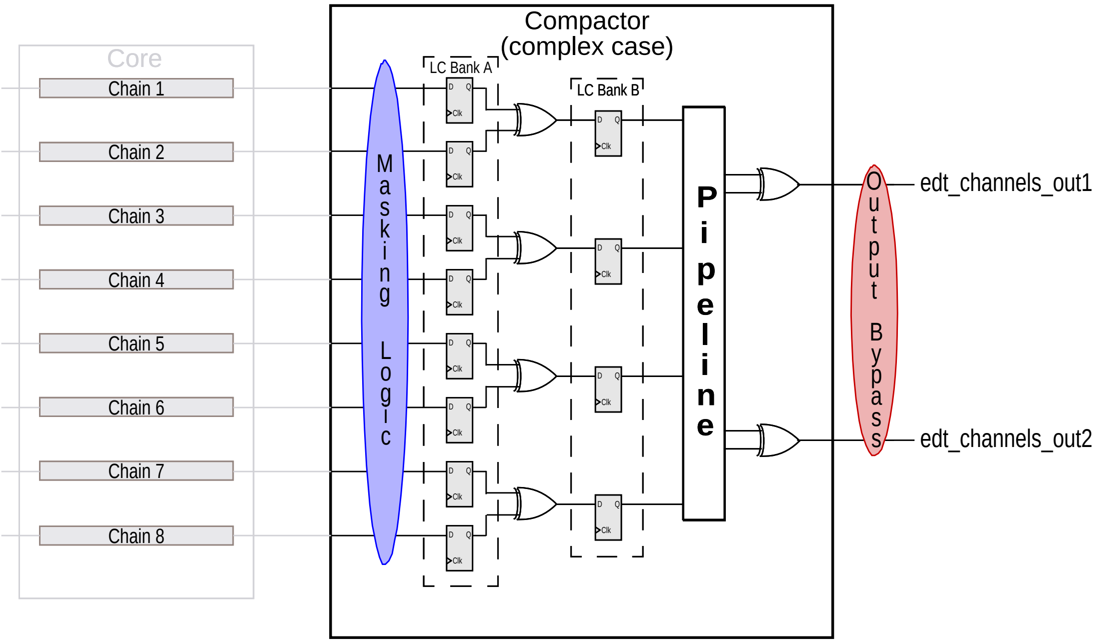
# scan channels < # of lockups < # scan chains (lockups are always inserted here)

0 < # lockups < # scan chains
# depends on scan chain arrangement & clocking of lockups ahead of phase shifter

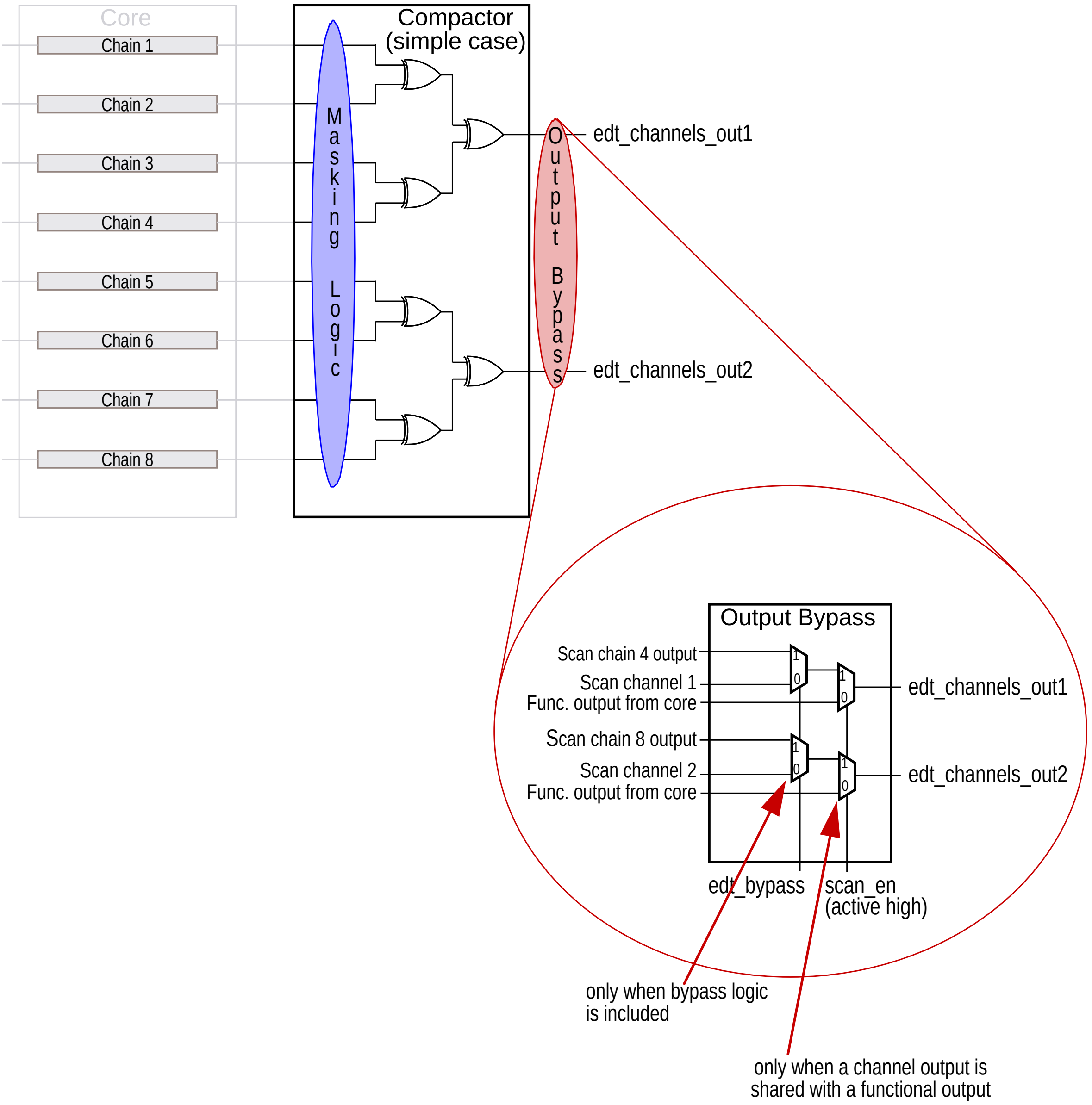
Input Bypass Logic



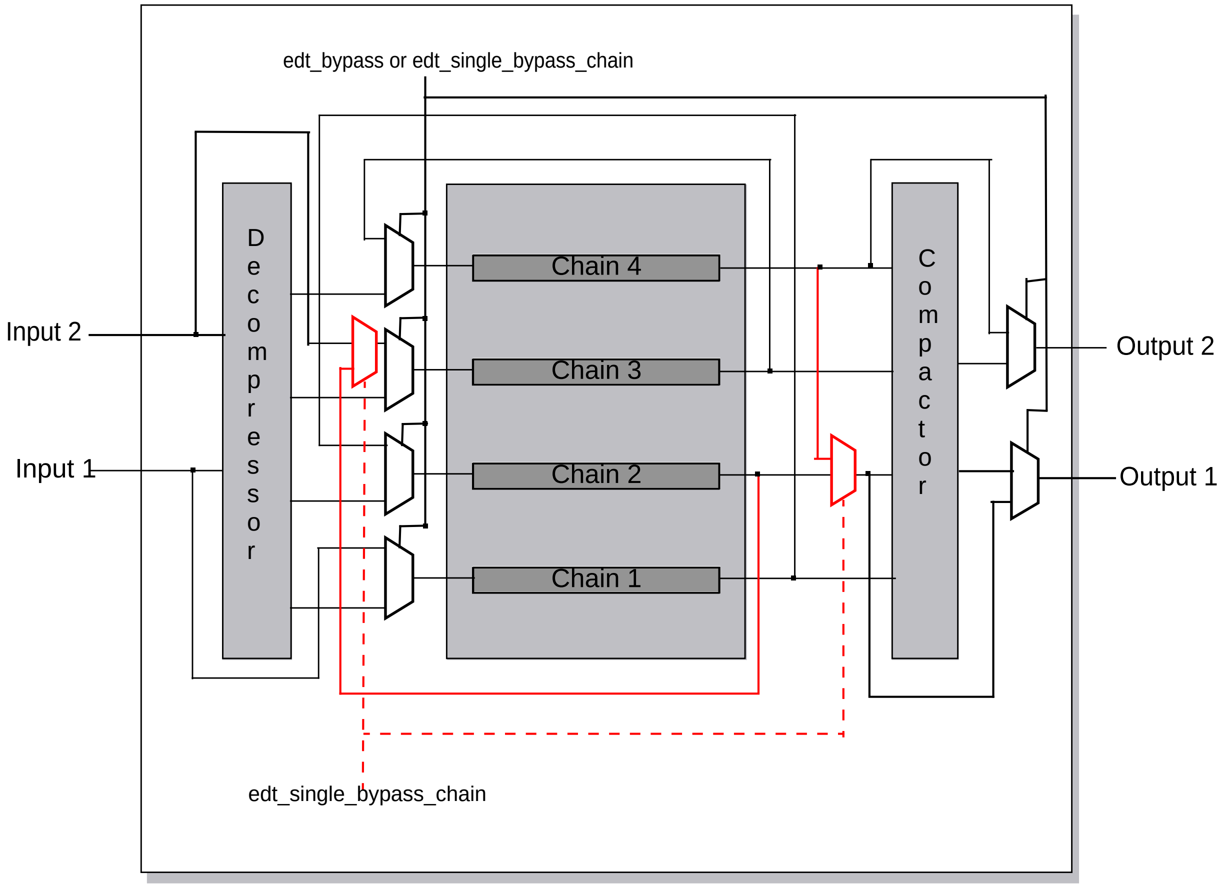
Compactor Module



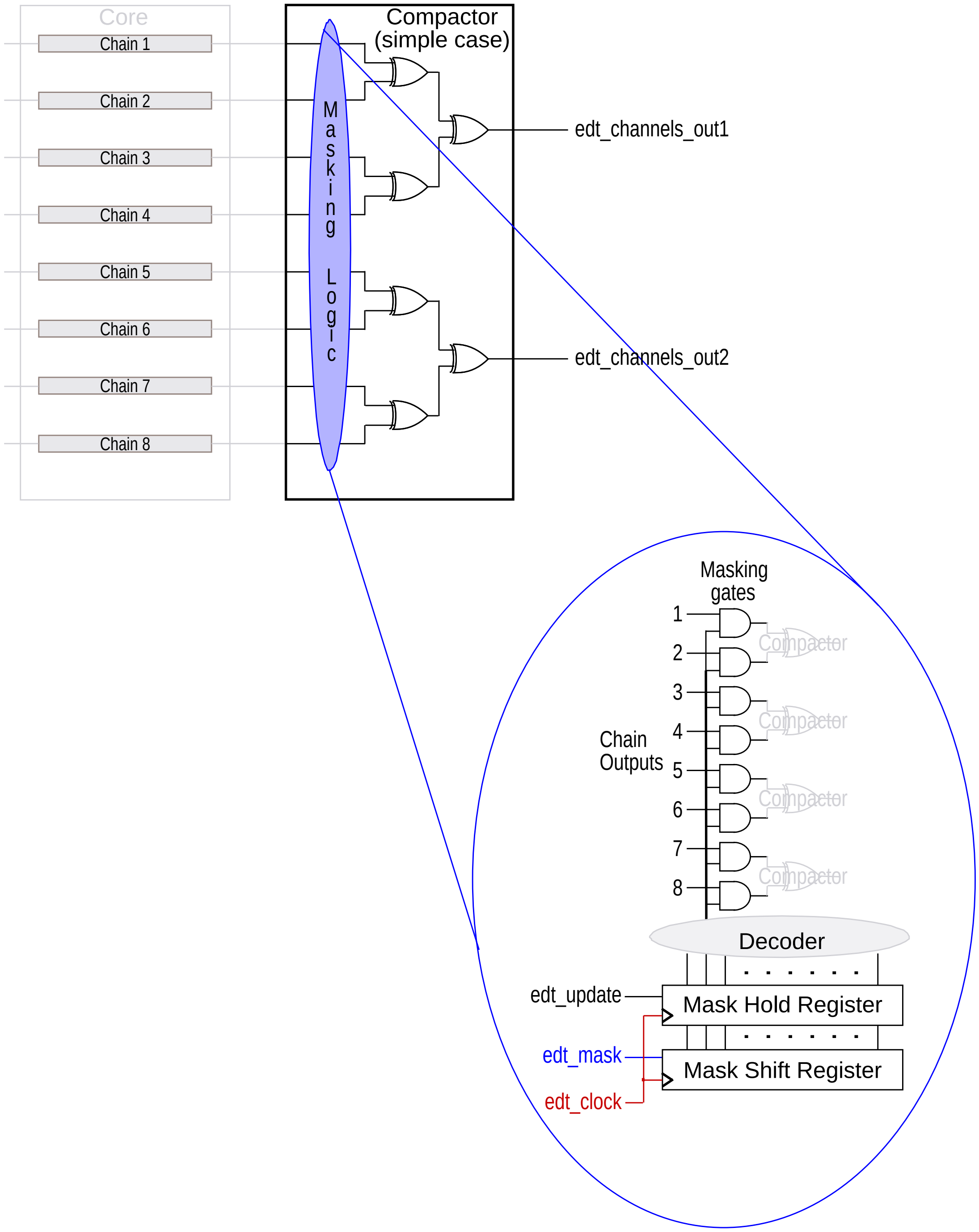
Output Bypass Logic



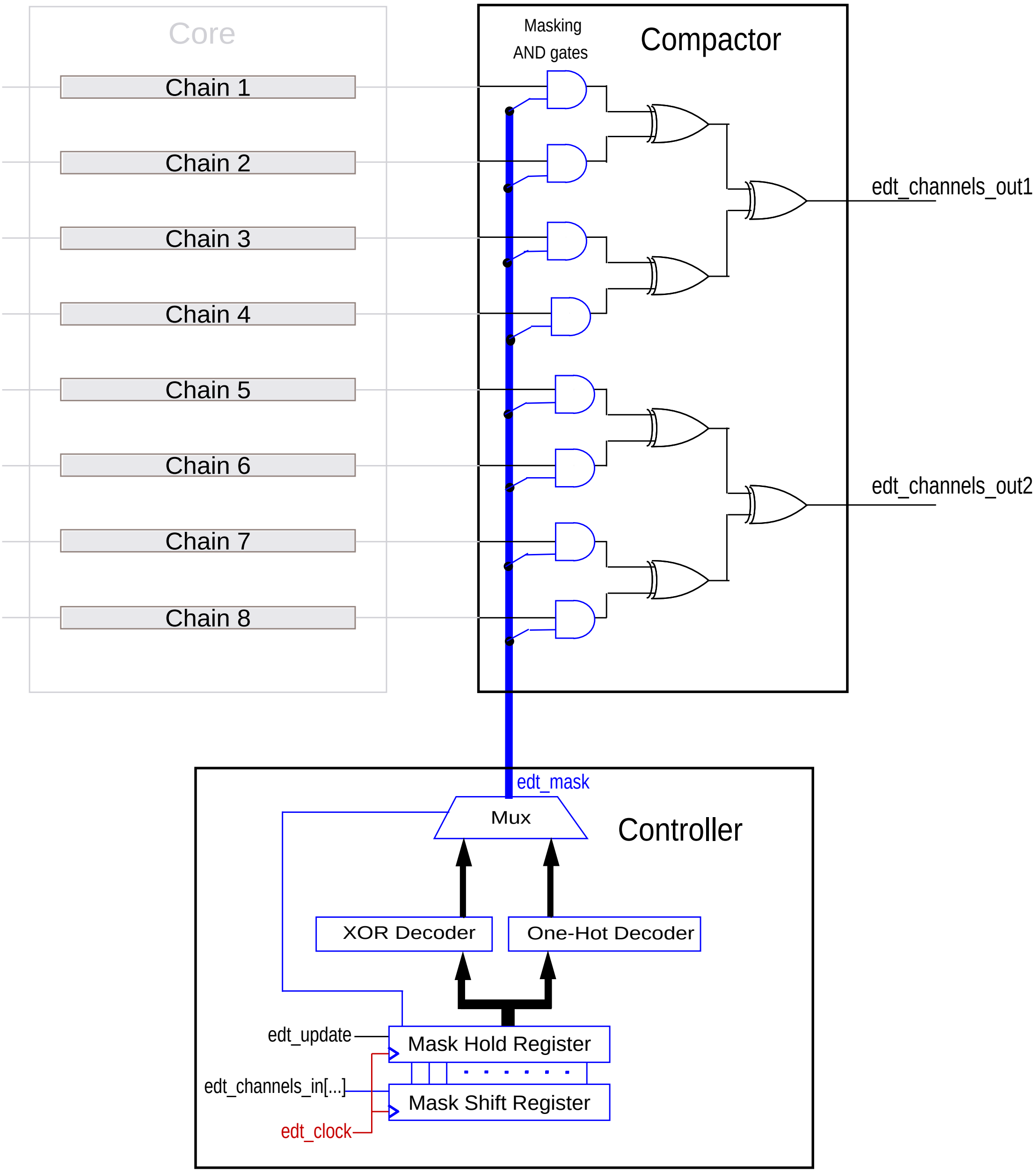
Single Chain Bypass Logic



Basic Compactor Masking Logic



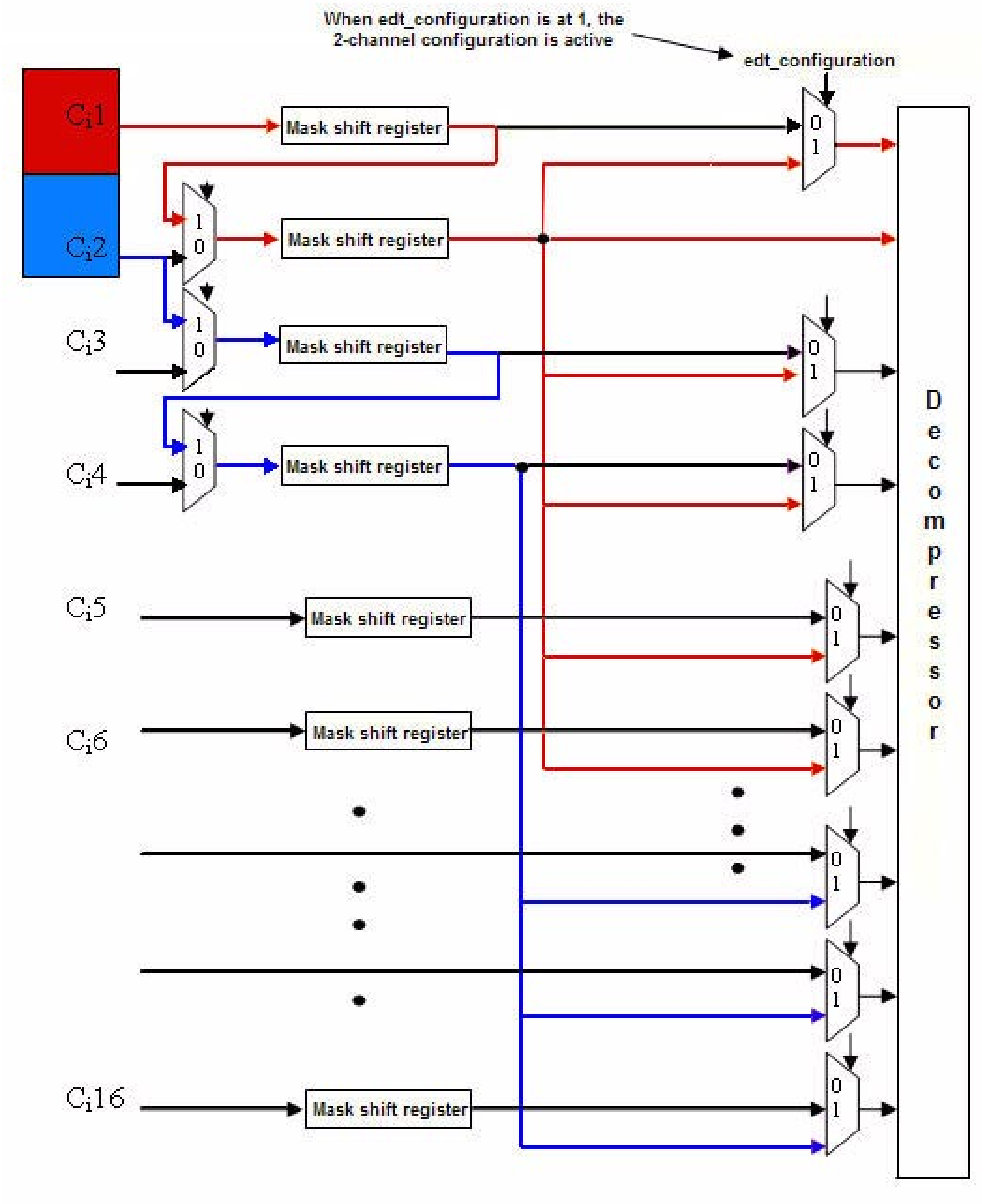
Xpress Compactor Controller Masking Logic



Dual Compression Configuration Input Logic

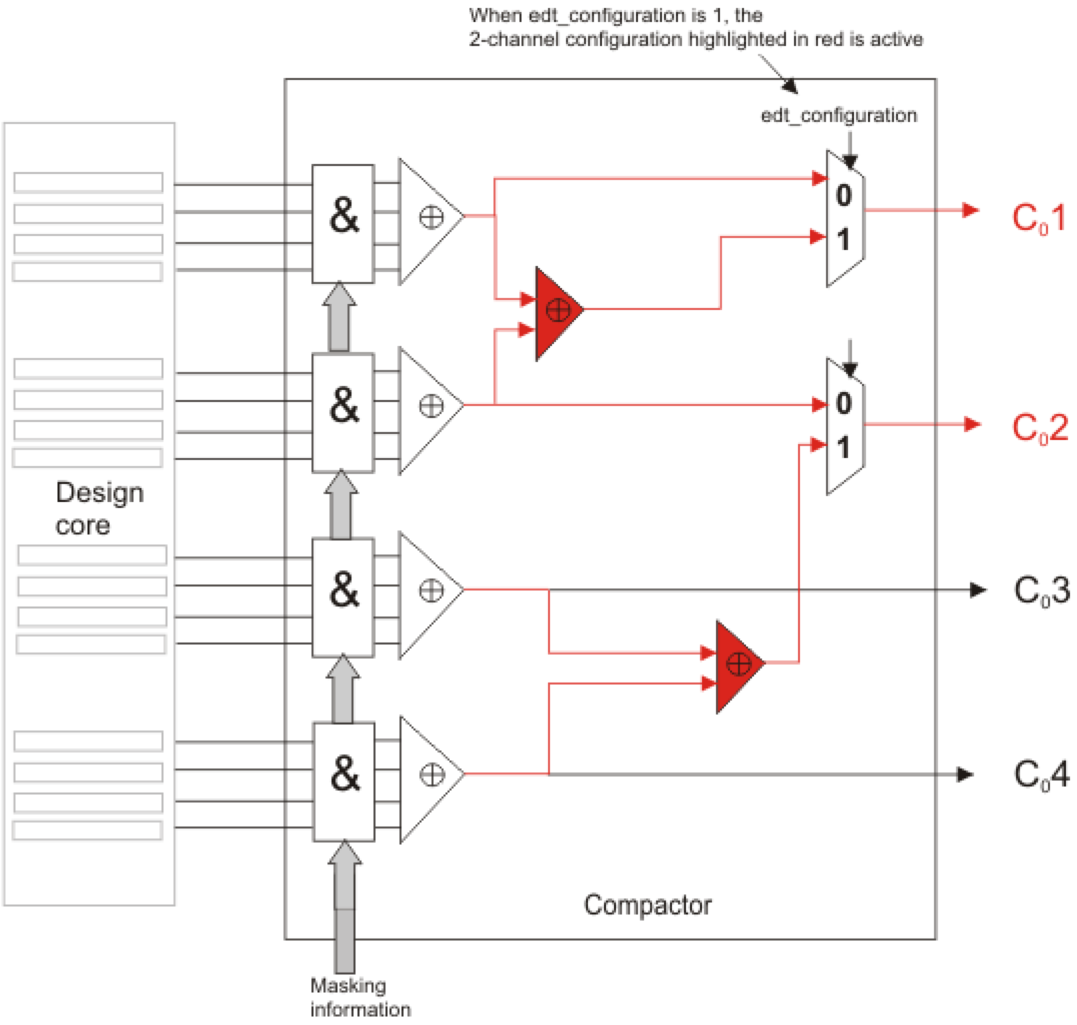
The following illustration shows input logic details when both a 2-channel and a 16-channel compression configuration are defined. Note that the first 2 channels of the 16-channel configuration are always used for the 2-channel configuration.

Red highlights the path for channel 1 when the 2-channel configuration is active. Blue highlights the path for channel 2 when the 2-channel input configuration is active.

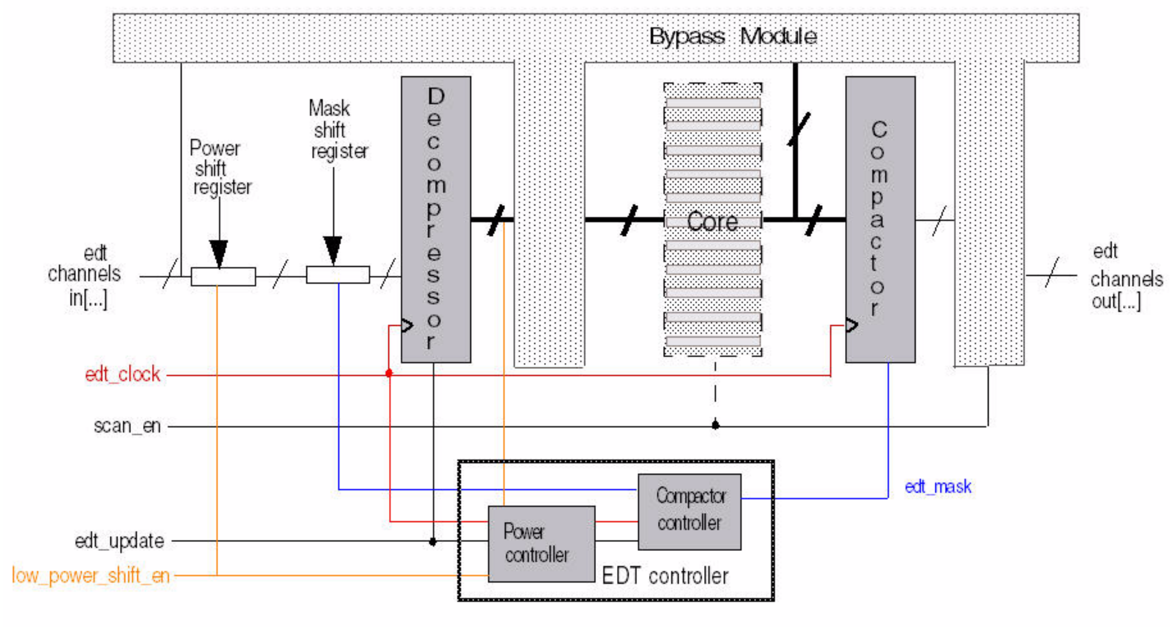


Dual Compression Configuration Output Logic

The following illustration shows output logic details when both a 2-channel and a 4-channel compression configuration are defined. Note that the first 2 channels of the 4-channel configuration are always used for the 2-channel configuration.



EDT Logic with Power Controller



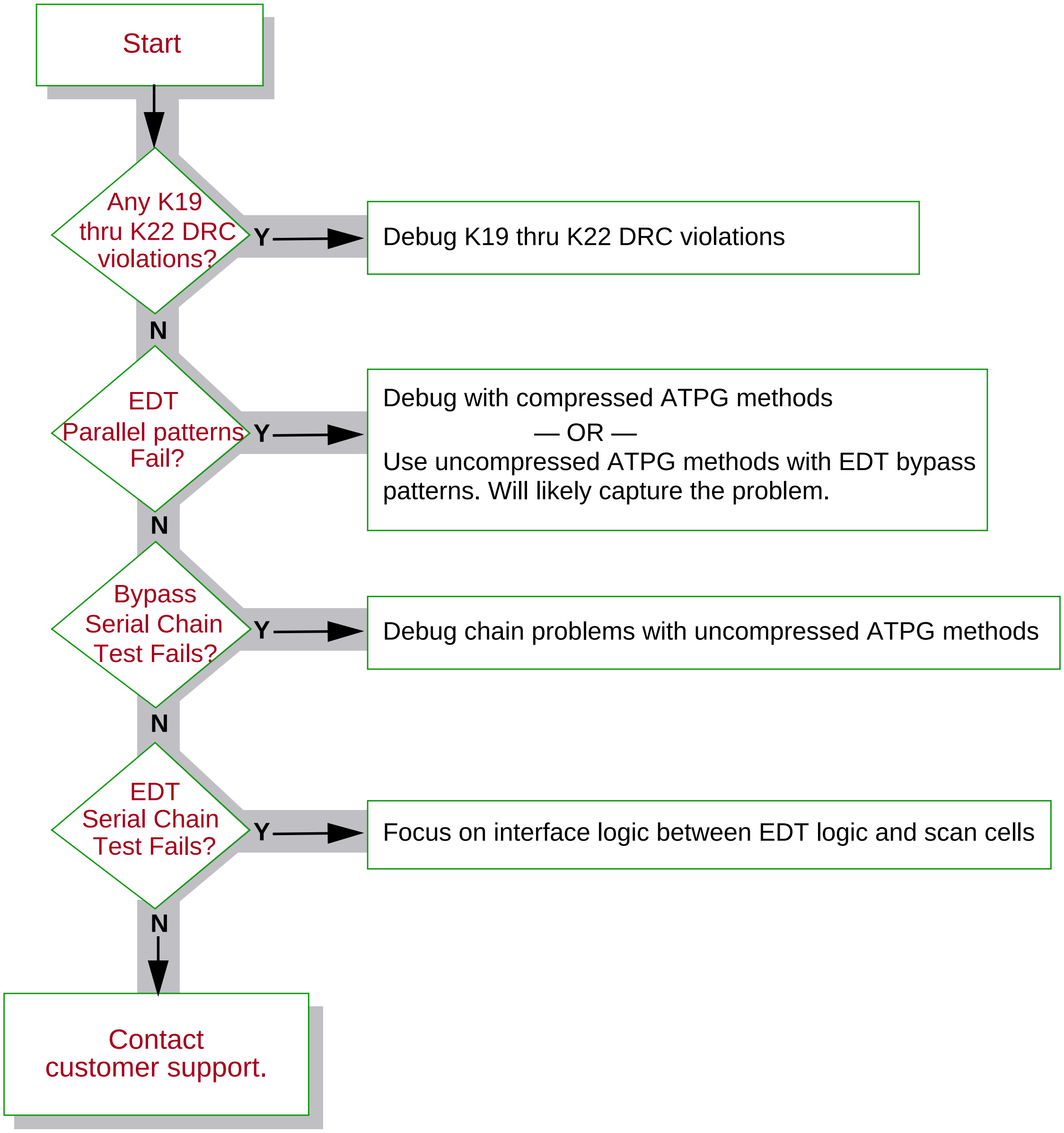
Appendix C
Troubleshooting

This appendix is divided into three parts. The first part, "Debugging Simulation Mismatches," lists some EDT-specific aspects of debugging simulation mismatches that you may already be familiar with from past ATPG experience. The second part, "Resolving DRC Issues," contains some explanation (and examples where applicable) of causes of EDT-specific design rule violations and possible solutions. The "Miscellaneous" section covers a few topics that are unrelated to simulation mismatches or DRC.

Debugging Simulation Mismatches

This section provides a suggested flow for debugging simulation mismatches in a design that uses EDT. You are assumed to be familiar with the information provided in the "Debugging Simulation Mismatches in Tessent FastScan" section of the *Tessent Scan and ATPG User's Manual*, so that information is not repeated here. Your first step with EDT should be to determine if the source of the mismatch is the EDT logic or the core design. Figure C-1 shows a suggested flow to help you begin this process.

Figure C-1. Flow for Debugging Simulation Mismatches


Start
Any K19 thru K22 DRC violations?
Y
Debug K19 thru K22 DRC violations
N
EDT Parallel patterns Fail?
Y
Debug with compressed ATPG methods
— OR —
Use uncompressed ATPG methods with EDT bypass patterns. Will likely capture the problem.
N
Bypass Serial Chain Test Fails?
Y
Debug chain problems with uncompressed ATPG methods
N
EDT Serial Chain Test Fails?
Y
Focus on interface logic between EDT logic and scan cells
N
Contact customer support.

If the core design is the source of the mismatch, then you can use uncompressed ATPG troubleshooting methods to pinpoint the problem. This entails saving bypass patterns from compressed ATPG, which you then process and simulate in uncompressed ATPG with the design configured to operate in bypass mode. Alternatively, you can invoke Tessent Shell with the circuit (configured to run in bypass mode) and generate another set of uncompressed patterns. For more information, refer to “Compression Bypass Logic” in Chapter 7.

Resolving DRC Issues

“Design Rule Checking” in the *Tessent Shell Reference Manual* provides full descriptions of the EDT-specific “K” rules. This section supplements that information with some suggestions to help you reduce the occurrence of certain DRC violations.

K19 through K22 DRC Violations

K19 through K22 are simulation-based DRCs. They verify the decompressor and compactor through zero-delay serial simulation and analyze mismatches to try to determine the source of each mismatch. As a troubleshooting aid, these DRCs transcript detailed messages listing the gates where the tool’s analysis determined each mismatch originated, and specific simulation results for these gates.

The tool can provide the most debugging information if you have preserved the EDT logic hierarchy, including pin pathnames and instance names, during synthesis. When this is not the case and either rule check fails, the tool transcripts a message that begins with the following reminder (K22 would be similar):

```
Warning: Rule K19 can provide the most debug information if the EDT logic
         hierarchy, including pin and instance names, is preserved during
         synthesis and can be found by Tessent TestKompress.
```

The message then lists specifics about instance(s) and/or pin pathname(s) the tool cannot resolve, so you can make adjustments in tool setups or your design if you choose. For example, if the message continues:

```
The following could not be resolved:
   EDT logic top instance "edt_i" not found.
   EDT decompressor instance "edt_decompressor_i" not found.
```

you can use the set_edt_instances command to provide the tool with the necessary information. Use the report_edt_instances command to double-check the information.

If the tool can find the EDT logic top, decompressor and compactor instances, but cannot find expected EDT pins on one or more of these instances, the specifics would tell you about the pins as in this example for an EDT design with two channels:

```
The following could not be resolved:
   EDT logic top instance "edt_i" exists, but could not find
      2-bit channel pin vector "edt_channels_in" on the instance.
   EDT decompressor instance "edt_decompressor_i" exists, but
      could not find 2-bit channel pin vector "edt_channels_in"
      on the instance.
```

When the tool is able to find the EDT logic top, decompressor and compactor instances, but cannot resolve a pin name within the EDT logic hierarchy, it is typically because the name was changed during synthesis of the EDT RTL. To help prevent interruptions of the pattern creation flow to fix a pin naming issue, you are urged to preserve during synthesis, the pin names the tool created in the EDT logic hierarchy. For additional information about the synthesis step, refer to "The EDT Logic Synthesis Script" on page 100.

Debugging Best Practices

For most common K19 and K22 debug tasks, you can report gate simulation values with the set_gate_report Drc_pattern command.

Typical debug tasks include checking for correct values on:

- EDT control signals (edt_clock, edt_update, edt_bypass, edt_reset)
- Sensitized paths from:
    - Input channel pins to the decompressor and from the decompressor to the scan chains during shift. (K19)
    - Scan chains to the compactor and from the compactor to the output channel pins during shift. (K22)

When you use the Drc_pattern option the gate simulation data for different procedures in the test procedure file display. For more information on the use of Drc_pattern reporting, refer to "Debugging State Stability" in the *Tessent Scan and ATPG User's Manual*.

In rare cases, you may need to see the distinct simulation values applied in every shift cycle. For these special cases, you can force the tool to simulate every event specified in the test procedure file by issuing the set_gate_report command with the K19 or K22 argument while in setup mode.

Tip: Use set_gate_report with the K19 or K22 argument only when necessary. Because the tool has to log simulation data for all simulated setup and shift cycles, set_gate_report K19/K22 reporting can slow EDT DRC run time and increase memory usage compared to set_gate_report Drc_pattern reporting.

The following two subsections provide detailed discussion of the K19 and K22 DRCs, with debugging examples utilizing the Drc_pattern, K19 and K22 options to the set_gate_report command.

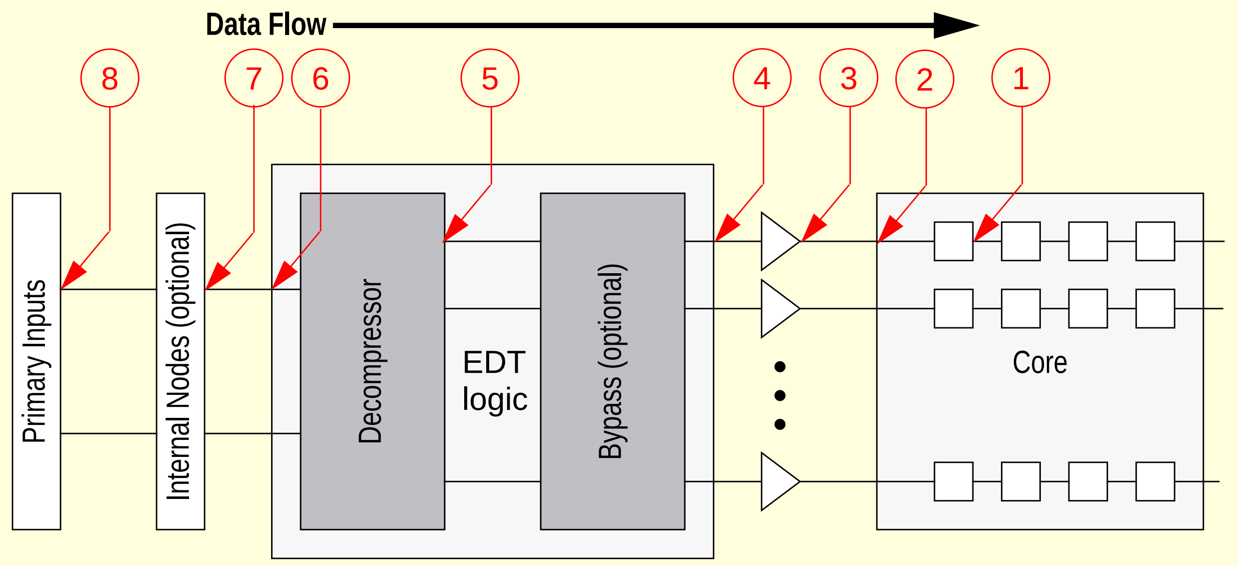
"Understanding K19 Rule Violations" on page 286
"Understanding K22 Rule Violations" on page 297

Understanding K19 Rule Violations

DRC K19 simulates the test_setup, load_unload, shift and capture procedures as defined in the test procedure file. By default, this simulation is performed with constrained pins initialized to their constrained values. To speed up simulation times, however, the rule simulates only a small number of shift cycles. If the first scan cell of each scan chain is loaded with the correct values, then the EDT decompressor works properly and this rule check passes.

If the first scan cell of any scan chain is loaded with incorrect data, the K19 rule check fails. The tool then automatically performs an initial diagnosis to determine where along the path from the channel inputs to the core chain inputs the problem originated. Figure C-2 shows the data flow through the decompressor and where in this flow the K19 rule check validates the signals.

Figure C-2. Order of Diagnostic Checks by the K19 DRC



1: Core chain <*index*> first cell
2: Core chain <*index*> input
3: Core chain <*index*> input driver
4: EDT module chain <*index*> input (source)
5: Decompressor chain <*index*> output
6: EDT module channel <*index*> input
7: Channel <*index*> input internal node
8: Channel <*index*> input pin

For example, if the K19 rule detected erroneous data at the output of the first scan cell (1) in scan chain 2, the rule would check whether data applied to the core chain input (2) is correct. If the data is correct at the core chain input, the tool would issue an error message similar to this:

```
Erroneous bit(s) detected at core chain 2 first cell
   /cpu_i/option_reg_2/DFF1/ (7021).
Data at core chain 2 input /cpu_i/edt_si2 (43) is correct.
   Expected: 0011101011101001X
   Simulated:01100110001110101
```

The error message reports the value the tool expected at the output of the first cell in scan chain 2 for each shift cycle. For comparison, the tool also lists the values that occurred during the

DRC’s simulation of the circuitry. If the data is correct at the first scan cell (1) and at the core chain inputs (2), the rule next checks the data at the outputs of the core chain input drivers (3).

Note

The term, “core chain input drivers” refers to any logic that drives the scan chain inputs. Usually, the core chain input drivers are part of the EDT logic. However, if a circuit designer inserts logic between the EDT logic and the core scan chain inputs, the drivers might be outside the EDT module.

The signals at (3) should always be the same as the signals at the core chain inputs (2). The tool checks that this is so, however, because the connection between these two points is emulated and not actually a physical connection. Figure 6-4 and the explanation accompanying it detail why the tool emulates this connection.

Note

Due to the tool’s emulation of the connection between points (2) and (3), you cannot obtain the gate names at these points by tracing between them with a “report_gates -backward” or “report_gates -forward” command. However, reporting a gate that has an emulated connection to another gate at this point will display the name and gate ID# of the other gate; you can then issue report_gates for the other gate and continue the trace from there.

If the data at the outputs of the core chain input drivers (3) is correct, the rule next checks the chain input data at the outputs of the EDT module (4). For each scan chain, if the data is correct at (4), but incorrect at the core chain input (2), the tool issues a message similar to the following:

```
Erroneous bit(s) detected at core chain 1 input /tiny_i/scan_in1 (11).
Data at EDT module chain 1 input (source) /edt_i/edt_bypass_logic_i/ix31/Y
   (216) is correct.
   Expected: 10011101011101001
   Simulated:10110011000111010
```

In this message, “EDT module chain 1 input (source)” refers to the output of the EDT module that drives the “core chain 1 input.” The word “source” indicates this is the pattern source for chain 1. Also, notice the gate name “/edt_i/edt_bypass_logic_i/ix31/Y” for the EDT module chain 1 input. Because the tool simulates the flattened netlist and does not model the hierarchical module pins, the tool reports the gate driving the EDT module output.

Note

The K19 and K22 rules always report gates driving EDT module inputs or outputs. Again, this is because in the flattened netlist there is no special gate that represents module pins.

The K19 rule verifies the data at the EDT module chain inputs (4) only if the EDT module hierarchy is preserved. If the netlist is flattened, or the EDT module name or pin names are changed during synthesis, the tool will no longer be able to identify the EDT module and its pins.

Tip: Preserving the EDT module during synthesis allows for better diagnostic messages if the simulation-based DRCs (K19 and K22) fail during the Pattern Generation Phase.

The K19 rule continues comparing the simulated data to what is expected for all eight locations shown in Figure C-2 until it finds a location where the simulated data matches the expected data. The tool then issues an error message that describes where the problem first occurred, and where the data was verified successfully.

This rule check not only reports erroneous data, but also reports unexpected X or Z values, as well as inverted signals. This information can be very useful when you are debugging the circuit.

Examples of some specific K19 problems, with suggestions for how to debug them, are detailed in the following sections:

Incorrect Control Signals
Inverted Signals
Incorrect EDT Channel Signal Order
Incorrect Scan Chain Order
X Generated by EDT Decompressor
Using set_gate_report K19

Incorrect Control Signals

Fixing incorrect values on EDT control signals often resolves other K19 violations. Problems with control signals may be detected by other K rules, so it is a good practice to check for these in the transcript prior to the K19 failure(s) and fix them first. At minimum, the other K rule failures may provide clues that help you solve the K19 issues.

If K19 detects incorrect values on an EDT control signal, the tool will issue a message similar to this one for the EDT bypass signal (edt_bypass by default):

```
1 EDT module control signals failed. (K19-1)
Inverted data detected at EDT module bypass /edt_bypass (37).
   Expected:  0000000000000000000000
   Simulated: 1111111111111111111111
```

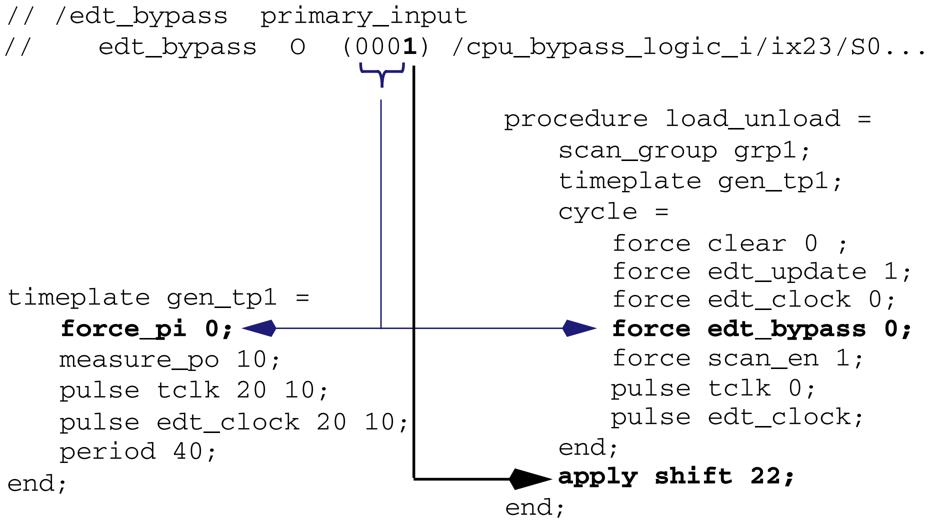
Because the edt_bypass signal is a primary input, and the message indicates it is at a constant incorrect value, it is reasonable to suspect that the load_unload or shift procedure in the test procedure file is applying an incorrect value to this pin. The edt_bypass signal should be 0 during load_unload and shift (see Figure 6-2), so you could use the following command sequence to check the pin's value after DRC.

1. set_gate_report drc_pattern load_unload
2. report_gates /edt_bypass
3. set_gate_report drc_pattern shift
4. report_gates /edt_bypass

The following transcript excerpt shows an example of the use of this command sequence, along with examples of procedures you would be examining for errors:

set_gate_report drc_pattern load_unload

report gate /edt_bypass

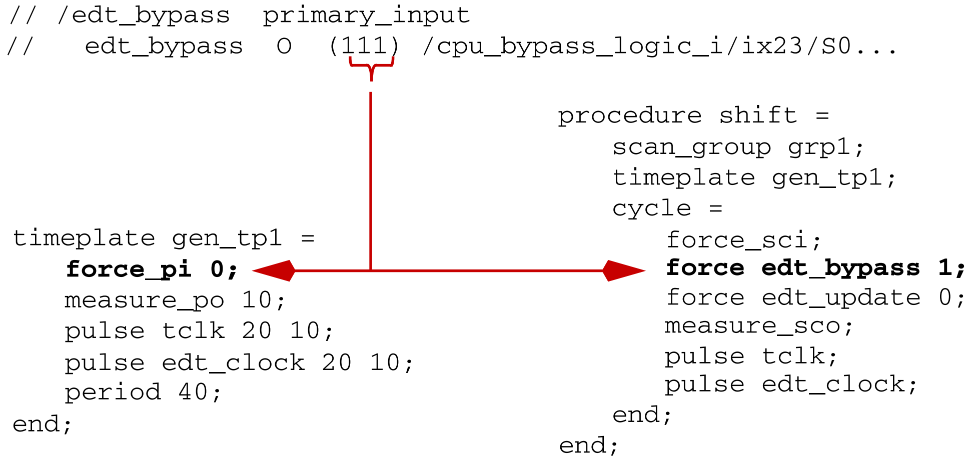


```
// /edt_bypass  primary_input
//    edt_bypass  O  (0001) /cpu_bypass_logic_i/ix23/S0...

                               procedure load_unload =
                                  scan_group grp1;
                                  timeplate gen_tp1;
                                  cycle =
                                     force clear 0 ;
                                     force edt_update 1;
timeplate gen_tp1 =                  force edt_clock 0;
   force_pi 0;                       force edt_bypass 0;
   measure_po 10;                    force scan_en 1;
   pulse tclk 20 10;                 pulse tclk 0;
   pulse edt_clock 20 10;            pulse edt_clock;
   period 40;                     end;
end;                              apply shift 22;
                               end;
```

The values reported for the load_unload are okay, but in the first “apply shift” (shown in bold font), edt_bypass is 1 when it should be 0. This points to the shift procedure as the source of the problem.

You can use the following commands to confirm:

set_gate_report drc_pattern shift
report_gate /edt_bypass



```
// /edt_bypass  primary_input
//   edt_bypass  O  (111) /cpu_bypass_logic_i/ix23/S0...
```

```
timeplate gen_tp1 =
    force_pi 0;
    measure_po 10;
    pulse tclk 20 10;
    pulse edt_clock 20 10;
    period 40;
end;
```

```
procedure shift =
    scan_group grp1;
    timeplate gen_tp1;
    cycle =
        force_sci;
        force edt_bypass 1;
        force edt_update 0;
        measure_sco;
        pulse tclk;
        pulse edt_clock;
    end;
end;
```

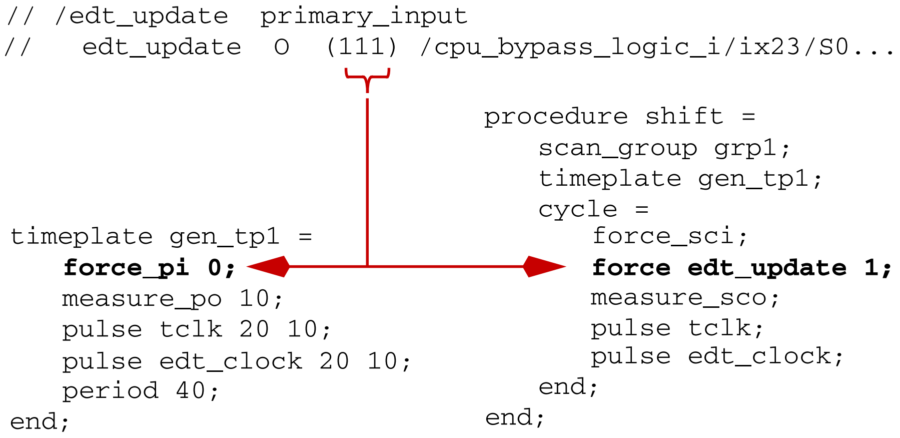
The DRC simulation data for the shift procedure shows it is forcing the edt_bypass signal to the wrong value (1 instead of 0). The remedy is to change the force statement to “force edt_bypass 0”.

Following is another example of the tool’s K19 messaging—for an incorrect value on the EDT update signal (highlighted in bold).

```
EDT update pin "edt_update" is not reset before pulse of EDT clock pin
   "edt_clock" in shift procedure. (K18-1)
1 error in test procedures. (K18)
...
1 EDT module control signals failed. (K19-1)
Inverted data detected at EDT module update /edt_update (36).
   Expected:  0000000000000000000000
   Simulated: 1111111111111111111111
4 of 4 EDT decompressor chain outputs (bus
   /cpu_edt_i/cpu_edt_decompressor_i/edt_scan_in) failed. (K19-2)
Erroneous bit(s) detected at EDT decompressor chain 1 output
   /cpu_edt_i/cpu_edt_decompressor_i/ix97/Y (282).
Data at EDT module channel inputs (signal /cpu_edt_i/edt_channels_in)
   is correct.
   Expected:  11010110111101010000 1X
   Simulated: 0000000000000000000000
...
```

Notice that earlier in the transcript there is a K18 message that mentions the same control signal and describes an error in the shift procedure. A glance at Figure 6-2 shows the EDT update signal should be 1 during load_unload and 0 for shift. You could now check the value of this signal as follows (relevant procedure file excerpts are shown below the example commands):

set_gate_report drc_pattern shift
report_gate /edt_update



```
// /edt_update  primary_input
//   edt_update  O  (111) /cpu_bypass_logic_i/ix23/S0...
```

```
procedure shift =
   scan_group grp1;
   timeplate gen_tp1;
   cycle =
      force_sci;
      force edt_update 1;
      measure_sco;
      pulse tclk;
      pulse edt_clock;
   end;
end;
```

```
timeplate gen_tp1 =
   force_pi 0;
   measure_po 10;
   pulse tclk 20 10;
   pulse edt_clock 20 10;
   period 40;
end;
```

The output of the gate report for the shift procedure shows the EDT update signal is 1 during shift. The reason is an incorrect force statement in the shift procedure, shown in the procedure excerpt below the example. Changing “force edt_update 1;” to “force edt_update 0;” in the shift procedure would resolve these K18 and K19 violations.

Inverted Signals

You can use inverting input pads to drive the EDT decompressor. However, you must specify the inversion using the set_edt_pins command. (This actually is true of any source of inversion added on the input side of the decompressor.) Without this information, the decompressor will generate incorrect data and the K19 rule check will transcript a message similar to this:

```
1 of 1 EDT module channel inputs (signal /cpu_edt_i/edt_channels_in)
   failed. (K19-1)
Inverted data detected at EDT module channel 1 input /U$1/Y (237).
Data at channel 1 input pin /edt_channels_in1 (38) is correct.
   Expected:  1000001011011000010000
   Simulated: 0111110100100111101111
```

The occurrence message lists the name and ID of the gate where the inversion was detected (point 6 in Figure C-2). It also lists the upstream gate where the data was correct (point 8 in Figure C-2). To debug, trace back from point 6 looking for the source of the inversion. For example:

report_gates /U$1/Y

```
// /U$1  inv02
//     A     I /edt_channels_in1
//     Y     O /cpu_edt_i/cpu_edt_decompressor_i/ix199/A1
               /cpu_edt_i/cpu_edt_decompressor_i/ix191/A1
               /cpu_edt_i/cpu_edt_decompressor_i/ix183/A1
```

b

```
// /edt_channels_in1 primary_input
//    edt_channels_in1   O   /U$1/Y
```

The trace shows there are no gates between the primary input where the data is correct and the gate (an inverter) where the inversion was detected, so the latter is the source of this K19 violation. You can use the -*Inv* switch with the set_edt_pins command to solve the problem.

report_edt_pins

```
//
// Pin description          Pin name          Inversion
// ---------------          --------          ---------
// Clock                    edt_clock             -
// Update                   edt_update            -
// Scan channel 1 input     edt_channels_in1      -
//  "       "     " output   edt_channels_out1     -
//
```

set_edt_pins input_channel 1 -inv
report edt pins

```
//
// Pin description          Pin name          Inversion
// ---------------          --------          ---------
// Clock                    edt_clock             -
// Update                   edt_update            -
// Scan channel 1 input     edt_channels_in1     inv
//  "       "     " output   edt_channels_out1     -
//
```

Incorrect EDT Channel Signal Order

If you manually connect the EDT module to the core scan chains, it is easy to connect signals in the wrong order. If the K19 rule check detects incorrectly ordered signals at any point, it issues messages similar to the following; notice the statement that signals appear to be connected in the wrong order:

```
2 of 2 EDT module channel inputs (bus /edt_i/edt_channels_in) failed.
   (K19-1)
Erroneous bit(s) detected at EDT module channel 1 input
   /edt_channels_in2 (9).
Data at channel 1 input pin /edt_channels_in1 (8) is correct.
   Expected:   010000000
   Simulated:  000000000
Erroneous bit(s) detected at EDT module channel 2 input
   /edt_channels_in1 (8).
Data at channel 2 input pin /edt_channels_in2 (9) is correct.
   Expected:   000000000
   Simulated:  010000000
2 signals appear to be connected in the wrong order at EDT module
   channel inputs (bus /edt_i/edt_channels_in). (K19-2)
Data at EDT module channel 2 input /edt_channels_in1 (8) match those
   expected at EDT module channel 1 input /edt_channels_in2 (9).
Data at EDT module channel 1 input /edt_channels_in2 (9) match those
   expected at EDT module channel 2 input /edt_channels_in1 (8).
```

DRC reports this as two K19 occurrences, but the same signals are mentioned in both occurrence messages. Notice also that the Expected and Simulated values are the same, but reversed for each signal, a corroborating clue. The fix is to reconnect the signals in the correct order in the netlist.

Incorrect Scan Chain Order

The tool enables you to add and delete scan chain definitions with the commands, add_scan_chains and delete_scan_chains. If you use these commands, it is mandatory that you keep the scan chains in exactly the same order in which they are connected to the EDT module: for example, the input of the scan chain added first must be connected to the least significant bit of the EDT module chain input port (point 4 in Figure C-2). Deleting a scan chain with the delete_scan_chains command, then adding it back again with add_scan_chains, will change the defined order of the scan chains, resulting in K19 violations. If scan chains are not added in the right order, the K19 rule check will issue a message similar to the following:

```
2 signals appear to be connected in the wrong order at core chain
inputs. Check if scan chains were added in the wrong order. (K19-2)
Data at core chain 6 input /cpu_i/edt_si6 (39)
   match those expected at core chain 5 input /cpu_i/edt_si5 (40).
Data at core chain 5 input /cpu_i/edt_si5 (40)
   match those expected at core chain 6 input /cpu_i/edt_si6 (39).
```

To check if scan chains were added in the wrong order, issue the report_scan_chains command and compare the displayed order with the order in the dofile the tool wrote out when the EDT logic was created. For example:

report_scan_chains

```
chain = chain1   group = grp1
   input = /cpu_i/scan_in1  output = /cpu_i/scan_out1  length = unknown
chain = chain2   group = grp1
   input = /cpu_i/scan_in2  output = /cpu_i/scan_out2  length = unknown
...
chain = chain6   group = grp1
   input = /cpu_i/scan_in6  output = /cpu_i/scan_out6  length = unknown
chain = chain5   group = grp1
   input = /cpu_i/scan_in5  output = /cpu_i/scan_out5  length = unknown
```

shows chains 5 and 6 reversed from the order in this excerpt of the original tool-generated dofile:

```
//
// Define the instance names of the decompressor, compactor, and the
// container module which instantiates the decompressor and compactor.
// Locating those instances in the design allows DRC to provide more debug
// information in the event of a violation.
// If multiple instances exist with the same name, subtitute the instance
// name of the container module with the instance's hierarchical path
// name.
```

```
set_edt_instances -edt_logic_top test_design_edt_i
set_edt_instances -decompressor  test_design_edt_decompressor_i
set_edt_instances -compactor     test_design_edt_compactor_i


add_scan_groups grp1 testproc
add_scan_chains -internal chain1 grp1 /cpu_i/scan_in1 /cpu_i/scan_out1
add_scan_chains -internal chain2 grp1 /cpu_i/scan_in2 /cpu_i/scan_out2
...
add_scan_chains -internal chain5 grp1 /cpu_i/scan_in5 /cpu_i/scan_out5
add_scan_chains -internal chain6 grp1 /cpu_i/scan_in6 /cpu_i/scan_out6
```

The easiest way to solve this problem is either to delete all scan chains and add them in the right order:

```
delete_scan_chains -all
add_scan_chains -internal chain1 grp1 /cpu_i/scan_in1 /cpu_i/scan_out1
add_scan_chains -internal chain2 grp1 /cpu_i/scan_in2 /cpu_i/scan_out2
...
add_scan_chains -internal chain5 grp1 /cpu_i/scan_in5 /cpu_i/scan_out5
add_scan_chains -internal chain6 grp1 /cpu_i/scan_in6 /cpu_i/scan_out6
```

or exit the tool, correct the order of add_scan_chains commands in the dofile and start the tool with the corrected dofile.

X Generated by EDT Decompressor

Xs should never be applied to the scan chain inputs. If this occurs, the K19 rule check issues a message similar to this:

```
X detected at EDT module chain 1 input (source)
   /edt_i/edt_bypass_logic_i/U86/Z (3303).
Data at EDT decompressor chain 1 output
   /edt_i/edt_decompressor_i/U83/Z (2727) is correct.
   Expected:  10100010000000010001
   Simulated: X0X000X00000000X000X
```

Provided the EDT module hierarchy is preserved, the message describes the origin of the X signals. The preceding message, for example, indicates the EDT bypass logic generates X signals, while the EDT decompressor works properly.

To debug these problems, check the following:

- Are the core chain inputs correctly connected to the EDT module chain input port? Floating core chain inputs could lead to an X.
- Are the channel inputs correctly connected to the EDT module channel input ports? Floating EDT module channel inputs could lead to an X.
- Are the EDT control signals (edt_clock, edt_update and edt_bypass by default) correctly connected to the EDT module? If the EDT decompressor is not reset properly, X signals might be generated.

- Is the EDT update signal (edt_update by default) asserted in the load_unload procedure so that the decompressor is reset? If the decompressor is not reset properly, X signals might be generated.
- Is the EDT bypass signal (edt_bypass by default) forced to 0 in the shift procedure? If the edt_bypass signal is not 0, X signals from un-initialized scan chains might be switched to the inputs of the core chains.
- If the EDT control signals are generated on chip (by means of a TAP controller, for example), are they forced to their proper values so the decompressor is reset in the load_unload procedure?

You can report the K19 simulation results for gates of interest by issuing “set_gate_report k19” in setup system mode, then using “report_gates” on the gates after the K19 rule check fails. You can also use an HDL simulator like ModelSim. In order to do that, ignore failing K19 DRCs by issuing a “set_drc_handling k19 ignore” command. Next, generate three random patterns in analysis system mode and save the patterns as serial Verilog patterns. Then simulate the circuit with an HDL simulator and analyze the signals of interest.

Using set_gate_report K19

If you issue a set_gate_report command with the K19 argument prior to DRC, you can use report_gates to view the simulated values for the entire sequence of events in the test procedure file for any K19-simulated gate. The K19 argument also has several options that enable you to limit the content of the displayed data.

Tip: Use set_gate_report with the K19 argument only when necessary. Because the tool has to log simulation data for all simulated setup and shift cycles, set_gate_report K19 reporting can slow EDT DRC run time and increase memory usage compared to set_gate_report Drc_pattern reporting.

The following shows how you might report on the simulated values for the “core chain 2 first cell” mentioned in the first error message example of this section (see “Understanding K19 Rule Violations” on page 286):

set_gate_report k19

```
// Warning: Data will be accessible after running DRC.
```

set_system_mode analysis

```
   ...
Erroneous bits detected at core chain 2 first cell
   /cpu_i/option_reg_2/DFF1/ (7021).
Data at core chain 2 input /cpu_i/edt_si2 (43) is correct.
   Expected: 0011101011101001X
   Simulated:01100110001110101
   ...
```

report_gates 7021

```
// /cpu_i/option_reg_2/DFF1 (7021) DFF
// "S"   I  50-
// "R"   I  46-
// CLK   I  1-/clk
// "D0"  I  1774-
// "OUT" O  52- 53-
//
// Proc: ts ld_u sh 1 sh 2 sh 3 sh 4 sh 5 sh 6... cap
// ----- -- ---- ---- ---- ---- ---- ---- ----     ---
// Time:  i  234  123  123  123  123  123  123... o o
//       n0 0000 0000 0000 0000 0000 0000 0000... fXf
// ----- -- ---- ---- ---- ---- ---- ---- ----     ---
//  Sim: XX XXXX XX00 0001 0011 0010 1110 1111... XXX
//  Emu: -- ---- ---0 ---0 ---1 ---1 ---1 ---0... ---
// Mism:                 *         *    *    *
// Monitor: core chain 1 first cell.
//
// Inputs:
// S     00 0000 0000 0000 0000 0000 0000 0000... 0X0
// R     00 0000 0000 0000 0000 0000 0000 0000... 0X0
// CLK   X0 0000 0010 0010 0010 0010 0010 0010... 0X0
// DO    XX XXXX XX00 0001 0011 0010 1110 1111... XXX
```

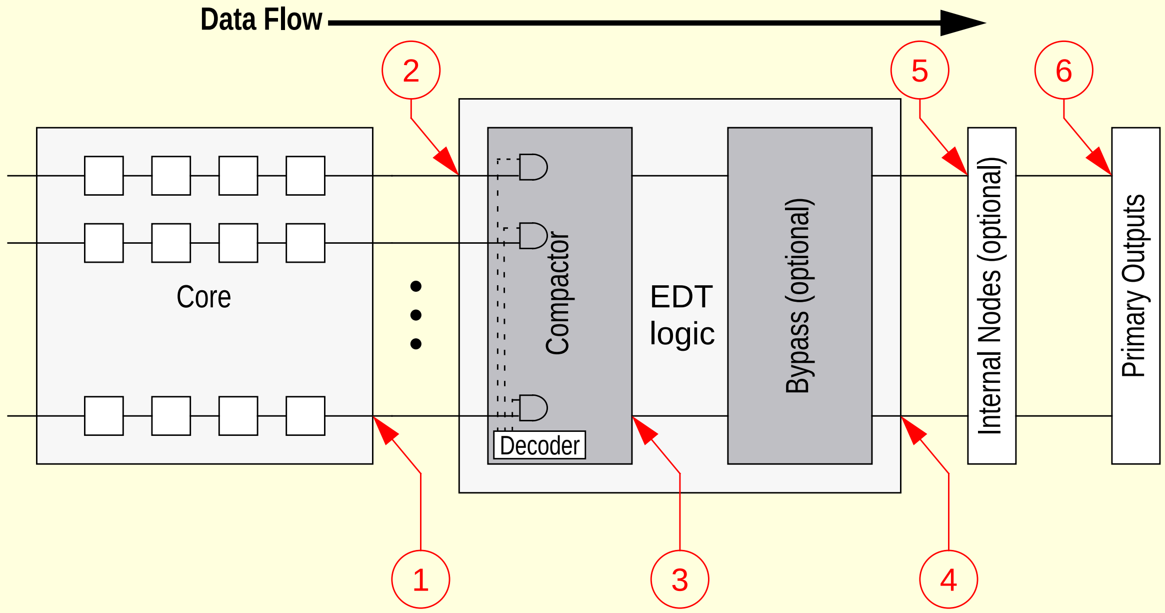
You can see from this report the effect each event in each shift cycle had on the gate's value during simulation. The time numbers (read vertically) indicate the relative time events occurred within each cycle, as determined from the procedure file. If the gate is used by DRC as a reference point in its automated analysis of K19 mismatches, the report lists the value the tool expected at the end of each cycle and whether it matched the simulated value. The last line reminds you the gate is a monitor gate (a reference point in its automated analysis) and tells you its location in the data path. These monitor points correspond to the eight points illustrated in Figure C-2.

Understanding K22 Rule Violations

Like DRC K19, the K22 rule check simulates the test_setup, load_unload and shift procedures, as defined in the test procedure file. But the K22 rule check performs more simulations than K19; one simulation in non-masking mode and a number of simulations in masking mode. If the correct values are shifted out of the channel outputs in both modes, then the EDT compactor works properly and this rule check passes.

If erroneous data is observed at any channel output, either in non-masking or masking mode, the K22 rule check fails. The tool then automatically performs an initial diagnosis to determine where along the path from the core scan chains to the channel outputs the problem originated. Figure C-3 shows the data flow through the compactor and where in this flow the K22 rule check validates the signals.

Figure C-3. Order of Diagnostic Checks by the K22 DRC



1: Core chain <*index*> output
2: EDT module chain <*index*> output (sink)
3: EDT compactor channel <*index*> output
4: EDT module channel <*index*> output
5: Channel <*index*> output internal node
6: Channel <*index*> output pin

For example, if the K22 rule detected erroneous data at the channel outputs (6), the tool would begin a search for the origin of the problem. First, it checks if the core chain outputs (1) have the correct values. If the data at (1) is correct, the tool next checks the data at the inputs of the EDT module (2). If the simulated data does not match the expected data here, the tool stops the diagnosis and issues a message similar to the following:

```
Error:Non-masking mode: 1 of 8 EDT module chain outputs (sink)
         (bus /edt_i/edt_scan_out) failed. (K22-1)
      Erroneous bit(s) detected at EDT module chain 3 output (sink)
         /cpu_i/stack2_reg_8/Q (1516).
      Data at core chain 3 output /cpu_i/edt_so3 (7233) is correct.
      Check if core chain 3 output is properly connected to EDT module
         chain 3 output (sink).
         Expected:  111101001101100100000000001000100000
         Simulated: 110100100101101000101011010111001111

Error:Masking mode (mask 3): 1 of 8 EDT module chain outputs (sink)
      (bus /edt_i/edt_scan_out) failed. (K22-2)
      Erroneous bit(s) detected at EDT module chain 3 output (sink)
         /cpu_i/stack2_reg_8/Q (1516).
      Data at core chain 3 output /cpu_i/edt_so3 (7233) is correct.
      Check if core chain 3 output is properly connected to EDT module
         chain 3 output (sink).
```

```
Expected:  110001001011000000000000000000110001
Simulated: 00011000111010000000001111001101100
```

In this message, “EDT module chain 3 output (sink)” refers to the input of the EDT module that is driven by the “core chain 3 output.” The word “sink” indicates this is the sink for the responses captured in chain 3. Also, notice the gate name “/cpu_i/stack2_reg_8/Q” for the EDT module chain 3 output. Because the tool simulates the flattened netlist and does not model hierarchical module pins, the tool reports the gate driving the EDT module’s input.

Note

The K19 and K22 rules always report_gates driving EDT module inputs or outputs. This is because in the flattened netlist there is no special gate that represents module pins.

The message has two parts; the first part reporting problems in non-masking mode, the second reporting problems in masking mode. The preceding example tells you the masking mode fails when the mask is set to 3; that is, when the third core chain is selected for observation.

Note

In masking mode, only one core chain per compactor group is observed at the channel output for the group. In non-masking mode, the output from all core chains in a compactor group are compacted and observed at the channel output for the group.

Given the error message, it is easy to debug the problem. Check the connection between the core chain output (1 in Figure C-3) and the EDT module, making sure any logic in between is controlled correctly. Usually, there is no logic between the core chain outputs and the EDT module.

The K22 rule verifies data at the EDT module chain outputs (2) only if the EDT module hierarchy is preserved. If the netlist is flattened or the EDT module’s name or pin names are changed during synthesis, the tool will no longer be able to identify the EDT module and its pins.

Note

Preserving the EDT module during synthesis allows for better diagnostic messages if the simulation-based DRCs (K19 and K22) fail during the Pattern Generation Phase.

If the data at the EDT module chain outputs (2) is correct, the K22 rule continues comparing the simulated data to the expected data for the EDT compactor outputs (3), the EDT module channel outputs(4), and so on until the tool identifies the source of the problem. This approach is analogous to that used for the K19 rule checks described in the section, “Understanding K19 Rule Violations” on page 286.

For guidance on methods of debugging incorrect or inverted signals, X signals, and signals or scan chains in the wrong order, the discussion of these topics in the section, “Understanding

K19 Rule Violations," is good background information for K22 rule violations. Examples of some specific K19 problems, with example debugging steps, are detailed in these sections:

Incorrect Control Signals
Inverted Signals
Incorrect EDT Channel Signal Order
Incorrect Scan Chain Order
X Generated by EDT Decompressor

Some specific K22 problems, with example debugging steps, are detailed in the following sections:

Inverted Signals
Incorrect Scan Chain Order
Masking Problems
Using set_gate_report K22

Inverted Signals

You can use inverting pads on EDT channel outputs. However, you must specify the inversion using the set_edt_pins command. (This actually is true of any source of inversion added on the output side of the compactor.) Without this information, the compactor will generate incorrect data and the K22 rule check will transcript a message similar to this (for a design with one scan channel and four core scan chains):

```
Non-masking mode: 1 of 1 channel output pins failed. (K22-1)
Inverted data detected at channel 1 output pin /edt_channels_out1 (564).
Data at EDT module channel 1 output /cpu_edt_i/edt_bypass_logic_i/ix23/Y
   (458) is correct.
   Expected:  X000001101110000100111
   Simulated: X111110010001111011000

Masking mode (mask 1): 1 of 1 channel output pins failed. (K22-2)
Inverted data detected at channel 1 output pin /edt_channels_out1 (564).
Data at EDT module channel 1 output /cpu_edt_i/edt_bypass_logic_i/ix23/Y
   (458) is correct.
   Expected:  X111101001010010011001
   Simulated: X000010110101101100110

Masking mode (mask 2): 1 of 1 channel output pins failed. (K22-3)
Inverted data detected at channel 1 output pin /edt_channels_out1 (564).
Data at EDT module channel 1 output /cpu_edt_i/edt_bypass_logic_i/ix23/Y
   (458) is correct.
   Expected:  X111111110000000010010
   Simulated: X000000001111111101101

Masking mode (mask 3): 1 of 1 channel output pins failed. (K22-4)
Inverted data detected at channel 1 output pin /edt_channels_out1 (564).
Data at EDT module channel 1 output /cpu_edt_i/edt_bypass_logic_i/ix23/Y
   (458) is correct.
   Expected:  X010001010000110011101
   Simulated: X101110101111001100010
```

```
Masking mode (mask 4): 1 of 1 channel output pins failed. (K22-5)
Inverted data detected at channel 1 output pin /edt_channels_out1 (564).
Data at EDT module channel 1 output /cpu_edt_i/edt_bypass_logic_i/ix23/Y
   (458) is correct.
   Expected:  X110101011110011101110
   Simulated: X001010100001100010001
```

Notice the separate occurrence messages are identifying the same problem.

The occurrence messages list the name and ID of the gate where the inversion was detected (point 6 in Figure C-3). It also lists the upstream gate where the data was correct (point 4 in Figure C-3). To debug, simply trace back from point 6 looking for the source of the inversion. For example:

report_gates /edt_channels_out1

```
//  /edt_channels_out1  primary_output
//      edt_channels_out1  I  /ix77/Y
```

b

```
//  /ix77  inv02
//      A      I  /cpu_edt_i/edt_bypass_logic_i/ix23/Y
//      Y      O  /edt_channels_out1
```

The trace shows there are no gates between the primary output where the inversion was detected and the gate (an inverter) where the data is correct, so the latter is the source of this K22 violation. You can use the -Inv switch with the set_edt_pins command to solve the problem.

report_edt_pins

```
//
// Pin description          Pin name          Inversion
// ---------------          --------          ---------
// Clock                    edt_clock             -
// Update                   edt_update            -
// Scan channel 1 input     edt_channels_in1      -
//   "       "    " output  edt_channels_out1     -
//
```

set_edt_pins output_channel 1 -inv
report edt pins

```
//
// Pin description          Pin name          Inversion
// ---------------          --------          ---------
// Clock                    edt_clock             -
// Update                   edt_update            -
// Scan channel 1 input     edt_channels_in1      -
//   "       "    " output  edt_channels_out1    inv
//
```

Incorrect Scan Chain Order

You can add and delete scan chain definitions with the commands add_scan_chains and delete_scan_chains. If you use these commands, it is mandatory that you keep the scan chains in exactly the same order in which they are connected to the EDT module: for example, the output of the scan chain added first must be connected to the least significant bit of the EDT module chain output port (point 2 in Figure C-3). Deleting a scan chain with the delete_scan_chains command, then adding it again with add_scan_chains, will change the defined order of the scan chains, resulting in K22 violations. If scan chains are not added in the right order, the K22 rule check will issue a message similar to the following:

```
4 signals appear to be connected in the wrong order at EDT module chain
outputs (sink) (bus/cpu_edt_i/edt_so). (K22-8)
Data at EDT module chain 2 output (sink) /cpu_i/datai/uu1/Y (254)
   match those expected at EDT module chain 1 output (sink)
   /cpu_i/datao/uu1/Y (256).
Data at EDT module chain 3 output (sink) /cpu_i/datai1/uu1/Y (253)
   match those expected at EDT module chain 2 output (sink)
   /cpu_i/datai/uu1/Y (254).
Data at EDT module chain 4 output (sink) /cpu_i/addr_0/uu1/Y (245)
   match those expected at EDT module chain 3 output (sink)
   /cpu_i/datai1/uu1/Y (253).
Data at EDT module chain 1 output (sink) /cpu_i/datao/uu1/Y (256)
   match those expected at EDT module chain 4 output (sink)
   /cpu_i/addr_0/uu1/Y (245).
```

To check if scan chains were added in the wrong order, issue the report_scan_chains command and compare the displayed order with the order in the dofile the tool wrote out when the EDT logic was created. For example:

report_scan_chains

```
chain = chain2 group = grp1
   input = /cpu_i/scan_in2 output = /cpu_i/scan_out2 length = unknown
chain = chain3 group = grp1
   input = /cpu_i/scan_in3 output = /cpu_i/scan_out3 length = unknown
chain = chain4 group = grp1
   input = /cpu_i/scan_in4 output = /cpu_i/scan_out4 length = unknown
chain = chain1 group = grp1
   input = /cpu_i/scan_in1 output = /cpu_i/scan_out1 length = unknown
```

shows chain1 added last instead of first, chain2 added first instead of second, and so on; not the order in this excerpt of the original tool-generated dofile:

```
//
// Define the instance names of the decompressor, compactor, and the
// container module which instantiates the decompressor and compactor.
// Locating those instances in the design allows DRC to provide more debug
// information in the event of a violation.
// If multiple instances exist with the same name, subtitute the instance
// name of the container module with the instance’s hierarchical path
// name.

set_edt_instances -edt_logic_top test_design_edt_i
set_edt_instances -decompressor  test_design_edt_decompressor_i
set_edt_instances -compactor     test_design_edt_compactor_i

add_scan_groups grp1 testproc
add_scan_chains -internal chain1 grp1 /cpu_i/scan_in1 /cpu_i/scan_out1
add_scan_chains -internal chain2 grp1 /cpu_i/scan_in2 /cpu_i/scan_out2
add_scan_chains -internal chain3 grp1 /cpu_i/scan_in3 /cpu_i/scan_out3
add_scan_chains -internal chain4 grp1 /cpu_i/scan_in4 /cpu_i/scan_out4
...
```

The easiest way to solve this problem is either to delete all scan chains and add them in the right order:

```
delete_scan_chains -all
add_scan_chains -internal chain1 grp1 /cpu_i/scan_in1 /cpu_i/scan_out1
add_scan_chains -internal chain2 grp1 /cpu_i/scan_in2 /cpu_i/scan_out2
add_scan_chains -internal chain3 grp1 /cpu_i/scan_in3 /cpu_i/scan_out3
add_scan_chains -internal chain4 grp1 /cpu_i/scan_in4 /cpu_i/scan_out4
```

or exit the tool, correct the order of add_scan_chains commands in the dofile and start the tool with the corrected dofile.

Note

When the tool is set up to treat K19 violations as errors, the invocation default, incorrect scan chain order will be detected by the K19 rule check, since the tool performs K19 checks before K22. (See "Incorrect Scan Chain Order" in the K19 section for example tool messages). In this case, the tool will stop before issuing any K22 messages related to the incorrect order.

If the issue was actually one of incorrect signal order only at the outputs of the internal scan chains and the inputs were in the correct order, you would get K22 messages similar to the preceding and no K19 messages about scan chains being "added in the wrong order."

Masking Problems

Most masking problems are caused by disturbances in the operation of the mask hold and shift registers. One such problem results in the following message for the decoded masking signals:

```
Non-masking mode: 4 of 4 EDT decoded masking signals failed. (K22-1)
Constant X detected at EDT decoded masking signal 1
   /cpu_edt_i/cpu_edt_compactor_i/decoder1/ix63/Y (343).
   Expected:  1111111111111111111111
   Simulated: XXXXXXXXXXXXXXXXXXXXXX
```

You can usually find the source of masking problems by analyzing the mask hold and shift registers. In this example, you could begin by tracing back to find the source of the Xs:

```
set_gate_level primitive
set_gate_report drc_pattern state_stability
report_gates /cpu_edt_i/cpu_edt_compactor_i/decoder1/ix63/Y
```

```
// /cpu_edt_i/cpu_edt_compactor_i/decoder1/ix63 (343)  NAND
//             (ts)( ld)(shift)(cap)(stbl)
//   "I0"   I  ( X)(XXX)(XXX~X)(XXX)(   X) 294-
//   B0     I  ( X)(XXX)(XXX~X)(XXX)(   X) 291- ../decoder1/ix107/Y
//   Y      O  ( X)(XXX)(XXX~X)(XXX)(   X) 419- ../ix41/A1
```

b

```
// /cpu_edt_i/cpu_edt_compactor_i/decoder1/ix63 (294)  OR
//             (ts)( ld)(shift)(cap)(stbl)
//   A0     I  ( X)(XXX)(XXX~X)(XXX)(   X) 208- ../reg_masks_hold_reg_0_/Q
//   A1     I  ( X)(XXX)(XXX~X)(XXX)(   X) 214- ../reg_masks_hold_reg_1_/Q
//   "OUT"  O  ( X)(XXX)(XXX~X)(XXX)(   X) 343-
```

b

```
// /cpu_edt_i/cpu_edt_compactor_i/reg_masks_hold_reg_0_ (208)  BUF
//             (ts)( ld)(shift)(cap)(stbl)
//   "I0"   I  ( X)(XXX)(XXX~X)(XXX)(   X) 538-
//   Q      O  ( X)(XXX)(XXX~X)(XXX)(   X) 235- ../ix102/A0
//                                         292- ../decoder1/ix57/A0
//                                         293- ../decoder1/ix113/A
//                                         346- ../decoder1/ix61/A0
//                                         294- ../decoder1/ix63/A0
```

b

```
// /cpu_edt_i/cpu_edt_compactor_i/reg_masks_hold_reg_0_ (538)  DFF
//             (ts)( ld)(shift)(cap)(stbl)
//   "S"    I  ( 0)(000)(000~0)(000)(   0) 48-
//   "R"    I  ( 0)(000)(000~0)(000)(   0) 150-
//   CLK    I  ( 0)(000)(000~0)(000)(   0) 47-
//   D      I  ( X)(XXX)(XXX~X)(XXX)(   X) 235- ../ix102/Y
//   "OUT"  O  ( X)(XXX)(XXX~X)(XXX)(   X) 208- 209-
```

The trace shows the clock for the mask hold register is inactive. Trace back on the clock to find out why:

report_gates 47

```
// /cpu_edt_i (47)  TIE0
//           (ts)( ld)(shift)(cap)(stbl)
// "OUT" O ( 0)(000)(000~0)(000)(   0) 541-../reg_masks_hold_reg_1_/CLK
//                                     540-../reg_masks_shift_reg_1_/CLK
//                                     539-../reg_masks_shift_reg_0_/CLK
//                                     538-../reg_masks_hold_reg_0_/CLK
//                                     537 ../reg_masks_shift_reg_2_/CLK
//                                     536-../reg_masks_hold_reg_2_/CLK
```

The information for the clock source shows it is tied. As the EDT clock should be connected to the hold register, you could next report on the EDT clock primary input at the compactor and check for a connection to the hold register:

report_gates /cpu_edt_i/cpu_edt_compactor_i/edt_clock
. . .

Based on the preceding traces, you would expect to find that the EDT clock was *not* connected to the hold register. Because an inactive clock signal to the mask hold register would cause masking to fail, check the transcript for corroborating messages that indicate multiple similar masking failures. These DRC messages, which preceded the K22 message in this example, provide such a clue:

```
Pipeline identification for channel output pins failed. (K20-1)
Non-masking mode: Failed to identify pipeline stage(s) at channel 1 output
   pin /edt_channels_out1 (563).
Masking mode (mask 1, chain1): Failed to identify pipeline stage(s) at
   channel 1 output pin /edt_channels_out1 (563).
Masking mode (mask 2, chain2): Failed to identify pipeline stage(s) at
   channel 1 output pin /edt_channels_out1 (563).
Masking mode (mask 3, chain3): Failed to identify pipeline stage(s) at
   channel 1 output pin /edt_channels_out1 (563).
Masking mode (mask 4, chain4): Failed to identify pipeline stage(s) at
   channel 1 output pin /edt_channels_out1 (563).

Error during identification of pipeline stages. (K20)
Rule K21 (lockup cells) not performed for the compactor side since
   pipeline identification failed.
```

Notice the same failure was reported in masking mode for all scan chains. To fix this particular problem, you would need to connect the EDT clock to the mask hold register in the netlist.

Using set_gate_report K22

The set_gate_report command has a K22 argument similar to the K19 argument described in "Using set_gate_report K19" on page 296. If you issue the command prior to DRC, you can use "report_gates" to view the simulated values for the entire sequence of events in the test procedure file for any K22-simulated gate. Like the K19 argument, the K22 argument also has several options that enable you to limit the content of the displayed data.

Tip: Use set_gate_report with the K22 argument only when necessary. Because the tool has to log simulation data for all simulated setup and shift cycles, "set_gate_report k22" reporting can slow EDT DRC run time and increase memory usage compared to "set_gate_report drc_pattern" reporting.

Miscellaneous

Incorrect References in Synthesized Netlist

Use the information in this section troubleshoot problems that cause Design Compiler to insert **TSGEN** references in a synthesized netlist.

Run Design Compiler to synthesize the netlist and verify that no errors occurred and check that tri-state buffers were correctly synthesized. For certain technologies, Design Compiler is unable to correctly synthesize tri-state buffers and inserts an incorrect reference to “**TSGEN**” instead. You can run the UNIX **grep** command to check for TSGEN:

```
grep TSGEN created_edt_bs_top_gate.v
```

If TSGEN is found, as shown in bold font in the following example Verilog code,

```
module tri_enable_high ( dout, oe, pin );
input dout, oe;
output pin;
   wire pin_tri_enable;
   tri pin_wire;
   assign pin = pin_wire;
   \**TSGEN** pin_tri ( .\function (dout),
      .three_state(pin_tri_enable), .\output (pin_wire) );
   N1L U16 ( .Z(pin_tri_enable), .A(oe) );
endmodule
```

you need to change the line of code that contains the reference to a correct instantiation of a tri-state buffer. The next example corrects the previous instantiation to the LSI lcbg10p technology (shown in bold font):

```
module tri_enable_high ( dout, oe, pin );
input dout, oe;
output pin;
   wire pin_tri_enable;
   tri pin_wire;
   assign pin = pin_wire;
   BTS4A pin_tri ( .A (dout), .E (pin_tri_enable), .Z
      (pin_wire) );
   N1A U16 ( .Z(pin_tri_enable), .A(oe) );
endmodule
```

Limiting Observable Xs for a Compact Pattern Set

EDT can handle Xs, but you may want to limit them in order to enhance compression. To achieve a compact pattern set (and decrease runtime as well), ensure the circuit has few, or no, X generators that are observable on the scan chains. For example, if you bypass a RAM that is tested by memory BIST, X sources are reduced because the RAM will no longer be an X generator in analysis mode.

If no Xs are captured on the scan chains, usually no fault effects are lost due to the compactors and the tool does not have to generate patterns that use scan chain output masking. For circuits with no Xs observable on the scan chains, the effective compression is usually much higher (everything else being equal) and the number of patterns is only slightly more than what ATPG generates without EDT. DRC’s rule E5 identifies sources of observable Xs.

One clue you probably have many observable Xs is usually apparent in the transcript for an EDT pattern generation run. With few or no observable Xs, the number of effective patterns in each simulation pass without scan chain masking will (ideally) be 32 for 32-bit invocations and 64 for 64-bit invocations. Numbers significantly lower indicate Xs are reducing test effectiveness. This is confirmed if the number of effective patterns rises significantly when the tool uses masking to block the observable Xs. This is shown in the following excerpt from the transcript for a 32-bit invocation run:

```
//  #patterns     test     #faults     #faults     # eff.      # test
//  simulated     cvrg     in list     detected    patterns    patterns
//  deterministic ATPG invoked with abort limit 30
//  EDT without scan masking. Dynamic compaction enabled.
...
//   ---        ------       ---         ---         ---         ---
//   608        61.51%      3301          58          17          93
//   ---        ------       ---         ---         ---         ---
//   640        63.17%      3249          52          14         107
//   ---        ------       ---         ---         ---         ---
//   672        65.63%      3211          38          18         125
//   ---        ------       ---         ---         ---         ---
//  Warning: Unsuccessful test for 972 faults.
//  deterministic ATPG invoked with abort limit = 30
//  EDT with scan masking. Dynamic compaction disabled.
//   736        82.06%      2007         638          32         157
//   ---        ------       ---         ---         ---         ---
//   768        84.42%      1638         369          32         189
//   ---        ------       ---         ---         ---         ---
//   800        86.16%      1221         417          32         221
//   ---        ------       ---         ---         ---         ---
...
```

Applying Uncompressable Patterns Thru Bypass Mode

Occasionally, the tool will generate an effective pattern that cannot be compressed using EDT technology. Although it is a rare occurrence, if many faults generate such patterns, it can have an impact on test coverage. Decreasing the number of scan chains usually remedies the problem. Alternatively, you can bypass the EDT logic, which reconfigures the scan chains into fewer, longer scan chains. This requires an uncompressed ATPG run on the remaining faults.

Note

You can use bypass mode to apply uncompressed patterns. You can also use bypass mode for system debugging purposes.

If Compression is Less Than Expected

If you find effective compression is much less than you targeted, taking steps to remedy or reduce the following should improve the compression:

- Many observable Xs—EDT can handle observable Xs but their occurrence requires the tool to use masking patterns. Masking patterns observe fewer faults than non-masking patterns, so more of them are required. More patterns lowers effective compression.

  If the session transcript shows all patterns are non-masking, then observable Xs are not the cause of the lower than expected compression. If the tool generated both masking and non-masking patterns and the percentage of masking patterns exceeds 25% of the total, then there are probably many observable Xs. To find them, look for E5 DRC messages. You activate E5 messages by issuing a "set_drc_handling e5 note" command.

Note

If there are many observable Xs, you will probably see a much higher runtime compared to uncompressed ATPG. You will probably also see a much lower number of effective patterns reported in the transcript when compressed ATPG is not using scan chain masking, compared to when the tool is using masking.

The Chapter 7 section, "Resolving X Blocking with Scan Chain Masking," describes masking patterns. It also shows how the tool reports their use in the session transcript, and illustrates how masked patterns appear in an ASCII pattern file. See also "Limiting Observable Xs for a Compact Pattern Set" earlier in this chapter.

- EDT Aborted Faults—For information about these types of faults, refer to "If there are EDT aborted faults" in the next section.
- If there are no EDT aborted faults, try a more aggressive compression configuration by increasing the number of scan chains.

If Test Coverage is Less Than Expected

If you find test coverage is much less than you expected, first compare it to the test coverage obtainable without EDT. If the test coverage with EDT is less than you obtain with uncompressed ATPG, the following sections list steps you can take to raise it to the same level as uncompressed ATPG:

If there are EDT aborted faults

When the tool generates an effective fault test, but is unable to compress the pattern, the fault is classified as an EDT aborted fault.

A warning is issued at the end of the run for EDT aborted faults and reports the resultant loss of coverage. You can also obtain this information by issuing the report_aborted_faults command and looking for the "edt" class of aborted faults. Each of the following increases the probability of EDT aborted faults:

- Relatively aggressive compression (large chain-to-channel ratio)
- Large number of ATPG constraints
- Relatively small design

If the number of undetected faults is large enough to cause a relevant decrease of test coverage, try re-inserting a fewer number of scan chains.

Internal Scan Chain Pins Incorrectly Shared with Functional Pins

Relatively low test coverage can indicate internal scan chain pins are shared with functional pins. These pins must not be shared because the internal scan chain pins are connected to the EDT logic and not to the top level. Also, the tool constrains internal scan chain input pins to X, and masks internal scan chain output pins. This has minimal impact on test coverage only if these are dedicated pins. By default, DRC issues a warning if scan chain pins are not dedicated pins.

Be sure none of the internal scan chain input or output pins are shared with functional pins. Only scan *channel* pins may be shared with functional pins. Refer to "Scan Chain Pins" on page 41 for additional information.

Masking Broken Scan Chains in the EDT Logic

You can set up the EDT logic to mask the load, capture, and/or unload values on specified scan chains by inserting custom logic between the scan chain outputs and the compactor. The custom logic allows you to either feed the desired circuit response (0/1) to the compactor or tie the scan chain output to an unknown value (X). For more information, see the add_chain_masks command.

Index

Third-Party Information

For information about third-party software included with this release of Tessent products, refer to the *Third-Party Software for Tessent Products*.

End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/eula

IMPORTANT INFORMATION

USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Software received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. **ORDERS, FEES AND PAYMENT.**

1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement ("Order(s)"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not these documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order or presented via any electronic portal or other automated order management system will not be effective unless agreed in writing by an authorized representative of Customer and Mentor Graphics.

1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice(s). Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.

1.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 5.2); (b) for Customer's internal business purposes; (c) for the term of the license; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if Customer provides any feedback or requests any change or enhancement to Products,

whether in the course of receiving support or consulting services, evaluating Products, performing beta testing or otherwise, any inventions, product improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.

3. **ESC SOFTWARE.** If Customer purchases a license to use development or prototyping tools of Mentor Graphics' Embedded Software Channel ("ESC"), Mentor Graphics grants to Customer a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESC compilers, including the ESC run-time libraries distributed with ESC C and C++ compiler Software that are linked into a composite program as an integral part of Customer's compiled computer program, provided that Customer distributes these files only in conjunction with Customer's compiled computer program. Mentor Graphics does NOT grant Customer any right to duplicate, incorporate or embed copies of Mentor Graphics' real-time operating systems or other embedded software products into Customer's products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.

4. **BETA CODE.**

4.1. Portions or all of certain Software may contain code for experimental testing and evaluation (which may be either alpha or beta, collectively "Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and Customer's use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.

4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 4.3 shall survive termination of this Agreement.

5. **RESTRICTIONS ON USE.**

5.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer becomes aware of such unauthorized disclosure or use. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any source code from Software. Log files, data files, rule files and script files generated by or for the Software (collectively "Files"), including without limitation files containing Standard Verification Rule Format ("SVRF") and Tcl Verification Format ("TVF") which are Mentor Graphics' proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Software or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Software, or disclose to any third party the results of, or information pertaining to, any benchmark.

5.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer's employees or on-site contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.

5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.

5.4. The provisions of this Section 5 shall survive the termination of this Agreement.

6. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer with updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics' then current End-User Support Terms located at http://supportnet.mentor.com/about/legal/.

7. **LIMITED WARRANTY.**

7.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification, improper installation or Customer is not in compliance with this Agreement. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

7.2. THE WARRANTIES SET FORTH IN THIS SECTION 7 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

8. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

9. **HAZARDOUS APPLICATIONS.** CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING ITS PRODUCTS USED IN APPLICATIONS WHERE THE FAILURE OR INACCURACY OF ITS PRODUCTS MIGHT RESULT IN DEATH OR PERSONAL INJURY ("HAZARDOUS APPLICATIONS"). EXCEPT TO THE EXTENT THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF MENTOR GRAPHICS PRODUCTS IN OR FOR HAZARDOUS APPLICATIONS. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

10. **INDEMNIFICATION.** CUSTOMER AGREES TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH THE USE OF MENTOR GRAPHICS PRODUCTS IN OR FOR HAZARDOUS APPLICATIONS. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

11. **INFRINGEMENT.**

11.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to such action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

11.2. If a claim is made under Subsection 11.1 Mentor Graphics may, at its option and expense: (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.

11.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by Customer that is deemed willful. In the case of (h), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.

11.4. THIS SECTION 11 IS SUBJECT TO SECTION 8 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, FOR DEFENSE, SETTLEMENT AND DAMAGES, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

12. **TERMINATION AND EFFECT OF TERMINATION.**

12.1. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.

12.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.

13. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and United States ("U.S.") government agencies, which prohibit export, re-export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export or re-export Products in any manner without first obtaining all necessary approval from appropriate local and U.S. government agencies. If Customer wishes to disclose any information to Mentor Graphics that is subject to any U.S. or other applicable export restrictions, including without limitation the U.S. International Traffic in Arms Regulations (ITAR) or special controls under the Export Administration Regulations (EAR), Customer will notify Mentor Graphics personnel, in advance of each instance of disclosure, that such information is subject to such export restrictions.

14. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. The parties agree that all Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to U.S. FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. government or a U.S. government subcontractor is subject solely to the terms and conditions set forth in this Agreement, which shall supersede any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.

15. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.

16. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 16 shall survive the termination of this Agreement.

17. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the U.S. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, U.S., if Customer is located in North or South America, and the laws of Ireland if

Customer is located outside of North or South America. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply. Notwithstanding the foregoing, all disputes in Asia arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

18. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

19. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements. Some Software may contain code distributed under a third party license agreement that may provide additional rights to Customer. Please see the applicable Software documentation for details. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

Rev. 130502, Part No. 255853