Tessent® Shell User's Manual

Software Version 2014.2

June 2014

This manual is part of a fully-indexed Tessent documentation set. To search across all Tessent manuals, click on the "binocular" icon or press Shift-Ctrl-F. Note that this index is not available if you are viewing this PDF in a web browser.

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

List of Figures

List of Tables

Chapter 1
Tessent Shell Introduction

Tessent Shell is a Tcl shell environment and design data model that provides a unified Tcl command set and command naming convention.

Tessent Shell Overview

Tessent Shell enables you to manipulate and to query your design data.

Using Tessent Shell, you can perform the following operations on your design:

- **Design editing** — Provides a robust set of design editing commands that you can use to manipulate the design's modules, instances, nets, ports, and pins, either interactively or through Tcl scripting.
- **Hierarchical design introspection** — Intuitively named "get_" commands return collections of objects from the design data model and the model's built-in attributes that you can use for design introspection of various design objects.
- **Setting and reading attributes** — Provides a set of commands to allow for creating, manipulating, and querying attributes on the design objects.
- **ATPG functions** — Provides features for creating patterns, and performing good and fault simulation. This includes all the functionality in Tessent FastScan and Tessent TestKompress, including EDT (Embedded Deterministic Testing) IP creation.

- **IJTAG support** — Provides features for PDL (Procedural Description Language) command retargeting for IJTAG (IEEE P1687 standard) plus extraction of the ICL network from the design to create the ICL for the design.
- **Scan analysis and insertion** — Provides features for scan analysis and scan chain insertion. This includes all of the functionality in Tessent Scan.
- **Scan pattern retargeting support** — Provides features for retargeting core-level test patterns at the top level.
- **Diagnosis** — Provides test failure diagnosis to determine a defect's most probable failure mechanism, logic location, and physical location. Uses failure data from manufacturing test, scan test patterns, and design information. This includes all of the functionality in Tessent Diagnosis.

Tool Invocation

You can invoke Tessent Shell from a Linux shell. The tool's system mode defaults to "setup" after invocation.

You invoke Tessent Shell from a Linux shell using the following syntax:

```
% tessent -shell
```

To use most Tessent Shell functionality, you must load a cell library after invocation, which you can do with the read_cell_library command.

Refer to the tessent shell command description in the *Tessent Shell Reference Manual* for additional invocation options.

Tessent Startup File

Tessent Shell supports the use of the *.tessent_startup* startup file.

This startup file is common to the contexts listed in Table 1-2 on page 14, which correspond to the following products: Tessent FastScan, Tessent TestKompress, Tessent Scan, Tessent Diagnosis, Tessent IJTAG, and Tessent SiliconInsight.

You can use this startup file for both general and tool-specific settings. Tessent Shell reads the startup file during invocation. The default location for the startup file is the home directory as shown here: $HOME/*.tessent_startup*.

Application-Specific Environment Variables

Tessent Shell provides some environment variables that specifically affect the environment in which Tessent applications operate. During invocation, the tool reads any application-specific environment variables that are set.

Table 1-1 lists the environment variables specific to Tessent Shell. For information about all other Tessent application environment variables, see the *Managing Mentor Graphics Tessent Software* manual.

Table 1-1. Application-Specific Environment Variables

Variable	Default Value and Description
TESSENT_STARTUP	Default: No default The pathname of a directory that contains a Tessent tool startup file.
TESSENT_TMP_LOCATION	Specifies the location at which the tool creates the *.tessent.tmp.hostname.process_id* directory. The tool creates the scratch directory at *.tessent.tmp.hostname.process_id.*
TESSENT_UNDERSCORE_COMMANDS_ONLY	Directs the tool to only allow commands that use the underscore style. When this environment variable is set to any value, the legacy commands that used spaces are disabled. For example, the tool would accept analyze_drc_violation but would not accept “analyze drc violation”.

Contexts and System Modes

In Tessent Shell, the term “context” refers to a broad category of functionality that often corresponds to a specific point tool, product, or license feature, such as Tessent FastScan. Each context includes several system modes, which designate the tool’s current state of operation. By setting the context and then the system mode, you indicate the type of task you want Tessent Shell to perform.

Contexts

The context specifies the functional category of tasks you want to perform with Tessent Shell.

Table 1-2 lists the contexts that are currently available.

Table 1-2. Tessent Shell Contexts

Context	Description
dft	Editing and introspection of the following types of designs: gate-level Verilog, RTL Verilog, RTL System Verilog, and RTL VHDL.
dft -edt	EDT IP generation and optional insertion. This corresponds to the IP creation phase of Tessent TestKompress.
dft -scan	Scan analysis and scan chain insertion. This corresponds to Tessent Scan.
dft -logic_bist -edt	Configuration, generation, and insertion of the EDT/LBIST hybrid controller IP. This corresponds to Tessent LogicBIST.
dft -test_points	Test point identification and insertion. The test point identification algorithm focuses on improving random pattern testability of the design.
patterns -failure_mapping	Reverse map top-level failures to the core so that you can perform diagnosis with Tessent Diagnosis. Used after performing scan pattern retargeting within the patterns -scan_retargeting context.
patterns -ijtag	PDL command retargeting for IJTAG (IEEE P1687) plus extraction of the ICL network from the design.
patterns -scan	Test pattern generation and good and fault simulation. This corresponds to Tessent FastScan and the test pattern generation phase of Tessent TestKompress. The patterns -scan context supports uncompressed scan ATPG/fault simulation, EDT ATPG/fault simulation, and Logic BIST fault simulation.
patterns -scan_diagnosis	Test failure diagnosis to determine a defect's failure mechanism and location. This corresponds to Tessent Diagnosis.
patterns -scan_retargeting	Scan pattern retargeting for retargeting core-level test patterns at the top level.
patterns -silicon_insight	Control, simulate, debug, and characterize BIST-related memories, logic, PLLs, and SerDes. This corresponds to Tessent SiliconInsight.

Context Specification

You set the Tessent Shell context using the set_context command. For example:

```
SETUP> set_context dft -scan
```

You must set the context after you invoke Tessent Shell and before you can enter most commands. The set_context command is available only in setup mode. You can use the get_context command to see the current context.

Prior to setting the context, you can only run a small set of setup commands; these commands are those that you would typically place inside the startup file.

Context and Licensing

When you set the context, the tool automatically acquires the appropriate license, if available. Alternatively, you can directly specify the license Tessent Shell acquires by using the set_context command with the optional -license feature_name switch.

For more information about specifying a license feature, refer to the set_context command description in the *Tessent Shell Reference Manual*. For a complete list of license feature names, refer to *Managing Mentor Graphics Tessent Software*.

System Modes

A system mode in Tessent Shell defines the operational state of the tool. The default system mode is setup. The available system mode depends on the current context of the tool.

Table 1-3 lists the available system modes.

Table 1-3. Tessent Shell System Modes

System Mode	Description
setup	Used as the entry point into the tool. Used to define the current context and specify the design information.
analysis	Used to perform design analysis, test pattern generation, PDL retargeting, and simulation.
insertion	Used to perform design editing and introspection.

System Mode Specification

You change the Tessent Shell system mode using the set_system_mode command. For example:

```
SETUP> set_system_mode insertion
```

Contexts and System Mode Combinations

The specification of a context and a mode together provide a unique set of available features. The tool provides a set of commands that allow you to interact with contexts and modes.

The following matrix lists the features supported by the combinations of contexts and system modes:

Context	Setup Mode	Analysis Mode	Insertion Mode
dft	• Read and configure design • Prepare for design editing, EDT IP creation, scan insertion	• Scan analysis • EDT IP creation	• RTL or gate-level design editing with design introspection
patterns	• Read and configure design • Define test pattern generation type to perform: Scan (EDT, uncompressed, or LBIST) • IJTAG • Execute an IJTAG test pattern • Perform SimDUT simulation	• ATPG/fault simulation • PDL retargeting • Scan pattern retargeting • Execute an IJTAG test pattern • Perform SimDUT simulation	(not applicable)

Table 1-4 lists the available context and system mode commands.

Table 1-4. Tessent Shell Context and System Mode Commands

Command	Description
get_context	Returns the current context as specified by the set_context command. Also returns inferred subcontexts, such as whether patterns -scan is configured to perform Logic BIST or ATPG.
set_context	Sets the current context and its options.
set_system_mode	Sets the system mode.

Tessent Shell Tcl Interface

Tessent Shell is a Tcl-based tool that you use interactively from the command line or by using a dofile script.

The Tcl interface supports Tcl constructs, such as variables, command substitution, flow control, and procedures. You can embed Tcl constructs in tool commands and embed tool commands within Tcl constructs the same as any Tcl command.

Command Conventions

Tessent Shell provides a unified Tcl-style command set and naming convention. Commands that begin with a certain first word (for example, “get” in get_attribute_value_list) perform operations on the current design data model.

Table 1-5 provides a summary listing by command first word of the Tessent Shell commands. Refer to the *Tessent Shell Reference Manual* for a complete list of commands and options. You

can also use the “help” command with a wildcard to see a complete list of commands that start with the same word:

```
> help get_*
```

Table 1-5. Tessent Shell Command Conventions

Command First Word	Types of Operations Performed
append_ compare_ copy_ filter_ foreach_ index_ remove_ sizeof_ sort_	Operates on collections.
get_	Returns data strings, or collections of lists, objects, or object attributes from the design object model for introspection.
read_	Reads files into memory, such as libraries and netlists.
report_	Displays information about a specific item, such as the current context or licenses currently checked out.
set_	Specifies options, contexts, modes, attributes, and the current design.
write_	Writes design data to a file or set of files.

Command Completion

In the Tessent Shell interface, you can use the Tab key to complete command names. If the command you type is ambiguous, pressing the Tab key lists all matching commands. Additionally, within a command, pressing the Tab key can match variable names with $.

For example, you can abbreviate the get_attribute_list command in the Tessent Shell interface by typing the following:

```
SETUP> g_a_l <tab>
```

Pressing the Tab key after the string completes and expands the command.

Dofile Entries

You can use minimum typing of commands in a Tessent Shell dofile. For example, you can specify the set_system_mode command in a dofile as follows:

```
SETUP> set_s_m insertion
```

However, you should expand the command to its complete name when using it inside a large loop, because it consumes milliseconds to look up the command when the name is not fully specified. Additionally, avoid extreme minimum typing of commands in your dofiles. This is because there is no fixed minimum typing for any command or option, and current minimum typing can change in a future release due to the addition of new commands or options.

Dofile Transcription

By default, the transcript style is Full, which means the tool writes out all commands read from a dofile before any Tcl evaluation occurs. The command appears in the transcript as entered but is commented out. During the Tcl evaluation, Tcl commands are written to the transcript as follows:

- Writes all commands from the dofile with "// *command*:" prefix. This includes Tessent Shell commands, Tcl commands, and complete loop, if/else constructs.
- Tool commands embedded in Tcl constructs are written to the transcript as executed with a "// *sub_command*:" prefix. Tessent Shell completes the variable and command substitutions and writes the resolved values in the loop to the transcript.

Note

This does not apply to introspection commands: introspection commands are not written to the transcript again as subcommands.

- The tool indents nested dofiles when they are written to the logfile.
- The tool can read a Tcl routine similar to a dofile. If you issue the source command (>source my_report_env), then all the Tcl commands in the routine are not transcripted.

You control the Tessent Shell transcription behavior using the set_transcript_style command.

Tcl Command Registration

You can convert any Tcl procedure into a Tessent Shell application command. This enables you to implement functionality in Tcl and register that functionality as a command within the tool such that it is handled like any other built-in command. Like any built-in command, the newly registered Tcl command supports Tab completion, and its availability can be controlled relative to the context and system mode; the help system also includes the new command.

Tessent Shell provides the following commands to enable you to register a new command, and to define the arguments and options of that command.

Table 1-6. Commands for Tcl Command Registration

Command	Description
display_message	Controls messages produced by Tcl commands.

Table 1-6. Commands for Tcl Command Registration

Command	Description
lock_current_registration	Lock all current Tcl commands.
register_tcl_command	Register a Tcl command.
unregister_tcl_command	Unregister a Tcl command.

When you later execute the new command, the tool performs syntactic and semantic checks, and provides the parsed results to your Tcl implementation of the command. The tool also provides a mechanism to automatically define and register user-defined commands at tool invocation.

For more information about creating your own application commands, refer to the examples included with the register_tcl_command description in the *Tessent Shell Reference Manual*. The second example shows how to make your application commands automatically load during tool initialization.

Chapter 2
Design Data Model

Tessent Shell has an internal database that is organized into data models, which are broad categories of design information.

Design Data Models in Tessent Shell

Tessent Shell has the following data models: the hierarchical design data model, the flat design data model, and the ICL data model. Each of these data models contains one or more types of design objects (such as pins and modules), and each design object has a set of attributes (such as an ID or bit-width of a bus).

For a complete description of the various data models, object types, and attributes, refer to the "Data Model" chapter in the *Tessent Shell Reference Manual*.

Hierarchical Design Data Model

Tessent Shell translates your design netlist into a hierarchical design data model in memory when you load the design into the tool using a command such as read_verilog. The design data remains in memory during the Tessent Shell session until you delete the model or exit the tool.

The hierarchical design data model contains the following object types:

- Module — a basic building block of your design. A module can be a Verilog module, a Tessent library model, or a built-in primitive.
- Instance — a single instantiation of a module.
- Port — an input, output, or inout interface of a module.
- Pin — an input or output interface of an instance.
- Net — a wire that connects the pins of instances.

- Pseudo_port — represents a user-added primary input or primary output.

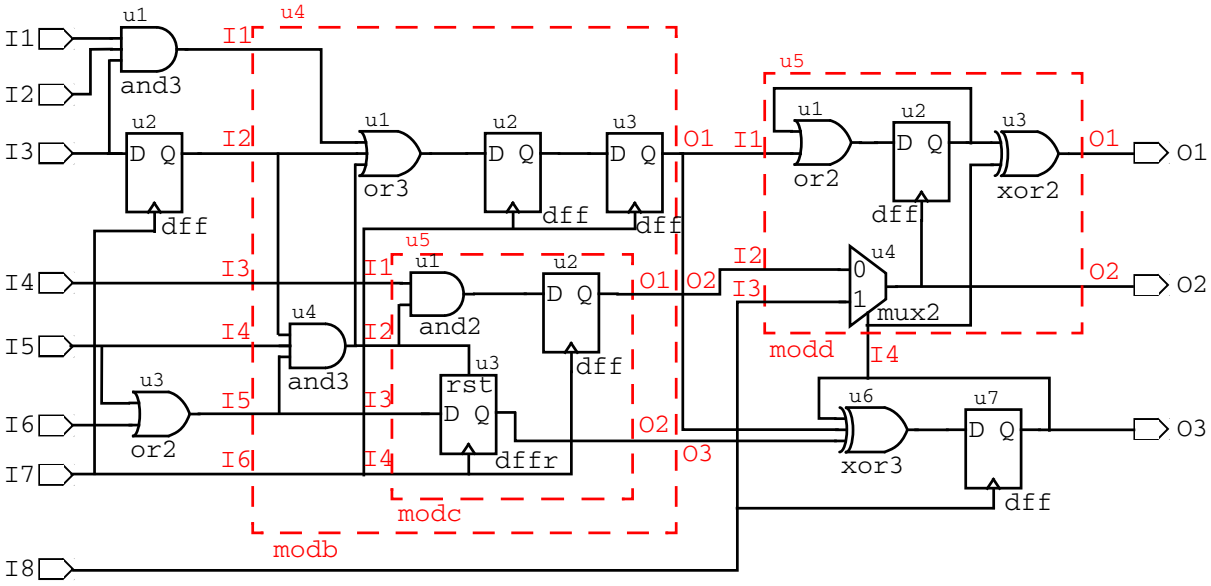
These object types are all described in detail in the "Data Models" chapter of the *Tessent Shell Reference Manual*.

You use the set_current_design command to designate one module within your design as the current design. Instances, pins, and nets are hierarchical objects defined relative to the current design.

Figure 2-1 shows a hierarchical design upon which many of the examples in this chapter are based. The label above the elements is the instance name while the label below the elements is the module name. For example, the module name of the AND gate in the upper-left corner is and3, and its instance name is u1.

The dashed boxes are design modules instantiated in the parent module. For example, module modc is instantiated inside of module modb. Its complete instance path is u4/u5.

Figure 2-1. Hierarchical Design Example



Flat Design Data Model

The flat design data model is an internal, flattened representation of the hierarchical design the tool creates when entering analysis mode. You can also explicitly create the flat model using the create_flat_model command.

The flat design data model consists of gates that are connected together. A gate is an instance of a primitive module, and a gate_pin object represents a pin on a gate instance. Gate_pin objects have no unique instance name. Instead they have a unique ID that differentiates one from another. The format of the ID is two integers separated by a period character. The first integer represents the gate ID, and the second integer represents the pin index (where 0 is the output pin, 1 is first input pin, and so on). However, this ID does not remain in place from one netlist

version to the next or from one Tessent Shell invocation to the next. For this reason, you should avoid hardcoding this ID into a script.

Note

You can preserve hierarchical pins in the flat model by using the set_attribute_options -preserve_boundary_in_flat_model option.

For more information about the flat design data model and the gate_pin object type, refer to "Flat Design Data Model" in the *Tessent Shell Reference Manual*.

ICL Data Model

The purpose of the Instrument Connectivity Language (ICL) is to describe the elements that comprise the 1687 network as well as their logical (though not necessarily physical) connections to each other and to the instruments at the endpoints of the network.

ICL bears a loose resemblance to a hierarchical netlist such as Verilog; it is organized by modules that may contain instances of other modules, and it describes the connections between the pins of the instances. However, it is important to note that ICL is not a complete netlist; connections are port-to-port rather than through nets, and ICL freely uses abstraction in order to omit the detailed physical construction of the circuitry — only the behavioral operation of the network is represented.

The ICL data model defines objects as one of the following four data types: icl_module, icl_instance, icl_port, and icl_pin.

Each icl_module object has its list of icl_port objects. Once the current design is set using the set_current_design command, the ICL data model is elaborated downward from the current design and icl_instance objects are created for each instance of icl_module, and icl_pin objects are created for each port of the modules associated to the icl_instances. The icl_instance and icl_pin objects are hierarchical objects and therefore only exist after the current design is set.

For more information about the ICL data model and the icl_module, icl_instance, icl_port, and icl_pin object types, refer to "ICL Data Model" in the *Tessent Shell Reference Manual*.

Object Attributes

Each design object has a list of characteristics, called attributes, attached to that object. For example, all pins have an attribute that specifies its hierarchical name and its parent instance. There are both predefined and user-defined attributes.

Predefined attributes provide access to information known to the tool such as the name of a module or the direction of a pin. All design objects have some predefined attributes, and every design object can have user-defined attributes as well. For a complete list of attributes for each type of data model, refer to "Data Models" chapter in the *Tessent Shell Reference Manual*.

The process for creating a new user-defined attribute is called *registration*. The predefined attributes, unlike the user-defined attributes, do not need to be registered.

You must register each new user-defined attribute and specify a default value before you can use the attribute. And if you want to later change the default value, you must unregister and then re-register the attribute. For more information about the registration process, refer to the description of the register_attribute command in the *Tessent Shell Reference Manual*.

You can query any attribute and change any user-defined attribute. Most predefined attributes are read-only, although some are read-write, such as the is_hard_module attribute.

You can filter and sort objects based on the attributes and their values. The filter_collection and sort_collection commands perform these operations. For more information about filtering attributes, refer to "Attribute Filtering Equation Syntax" in the *Tessent Shell Reference Manual*.

Object Specification in Tessent Shell

Tessent Shell commands introspect (or examine) and manipulate the various data models and object types. These commands operate on an "object specification" and return "collections" as described in the following sections.

Object Specification

All Tessent commands operating on user-specified objects accept an object specification (sometimes referred to as an object_spec) as an argument.

An object specification designates one or more objects (such as instances, pins, or nets). The object specification can be the name of an object, a Tcl list of names of objects, or a collection of objects. For object types that have a unique ID, like instances, you can use the ID as the object name.

The following examples show the add_display_instances command with valid object specifications:

```
add_display_instances [get_instances u*] -display design
add_display_instances [list u1 u2 u3] -display design
add_display_instances {u1 u2 u3} -display design
```

These commands display the results in the Design window of DFTVisualizer.

Collections

Collections are an extension of Tcl that are specific to Tessent applications. Native Tcl commands like "foreach" and "puts" do not recognize collections. A collection represents a

group of zero (an empty set) or more design objects that you can access via the Tessent Shell command interface.

Design introspection commands return collections of objects. The objects are stored in the internal data structures, and the tool returns a string handle to this collection. The string handle is a "@" character followed by a numeric ID (in this case, "@1"). The entire data volume remains in the tool's internal data structures, so the Tcl interface is not overloaded with large amounts of data. Tessent Shell commands provide access to these data structures. This gives you the flexibility of Tcl scripting while keeping all computation-intensive processing in the tool's back end.

Persistence of Collections

The tool references a collection when the collection is stored in a Tcl variable or when it is passed to a command or procedure. The tool automatically deletes a collection when it is no longer referenced.

You should not use Tcl built-in commands that expect a Tcl list, such as the foreach command, with collections. When you pass a collection to this type of command, the command dereferences the collection and, as a result, deletes the collection.

Example 1:

```
set instCollection [get_instances -of_type cell]
```

The collection created by the command get_instances is stored in the variable instCollection (which means that the collection is referenced). Tessent Shell deletes the collection when you unset the variable instCollection, or set the variable to a new value, or when the Tcl variable(s) referring to the collection go out of scope.

Example 2:

```
get_attribute_value_list [get_pins u1/a*] -name direction
```

The collection created by the command get_pins is passed to the get_attribute_value_list command; this means that the collection is referenced. Tessent Shell deletes the collection when the command get_attribute_value_list returns a value.

Collections can refer to objects that no longer exist because they have been deleted using editing commands, such as delete_pins. The built-in attribute "is_valid" is set to false when an object has been deleted. All commands that accept collection pointers as input automatically ignore objects with the "is_valid" attribute set to false.

Transcripting the Contents of Collections

You can use the get_name_list command to return a list of the names of all the elements in the collection. So, to transcript the names in the log file, you can use the "puts" command with get_name_list (or any other Tessent get_* command):

```
SETUP> puts [get_name_list $var1]
u1 u2 u3 u4 u5 u6 u7
```

Commands That Work With Collections or Tcl Lists

Tessent Shell provides commands that create, manipulate, and query collections. Table 2-1 presents some Tessent Shell commands based on how they interact with collections.

Table 2-1. Commands That Interact With Collections

Commands that...	Command Name
Create collections or Tcl lists[1]	get_attribute_list get_attribute_option get_attribute_value_list get_current_design get_fanins get_fanouts get_gate_pins get_instances get_modules get_name_list get_nets get_pins get_ports
Operate on collections	add_to_collection append_to_collection compare_collections copy_collection filter_collection foreach_in_collection index_collection remove_from_collection sizeof_collection sort_collection
Read and write attributes of objects found within collections or Tcl lists	get_attribute_list get_attribute_value_list report_attributes reset_attribute_value set_attribute_value

1. Note, this is not a comprehensive list.

For more information about these commands, refer to "Command Dictionary" in the *Tessent Shell Reference Manual*.

Chapter 3
Design Introspection and Design Editing

Tessent Shell commands can introspect (or examine) and modify the data structures described in the previous chapter.

Design Introspection

Tessent Shell introspection commands introspect the various data models and object types found within Tessent Shell. Introspection commands operate on an “object specification” and return “collections.”

Introspection commands issued directly from the shell display the “name” attribute of the first 50 elements of the collection, because the name is more useful than displaying the collection’s ID. However, names are not displayed in non-interactive mode such as when executing a dofile. The following example demonstrates the use of the get_instances and get_name_list commands to return collections of objects:

```
SETUP> set var1 [get_instances u* -hierarchical –of_modules MOD1]
{u1 u2 u3 u4 u5 u6 u7}

SETUP> puts $var1
@1

SETUP> puts [get_name_list $var1]
u1 u2 u3 u4 u5 u6 u7
```

In this example, the first command returns a collection of names. The objects are stored in internal data structures, and we return to Tcl a string handle to this collection which is “@” followed by a number ID (“@1”). The entire data volume remains in the tool’s internal data structures, so the Tcl interface is not overloaded with large amounts of data. The design introspection commands described in this section provide access to those data structures, giving you the flexibility of Tcl scripting while keeping all compute-intensive processing in the tool’s backend.

Introspection Examples

The following example demonstrates how to use collections and introspection commands together in a Tcl scripting environment.

Example of Creating a Procedure to Show Flip-Flop Fanouts

The following example is based on Figure 2-1 on page 22 and creates a procedure called show_fanouts that returns the fanouts of a specified flip-flop:

```
proc show_fanouts {flop} {
   set ff_fanout [get_fanouts $flop/Q]
   puts "$flop/Q is connected to: [get_name_list $ff_fanout]"
}
```

The following two examples show how to use the show_fanouts procedure you just created:

> **show_fanouts u2**
u2/Q is connected to : u4/u4/A0 u4/u1/A1

> **foreach_in_collection ff [get_instances * -of_module dff* -hierarchical] {**
show_fanouts [get_name_list $ff]
}

u2/Q is connected to: u4/u1/A1 u4/u4/A0
u7/Q is connected to: O3 u6/A0 u5/u3/A1 u5/u4/S0
u4/u2/Q is connected to: u4/u3/D
u4/u3/Q is connected to: u6/A1 u5/u1/A1
u4/u5/u2/Q is connected to: u5/u4/A0
u5/u2/Q is connected to: u5/u1/A0 u5/u3/A0
u4/u5/u3/Q is connected to: u6/A2

Example of Displaying the Number of Input Ports

The following example displays the number of input ports on modules (objects with names that begin with “mod” in Figure 2-1):

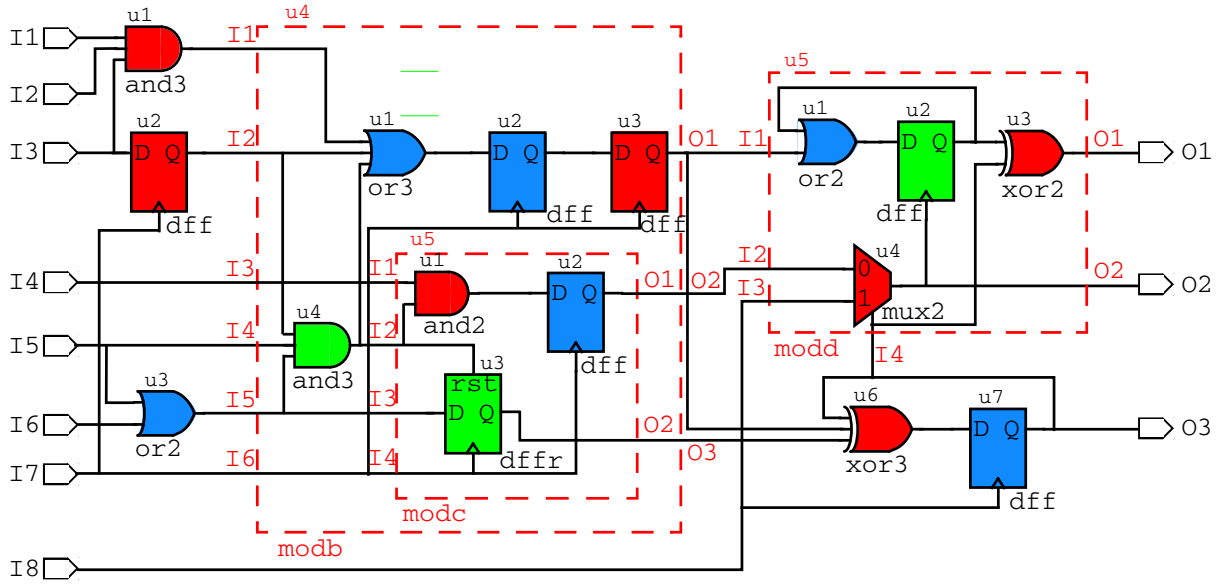
> **set mods [get_modules mod* -of_type design]**

> **foreach_in_collection itr $mods **
**{puts "The number of input ports of [get_attribute_value_list $itr -name name] **
is [sizeof_collection [get_ports -of_module $itr -direction input]]"}

The number of input ports of modb is 6
The number of input ports of modc is 4
The number of input ports of modd is 4

The design in Figure 3-1 is identical to Figure 2-1, with the addition of colors to help you understand the examples that follow.

Figure 3-1. Hierarchical Design Example With Colors



Example of Creating Attributes and Manipulating Collections

The following example is based on Figure 3-1 and shows the complete process of creating and setting user-defined attributes, and then creating and manipulating collections:

register_attribute -name color -value_type string -default "blue" -enum "red green blue"

Set the “color” attribute of each instance to match Figure 3-1
set_attribute_value {/u4/u4 /u4/u5/u3 /u5/u2} -name color -value "green"
{u4/u4 u4/u5/u3 u5/u2}
set_attribute_value {u1 u2 u6 /u4/u5/u1 /u4/u3 /u5/u4 /u5/u3} -name color -value "red"
{u1 u2 u6 u4/u5/u1 u4/u3 u5/u4 u5/u3}
set_attribute_value {u3 u7 /u4/u1 /u4/u2 /u4/u5/u2 /u5/u1} -name color -value "blue"
{u3 u7 u4/u1 u4/u2 u4/u5/u2 u5/u1}

Create collections of instances with the same color values
set all_inst [get_instances -of_type cell]
{u1 u2 u3 u6 u7 u4/u1 u4/u2 u4/u3 u4/u4 u4/u5/u1 u4/u5/u2 u4/u5/u3 u5/u1 u5/u2 u5/u3 u5/u4}
set red_inst [filter_collection $all_inst {color == "red"}]
{u1 u2 u6 u4/u3 u4/u5/u1 u5/u3 u5/u4}
set blue_inst [filter_collection $all_inst {color == "blue"}]
{u3 u7 u4/u1 u4/u2 u4/u5/u2 u5/u1}
set green_inst [filter_collection $all_inst {color == "green"}]
{u4/u4 u4/u5/u3 u5/u2}

Manipulate the collections and create additional collections
puts "Number of red cells in the design: [sizeof_collection $red_inst]"
Number of red cells in the design: 7
puts [compare_collections $red_inst $blue_inst]
1
set red_green [add_to_collection $red_inst $green_inst]
{u1 u2 u6 u4/u3 u4/u5/u1 u5/u3 u5/u4 u4/u4 u4/u5/u3 u5/u2}
set sort_red [sort_collection $red_inst name -descending]
{u6 u5/u4 u5/u3 u4/u5/u1 u4/u3 u2 u1}

Example of Displaying Attribute Values of a Collection

After Tessent Shell has executed the previous commands, the following commands display the colors assigned to gates in the collection named ANDs:

```
> set ANDs ""
> append_to_collection ANDs [get_instances -of_type cell -filter {module_name == "AND"}]

> foreach_in_collection itr $ANDs {puts "The color value for \
        [get_attribute_value_list $itr -name name] is [get_attribute_value_list \
            $itr -name color] "}

The color value for u1 is red
The color value for u4/u4 is green
The color value for u4/u5/u1 is red
```

As an alternate way to register the attribute in the previous example, you can use the register_attribute command as shown here:

```
register_attribute -name is_red -value_type bool \
-object_types “instance" –description "True if color is red."
```

So, instead of using the color attribute with enumerated values of red, green, and blue as shown previously, you could instead register three Boolean type attributes of is_red, is_green, and is_blue. This allows you to use Boolean values for filtering and tracing. For example:

```
> set red_inst [filter_collection $all_inst {is_red}]
```

Example of Changing an Attribute of a Collection

The following command example is based on Figure 3-1 and shows how to change the color attribute of a collection of DFFs to green:

```
> set DFFs [get_instances -of_modules dff]

# display all of the instances that have a color attribute of blue
> set all_blue [filter_collection $all_inst "color == blue"]
{u7 u4/u2 u4/u5/u2}

# change all instances in the collection DFFs to have a “color” attribute value of green
> set_attribute_value $DFFs -name color -value green

# now check the results
> set all_green [ filter_collection [$all_inst {color == "green"}] ]
{u2 u7 u5/u2 u4/u4 u4/u2 u4/u3 u4/u5/u2 u4/u5/u3}
```

Example of Removing Duplicate Objects from a Collection

The following example shows how to create a collection containing three duplicate objects, and then remove the duplicate objects from the collection:

```
> set t [get_instances {u3 u3 u3}]
{u3 u3 u3}

> sizeof_collection $t
3

> set tt [add_to_collection "" $t -unique]
{u3}
```

```
> sizeof_collection $tt
1
```

Example of Creating a Custom Report

The following example creates a collection of all instance objects with a leaf name starting with "u," and then displays ten instance names per row.

```
set cnt 0
set line ""
foreach_in_collection element [get_instances u* -hierarchical] {
  if {$cnt < 10} {
    append line "[lindex [get_name_list $element] 0] "
    incr cnt
  } else {
    set cnt 0
    puts $line
    set line ""
  }
}
if {$cnt > 0} {puts $line}
```

Design Introspection Command Summary

In this section, introspection commands are presented by category based on which objects they introspect.

Design Introspection Command Summary

Table 3-1 lists all of the design introspection commands.

Table 3-1. Design Introspection Commands

Command Name	Description
get_current_design	Returns a collection of one element that specifies the top-level design when previously set.
get_fanins	Returns a collection of all objects found in the fanin of the specified pin, net, or port objects.
get_gate_pins	Returns a collection of all gate_pin objects found below the current design in the flat model.
get_instances	Returns a collection of instances relative to the current design.
get_modules	Returns a collection of modules.
get_nets	Returns a collection of nets relative to the current design.
get_pins	Returns a collection of pins relative to the current design.
get_ports	Returns a collection of ports on a given module.

Attribute Introspection Command Summary

Table 3-2 lists all of the commands that introspect attributes.

Table 3-2. Attribute Introspection Commands

Command Name	Description
get_attribute_list	Lists registered attributes.
get_attribute_option	Returns the current setting of an attribute's option.
get_attribute_value_list	Returns the attribute's value.
get_name_list	Returns the name attribute of the specified objects.
register_attribute	Registers a new attribute by adding it to the attribute manager.
report_attributes	Prints a report for the registered attributes.
reset_attribute_value	Resets the attribute to its default value.
set_attribute_options	Configures attribute options.
set_attribute_value	Defines the attribute's value.
unregister_attribute	Makes an attribute unusable by removing it from the attribute manager.

Design Editing

You use Tessent Shell design editing commands to modify your design after reading in the RTL or gate-level netlist. Tessent Shell supports gate-level netlist editing or RTL design editing with full language support, including multiple logical libraries, VHDL, Verilog, and System Verilog. Parameterized modules are also fully supported.

Design editing commands work with collections and introspection commands, as well as native Tcl commands, to automate many tasks. Table 3-3 presents the design editing commands based on the function they perform— that is, whether they create, modify, or remove elements and so on. For more information on collections and introspection commands, see “Collections” on page 24 and “Design Introspection” on page 27.

Table 3-3. Design Editing Commands

Commands that...	Command Name
Read netlists and specify logical libraries	read_verilog read_vhdl set_logical_design_libraries

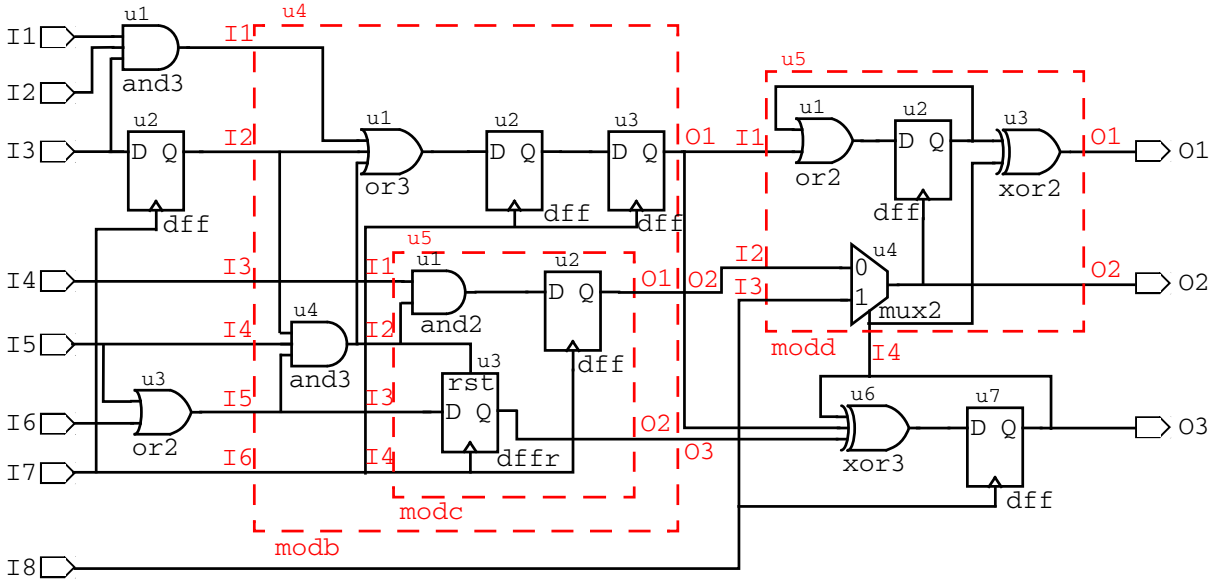
Table 3-3. Design Editing Commands

Commands that...	Command Name
Create design elements	create_connections create_instance create_module create_net create_pin create_port
Remove design elements	delete_connections delete_instances delete_nets delete_pins delete_ports
Modify the design	copy_module intercept_connection move_connections rename_instance replace_instances uniquify_instances
Set or get editing options	get_insertion_option set_insertion_options

Design Editing Examples

There are many different ways to create, modify, and remove elements from your design in a Tcl scripting environment. The following examples that show some of these ways are based on the following hierarchical design.

Figure 3-2. Hierarchical Design Example



Example of Creating a Module From Scratch

The following example shows how to create module C (modc) in Figure 3-2 as a standalone module:

```
set_context dft -no_rtl
set_system_mode insertion
create_module modc -language verilog
set_current_design modc

create_port I1 -on_module modc -direction input
create_port I2 -on_module modc -direction input
create_port I3 -on_module modc -direction input
create_port I4 -on_module modc -direction input
create_port O1 -on_module modc -direction output
create_port O2 -on_module modc -direction output

create_instance u1 -of_module and2
create_instance u2 -of_module dff
create_instance u3 -of_module dffr

create_connection  I1  u1/A0
create_connection  I2  u1/A1
create_connection  I2  u3/R
create_connection  I3  u3/D
create_connection  I4  u3/CLK
create_connection  I4  u2/CLK
create_connection  u1/Y  u2/D
create_connection  u2/Q  O1
create_connection  u3/Q  O2

write_design -output_file MODC.v -replace
```

Example of Replacing a Module With Another Module

The following example replaces the module definition for instance u1/u2 to ModB. The tool preserves the original connection to pin u1/u2/a while leaving the new pin u1/u2/b open. (Note that this example is not based on Figure 3-2.)

get_attribute_value_list u1/u2 –name module_name
{ModA}

get_fanin u1/u2/a –stop_on net
{u1/n45}

replace_instances u1/u2 –with_module ModB
{u1/u2}

get_fanin u1/u2/a –stop_on net
{u1/n45}

get_fanin u1/u2/b –stop_on net
{}

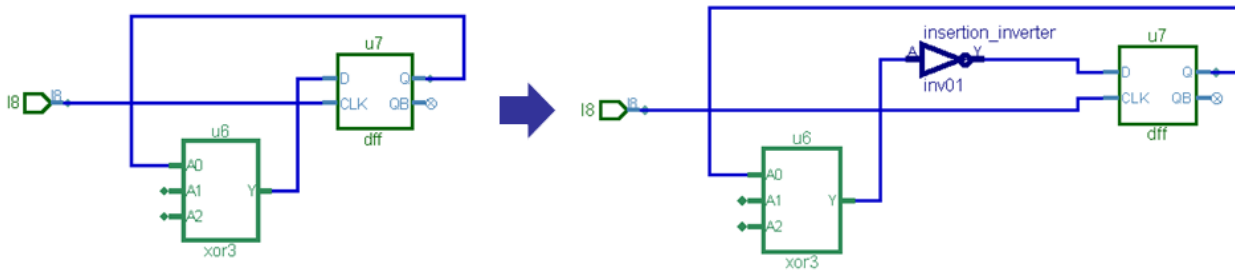
Example of Intercepting a Connection

The following example inserts a simple inverter gate by intercepting the existing connection between the u7/D pin and the u6/Y pin, shown on the left side of Figure 3-3. After the interception, the inverter A pin is connected to the u6/Y pin and the inverter Y pin is connected to the u7/D pin as shown on the right side of Figure 3-3.

INSERTION> intercept_connection u7/D -cell_function_name inverter

```
{insertion_inverter}
```

Figure 3-3. Inverter Interception



Example of Adding Input and Output Pads to a Design

The following commands add input and output pads to a design by creating a collection of port names from the design, creating a collection of pad names that match the ports, then connecting the ports and pads together in the design. The example also adds the TAP's IO ports, including the respective pads.

```
# Setting the context to netlist editing
set_context dft -no_rtl

# Reading the library and all necessary Verilog files
read_cell_library ../libs/libs/adk.atpg
read_verilog       ./counter_block2_edt_top_gate.v
```

```
read_verilog      ../libs/pads/*.v

# Telling the tool, which module 'top' is
set top_module "counter_block2"
set_current_design $top_module

# Now for the editing
set_sys_mode insertion

# First, insert the pad cells for all inputs of the netlist
set inputPortList [ get_ports -of_module $top_module -direction input ]
foreach_in_collection inputPort $inputPortList {

    # Creating the name of the pad cell:
    #   The following lines take care of '[' and ']' in the portname,
    #   substituting these by '_'. This makes the Tcl-life easier
    set portName [lindex [get_name_list $inputPort] 0]
    set padName  [ string map { \[ _ \] _ } ${portName}_PAD ]

    # Adding the pad cell
    create_instance $padName -of_module INPAD

# Connecting it up in the netlist
# The first line moves the existing net from the input port to the output
# of the pad cell. The second line creates a new net and connects the
# input port to the input of the pad cell
# Note: We are not connecting anything else here.
#       We leave this to boundary scan insertion
    move_connection -from $inputPort -to $padName/FP
    create_connection $inputPort $padName/IO
}

# Second, insert the pad cells for all outputs of the netlist
set outputPortList [ get_ports -of_module $top_module -direction output ]
foreach_in_collection outputPort $outputPortList {

    # Creating the name of the pad cell:
    #   The following lines take care of '[' and ']' in the portname,
    #   substituting these by '_'.
    set portName [lindex [ get_name_list $outputPort ] 0]
    set padName  [ string map { \[ _ \] _ } ${portName}_PAD ]

    # Adding the pad cell
    create_instance $padName -of_module OUTPAD_2S

   # Connecting it up in the netlist
   # The first line moves the existing net from the IO port to the input
   # of the pad cell. The second line creates a new net and connects the
   # IO port to the output of the pad cell.
   # Note: We are not connecting anything else here.
   #       We leave this to boundary scan insertion
    move_connection -from  $outputPort -to $padName/TP
    create_connection $outputPort $padName/IO
}

# Next, add the JTAG IO pins and their respective PAD cell. Connecting #
them up.
```

```
set inputPortList  { TDI TMS TRST TCK }
set outputPortList { TDO }

foreach portName $inputPortList {

    # Adding the pad cell
    set padName     ${portName}_PAD
    create_instance $padName -of_module INPAD

    # Creating an Input IO pin and connecting the pad
    create_port $portName -direction input
    create_connection $portName $padName/IO
}

foreach portName $outputPortList {

    # Adding the pad cell
    set padName     ${portName}_PAD
    create_instance $padName -of_module OUTPAD_EN1

    # Creating an output pin and connecting the pad
    create_port $portName -direction output
    create_connection $portName $padName/IO
}

# Write the new netlist
write_design -output_file "${top_module}.v" -replace
exit
```

Design Editing Command Summary

The Tessent Shell commands that are available for design editing are listed in the following table.

Table 3-4. Design Editing Command Summary

Command Name	Description
copy_module	Creates an exact copy of a design module and gives it a new name, which the tool can use as part of create_instance and replace_instances operations.
create_connections	Creates a connection between pin, net, or port objects.
create_instance	Instantiates a module (design or cell type) inside of a design module that is part of the current design.
create_module	Creates a new design module.
create_net	Creates a net inside an instance of a design module.
create_pin	Creates a pin on an instance of a design module.
create_port	Creates a port on a design module.
delete_connections	Disconnects the specified pin objects.

Table 3-4. Design Editing Command Summary

Command Name	Description
delete_instances	Removes an instance of a module.
delete_nets	Removes net objects inside an instance of a design module.
delete_pins	Removes pin objects on an instance of a design module.
delete_ports	Removes port objects on a design module.
get_insertion_option	Introspects the default values of options affecting many design editing commands.
intercept_connection	Using the get_dft_cell command, obtains a cell with the specified function name and uses it to intercept a connection to a pin, port, or net.
move_connections	Moves a net connected on one pin or port to another pin or port. The first pin is left open after the move.
rename_instance	Renames the leaf name of an instance object.
replace_instances	Replaces the module object used in an instantiation.
set_insertion_options	Specifies default values of options affecting many design editing commands.
uniquify_instances	If the module of the specified instance has other instantiations in the design, the tool copies the module and the specified instance becomes an instance of the copied module.

Chapter 4
DFTVisualizer

DFTVisualizer is the tool you use for viewing and debugging design and simulation data in Tessent Shell.

DFTVisualizer Overview

DFTVisualizer provides a visual means of browsing and troubleshooting designs in the following Tessent Shell contexts and subcontexts:

- dft -scan (Tessent Scan)
- dft -edt (Tessent TestKompress IP Creation)
- patterns -scan (Tessent FastScan, TestKompress Test Pattern Generation)
- patterns -scan_diagnosis (Tessent Diagnosis™)
- patterns -scan_retargeting
- patterns -failure_mapping

For more information on Tessent Shell contexts and subcontexts, see "Contexts and System Modes" on page 13.

DFTVisualizer Invocation

Prior to invoking DFTVisualizer, you should perform the following steps:

1. Set a context using the set_context command.
2. Load a design using the read_verilog command.
3. Read a library using the read_cell_library command.
4. Set the current design using the set_current_design.

Note

Note, you can invoke DFTVisualizer without issuing these commands but most of the user interface will be disabled until you issue these commands. You can also issue these commands from the Transcript window after you have invoked the tool.

After you have done this, you can invoke DFTVisualizer using one of the following methods depending on the task you want to perform.

- Invoke DFTVisualizer explicitly by issuing any command that provides a -Display argument. For example:

  open_visualizer -display browser design data

- Issue a command that requests information from the tool. In this case, the tool invokes DFTVisualizer to analyze and then display the requested data. For example:

  analyze_drc_violation c3-3

  The analyze_drc_violation command opens DFTVisualizer with the appropriate windows displaying a schematic of the parts of the design associated with the specified DRC violation.

- Issue a command that opens the DFTVisualizer and displays DftSpecification configuration data. For example, if you have already read a specification into memory, issuing this command opens DFTVisualizer with the configuration data loaded:

  display_specification

DFTVisualizer Window Overview

DFTVisualizer contains windows for viewing and debugging design and simulation data. You can access these windows from the **Windows** pulldown menu.

- **Task Manager Window** — Displays a quick list of tasks to choose from.
- **Debug Window** — Displays a schematic representation of the flattened model of your design. The schematic can be at the design or primitive level.
- **Design Window** — Displays a hierarchical schematic of the design as described in the input netlist. Includes net names and hierarchical ports down to library-level instances.
- **Browser Window** — Displays tabs for accessing multiple windows:
  - **Hierarchy Browser** — Navigates the design hierarchy and displays coverage and DRC statistics for hierarchical blocks.
  - **Library Browser** — Displays statistics on the ATPG library models used in a design. By default, consolidated data for each ATPG library model displays. Each library model can be expanded to display statistics for the individual instances of the model.
  - **Clocks Browser** — Displays all of the clocks in the design and their attributes. The Clocks Browser allows you to navigate through the design hierarchy and view the faults for each individual clock and analyze the distribution of faults between clock domains.

Note

Currently, the Clocks Browser does not support the UDFM fault type.

- **Signals Window** — Displays pins and signals for instance selected in any tab of the Browser window.
- **Data Window** —Displays DRC and pattern data for specific instances and signals.

- **Wave Window** — Provides a waveform representation of test_setup data and named capture procedures (related to Data window).
- **Global Search Window** — Allows you to search for any instance, net, or pin in the active design.
- **Global Search Window** — Provides illustrative waveforms of what the pattern data displayed in the Data or Debug window means.
- **Text Editor Window** — Provides a basic text editor for creating new test procedure files and dofiles, or modifying existing netlists, test procedure files, dofiles, or current design files.
- **Transcript Window** — Displays notes, warnings, and errors applicable to the session.
- **Test Structures Window** — Displays graphical representation of the EDT logic inserted by Tessent TestKompress.

DFTVisualizer Quality Agent

When internal errors occur, it can be difficult for you to recall previous steps and recreate the problem. To assist in these cases, DFTVisualizer produces an error transcript that usually provides enough information to enable our Customer Support team to identify the problem.

The Quality Agent, shown in Figure 4-1, enables you to do the following:

- Automatically send transcripted error information to Mentor Graphics by clicking the Send Report button.

  You must enter information into the Steps field before you can successfully send the report to Mentor Graphics. Providing this feedback ensures that the problem you experienced is addressed as quickly as possible. This enables Mentor Graphics to provide the highest product quality.

Note

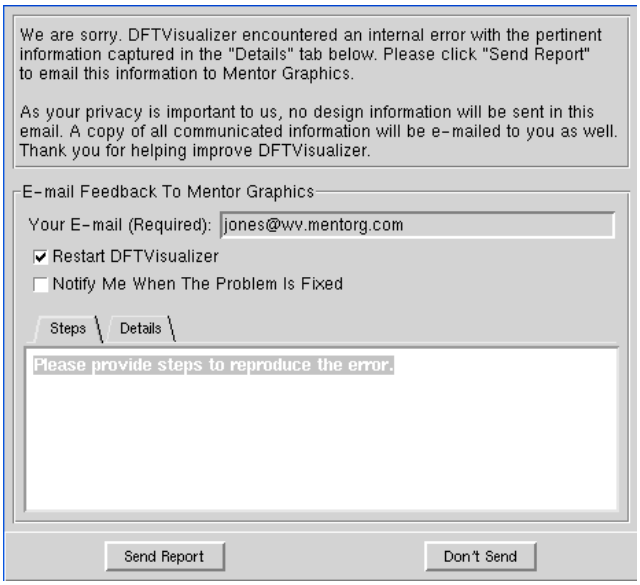
Please be aware that when you send feedback, absolutely NO DESIGN DATA is communicated to Mentor Graphics. To ensure your privacy, the transcripted error information is sent as an email in a text format; it is also sent to any email address you specify.

- Choose whether you want to restart DFTVisualizer following the internal error.

  You can use this option to automatically restart DFTVisualizer. This is usually the recommended action when an internal error occurs. If you want to preserve the displayed data for capturing screen shots or any other reason, you can decline this option and perform a manual restart using the **open_visualizer -restart** command.

- Choose whether to be notified when the problem has been addressed.

Figure 4-1. DFTVisualizer Quality Agent



Performing Basic Tasks

Typically, you will open a number of windows as you explore the tool's database, so developing good window management and navigation technique is helpful.

Saving and Restoring Displays

When you have a set of window and data displays you want to use again later, you can save the settings and then read them into a subsequent session of the tool. You restore the window and data displays by entering a dofile command and specifying the saved dofile as an argument.

Procedure

1. Select the **File > Create Dofile** menu item. The Create Dofile dialog box displays.
2. Specify the name of the dofile and click OK.

Results

The tool writes out the specified dofile containing the commands needed to recreate the instances and data sets currently displayed in DFTVisualizer windows.

Note

The commands in this dofile do not replicate the entire command sequence used in the tool session. They simply add the instances and data sets that were present in the DFTVisualizer windows when you wrote out the dofile.

Searching for an Instance, Net, or Pin in the Active Window

You can search the active window for a specified instance, pin, or net using the Find field on the toolbar.

Procedure

1. Enter a string in the Find 

text entry field to search on. By default, the tool implicitly applies wildcards to the search string and selects all of the design objects whose pathname includes that string in the active window. You can click the Exact  icon to search for the exact string without applying wildcard characters.
2. Click the  icon to search for the specified string.

Note

You can also click the  icon in this step. This opens the Global Search window and the search is applied to the entire design. Refer to “Searching for an Instance, Net, or Pin in the Design” on page 46” for more information on using this window.

Results

The objects matching the specified string are selected in the active window.

Searching for an Instance, Net, or Pin in the Design

You can search the entire design for a specified instance, pin, or net using the Global Search window.

Procedure

1. Click the  icon in the toolbar or select the **Windows > Global Search** menu item.
2. Enter a string in the Global Search text entry field to search on. The tool implicitly applies wildcards to the search string and returns all of the design objects whose pathnames include that string. You can click the Exact Search checkbox to search for the exact string without applying wildcard characters.
3. Optionally, click **Options** in the Global Search window to specify a search depth, search area, and an object type.
4. Click **Search**.

Results

The objects matching the specified string are listed in the Global Search window, or the tool reports that no matching netlist instances are found.

Interruption of Operations from DFTVisualizer

You can interrupt some DFTVisualizer operations at any time by pressing Control-C in the active DFTVisualizer window.

Alternately, you can also interrupt the same operations by pressing the Stop  icon.

The following operations can be interrupted/canceled:

- Data generation for faults, DRCs, gates and primitives in the Hierarchy Browser
- Searching in the Global Search window
- Tracing forward/backward to endpoints for instances and pins in the Design window
- Tracing down in the Design window
- Pattern creation

Undocking and Docking Windows

Each special purpose window can be undocked from the main window. When undocked, you can move the window around on your desktop independent of the main window.

Prerequisites

- DFTVisualizer is invoked. For more information, see the "DFTVisualizer Invocation" section in this chapter.
- At least one window is open.

Procedure

1. To undock a window, click the middle button of the three just outside the upper right corner of a window's display area.
2. To dock the window, click the middle button again.

Tip: You can also undock/dock a window by pressing the left mouse button over the center of the window's header bar and simultaneously dragging it outside/back into the main window.

Resizing Windows

Each window can be resized. Each special purpose window, when undocked, can also be maximized using the maximize button that expands to completely occupy the window. The maximize button is the middle button of the three just outside the upper-right corner of the window's display area.

Tip: You can resize a window using the little square in the upper left-hand area in the bar between the displayed windows.

Prerequisites

- DFTVisualizer is invoked. For more information, see the "DFTVisualizer Invocation" section in this chapter.
- At least one window is undocked (unattached to the session window).

Procedure

1. To expand an undocked window, click its maximize button.
2. To return (dock) an undocked window to its former position and size within the main window, click the button again.

Repositioning Windows

You can reposition windows in the DFTVisualizer session.

Prerequisites

- DFTVisualizer is invoked. For more information, see the "DFTVisualizer Invocation" section in this chapter.
- At least two windows are open.

Procedure

1. Press the left mouse button over the  icon in the center of the window's header bar and simultaneously drag it to the new location. A dynamic outline of the window in the new location appears when you have moved the mouse sufficiently for the tool to successfully determine the desired location.
2. Release the mouse button.

Results

The window is anchored in the new location.

Accessing Popup Menus

You can access most of a tool's available menu options from the window popup menu.

Prerequisites

- DFTVisualizer is invoked. For more information, see the "DFTVisualizer Invocation" section in this chapter.
- At least one window is open.

Procedure

1. Press the right mouse button in the display area of any window. The menu options on the popup menu vary as follows.
   - **Debug and Design window**

     If no objects are selected and the cursor is not positioned directly over an object, the menu is a simple one for adding instances to the window's display area. If an object is selected or the cursor is positioned over an object, the popup menu displays options needed for tracing and debugging.
   - **All windows except for the Debug and Design window**

     The object the cursor is positioned over when you open the popup menu (press the right mouse button) is automatically selected and the menu applies to it.
2. Select a menu option by moving the cursor over the menu item and releasing the right mouse button.

Object Name Copied from a Popup Menu to the System Clipboard

When you open a popup menu on a single selected object, the object's pathname is listed as the first menu option. If you choose the name from the menu, it is copied to the system clipboard.

You can then paste the name into other desktop locations, such as a shell window, tool command line, or dialog entry box.

Adding Instances to a Display Window

You can add instances to a window display by explicitly identifying the object or by copying the instance from another window.

Prerequisites

- DFTVisualizer is invoked and at least one window is open. For more information, see the "DFTVisualizer Invocation" section in this chapter.

Procedure

Add an instance to a window using one of the following methods:

If you want to...	Do the following:
Explicitly identify an instance to add	1. Activate the window you want to add the instance to. 2. Click the Add Instances  icon on the toolbar. You can use any number of asterisks (*) or question marks(?) as wildcard characters to specify this string enabling you to match many pathnames in the design.
Copy an instance from another window	1. Activate the window you want to copy the instance from. 2. Copy the instances from another window. See "Viewing Instances in Other Windows" on page 54 for details of this method.

Related Topics

add_display_instances command

Selecting Objects

You can select objects displayed in a window using several different methods.

Prerequisites

- DFTVisualizer is invoked and objects are displayed in one or more windows. For more information, see “Adding Instances to a Display Window” on page 49.

Procedure

Select objects using one of the following methods:

If you want to...	Do the following:
Select a single object	Move the cursor over the object and click the left mouse button. When selected, the object is highlighted in a different color. You can customize the highlight color using the **Edit > Preferences** menu item.
Select additional objects without unselecting previously selected objects	Press and hold the Ctrl key while selecting additional objects using the left mouse button.
Select a range of objects between a currently selected item(s) and a new object	Press and hold the Shift key while selecting the new object using the left mouse button. This selects the objects without unselecting the previously selected objects.
Select all displayed instances	Select the **Edit > Select All (Ctrl + A)** menu item. This action selects all objects in the window including nets. Note, this method is only valid in the Debug, Design, Data, Wave, or Text Editor windows. It is not supported in the Browser or Signals windows.

Cross-Selecting Objects

Cross-selection occurs when you select objects in one window and they are simultaneously selected in all windows in which they are already displayed. Cross-selection is useful for flagging an instance so you can identify it easily when viewing information about it in multiple windows.

Prerequisites

- DFTVisualizer is invoked and objects are displayed in one or more windows. For more information, see “Adding Instances to a Display Window” on page 49.

Procedure

1. Select the object(s) in the active window that you want to cross-select.
2. Press the right mouse button anywhere in the window's display area, and choose the **Cross Select** option from the popup menu.

Tip: If you prefer to have cross-selection automatically occur by default whenever you select an object, without requiring a menu pick, enable the Automatically Cross Select In All Windows option in the Global Preferences Dialog Box. You can access the dialog box by using the **Edit > Preferences** menu item.

Results

The selected object(s) is simultaneously selected in all windows in which it is already displayed.

Selecting Multiple Objects in the Debug and Design Windows

You can select objects in any DFTVisualizer window using different methods.

See the possible methods available in "Selecting Objects" on page 49. There is an additional way to select multiple objects (nets, pins, and/or instances) in the Debug and Design windows.

Prerequisites

- DFTVisualizer is invoked and objects are displayed in one or more windows. For more information, see "Adding Instances to a Display Window" on page 49.

Procedure

1. Press the left mouse button and drag the cursor so the bounding box contains all the objects you want selected.
2. Release the left mouse button.

Results

This action selects all objects on the schematic within the area of the bounding box.

Unselecting Objects

You can unselect currently selected objects using either the mouse or the tool menu.

Prerequisites

- DFTVisualizer is invoked and objects are displayed in one or more windows. For more information, see “Adding Instances to a Display Window” on page 49.
- At least one object is selected in the active window.

Procedure

Click in a blank part of the display area of the window, or select the **Edit > Undo** menu item if it is available for the window.

Results

All selected objects are unselected.

Moving Objects in the Debug or Design Window

Use this procedure to move an object to a new location in either the Debug or Design window.

Prerequisites

- DFTVisualizer is invoked and at least one object is displayed in the active window. For more information, see “Adding Instances to a Display Window” on page 49.

Procedure

1. Position your cursor over the object you want to move.
2. Simultaneously, press and hold the Shift key while pressing and dragging the right mouse button. A ghost image of the object(s) appears and moves across the screen as the cursor moves.
3. Release the right mouse button to anchor the object at the location of the cursor.

Note

If you add or delete an instance after moving an object, DFTVisualizer will re-optimize the view based on the new window content and your change will be lost.

Results

This action anchors the object at the final location of the cursor.

Customizing Marking Colors in the Schematic Windows

Use this procedure to specify the number of colors available to mark objects in the Debug, Design, and Test Structures windows, and also to customize which colors are available.

Prerequisites

- DFTVisualizer is invoked.

Procedure

1. Choose **Edit > Preferences** and click the **Colors** tab.
2. Click the window name in the Windows List field for which you want to customize colors. The window name becomes highlighted.
3. Click **Marked** in the Options List. The **No. of Colors** and **Color Index** buttons, which are only related to marking, display.
4. Click the **No. of Colors** and select a number to choose the number of colors you want to use when marking objects.
5. Click **Color Index** to select the number for which you want to assign a color and then click the desired color from the Color Palette.
6. Click **OK** to save your selections.

Results

The colors you set with this procedure are now available on the cascading menus of the specified window when you choose the **Display > Marking (Ctrl + M)** pulldown menu or **Marked** popup (RMB) menu.

Marking and Unmarking Objects in the Schematic Windows

Use this procedure to mark and unmark objects in the Debug, Design, and Test Structures windows using multiple colors.

Prerequisites

- DFTVisualizer is invoked, the Debug, Design, or Test Structures window is active, and at least one object is displayed in the active window. For more information, see "Adding Instances to a Display Window" on page 49.

Procedure

1. Select the objects you want to mark.
2. Press and hold down the Ctrl key; press M. A colored rectangle shows the active mark color in the upper left-hand corner of the window. Repeatedly press M until the rectangle shows the color you want to mark the selected objects with or shows gray to unmark the instance; release the Ctrl key.

Note

You can also mark or unmark objects using the following methods: (1) by selecting **Display > Marking (Ctrl + M)** from the pulldown menu or **Marking (Ctrl + M)** from the popup (RMB) menu, and (2) by using the mark_display_instances and unmark_display_instances commands.

3. Click the cursor in the active window to deselect the selected objects and display the objects as marked.

Tip: You can select **Display > Zoom > Marked** to reposition the schematic view to show marked objects.

Results

The selected objects display in the marked color.

Viewing Instances in Other Windows

You can view an instance that is visible in one window in a different window as listed in the following table. This can save time when you are viewing an instance in one window and the kind of data you want to see for the instance requires a different window.

Table 4-1. Windows Between Which You Can View Instances

Source Window	Destination Windows
Debug Window	Data/Wave, Design, Hierarchy Browser, Library Browser, Text Editor (Definition), and Text Editor (Instantiation)
Design Window	Data/Wave, Debug, Text Editor (Definition), and Text Editor (Instantiation)

Table 4-1. Windows Between Which You Can View Instances

Source Window	Destination Windows
Browser, **Hierarchy** tab, see "Usage Notes for the Hierarchy Browser"	Data/Wave, Debug, Design, Library Browser, Text Editor (Definition), and Text Editor (Instantiation)
Browser, **Library** tab, see "Usage Notes for the Library Browser"	Data/Wave, Debug, Design, Hierarchy Browser, Text Editor (Definition), and Text Editor (Instantiation)
Browser, **Clock** tab, see "Usage Notes for the Clock Browser"	Data/Wave, Debug, Design, Text Editor (Instantiation)
Signals Window	Data/Wave, Debug, and Design
Data Window	Debug, Design, Text Editor (Definition), and Text Editor (Instantiation)
Wave Window	Debug, Design, Text Editor (Definition), and Text Editor (Instantiation)
Global Search Window	Data/Wave, Debug, Design, Hierarchy Browser, Library Browser, Text Editor (Definition), and Text Editor (Instantiation)

Prerequisites

- Instances you wish to view in another window must display in the source window.

Procedure

Use one of the following methods to view an instance that is in one window (source) in another window (destination).

If you want to...	Do the following:
Drag-and-Drop	1. Select the objects you want to view in another window using instructions in "Selecting Objects" on page 49. 2. Click the left mouse button on any one of the selected items in the first window and then move the cursor into the display area of the destination window. The cursor in the destination window now includes a small box and plus (+) sign. 3. Release the left mouse button in the display area of the destination window to view the instances in the new window. **Note**: You cannot view an object(s) in the Browser or Text Editor window using this method. You should use one of the alternate methods to view objects in these windows.

If you want to...	Do the following:
Use the Right Mouse (Popup) Menu	1. Move the cursor over an instance in the source window. 2. Press the right mouse button and select the **View In** option on the popup menu to specify the destination. **Tip**: To view multiple instances, first select them, then move the cursor over one of the selected instances and press the right mouse button to access the popup menu.
Use the Main Pulldown Menu	1. Select one or more instances in the source window, then choose the **Edit > Copy** menu item. 2. Paste the selection into the destination window: a. Click on the window header bar of the target window and choose the **Edit > Paste** menu item, or press the right mouse button in the destination window and select **Add** from the popup menu. The Make Additions to the Display dialog box displays. b. Move the cursor over the entry box, press the right mouse button, and select Paste. c. Click the Add button and OK the dialog box.

Results

The selected objects are visible in the destination window.

Related Topics

Adding Instances to a Display Window

Copying and Pasting Object Names in the Design

Use this procedure in a DFTVisualizer window to select and copy an object's hierarchical pathname to a buffer which you can then paste into DFTVisualizer or any other application.

When you select more than one object, the selected object names are copied into a list surrounded by brackets and delimited by spaces as shown here: {obj_name1 obj_name2 ...}. If you select a single object whose name does not contain any special characters, the object name is copied into the buffer but is not surrounded by brackets.

Prerequisite

- DFTVisualizer is invoked and at least one window is open. For more information, see the "DFTVisualizer Invocation" section in this chapter.
- Object(s) whose name you wish to copy must display in the source window.

Procedure

1. Select the objects whose names you want to copy. Note, you can use the Shift key to select more than one object.

> **Tip**: If you are copying only one object name, you can position your cursor over the object, without selecting it, and click the right mouse button.

2. Click the right mouse button in the display area of the window and select the "<object_name(s)> (click to copy)" item at the top of the popup menu.
3. To paste the object(s) in DFTVisualizer, click the right mouse button in the location you want to paste and select **Edit > Paste** from the popup menu or type Ctrl-V.

Trace Options

The following table summarizes the trace options available in the Debug window (or within the same hierarchical level in the Design window).

When you click the diamond on a pin, rather than using the right mouse popup menu, you get the default trace behavior. You can change the forward trace default in the Preferences dialog box.

Table 4-2. Trace Options

Option	Selected	Trace Behavior
Trace Backward (default)	Pin	Trace backward one instance.
	Instance	Trace backward one instance from each input pin.
Trace Backward Endpoint	Pin	Trace backward to endpoints, showing all circuitry in between.[1]
	Instance	Trace from each input pin backward to endpoints and show all circuitry in between.[1]
Trace Forward One (default)	Pin	Trace forward one instance.
	Instance	Trace forward one instance from each output pin.
Trace Forward Fanout	Pin	Trace forward one instance on each fanout.
	Instance	Trace forward from each output pin one instance on each fanout.
Trace Forward Endpoint	Pin	Trace forward to endpoints, showing all circuitry in between.[1]
	Instance	Trace from each output pin forward to endpoints and show all circuitry in between.[1]

Table 4-2. Trace Options

Option	Selected	Trace Behavior
Trace Backward Value	Pin	Traces the value on a selected pin back to its source. The trace continues back from the selected pin until either the origin cannot be distinguished due to a complex path, or the origin is found.

1. An endpoint is defined as a primary input, primary output, scan cell, tie gate, or black box. RAMs and ROMs are also endpoints.

Compaction of Buffers and Inverters in Traced Circuitry

You can globally enable and disable compaction. Compaction applies only to those parts of the net that you have traced/added to the window. By default, compaction of buffers and inverters is enabled.

You enable and disable compaction in the Schematics Preferences Dialog Box which you access from the **Edit > Preferences** menu. When the global compaction option is enabled, you can selectively expand/collapse buffers/inverters on a per net basis. Compaction/un-compaction traverses from the symbol in all directions until a non-collapsible instance is reached.

Compaction symbols display on individual nets indicating whether buffers/inverters are expanded or collapsed on that net. You can view the number of inverters/buffers collapsed on a particular net by positioning your cursor over the compaction symbol and viewing the popup text; the number of inverters/buffers reported reflects only the section of the net that is visible in the current view even though there may be more buffers or inverters on that net.

Note

Be aware that compaction behavior may change if you trace additional parts of the net. For example, if you add a branch between two inverters that were compacted, those inverters may no longer be compacted after the branch is added which will result in the compaction markers disappearing from that net.

When global compaction is enabled, compaction symbols have the following meaning on individual nets:

- Compacted symbol  — indicates the existence of collapsed buffers and/or inverters on the net. Clicking on the symbol, expands the buffers and/or inverters and toggles the symbol. Positioning your cursor over the symbol, displays the number of inverters/buffers collapsed on the net.

- Uncompacted symbol  — indicates that buffers and/or inverters are expanded on that net. Clicking on the symbol, collapses the buffers and/or inverters and toggles the symbol.
- No Compacted or Uncompacted symbol — indicates that no buffers or inverters exist on the net.

When global compaction is enabled, the following is true:

- Buffers are not displayed. When you trace from an instance connected to a buffer, you trace to the next non-buffer instance. You can click the Compacted symbol to turn off compaction for the individual net and view the expanded objects.
- If an even number of inverters exists, zero inverters are displayed. If an odd number of inverters is compacted, a single inverter displays. When you trace from one instance connected to an inverter, you trace to the next non-inverter instance.

Note

If you explicitly add a buffer or an inverter using the add_display_instances command, the buffer/inverter displays on the net to which it is added, irrespective of whether compaction is enabled or disabled.

Note

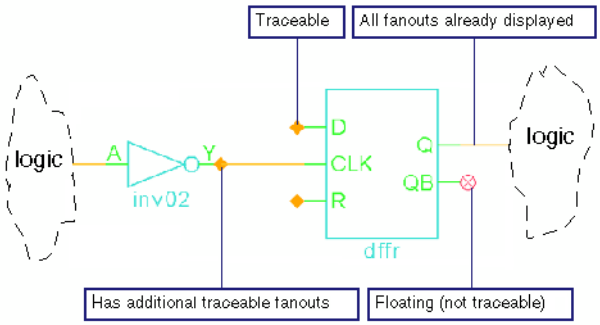
The maximum number of gates that can be inserted before requiring a confirmation includes the inverter buffer count. If you are displaying a large portion of circuitry with a lot of compaction, you might see the warning that the threshold has been exceeded even though what ends up being displayed is less than the maximum number of gates.

Tracing Signal Paths on a Schematic

You can often learn more about a circuit's behavior by displaying instances along a specific signal path (tracing).

Once one or more instances are visible in the Debug or Design window, you can trace from the instances. As shown in Figure 4-2, a diamond symbol on a pin or net indicates circuitry is connected there, is not yet displayed, and so can be traced. Table 4-2 summarizes the available trace options.

Figure 4-2. Trace Symbols



When tracing, be aware of the following:

- In the Debug window, buffers and certain inverters are not displayed by default in order to reduce screen clutter. See Compaction of Buffers and Inverters in Traced Circuitry for more information.
- In the Debug window, you can include annotated data. This is controlled by the set_gate_report command. See Annotation of Schematic Data in the Debug Window for more information.
- Instances added by the most recent trace are highlighted. You can control the highlight color using the Colors tab of the Preferences dialog box available from the **Edit > Preferences** menu.

Prerequisites

- DFTVisualizer is invoked and the Debug or Design window is open. For more information, see the "DFTVisualizer Invocation" section in this chapter.
- One or more instances must be visible in the Debug or Design window. To add instances to a window, see "Adding Instances to a Display Window" on page 49.

Procedure

Use one of the following methods to trace forward or backward:

If you want to...	Do the following:
Use the left mouse button	Move the cursor over the diamond on a pin and click the left mouse button.

If you want to...	Do the following:
Select a pin and use the **Trace** menu	1. Move the cursor over a pin (between the diamond and the instance) and click the left mouse button to select the pin. To select more than one pin, press the Shift key while simultaneously selecting the pins. For a single pin, move the cursor over the pin (anywhere, including the diamond) and press the right mouse button: this selects the pin and opens a popup menu. 2. Use the **Trace** menu or the popup menu to choose a trace option and initiate the trace. The trace will occur simultaneously from each selected pin.
Select an instance and use the **Trace** menu	1. Move the cursor over an instance and click the left mouse button to select it. To select more than one instance, press the Shift key while simultaneously selecting the instances. 2. Use the **Trace** menu or the right mouse popup menu to select a trace option and initiate the trace. The trace will occur simultaneously from all applicable input or output pins on the selected instances. **Tip**: When you open the popup menu for a single output pin (not instance), the total number of fanouts is in parentheses at the end of the "Trace Forward Fanout" choice.
Enter a command	1. Click the Transcript window to activate it. 2. Specify a path to trace by entering the add_display_instances command and specifying a beginning and ending gate using the *-Path* switch as shown here: **add_display_instances -path start_gate end_gate**

Results

In the Design window, the tool attempts to trace the path in one direction beginning at *start_gate;* the tool returns an error if *end_gate* is not found.

In the Debug window, the tool first attempts to trace the path in the direction specified by the user. If that path is not found, the tool then attempts to trace the path in the opposite direction. If that path in not found, the tool returns an error message.

Related Topics

Signal Path Tracing in the Design Window
Trace Options
Compaction of Buffers and Inverters in Traced Circuitry

Tracing a Specific Signal Value to the Source

Use this procedure to automatically trace a value on a selected pin to its source.

Prerequisites

- DFTVisualizer is invoked and the Debug window is open. For more information, see the "DFTVisualizer Invocation" section in this chapter.
- One or more instances is visible in the Debug window. See "Adding Instances to a Display Window" on page 49.
- Pin data is displayed. If needed, select an option from the **Data** menu to display pin data.

Procedure

1. Right click on the pin displaying the value you want to trace. A popup menu displays.
2. Select **Trace Backward Value**. A pop menu displays all the values on the pin. The left-most value on the pin displays at the top of the menu.
3. Select a value to trace.

Results

The value is automatically traced back from the selected pin either until a point is reached where the origin cannot be distinguished due to a complex path, or the origin is found.

Signal Path Tracing in the Design Window

The design window allows you to view and trace through the hierarchy in a multi-level design.

You can use the following methods for viewing and tracing a hierarchical design:

- Add the top level instance (/) by using any of the methods described in "Adding Instances to a Display Window" on page 49.
- When you add an instance, it is shown with all pins. Hierarchical modules added as a result of tracing, however, are shown with only the pins that are connected to other displayed instances. This allows you to trace up and down the hierarchy, displaying only pins from or through which you are tracing.

  To show all pins, move the cursor onto the instance, then press the right mouse button and choose **Show Hidden (#) Pins** from the popup menu. (The number in parentheses is the number of pins that are currently hidden.)

  To hide pins for which there are no connections currently displayed, choose **Hide Unconnected Pins** from the popup menu.

- Double-click on a hierarchical instance to display all instances inside it. The number of instances inside is shown in parentheses next to the name of the submodule.

  To hide all instances currently displayed inside a hierarchical instance, select it, then press the right mouse button and choose **Collapse** from the popup menu.

- To clean up the schematic, select an instance and choose the **Remove Other Instances** item from the right mouse button menu. Everything is deleted except the selected instance. If the selected instance is a submodule displaying instances inside it, they are kept.

Tracing Up and Down the Design Hierarchy

The design window allows you to trace through the design hierarchy in a multi-level design.

Prerequisites

- DFTVisualizer is invoked and the Design window is open. For more information, see the "DFTVisualizer Invocation" section in this chapter.
- One or more instances must be visible in the design window. If you need to add an instance, see "Adding Instances to a Display Window" on page 49.

Procedure

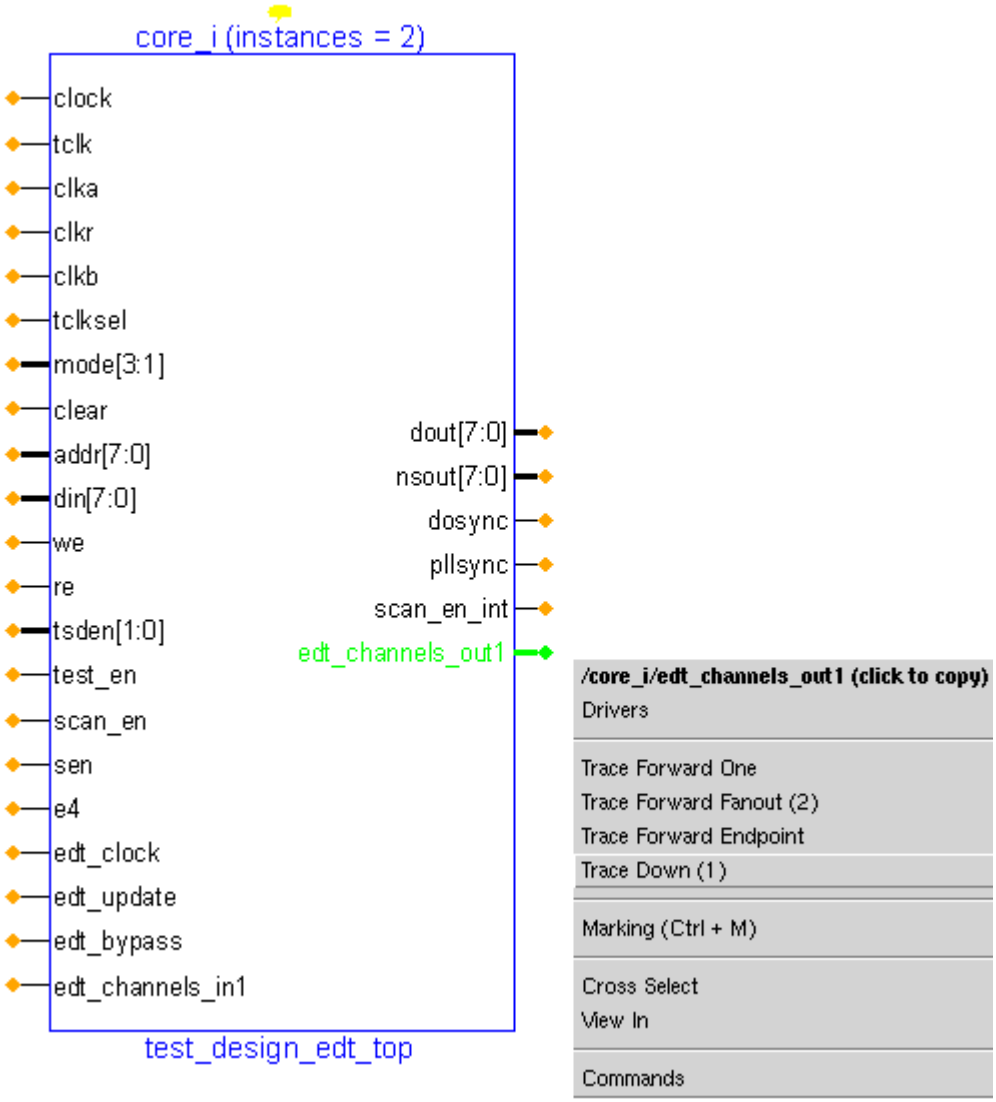
Use one of the following methods to trace up or down the design hierarchy:

If you want to...	Do the following:
Double-click	Double-click a hierarchical pin or hierarchical instance to trace down.
Use the **Trace Down Fanout** or **Trace Up** popup menu option	Select a hierarchical instance or a pin on a hierarchical instance, then use the **Trace Down Fanout** or **Trace Up** option from the right mouse popup menu.
Use the **Trace Up One** or **Trace Down** popup menu option	Select a hierarchical instance, then use the **Trace Up One** or **Trace Down** option from the right mouse popup menu.

Examples

The following example shows the result of selecting a pin and tracing down one hierarchical level and then up a level.

Figure 4-3. Tracing Down One Hierarchical Level from a Selected Pin



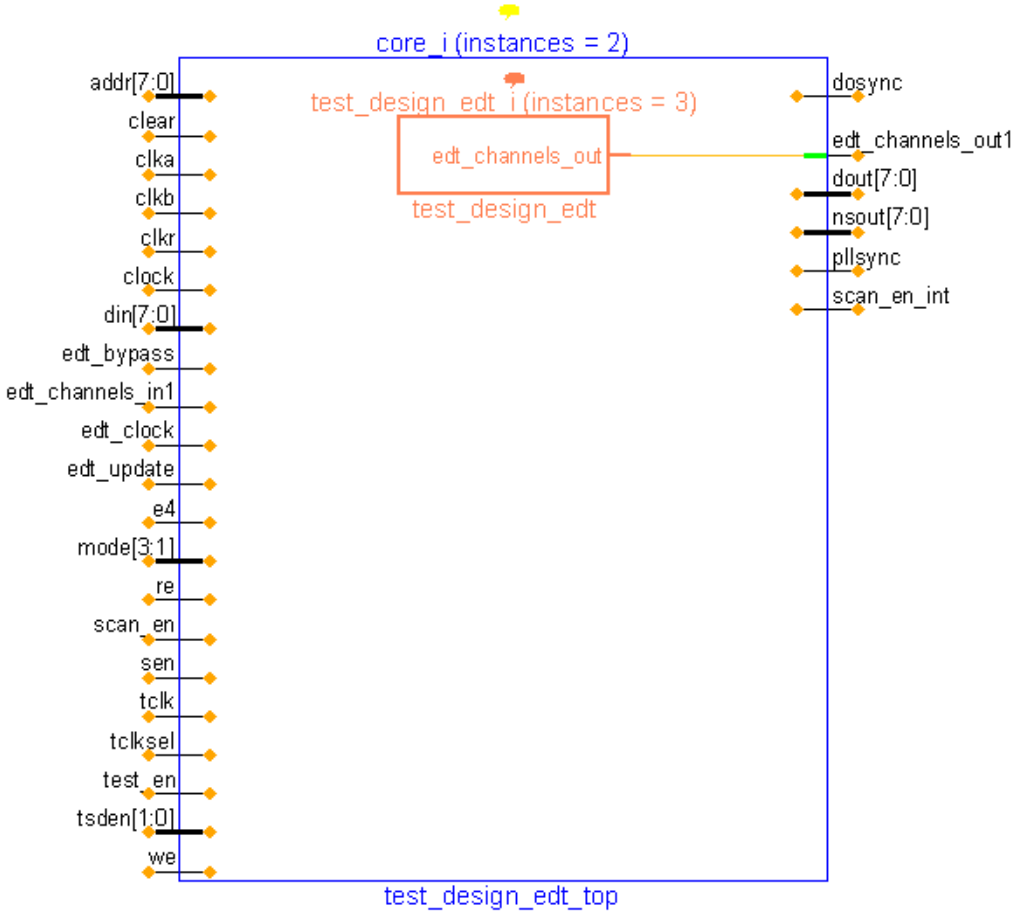
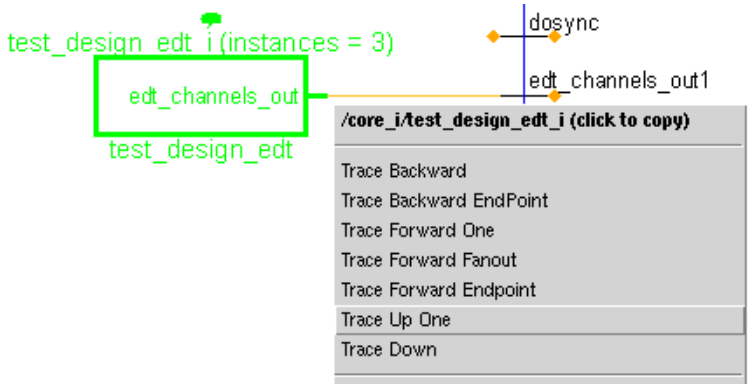
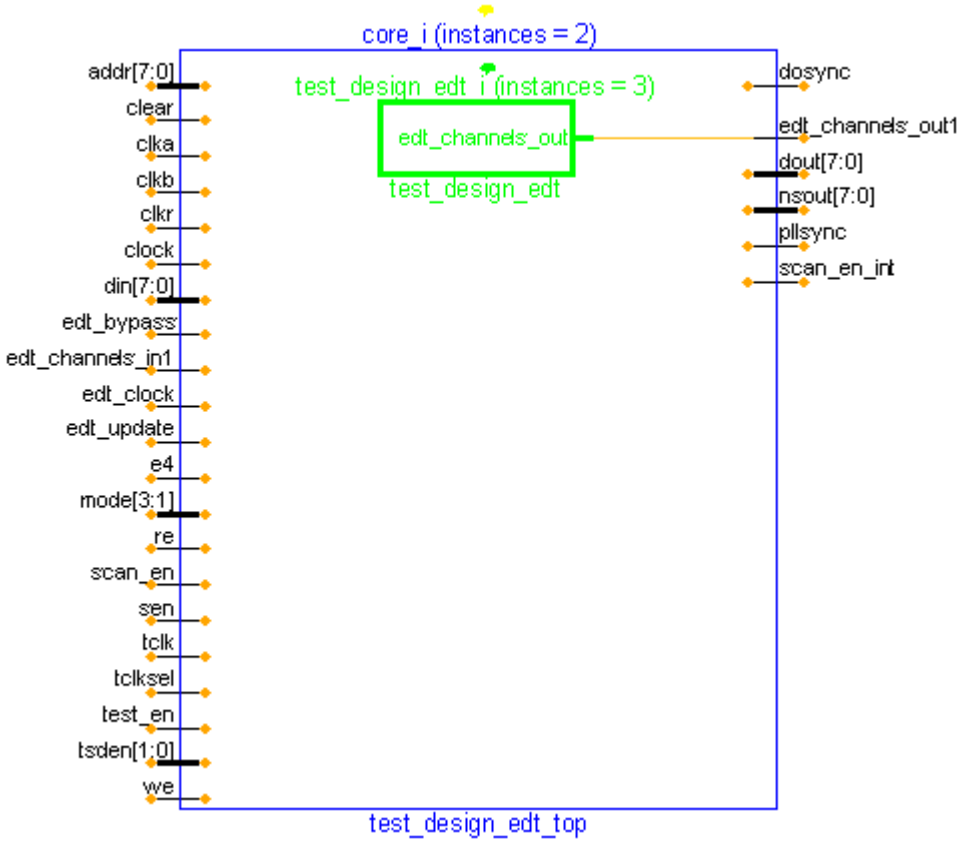


Figure 4-4. Tracing Up One Hierarchical Level from a Selected Pin





Bundling Nets

In some cases, you may not want to see all the nets that are connected between instances. For instances at higher levels of the hierarchy, where you typically have fewer instances with a large number of pins, you may be more interested in seeing between which blocks there are connections, than in seeing all the connections themselves.

To gather signals between instances into bundles represented by single thick lines, select the **Display > Net Bundle > On** menu item. The effect of net bundling is seen in the difference between Figure 4-5 and Figure 4-6.

Figure 4-5. Design Window Display with Net Bundling Off

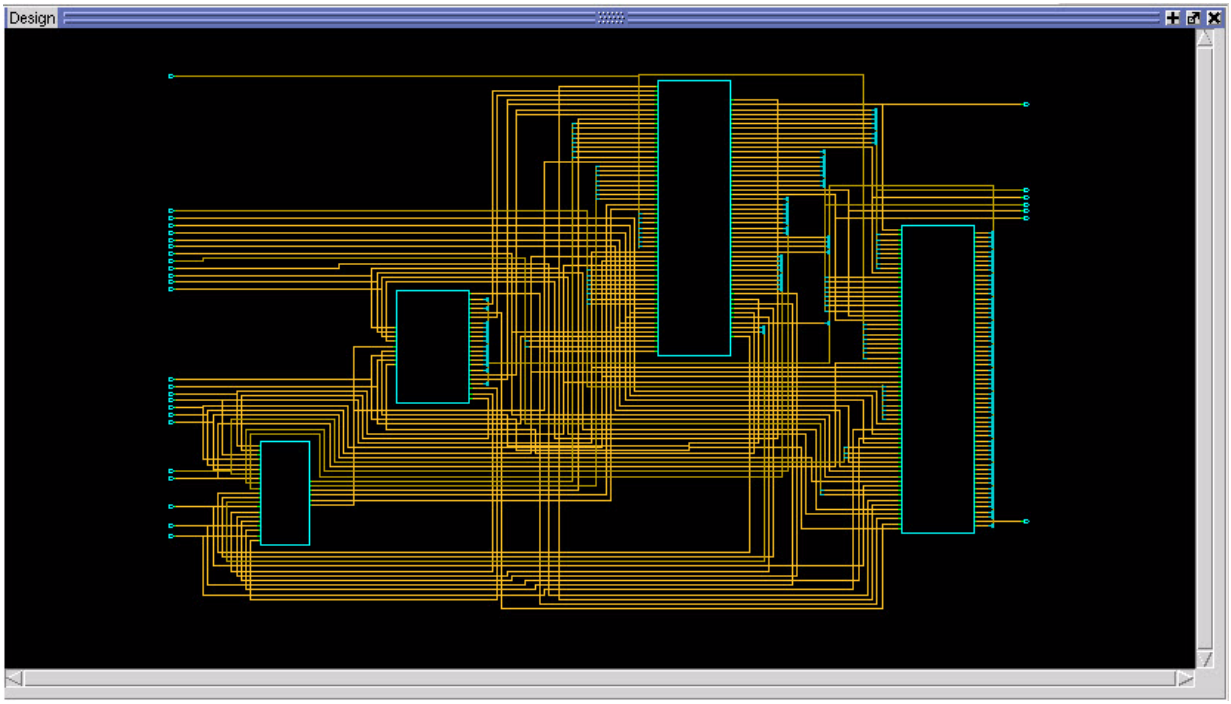
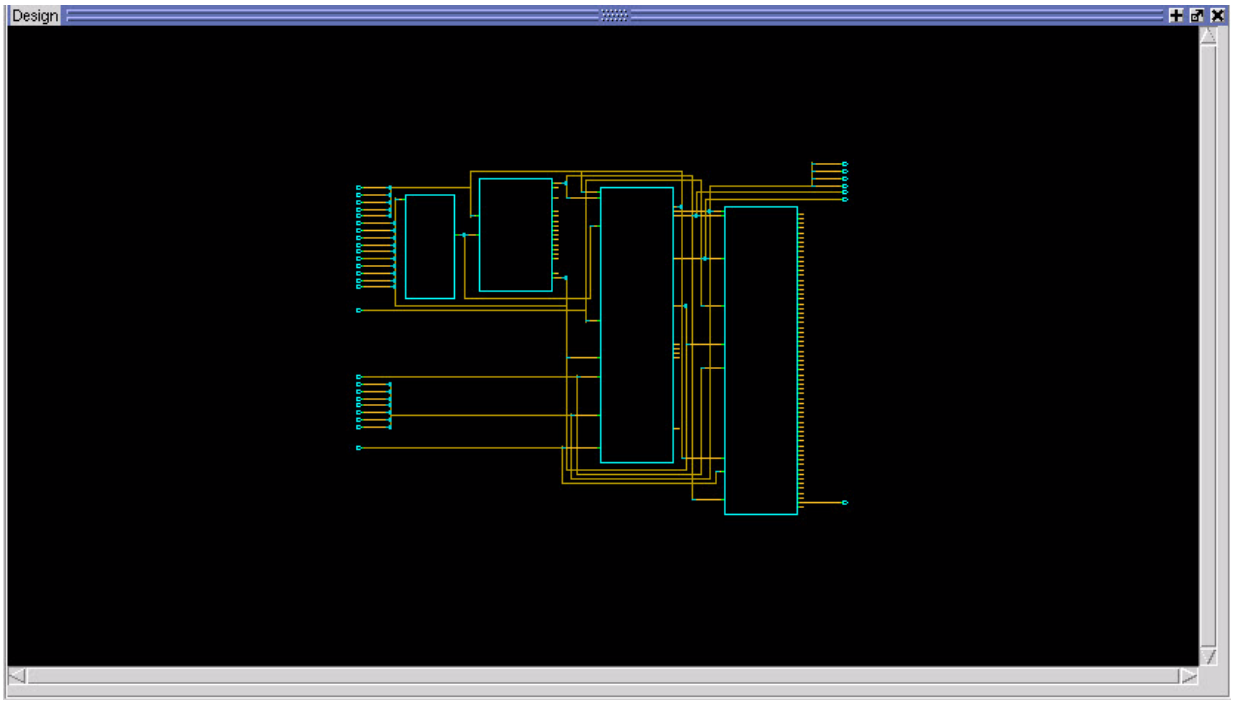


Figure 4-6. Same Display with Net Bundling On



Annotation of Schematic Data in the Debug Window

Data is automatically annotated to the schematic in the Debug window based on the current setting of the set_gate_report command which also controls the report_gates data output.

Most of the set_gate_report options are available through the **Data** menu and the most commonly used ones have buttons on the tool bar. The **Data** menu and tool bar are context sensitive: options shown might change depending on the availability of the associated data or procedure.

You can clean up the schematic by selecting an instance and choosing the **Remove Other Instances** item from the right mouse button menu. Everything is deleted except the selected instance. If the selected instance is a submodule displaying instances inside it, they are kept.

Note

If you select the right mouse button **Remove Other Primitives** menu item on a primitive gate inside an already expanded instance, all other primitives inside that instance are removed from the display but all design level instances remain.

Note

Some commands, such as analyze_drc_violation will automatically issue a set_gate_report command, so you do not have to manually enter the command or use the **Data** menu.

Adding User-Defined Annotations to Schematics

You can add user-defined text inside a callout box that can then be selectively displayed on design objects. You add user-defined text (annotations) by registering and adding attributes to objects, and then enabling the display of those attributes on the schematic. Annotations display in a callout box that is associated with a specific object or object type. The callout box moves with the object(s), and prints with the schematic.

You can use attributes in this way to easily identify a particular object, or with a collection of objects associated with an attribute using introspection (get_*) commands, as shown in step 3. You can also add annotation definitions to a dofile so that the desired text is displayed when the tool invokes. For more information, see “Working with Attributes” on page 73.

Prerequisites

- DFTVisualizer is invoked. For more information, see the “DFTVisualizer Invocation” section in this chapter.
- The current design is flattened. For more information, see the create_flat_model command.

Procedure

1. Register an attribute of type string for the object or object type that you want to annotate. For example:

   ```
   register_attribute -name notes -value_type string -default "" -object_type instance
   ```

   The name of the new attribute is “notes,” the type is “string,” the default is the empty string, and the attribute is only added to objects of type “instance.”

2. Enable the display of the new attribute in schematic windows.

   ```
   set_attribute_options -name notes -display_in_gui on -object_types instance
       -gui_marking_index 1
   ```

   The display of the “notes” attribute is “on” in schematic windows, and the background color of the attribute is set to 1 (light blue).

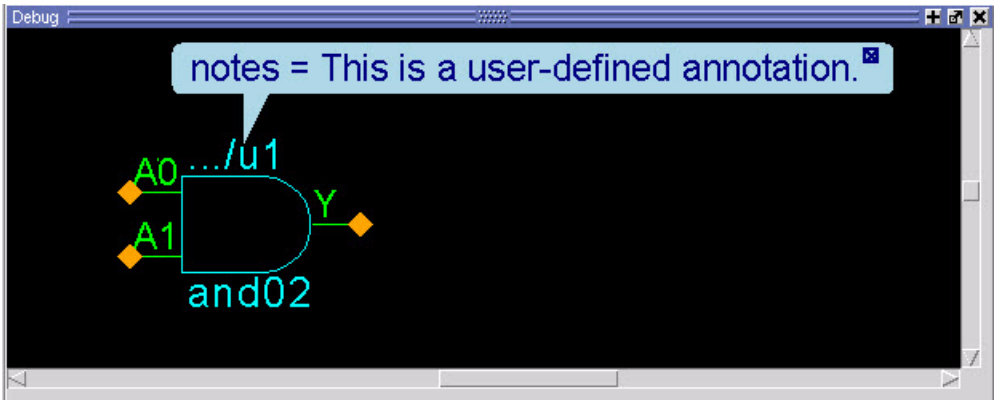
3. Assign the attribute and its value (annotation) to a set of design objects. For example:

   ```
   set_attribute_value inst_core/inst_ccu/inst_cgsri/rccu1 -name notes
       -value "This is a user-defined annotation."
   ```

   The “notes” attribute value “This is a user-defined annotation.” displays in the callout box for all instances in the rccu1 design as shown in Figure 4-7. Alternatively, you could use introspection to create a collection of objects to associate with the attribute as shown here:

   ```
   set_attribute_value [get_instances -of_modules CLK_GC_A*]
       -name notes -value "Clock Gating Cell"
   ```

Figure 4-7. User-defined Annotation



4. Optionally, add all instances with user-defined annotations to the schematic.

   ```
   add_display_instance [get_instances -filter {notes}]
   ```

   All instances that have a “notes” attribute with a non-default value will display in the Debug window.

Related Topics

add_display_instances command

Working with Attributes

Viewing K19 and K22 Simulation Data in the Debug Window

K19 and K22 simulation data is available in the Debug window for all simulation gates by using the **Data > K19** and **K22** menu items. By default, the tool displays load, shift, and capture procedures for K19 and K22 DRCs.

You can specify the number of characters of simulation data to be displayed at those gates by editing your preferences. The tool displays a [Pass] or [Fail] label at each monitor point indicating that the simulation for that monitor point has passed or failed. The tool displays a [Not sim] label for gates that are not simulated by the DRC.

For complete information on monitor points, see "Resolving DRC Issues" in the *Tessent TestKompress User's Manual*.

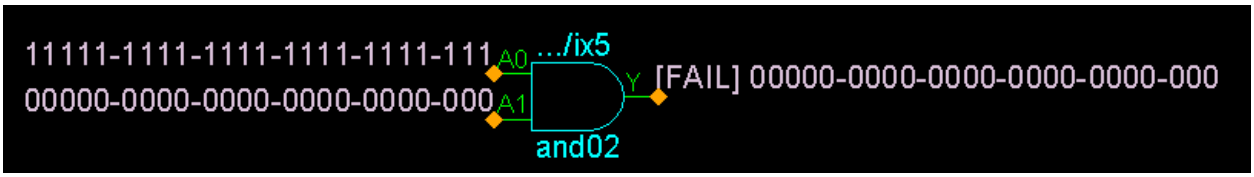
Prerequisites

- DFTVisualizer is invoked and the Debug window contains the design you are debugging. For more information, see the "DFTVisualizer Invocation" section in this chapter.

Procedure

1. Optionally, specify the number of characters of simulation data that you want to be displayed by selecting the **Edit > Preferences** menu, selecting the Schematic tab, clicking the Debug option, and entering a number in the "Show the First # Characters of Simulation Data" field; by default, the tool displays up to 32 characters of simulation data.
2. Issue the set_gate_report command or select the **Data > K22** menu item to instruct the tool to display the simulation data you want to see. For example, the following command instructs the tool to show five cycles of shift data for each gate. As a result of issuing this command, the following data is displayed in the Debug window:

```
set_gate_report k22 shift 1 5
```



3. Display the full simulation results for a gate by selecting the gate and entering Ctrl-R. The simulation data is displayed in the Transcript window as seen in this example:

```
SETUP> set_gate_report k22 shift 1 5
SETUP> //  command: report_gates
/tiny1_3/edt_i/edt_compactor_i/decoder1/ix5

//  /tiny1_3/edt_i/edt_compactor_i/decoder1/ix5  and02
//      A0      I   /tiny1_3/edt_i/edt_compactor_i/ix111/Q
//      A1      I   /tiny1_3/edt_i/edt_compactor_i/ix91/Q
//      Y       O   /tiny1_3/edt_i/edt_compactor_i/ix3/A0
//
//  Proc: shi 1 sh 2 sh 3 sh 4 sh 5 cap
//  ----- ----- ---- ---- ---- ---- ---
//  Time: i 123  123  123  123  123 o o
//        n0000 0000 0000 0000 0000 fXf
//  ----- ----- ---- ---- ---- ---- ---
//   Sim: 00000 0000 0000 0000 0000 000
//   Emu: --1-- -1-- -1-- -1-- -1-- ---
//  Mism:   *    *    *    *    *
//  Monitor: Block "tiny_1_of_3_cells" EDT decoded masking signal 2.
//
//  Inputs:
//  A0    11111 1111 1111 1111 1111 111
//  A1    00000 0000 0000 0000 0000 000
//
```

4. Examine the simulation data that displays in the Debug window and trace back to find the origins of the simulation failures.

Related Topics

K19 through K22 DRC Violations in the *Tessent TestKompress User's Manual*

Trace Options

Reporting Gates

Reporting Gates

You can report gate information (netlist and simulation data) for selected objects from the Browser, Design, Debug, Test Structures, Data, Signals, Wave, and Global Search windows.

Prerequisites

- DFTVisualizer is invoked and at least one window is open. For more information, see the "DFTVisualizer Invocation" section in this chapter.

Procedure

1. Select the object(s) to be reported on. You can use the Shift key together with the left mouse button to select multiple objects. For more information, see "Selecting Objects".
2. Press **Ctrl + R**, click the  Report Gates icon, or press the right mouse button and select **Commands > Report_Gates (Ctrl + R)**.

Results

The report_gates command executes and outputs data to the Transcript window. For more specific information on using the report_gates command to troubleshoot violations, see "Reporting Gate Data" and report_gates.

Expansion of Library Instances in the Debug Window

You can view the library primitives of a specific instance in the Debug window from design level by double-clicking on the instance. The instance expands to show the library primitives and is surrounded by a bounding box. Note, the progress bar in the lower-left of the Debug window indicates the progress of the expand operation. Double-clicking on the bounding box returns you to the instance view.

Display of Multiple Data Sets

The Data window provides simultaneous access to the several types of information controlled by the set_gate_report command. This is in contrast to the Debug window and the command line where you can report only the data corresponding to the current setting of the set_gate_report command.

For example, you can display simulation data for the load_unload and shift procedures for a particular pattern. The options are available through the **Data** menu and the buttons on the tool bar.

Working with Attributes

This section describes attributes and how to manage them in DFTVisualizer.

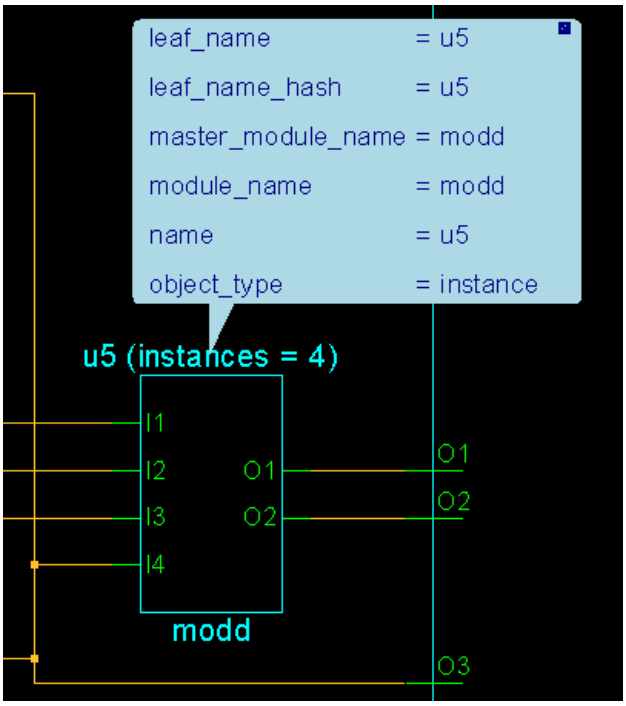
Attributes in the Design and Debug Windows

An attribute is visible in the DFTVisualizer Design and Debug windows if you have enabled the global display option for that particular attribute. By default, attributes do not display in DFTVisualizer windows.

You can enable/disable the global display of individual attributes using the Attributes Preferences dialog box, which you access from the **Edit > Preferences** menu or by clicking the  toolbar icon. You can also enable/disable the display of attributes by using the set_attribute_options command with appropriate switches.

When you globally enable the display of an attribute, the attribute displays in a *callout box* on the schematic. A callout box is a text box associated with a specific object that contains information about that object such as its attributes, as shown here in Figure 4-8. Note, in order to view attributes on the schematic, callout boxes must be showing.

Figure 4-8. Attributes in DFTVisualizer



Note

For information on how to add user-defined annotations to schematic callouts boxes, see "Adding User-Defined Annotations to Schematics" on page 68.

Table 4-3 lists the icons you can use in the Design and Debug windows to manage attributes.

Table 4-3. Icons for Managing Attributes and Callouts

Icon	Description
	Toggles between collapsing and expanding all callout boxes on the schematic. Callout markers  indicate the location of collapsed callout boxes.
	Hides all callout markers in the Design and Debug windows.
	Displays all callout markers in the Design and Debug windows.
	Displays the DFTVisualizer Preferences dialog box, Attributes tab.

DFTVisualizer displays attributes in the Design and Debug windows as follows:

- Only attributes with a value different than their default value display — regardless of whether you have globally set the attribute to display.
- Attributes with a type of Boolean display as a name only — no value is shown.
- Attributes registered on nets display; attributes inherited from ports/pins do not display because the attributes already show up on the ports/pins to which the net is connected.
- Attributes display with the background color assigned to them by the Color Index on the **Attributes** tab of the DFTVisualizer Preferences dialog box.

Related Topics

Attribute Preferences Dialog Box
Debug Window
Controlling the Display of Callouts in the Debug and Design Windows
Setting Attribute Background Display Colors
Setting Global Attribute Display Options

Setting Global Attribute Display Options

Use this procedure to specify which attributes are visible in DFTVisualizer. Note, only attributes with a value different than their default value display, regardless of whether you globally set the attribute to display.

Note

This procedure manages the display of attributes using the DFTVisualizer user interface. You can perform equivalent steps using set_attribute_options from the command line. For more information, see set_attribute_options in the *Tessent Shell Reference Manual*.

Prerequisites

- DFTVisualizer is invoked. For more information, see "DFTVisualizer Invocation" on page 40".

Procedure

1. Display the **Attributes** tab of the DFTVisualizer Preferences dialog box by doing *one* of the following:
   - Select **Edit > Preferences** and click the **Attributes** tab.
   - Activate either the Design or Debug window, and click the  icon.
2. Select an attribute in the left-hand column of the DFTVisualizer Preferences dialog box.
3. Click the  button to move the selected attribute into the Displayed Attributes column.

Note

By default, the attribute inherits the background color currently showing on the Color Index button. See "Setting Attribute Background Display Colors" to change this color.

4. Click OK.

Setting Attribute Background Display Colors

Use this procedure to specify the background color of attributes displayed in DFTVisualizer.

Prerequisites

- DFTVisualizer is invoked. For more information, see the "DFTVisualizer Invocation" on page 40.

Procedure

1. Display the **Attributes** tab of the DFTVisualizer Preferences dialog box by doing one of the following:
   - Select **Edit > Preferences** and click the **Attributes** tab.
   - Activate either the Design or Debug window, and click the  icon.
2. Select the attribute in the Displayed Attributes (right-hand) column whose background color you want to modify.
3. Click the Color Index button to display the color map and select the desired color.
4. Click OK.

Results

The background color of the specified attribute is set.

Controlling the Display of Callouts in the Debug and Design Windows

Use this procedure to control the display of callout boxes and markers in the Debug and Design windows. Attributes are displayed in callout boxes.

Prerequisites

- DFTVisualizer is invoked and a schematic window is active. For more information, see "DFTVisualizer Invocation" on page 40.

Procedure

Use one of the following icons to control the display of callout boxes and markers.

Click	To...
	Toggle between collapsing and expanding all callout boxes on the schematic.
	Hide all callout markers on the schematic.
	Display all callout markers on the schematic.

Attribute Preferences Dialog Box

To access: select **Edit > Preferences** and click the **Attributes** tab, or activate either the Design or Debug window and click the  toolbar icon.

The DFTVisualizer Preferences (**Attributes** tab) dialog box allows you to set preferences associated with the display of attributes in the Design and Debug windows.

Fields

Table 4-4. DFTVisualizer Preferences Dialog Box, Attributes Tab

Field	Description
Object Type	Specifies the object type to which the attributes listed in the left column of the dialog box are registered.
Object Attributes (left column)	Specifies the attributes defined for the selected Object Type. The attributes are divided into two groups: user-defined attributes (listed first) and predefined attributes (listed below the ---- divider). [1]
Displayed Attributes (right column)	Specifies the attributes that are visible in DFTVisualizer.[2] An object selected in this column displays in red text. Note, only attributes with a value different than their default value display on the schematic, regardless of whether you have globally set the attribute to display.
Color Index	Specifies the background color to use when displaying the selected attribute in schematic windows.[3] DFTVisualizer provides ten background colors.

1. These attributes use the following option: set_attribute_options -display_in_gui OFF.
2. These attributes use the following option: set_attribute_options -display_in_gui ON.
3. Attribute background color is specified by: set_attribute_options -gui_marking_index <index>.

Usage Notes

- Add and remove attributes from the Displayed Attributes column by clicking the  and  arrow buttons or by double-clicking on the column entry.
- Select multiple entries from a list using the Ctrl or Shift key together with the select mouse button.
- Set/change the background color of the attribute using the 

button.

Related Topics

Setting Attribute Background Display Colors Setting Global Attribute Display Options

Working With Specifications in the Configuration Data Window

The Configuration Data window allows you to view specifications and to modify the configuration tree elements and their properties using a graphical interface.

Modifying the Contents of the Configuration Data Window

You use the Configuration Data window to view and modify specifications.

Prerequisites

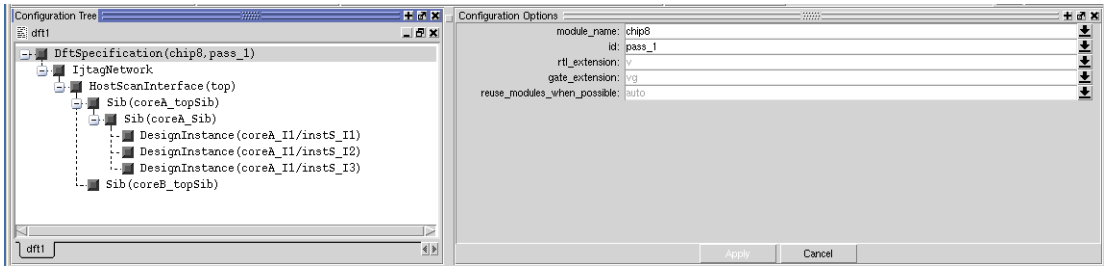
- Tessent Shell is invoked.

Procedure

1. Open the Configuration Data window by issuing the display_specification command to view and modify a DftSpecification that was read in using the read_config_data command, or issue the “display_specification -create” command to create one from scratch using the graphical interface.

   The Configuration Data window opens and displays the existing configuration data as shown in this example.

Figure 4-9. Configuration Data Window



2. Select an element from the Configuration Tree, click the right mouse button, and select one of the **Add** > menu items to add.

   The **Add >** menu displays a context-sensitive list of elements that shows the type of elements you can add based on the type of element you selected.

3. Specify the parameters of the newly added element in the Configuration Options panel:
   - Enter the values of the element’s listed properties; you can use the icon to select from pre-defined values for that object.
   - Click Interface [+] to define parameters of the element’s interface.
   - Click DataInPorts [+] and DataOutPorts [+] to define the number of ports on the element, their names, and connections.
   - Click Attributes [+] to define arbitrary attributes on the element’s ICL module.
4. Click **Apply** to update the specification.

5. When you have created and fully defined all of the elements that you want to add, click the  icon to validate the current specification and look for errors. You can also issue the "process_dft_specification -validate_only" command which is an equivalent action.
6. After you have validated the specification and resolved all errors, click the  icon to execute the specification. You can also issue the process_dft_specification command which is an equivalent action.

Results

You have created and defined new elements in your specification. You have validated the correctness of these changes, and created the specified elements by executing the specification.

Related Topics

Configuration Data Window

Modifying the Contents of the Configuration Data Window

Adding a Test Data Register to a SIB Example

Adding a Multiplexer to a SIB Example

Adding a Test Data Register to a SIB Example

You can add a Test Data Register (TDR) to a SIB using the graphical interface of the Configuration Data window.

Prerequisites

- Tessent Shell is invoked.

Procedure

1. Open the Configuration Data window from either the Tessent Shell command line or from DFTVisualizer.
2. Select the SIB to which you want to add the TDR.
3. Click the right mouse button and select **Add > Tdr**.

   You can also move the TDR by selecting it and dragging it to the desired location.You can also change its position using the **Move Up** and **Move Down** menu items.
4. In the Configuration Options panel, enter a name in the id field. You can specify the desired value of any of the other displayed fields.
5. Click DataInPorts [+] and DataOutPorts [+] and modify the value of the "count" property in each to change the number of ports on the TDR.
6. Click **Apply** to update the specification.

Results

You have added a TDR to a SIB and defined the number of ports on the TDR.

Related Topics

Configuration Data Window

Modifying the Contents of the Configuration Data Window

Adding a Multiplexer to a SIB Example

You can add a ScanMux to a SIB using the graphical interface of the Configuration Data window.

Prerequisites

- Tessent Shell is invoked.

Procedure

1. Open the Configuration Data window from either the Tessent Shell command line or from DFTVisualizer.
2. Add a ScanMux by selecting the SIB to which you want to add the ScanMux, clicking the right mouse button, and then selecting **Add > ScanMux**.

   You can also move the ScanMux by selecting it and dragging it to the desired location.You can also change its position using the **Move Up** and **Move Down** menu items.

   a. Name the ScanMux by entering a name in the id field of the Configuration Options panel. You can specify the desired value of any of the other displayed fields.

   b. Define the ScanMux connections by clicking Interface [+] and entering the connection for the Mux select line. If a TDR is defined, you may use any TDR data output port to control the mux select signal.

   c. Click **Apply** to update.
3. Add the first input port to the new ScanMux by positioning your cursor on the new ScanMux, clicking the right mouse button, and selecting **Add > Input**.

   a. Select the new input port and enter a number in the int field of the Configuration Options panel.

   b. Click **Apply** to update.

   c. Add a TDR that will be accessed when this mux input port is selected by positioning the cursor on the Input port, clicking the right mouse button, and selecting **Add > Tdr**.

i. Name the TDR by selecting the TDR and entering a name in the id field of the Configuration Options panel.

ii. Connect the TDR to the design by:

a. Clicking the left mouse button on the DataInPorts [+] button, clicking Connections [+], and then clicking the plus sign [+]. Enter the range that matches the design instance data bus size into the range field (for example "7:0"), and enter the pathname to the design instance in the pin_name field.

b. Clicking the left mouse button on the DataOutPorts [+] button, clicking Connections [+], and then clicking the plus sign [+]. Enter the range that matches the design instance data bus size into the range field, and enter the pathname to the design instance in the pin_name field.

4. Define the connections for the second input port of the ScanMux using the instructions in step 3.

Results

You have created a ScanMux and defined the select signal. You have added two Input wrappers to the ScanMux, each of which have a TDR defined.

Related Topics

Configuration Data Window

Modifying the Contents of the Configuration Data Window

Working with a Design

This section describes how to perform the analysis tasks in DFTVisualizer.

Analysis of a DRC Violation

You can use the analyze_drc_violation command from DFTVisualizer to report additional information on any of the following DRCs:

A1-A16
C1-C17

D1-D12
E2-E11, E14
F rules
R2, R4, and R5
S1-S4
T2-T6, T12, T16, T17,T24-26
W20-W31, W34-W38

Note

The following DRC rules do not have additional information that can be displayed with DFTVisualizer:

B1-B16
E1, E12, E13
G rules
K rules
P rules
T1, T8-T11, T13-T15, T18-T23
W1- W19, W32, W33

For more information on specific DRC rules, see "Design Rule Checking."

Step 1 - Run an Analysis of the Violation

Built into the tools (except for Tessent Diagnosis) is the capability to analyze DRC violations. At the end of the analysis, a schematic of the circuitry and/or instance associated with the violation displays in the Debug window. Data from the analysis, consistent with what the report_gates command would report for the displayed gates, is annotated on the schematic. Instructions for analyzing a violation are provided in "Running the Analysis" on page 86.

Step 2 - Find the Source of Problem Signals

If the cause of a DRC violation is not immediately apparent from reviewing the schematic in the preceding step, there are several things you can do to improve your understanding of the violation. These are primarily techniques for tracing the source of problem signals and are discussed in the following topics under "Performing Basic Tasks" on page 44:

- Tracing Signal Paths on a Schematic
- Signal Path Tracing in the Design Window
- Display of Multiple Data Sets

Example

The following example shows one way to utilize DFTVisualizer to trace the source of a bad signal value that produced a T3 DRC violation.

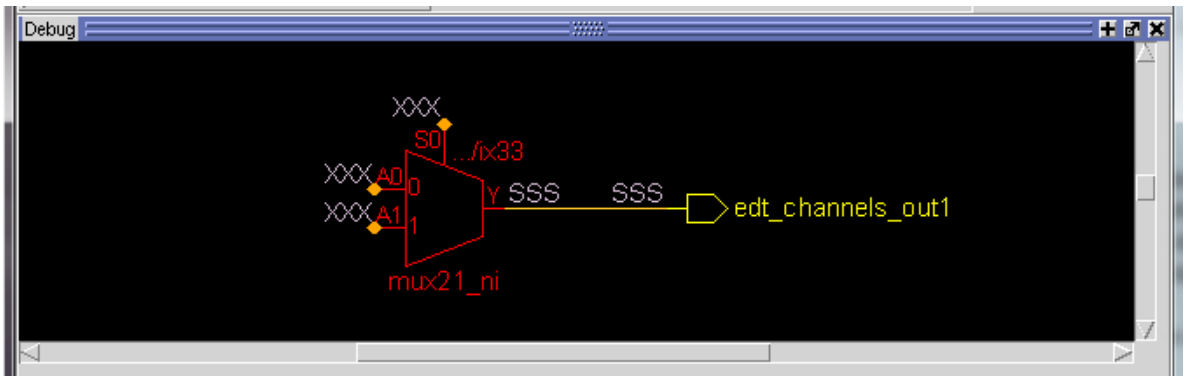
A DRC error is reported in the tool's session transcript:

```
...
//  Error: Scan chain chain1 blocked at gate
/bsr_i1/bsc_edt_channels_out1/ix33 (972) after tracing 0 cells. (T3-1)
//  Error: Rules checking unsuccessful, cannot exit SETUP mode.
//         'DOFile fs.do' aborted at line 19
```

1. Run an analysis of the violation:

```
SETUP> analyze_drc_violation t3-1
//  command: open_visualizer -display debug
//  Note: Gate report now set to trace.
```

Figure 4-10. Initial DRC Analysis Display

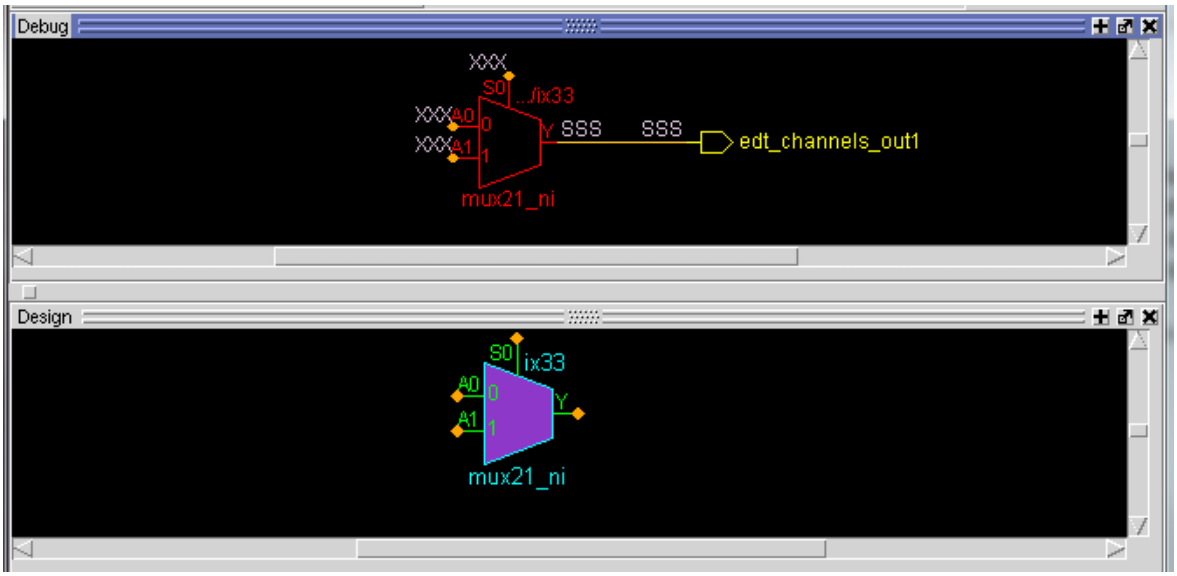


Because a trace error was generated (T3-1), the analyze_drc_violation command automatically sets gate reporting to Trace. The trace report shows the identified scan path nodes as 'S'.

- The Ss on the connection between the multiplexer (mux) and the primary output pin edt_channels_out1 indicate the scan path can successfully trace back to the mux.
- The Xs on the Select line of the mux result in Ss on the mux output regardless of the fact that the mux input is ambiguous.

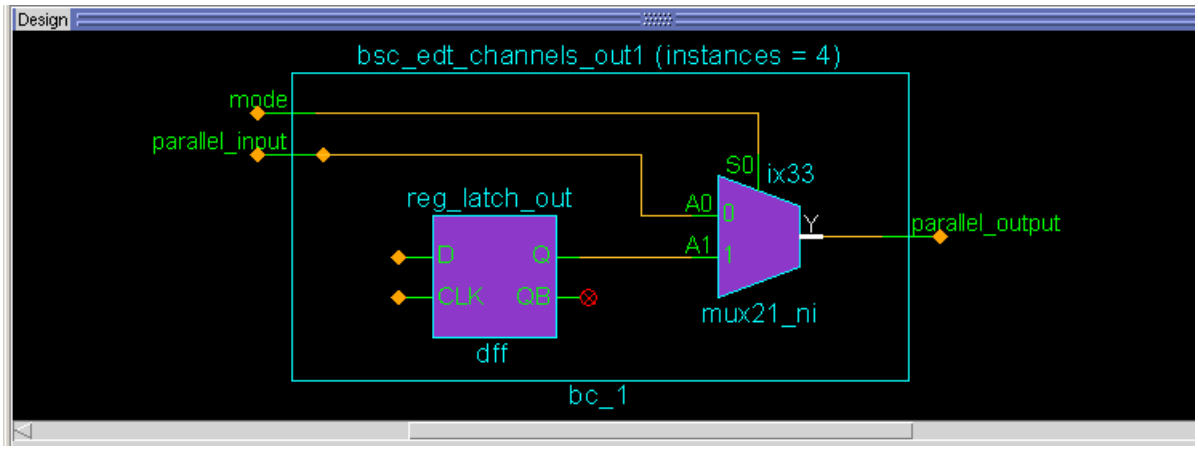
2. To maintain a netlist perspective in the following steps, copy the multiplexer to the Design window by selecting the multiplexer and choosing **View In > Design** from the popup menu. The multiplexer is copied to the Design window as shown in Figure 4-11.

Figure 4-11. Instance Copied to the Design Window



3. Determine the cause of the Xs on the mux Select line by selecting the Select line in the Design window and performing a backward trace. As you pass each hierarchical boundary, by default, the tool hides any pins that are not in the traced path. You can select a hierarchical instance and select **Show Hidden Pins** from the popup menu.

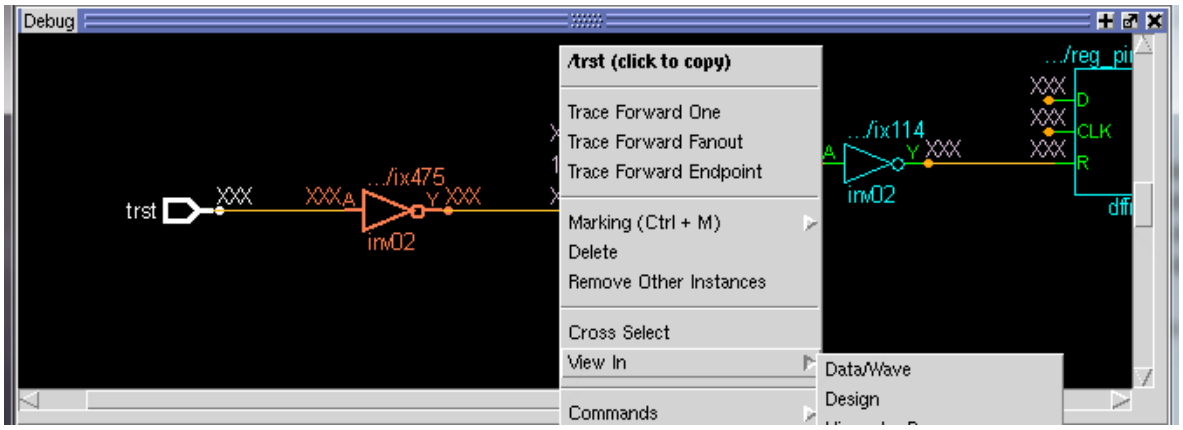
Figure 4-12. Tracing Back Using the Design Window



4. When the trace reaches a memory element, check its simulation values by selecting it and viewing it in the Debug window (right mouse button **View In > Debug**).

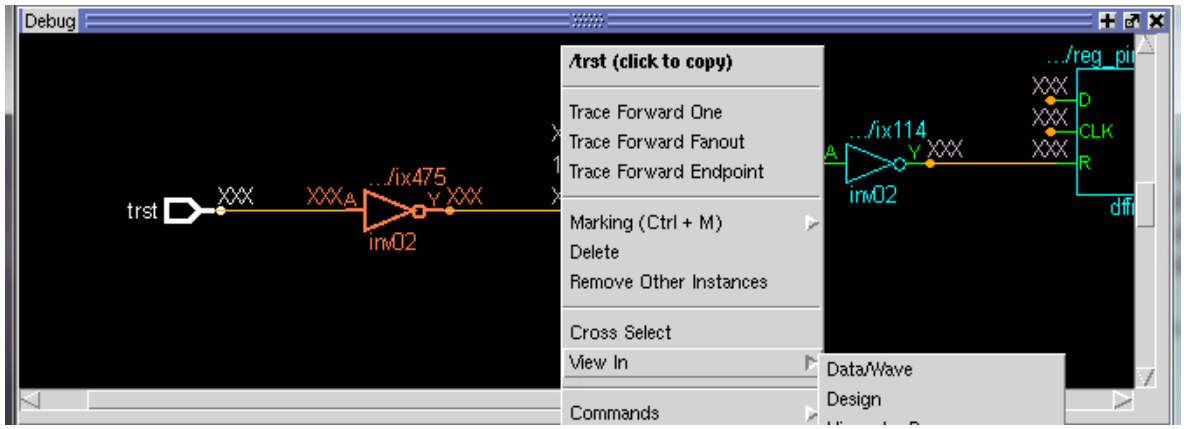
   Note, the Xs on NOR gate *.../ix124*. As expected, there are Xs on the output. We will trace the *A0* pin of this instance. The Xs on either the clock (CLK) or reset (R) input of instance *.../reg_pinst_0_* would cause Xs on the output.

Figure 4-13. Finding Xs on the Reset Input of a Memory Element on Trace Path



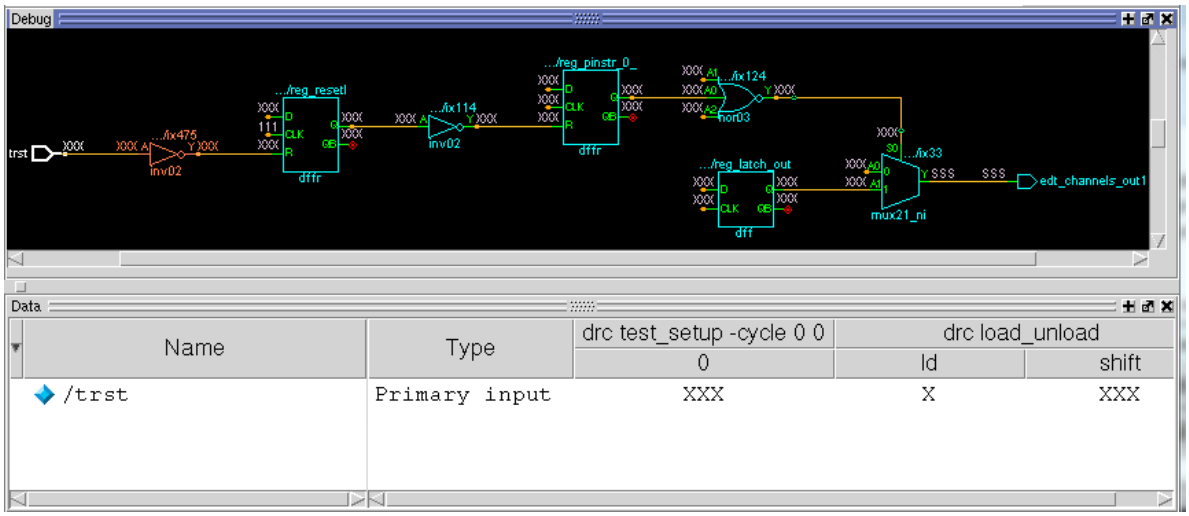
5. Continue the backward trace from the reset input of the memory element.
6. Because the *trst* primary input is sourcing Xs, select the PI instance and add it to the Data window (right mouse button **View In > Data/Wave**).

Figure 4-14. Completing the Trace to a PI and Reconfirming It is X Source



7. From the Data window, use the **Data** pulldown menu to look at simulation values during different test procedures. The Data window reveals that the *trst* pin is not initialized during the test_setup cycle.

Figure 4-15. Viewing Additional Simulation Data for the PI



Step 3 - Apply a Remedy and Rerun DRC

Once you have determined the reason for the DRC, you can make appropriate changes in your test procedure file, tool setups, or design to correct it.

Example

To remedy the erroneous signal of the preceding example, a force statement is added to the test_setup procedure to initialize the *trst* pin. DRC is then rerun to confirm the fix:

Before

```
procedure test_setup =
   timeplate bsda_timp1;
   cycle =
   ...
      force tck 0;
      force tdi 0;
      force tms 1;
   pulse tclk;
   end;
end;
```

After

```
procedure test_setup =
   timeplate bsda_timp1;
   cycle =
   ...
      force tck 0;
      force tdi 0;
      force tms 1;
      force trst 0;
   pulse tclk;
   end;
end;
```

Running the Analysis

Use this procedure to analyze a particular DRC violation.

Prerequisites

- The identification (*rule_id*) and specific occurrence number (*occurrence#*) of the DRC violation.

The *rule_id* is provided in parentheses at the end of each DRC violation summary message in the tool's session transcript. To get *occurrence#*s, enter a report_drc_rules command with a *rule_id* argument. This displays each occurrence associated with that *rule_id* rule. Each occurrence includes its *rule_id-occurrence#* in parentheses.

- DFTVisualizer is open.

Procedure

1. Select **Tools > Analyze DRC**. The Select a Violation ID dialog box displays.
2. Select a rule ID from the Failed DRCs list that you want to debug. All violation occurrences for that rule ID are displayed.
3. Double-click on any of the *occurrence#s* of the DRC in the Specific IDs list to start analyzing them in the Debug window. Alternatively, you can select the specific violation occurrence and click **Analyze** or **Analyze & Close** to see the analysis of the violation. Be aware that you can also issue the analyze_drc_violation command with the *rule_id-occurrence#* argument at the DFT tool's command line. For example:

   analyze_drc_violation c3-1

Tip: With either of these DRC analysis methods, the DRC ID is highlighted as a pink hyperlink in the Transcript window. However, if the DRC is a P (Procedure) or W (Timing) violation, clicking the DRC-ID-*occurrence#* opens the test procedure file in the DFTVisualizer Text Editor window and highlights the line generating the DRC violation.

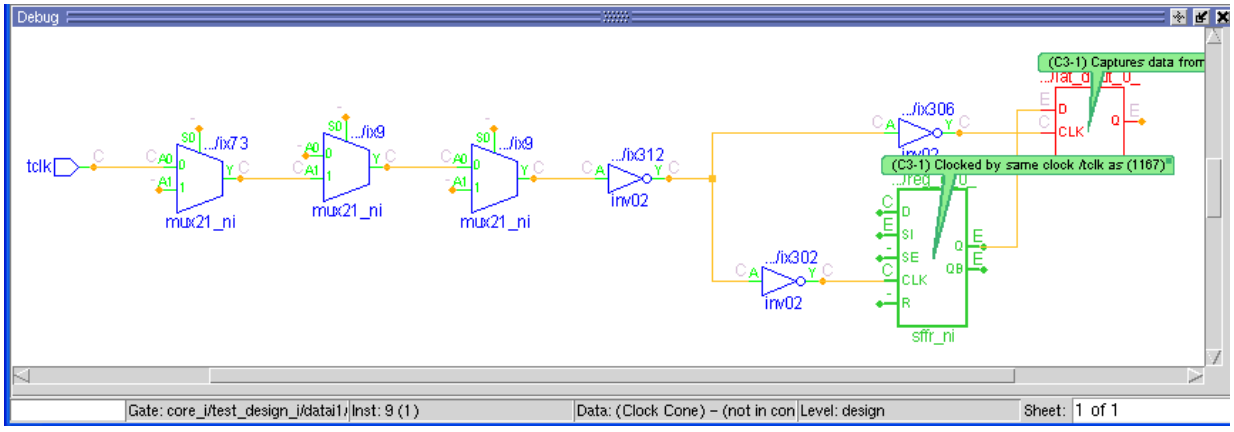
Note

In some situations, the tool's analysis may require significant CPU run time. You can interrupt the process and return to the command prompt using the Control-C key. Intermediate results are not retained if you interrupt the analysis.

Examples

Figure 4-16 shows the DFTVisualizer display resulting from analysis of a C3 violation:

Figure 4-16. DFTVisualizer Display Example



For some violations, the data annotated to the initial display will be enough for you to determine the exact cause of the violation. In the preceding display, the clock cone data (Cs and Es) show that under certain timing conditions (if captured data propagates through the .../reg_pstate_2_ flip-flop to its Q output in less than half a clock cycle for example), data will pass through both memory elements in a single clock cycle.

Related Topics

C3

Assessing Test Coverage in the Browser

The Browser window allows you to display test coverage, fault coverage, and DRC statistics for hierarchical design blocks using the **Hierarchy**, **Library**, and **Clocks** tabs. You use this procedure to display test coverage statistics. (The steps are similar for fault and DRC statistics.)

Prerequisites

- An ATPG process is completed and DFTVisualizer is invoked. For more information, see the "DFTVisualizer Invocation" section in this chapter and "The ATPG Process" in the *Tessent Scan and ATPG User's Manual*.

Procedure

1. Choose the **Windows > Browser** menu item in DFTVisualizer. The Browser window opens and displays the top level design instance in the **Hierarchy** tab as shown in Figure 4-17.

2. Optionally, select another Browser tab to use by selecting either the **Library** or the **Clocks** tab.

3. Choose **Data > Coverage Data > Test Coverage** or click **tc** on the toolbar. A test coverage column containing statistics for the displayed instance is added to the active Browser tab.

Note

In the Library Browser, test and fault coverage statistics include two subcolumns: one contains the maximum coverage of all instances of library models and the other contains the average test coverage of all instances of the model. Once a library model is expanded, the coverage for each instance of that model is displayed in both the Max and Avg columns.

4. Click the plus sign (+) next to the instance, library model, or clock name to expand the design hierarchy. You can see the blocks at the next level of hierarchy and the coverage for each as shown in Figure 4-18.

Tip: When showing high level statistics, it can be useful to just focus on the submodules (blue boxes). To do that, choose **Display > Group Instances > On**. This will group all gates (gray boxes) by themselves in a separate block marked $$Gates$$.

5. Repeat the preceding step for the additional blocks, looking for blocks with relatively low test coverage.

Examples

Figure 4-17. Browser Default Display Showing the Top-level Design

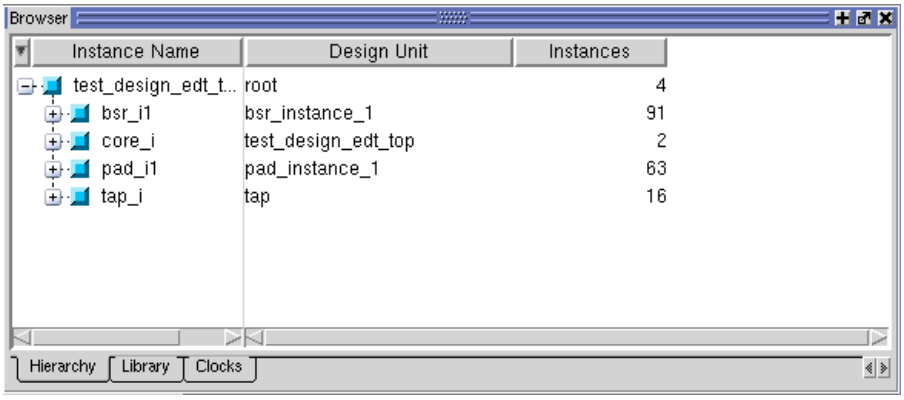
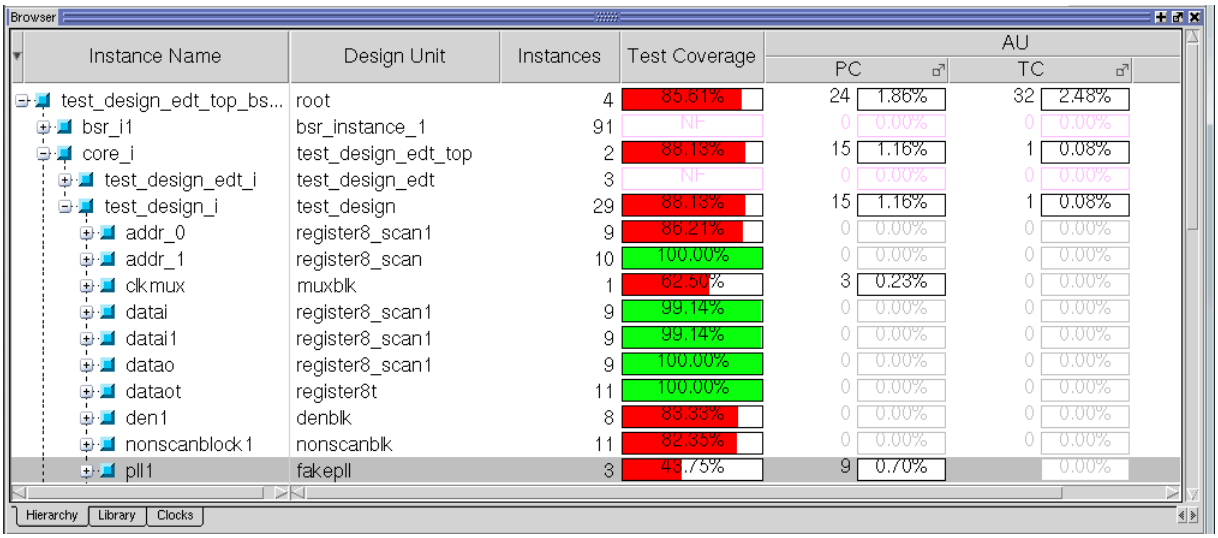


Figure 4-18. Browser Showing a Block with Low Test Coverage



Performing Clock Domain Analysis in the Browser

The Browser window allows you to display test coverage, fault, and DRC statistics for hierarchical design blocks using the **Hierarchy**, **Library**, and **Clock**s tabs. Use this procedure to analyze the impact of individual clocks on total test coverage and fault statistics.

Prerequisites

- An ATPG process is completed and DFTVisualizer is invoked. For more information, see “DFTVisualizer Invocation” on page 40 and “The ATPG Process” in the *Tessent Tessent Scan and ATPG User’s Manual*.

Procedure

1. Choose the **Windows > Browser** menu item in DFTVisualizer. The Browser window opens and displays the top level design instance in the **Hierarchy** tab.
2. Click the **Clocks** tab. The window displays a list of the clocks in the design in descending order of test coverage as shown in Figure 4-19. That is, the clock with the highest percent of test coverage is listed at the top of the list and the clock with the least coverage is listed at the bottom of the list. By default, the window displays three columns of data for each clock:
   - Clock attributes such as off state, constraints, and internal/external clock.
   - Total faults in each clock domain and the percentage of all faults in the design.

Note

If a fault is in multiple clock domains, the fault is attributed to each clock domain. Because of this, it is possible that the sum of all fault percentages (faults in a clock domain as a percent of the total faults in the design) can exceed 100%.

- Test coverage.

3. Choose **Data > Faults** and select any additional faults you want to display from the menu. Columns containing the faults you specified are added to the active Browser tab as shown in Figure 4-20.
4. Choose **Data > Coverage Data** and select the additional statistics you want to display from the menu. Columns containing the statistics you specified are added to the active Browser tab.
5. Click the plus sign (+) next to a clock name to expand the clock and see the individual instances within that clock's domain. You can see the instances at the next level of hierarchy and the test coverage and additional statistics for each as shown in Figure 4-20. Only instances with faults in the expanded clock domain are displayed.

Examples

Figure 4-19. Browser Display Showing the Clock Tab

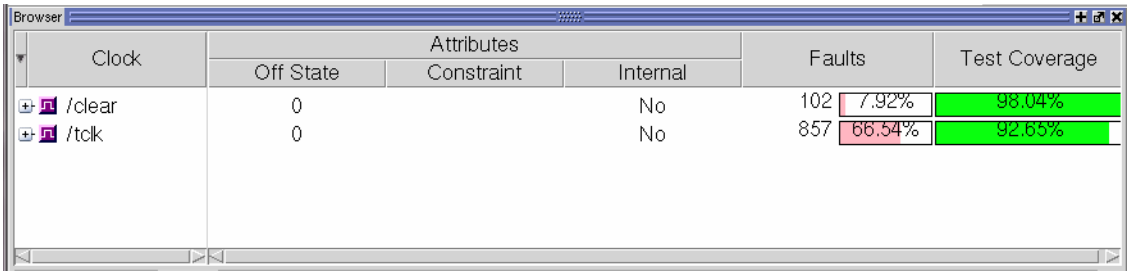
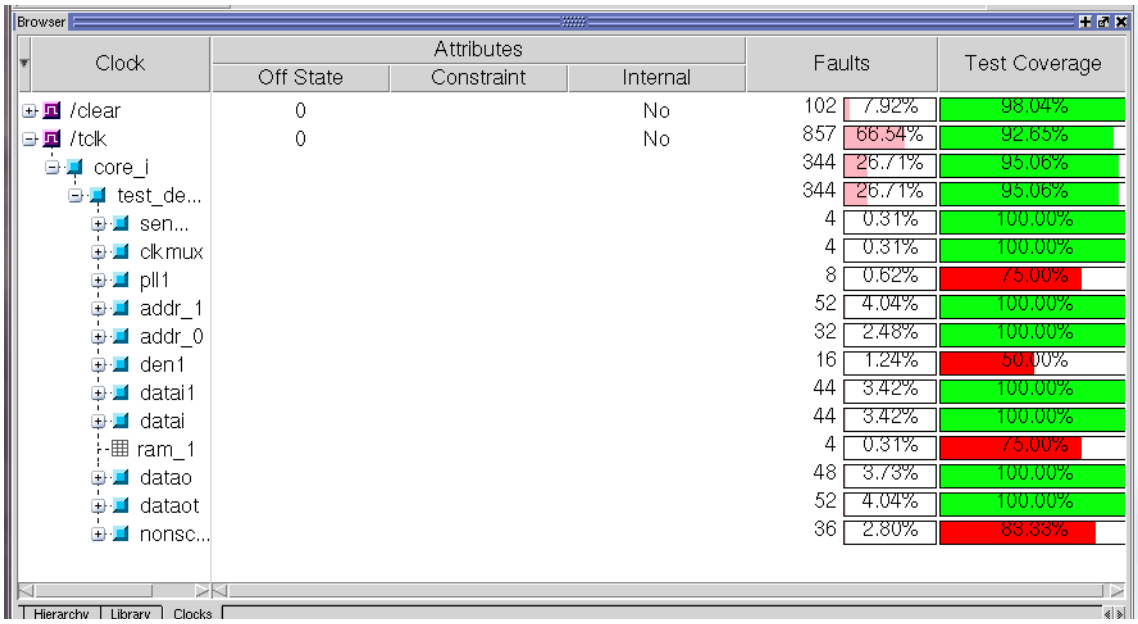


Figure 4-20. Browser with Expanded Clock Domains



Analyzing a Fault and Displaying its Location

The analyze_fault command allows you to identify why a fault is not detected. You can execute this command, as well as several related reporting commands from the DFTVisualizer Fault Analysis dialog box. You also can add the instance where the fault is located to the Debug window automatically, as part of the analysis.

Prerequisites

- You must have added faults to the current fault list and identified the instance where the pin whose faults you want to analyze is located.

Procedure

1. Choose **Tools > Analyze Faults** from the DFTVisualizer main menu. The Debug window opens if not already open, along with the DFTVisualizer Fault Analysis dialog box.
2. In the Faults & Statistics Options field, enter an instance pathname in the Specific Instance entry box using one of the following methods:
   - Select the instance in another window such as the Browser; the instance pathname will automatically appear in the entry box.

Tip: You can also select an instance in another window before you open the dialog box and it will automatically appear in the entry box.

- Copy and paste the pathname from somewhere else (session transcript for example, or the right mouse button pathname selection in another window).
- Alternatively, select the Entire Design option if you want to report on all the faults in the design.

3. In the Report Faults Options field, select an option for the Fault Type and choose the Fault Class using the dropdown list. If you want to display the fault class for equivalent faults, select that checkbox as well.
4. Click the Report Faults button to list faults of the selected type and class; the list appears in the display area of the Reported Data field.

   For convenience, there are also buttons for running the report_statistics and report_sequential_fault_depth commands.
5. In the Reported Data field, click the pin pathname of a fault you want to analyze. The pathname appears in the Fault entry box at the bottom left side of the Reported Data field.
6. Select a Stuck-at option, then click one of the Analyze buttons.

Results

The resulting analysis appears in the tool's session transcript and is equivalent to the analyze_fault command. If you chose the graphical analysis, the instance the pin is on is added to the Debug window automatically.

Determining Test Stimulus

You can use the report_test_stimulus command to create "what if" scenarios when debugging test coverage and other issues. This command attempts to generate a pattern that sets one or more pins to the value(s) you specify. You can execute this command graphically in the Debug window using this procedure.

Prerequisites

- The tool must be in ATPG, Fault, or Good mode.
- The instance(s) with the pins for which you want to specify values must be displayed in the Debug window.

Procedure

1. In the Debug window, select one or more pins as described in "Selecting Objects" on page 49.

Tip: If you want to specify values for multiple pins on an instance, selecting the instance is a fast way to automatically select all its pins; you can then get DFTVisualizer to ignore unwanted pins in step 3 below.

2. From the right mouse button menu, choose **Commands > Set Value**. The Set Value dialog box appears with a Pin Name field for each selected pin; each field contains the pathname to the pin.
3. Select a value for each pin using the Pin Value dropdown lists. Do not specify a Pin Value for those pins you want the tool to ignore.
4. Optionally, click "Append to previously set value(s)" to specify to append these new values to values specified in previous executions of this dialog box. The default is to discard the settings from previous executions of this dialog box (report_test_stimulus).
5. Click **OK**. The schematic shows the simulated values and propagates the results to the selected pins.

Results

If the test generation is not successful, a message in the session transcript will indicate why. If the test generation is successful, the session transcript reports the stimulus necessary to produce the pin values you specified.

Getting Oriented in a Large Design

Sometimes you do not have a specific problem to debug but want to get a better idea of how the design is put together and what the main components are. A good way to do this is by looking at how instances displayed in the **Hierarchy** tab of the Browser window are interconnected in the hierarchical schematic displayed in the Design window.

Prerequisites

- DFTVisualizer is invoked. For more information, see the "DFTVisualizer Invocation" section in this chapter.

Procedure

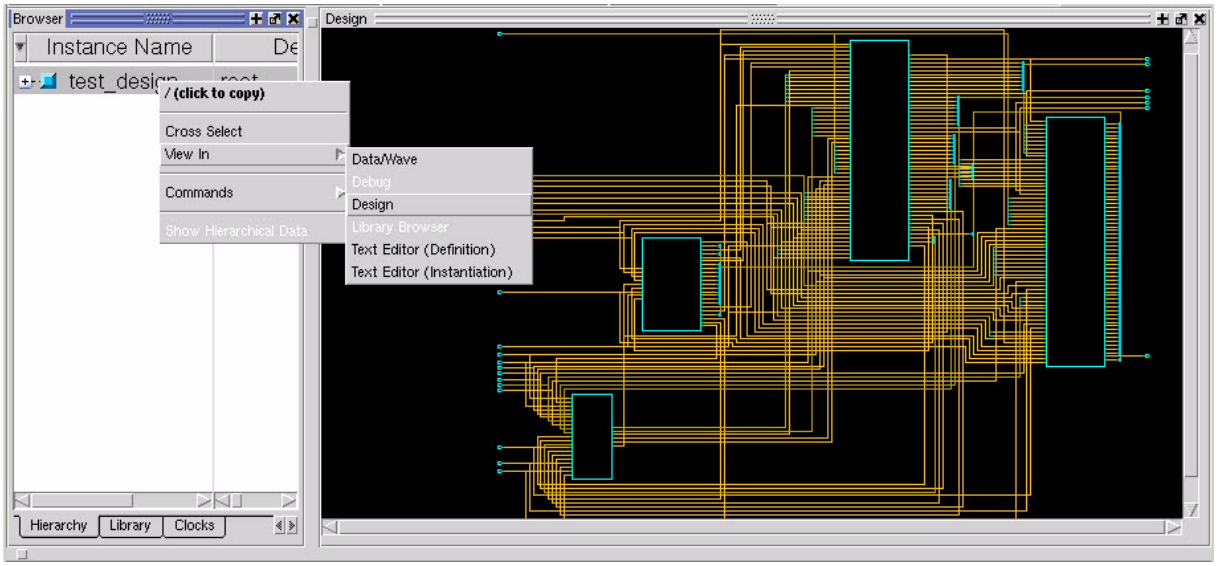
1. Open the Browser Window from the **Windows** menu.
2. Choose **View In > Design** from the right mouse popup menu.

   The top level of the design displays in the Design Window. You can see the major design blocks one hierarchical level below the design's top level, along with their interconnecting nets as illustrated in Figure 4-21 on page 95. The top level ports are represented by port symbols to make them easily distinguishable from pins.

3. Position the cursor over an object of interest and click the right mouse button to see the object's name which is displayed at the top of the popup menu.
4. Optionally, bundle nets connected between the same instances by choosing the **Display > Net Bundle > On** menu item. See "Bundling Nets" on page 66 for an example.
5. Trace down any of the displayed instances by clicking and holding the right mouse button with the cursor on the instance, and selecting any of the trace menu options.

Example

Figure 4-21. Adding the Top Level Instance to the Design Window



DFTVisualizer Preferences

This section describes different methods for customizing DFTVisualizer.

Setting DFTVisualizer Preferences

Use this procedure to customize the DFTVisualizer session as follows:

- Specify colors for display components.
- Add/remove toolbar components.
- Control text/data display.
- Specify editing behavior.

You can also save your preferences for use in subsequent sessions or to a special file that can be loaded after DFTVisualizer is invoked.

Prerequisites

- DFTVisualizer is invoked. For more information, see the "DFTVisualizer Invocation" section in this chapter.

Procedure

1. Select **Edit > Preferences**. The Global Preferences Dialog Box displays. Specify the desired global session attributes.
2. Click the Colors tab. The Colors Preferences Dialog Box displays. Specify the desired object colors.
3. Click the Schematics window tab. The Schematics Preferences Dialog Box displays. Specify the desired attributes for the Debug window, the Design window, or both.
4. Click the Browser Window tab. The Browser Window Preferences Dialog Box displays. Specify the desired attributes for the Browser window.
5. Click the Data Window tab. The Data Window Preferences Dialog Box displays. Specify the desired attributes for the Data window.
6. Click the Text Editor window tab. The Text Editor Window Preferences Dialog Box displays. Specify the desired attributes for the Text Editor window.
7. When you have set the desired preferences, click **OK**.

Results

The specified settings take effect and remain persistent for the current DFT tool session only.

Related Topics

Saving/Loading Session Preferences

Saving/Loading Session Preferences

Use this procedure to save specified DFTVisualizer preferences as the default settings used by all subsequent DFT tool sessions or to a specified file that can be loaded by other users or as alternate set of preferences. You can also use this procedure to set your preferences back to the system defaults.

Prerequisites

- New session preferences are selected. See “Setting DFTVisualizer Preferences” on page 96.

Procedure

1. Select **Edit > Preferences**. The Global Preferences Dialog Box displays.
2. Do one of the following:

Click...	To...
Write Preferences	Save the current settings as default.
Save Preference File	Save the current settings to a specified file in the current working directory. In the dialog box, specify a filename and click OK.
Reset to System Defaults	Change all preferences back to the systems defaults.

Related Topics

Setting DFTVisualizer Preferences

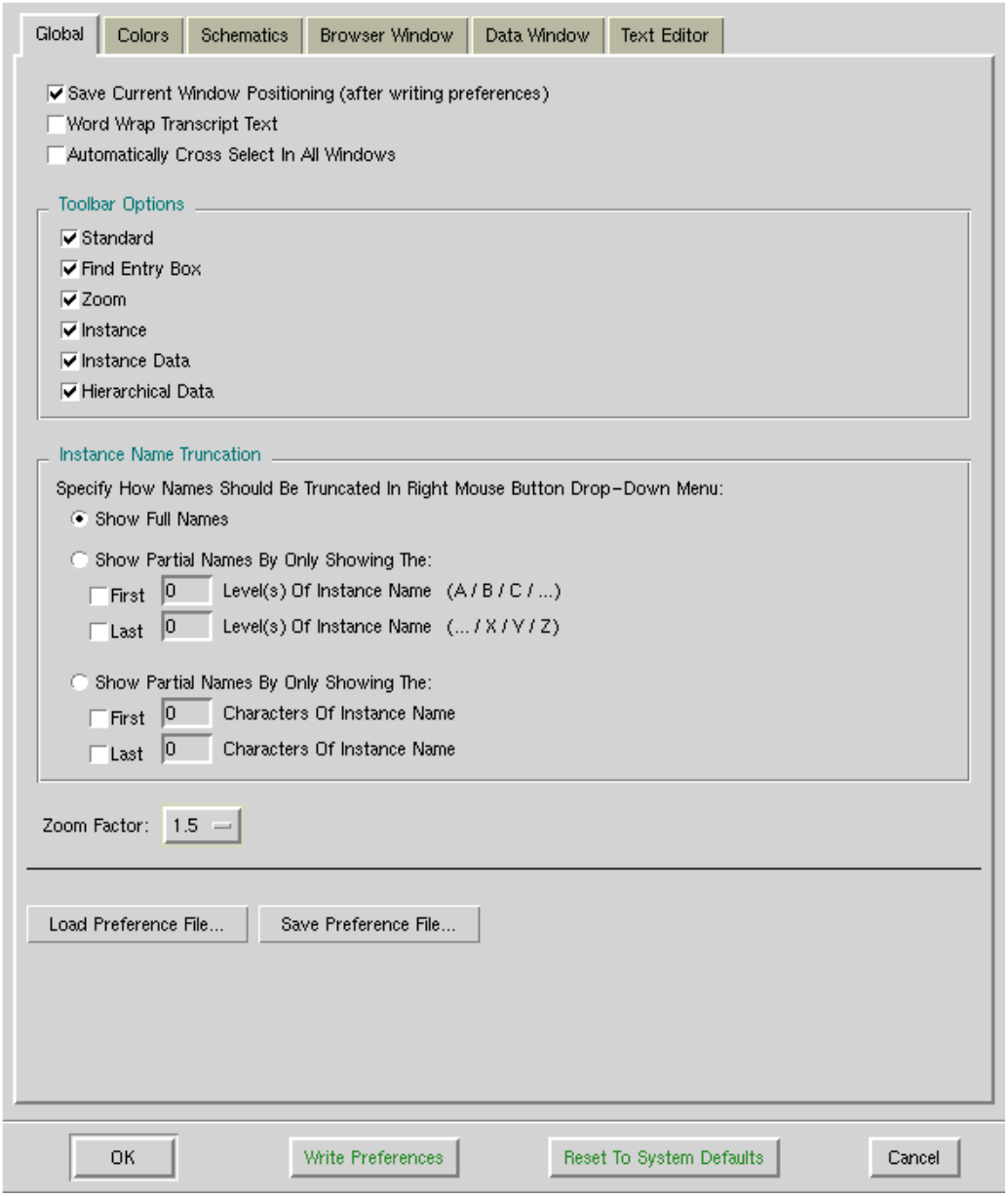
DFTVisualizer Preferences Dialog Box

The following sections describe each tab, left to right, of the DFTVisualizer Preferences dialog box. The following topics are available:

Global Preferences Dialog Box

To access: Select **Edit > Preferences** and click the **Global** tab.

The DFTVisualizer Preferences (Global tab) dialog box allows you to set preferences associated with all display windows.



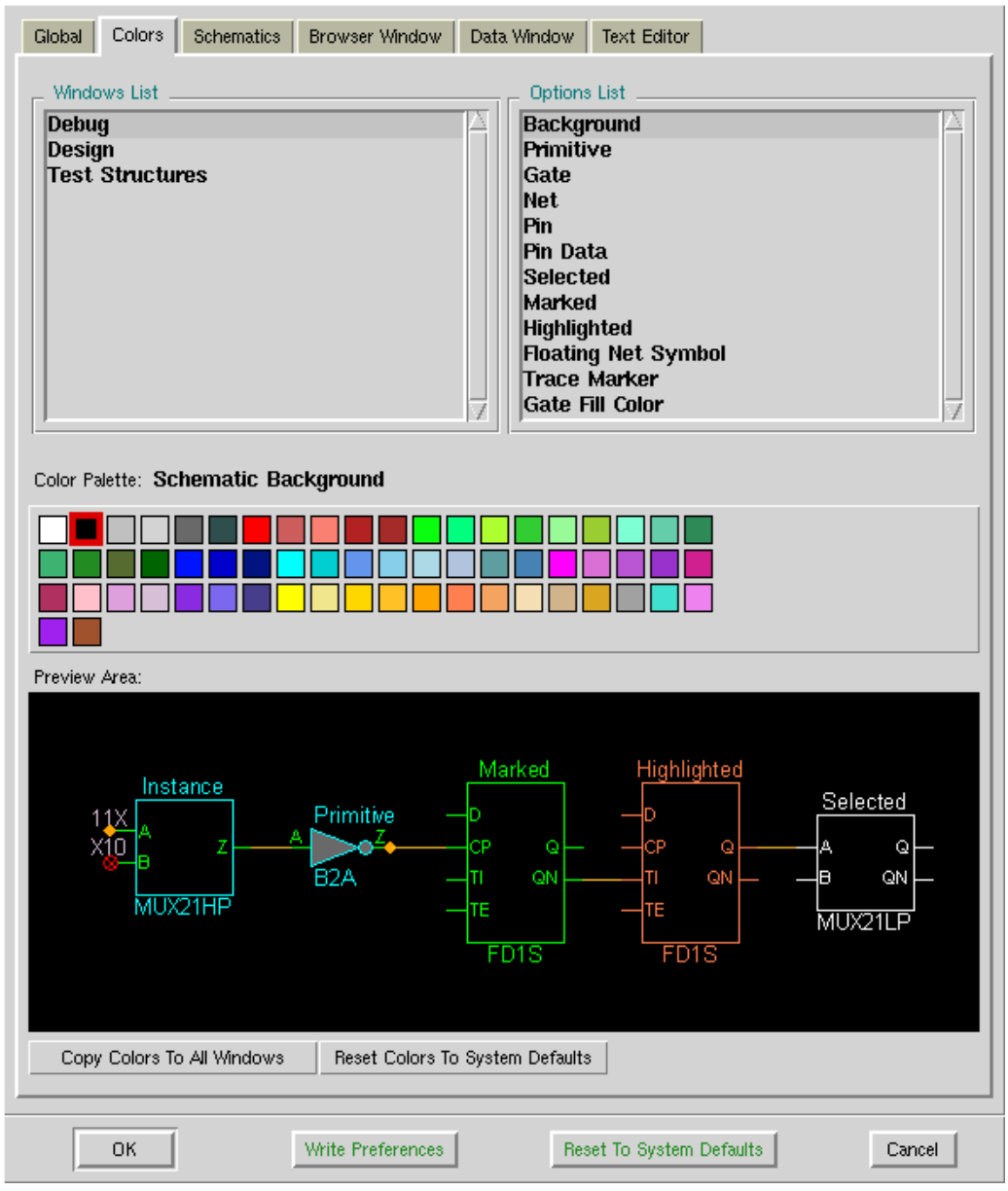
Fields

Field	Description
Save Current Window Positioning (after writing preferences)	Determines if window size and location are saved on exit and used for the next DFTVisualizer session.
Word Wrap Transcript Text	Determines if words wrap to the next line approximately every 70 characters.
Automatically Cross Select In All Windows	Determines if objects you select are automatically cross selected in all windows in which they are already displayed. See "Cross-Selecting Objects" on page 50 for more information.
Toolbar Options	Specifies which tool icons are available from the toolbar on the top of the display.
Show Full Names	Displays complete hierarchical pathname for each instance when the instance is clicked with the right mouse button. Default setting.
Show Partial Names By Only Showing The (Levels):	Specifies how many design levels display as part of instance names. You can specify how many of the first or last design levels are omitted. By default, full names display.
Show Partial Names By Only Showing The (Characters):	Specifies how many characters display in instance names when the instance is clicked with the right mouse button. By default, full names display.
Zoom Factor	Specifies by what percentage a window magnifies when the zoom option is used.
Load Preference File	Loads settings from a specified preference file into the current session.
Save Preference File	Saves the current preference settings to a specified file.
Write Preferences	Saves the current preference settings to a file named *.DftVisualizerrc* in your home directory. Preferences for subsequent tool sessions are read from this file by default.
Reset to System Defaults	Resets all preference settings to the factory default settings.

Colors Preferences Dialog Box

To access: Select **Edit > Preferences** and click the **Colors** tab.

The DFTVisualizer Preferences (Colors tab) dialog box allows you to set color preferences for backgrounds and graphical objects displayed in windows.



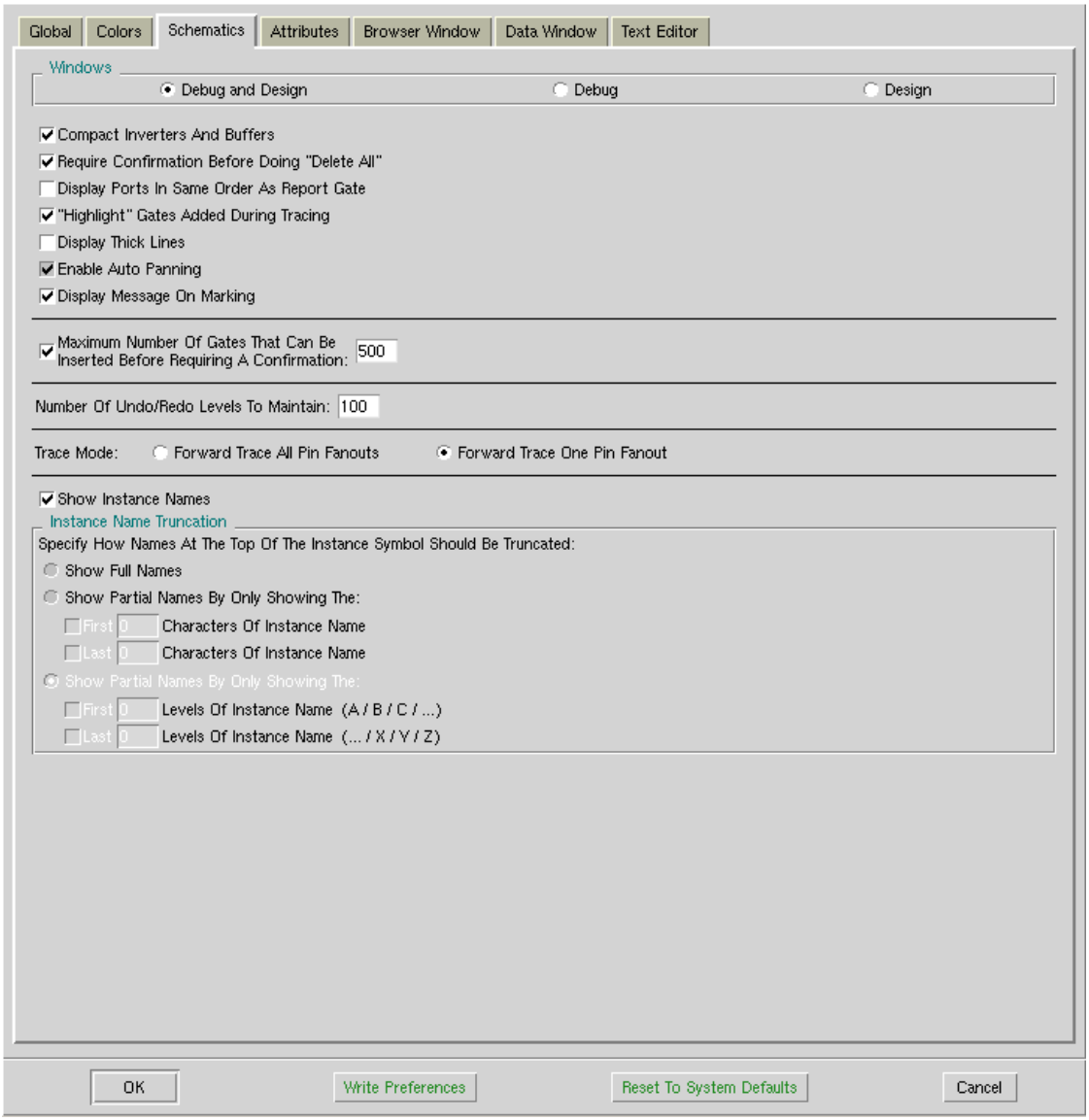
Fields

Field	Description
Windows List	Determines the window to which the Options List applies.
Options List	Displays a list of graphical window elements that you can select and specify a color for.
Color Palette	Palette that indicates the current color selection for the object type selected in the Windows List and Options List.
No. of Colors	Specifies the number of colors that can be used to mark objects in schematic windows (Debug and Design). This item is only available when Marked is selected in the Options List.
Color Index	Specifies a color to be used when marking objects. You can set as many colors as are specified by the No. of Colors field. By default, the following colors are used for marking: • Mark color 1: Green • Mark color 2: Blue • Mark color 3: Orange • Mark color 4: Yellow • Mark color 5: Red This item is only available when Marked is selected in the Options List. For more information, see "Customizing Marking Colors in the Schematic Windows" and "Marking and Unmarking Objects in the Schematic Windows."
Preview Area	Displays the current display color settings for instances, primitives (solid fill), marked objects, highlighted objects, and selected objects.
Copy Colors To All Windows	Applies the colors currently defined for the selected window to all other windows.
Reset Colors To System Defaults	Resets the color preferences for the currently selected window to system defaults.

Schematics Preferences Dialog Box

To access: Select **Edit > Preferences** and click the **Schematics** tab.

The DFTVisualizer Preferences (Schematics tab) dialog box allows you to set preferences associated with the Debug and Design windows.



Fields

Field	Window	Description
Windows	Debug and Design	Indicates whether the changes specified in the dialog box will be applied to the Debug window, Design window, or to both Debug and Design windows.
Compact Inverters And Buffers	Debug and Design	Global option that specifies whether buffers/inverters are collapsed or expanded. When enabled, redundant buffers/inverters are collapsed and omitted from the display; the next non-buffer/non-inverter instance displays. The following symbols indicate compaction status on an individual net:  — collapsed buffers and/or inverters exist on the net.  — buffers and/or inverters are expanded on the net. No symbol — no buffers or converters exist on the net.
Require Confirmation Before Doing “Delete All”	Debug and Design	Prompts for confirmation before a Delete All command is executed.
Display Ports In Same Order As Report Gate	Debug and Design	Connects and displays ports exactly as specified in the design netlist. By default, port connections are flipped/manipulated when possible to reduce clutter and optimize viewing.
“Highlight” Gates Added During Tracing	Debug and Design	Determines whether gates added in trace mode display highlighted.
Display Thick Lines	Debug and Design	Specifies to display all nets and instances on the schematic with thick lines to improve visibility.
Enable Auto Panning	Debug and Design	During tracing as new objects are added outside the window boundary, specifies to automatically bring the new objects into view and, if possible, also keep the original object in view.
Display Message On Marking	Debug and Design	Enables/disables the echoing of mark_display_instances commands, initiated through the graphical user interface, to the transcript. By default, this option is enabled.

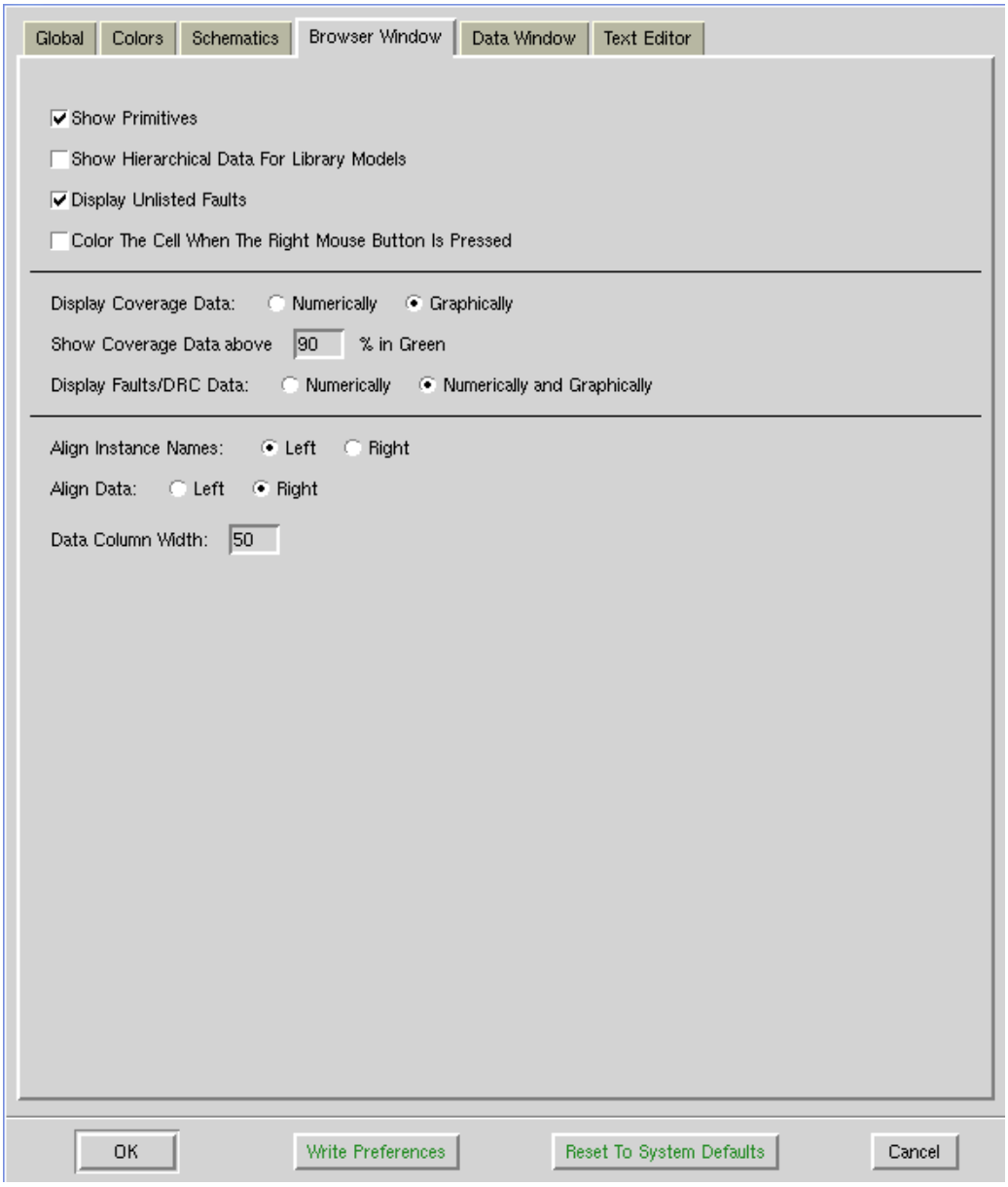
Field	Window	Description
Maximum Number Of Gates That Can Be Inserted Before Requiring A Confirmation	Debug and Design	Specifies the maximum number of gates that can added before prompting for confirmation. Default is 500.
Number Of Undo/Redo Levels To Maintain	Debug and Design	Specifies how many levels of Undo/Redo can be recalled from memory. Default is 20.
Forward Trace All Pin Fanouts	Debug and Design	Displays all levels of pin fanouts when a fanout marker is clicked.
Forward Trace One Pin Fanout	Debug and Design	Displays one level of pin fanout when a fanout marker is clicked.
Show Instance Names	Debug and Design	Determines whether instance names are displayed for each instance.
Show Full Names	Debug and Design	Displays complete hierarchical pathname for each instance when the instance is clicked with the right mouse button. Default setting.
Show Partial Names By Only Showing The: (Characters)	Debug and Design	Specifies how many characters display in instance names when the instance is clicked with the right mouse button. By default, full names display.
Show Partial Names By Only Showing The: (Levels)	Debug	Specifies how many design levels display as part of instance names. You can specify how many of the first or last design levels are omitted. By default, full names display.
Automatic Net Connection	Design	Determines how many nets display during tracing. You can choose to display only explicitly traced nets, a set number of nets, or all nets.
Split Schematic Into Multiple Pages	Debug	Determines whether the schematic is split into pages that display one at a time or displayed in its entirety. By default, the schematic displays in its entirety.
Show The First <32> Characters Of Simulation Data (0 to 252)	Debug	Specifies the number of characters of simulation data that are to be displayed in the Debug window. Maximum number of characters that can be displayed is 252. Default is 32.

Field	Window	Description
Display All Hierarchical Submodule Pins Except When Tracing	Design	Determines whether pins display when hierarchical submodules are added to the schematic.
Display Net Names On The Schematic	Design	Determines whether net names display on the schematic.

Browser Window Preferences Dialog Box

To access: Select **Edit > Preferences** and click the **Browser Window** tab.

The DFTVisualizer Preferences (Browser Window tab) dialog box allows you to set preferences associated with the Browser Window.


Global
Colors
Schematics
Browser Window
Data Window
Text Editor
Show Primitives
Show Hierarchical Data For Library Models
Display Unlisted Faults
Color The Cell When The Right Mouse Button Is Pressed
Display Coverage Data:
Numerically
Graphically
Show Coverage Data above
90
% in Green
Display Faults/DRC Data:
Numerically
Numerically and Graphically
Align Instance Names:
Left
Right
Align Data:
Left
Right
Data Column Width:
50
OK
Write Preferences
Reset To System Defaults
Cancel

Fields

Field	Description
Show Primitives	Determines if primitives can be displayed.
Show Hierarchical Data For Library Models	Determines if the **Library**, **Hierarchy**, and **Clocks** tabs of the Browser window display the fault data beneath library instances. When the fault data is shown as "Hidden" in the Browser window, you can show the fault data by clicking the right mouse button on the library instance and selecting **Show Hierarchical Data** from the popup menu. Note that the additional data may impact performance on large designs. Similarly, you can use the right mouse button option **Hide Hierarchical Data** to reduce the displayed data and improve performance.
Display Unlisted Faults	Specifies to include or exclude faults statistics for graybox instances that are not in the netlist (unlisted faults) from the statistical display of Browser data. Excluding unlisted faults from fault statistic calculations updates coverage calculations for both the graybox and all higher levels in the hierarchy whose statistics would normally include the graybox's statistics.
Color The Cell When The Right Mouse Button Is Pressed	Determines whether the part of the selection indicator in a data column cell displays highlighted when selected with a right mouse button press.
Display Coverage Data	Determines whether test/fault coverage data displays as a bar graph (graphically) or as text number.
Show Coverage Data above *nn* % in Green	Determines the display color for test coverage, fault coverage, and ATPG effectiveness columns. Values above the specified percentage display in green, and values equal to or below the specified percentage display in red. Note, this setting does not affect the display of the Test Coverage Loss column which always displays in red.
Display Faults/DRC Data	Determines whether fault/drc data displays as a bar graph (graphically) and a text number or text number only.
Align Instance Names	Determines whether the instance names display left or right justified.
Align Data	Determines whether data associated with instance names display left or right justified in the columns.
Data Column Width	Specifies the number of characters visible in the data columns. By default 50 characters display.

Data Window Preferences Dialog Box

To access: Select **Edit > Preferences** and click the **Data Window** tab.

The DFTVisualizer Preferences (Data Window tab) dialog box allows you to set preferences associated with the Data window.

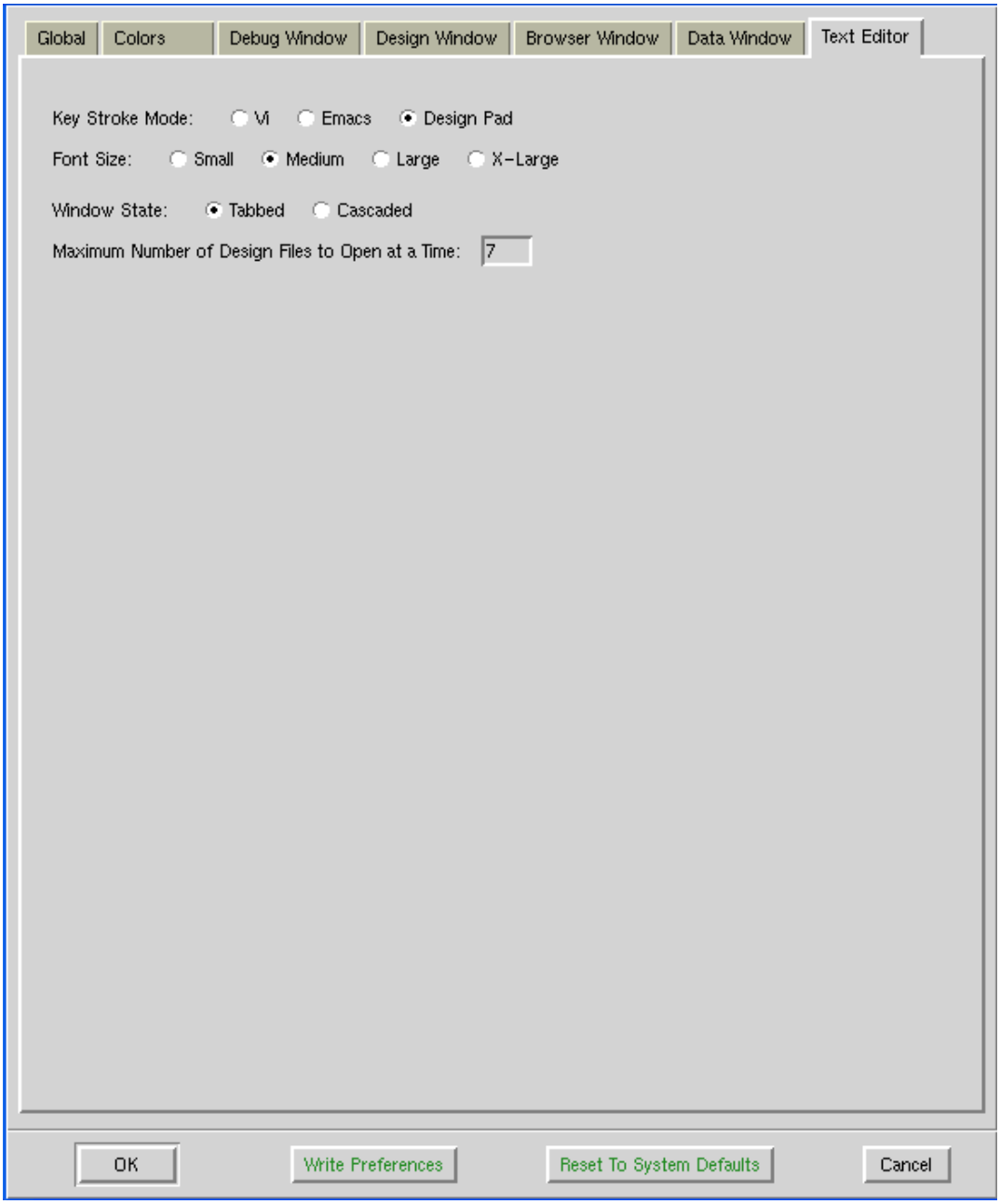
Fields

Field	Description
Align Names	Determines whether the names display left or right justified.
Align Data	Determines whether data associated with names display left or right justified in their columns.
Data Column Width	Specifies the number of characters visible in the data columns. By default 50 characters display.

Text Editor Window Preferences Dialog Box

To access: Select **Edit > Preferences** and click the **Text Editor Window** tab.

The DFTVisualizer Preferences (Text Editor Window tab) dialog box allows you to set preferences associated with the Text Editor window.



Fields

Field	Description
Key Stroke Mode	Determines whether key strokes are interpreted to be Vi, Emacs, or Design Pad commands.
Font Size	Specifies the font size used to display the contents of the window: • Small — 10 • Medium — 13 • Large — 16 • X-Large — 19
Window State	Specifies how windows display in the DFTVisualizer main window: • Tabbed — Windows align side by side and are accessed using a Tab at the bottom of the window. This is the default. • Cascaded — Windows stack with the top portion of each window visible.
Maximum Number of Design Files to Open at a Time	Specifies the maximum number of files that will be opened in the Text Editor window when **File > Open > Current Design Files** is selected.
Write Prefs	Saves the current preference settings to a file named *.DftVisualizerrc* in your home directory. Preferences for subsequent tool sessions are read from this file by default.

DFTVisualizer Windows

This section describes the DFTVisualizer windows.

Objects Added to DFTVisualizer Windows

When you add an object to the Debug, Design or Data window, you can refer to the object by either the hierarchical name (such as the port of a submodule, a net name, etc.) or the flattened model name.

The following table shows what will be added when you add a particular type of object to these windows.

Table 4-5. What is Added to the Debug, Design and Data Windows

Type of Object	Debug Window	Design Window	Data Window
Library level instance	Instance	Instance	All pins on instance
Pin on library instance	Instance	Instance	Specified instance pin
Bus on library instance	Instance	Instance	Specified bus
Primitive (gate ID)	Primitive (if gate level set to primitive) Design level instance the primitive is a part of (if gate level set to design)	Instance the primitive is a part of	All pins on primitive

Table 4-5. What is Added to the Debug, Design and Data Windows

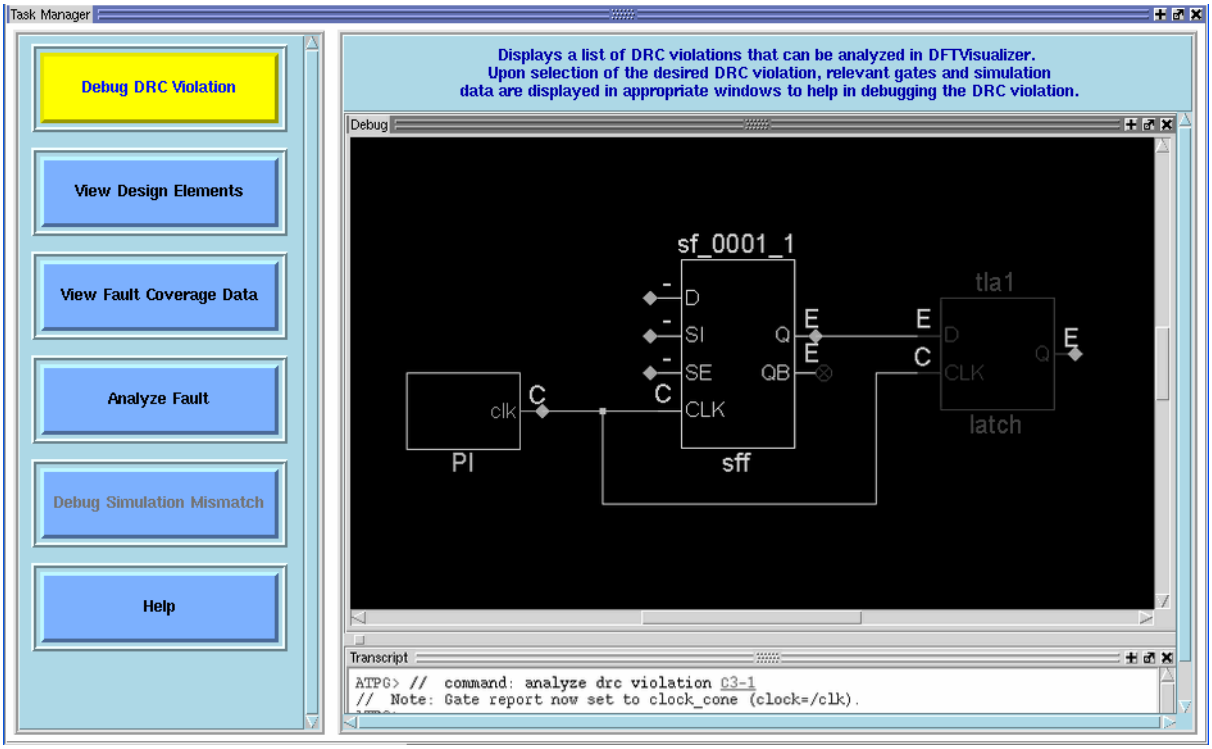
Type of Object	Debug Window	Design Window	Data Window
Pin on Primitive	Primitive (if gate level set to primitive) Design level instance the primitive is a part of (if gate level set to design)	Instance the primitive is a part of	Pin on primitive
Hierarchical instance (submodule)	Not supported	Hierarchical instance	All ports on the hierarchical instance
Port on hierarchical instance	Library level instance driven by the port (like report_gates)	Hierarchical instance	Port on hierarchical instance
Bus port on hierarchical instance	Library level instance driven by all members of the bus	Hierarchical instance	Bus on hierarchical instance
Net (wire)	Library level instance driving the net (like report_gates)	Instance driving the net (hierarchical)	Net
Bus (wire)	Library level instance driving the bus	Instance driving the bus (hierarchical)	Bus

Task Manager Window

To access: Choose **Windows > Task Manager** or click the button.

The Task Manager provides a graphical interface for displaying and analyzing DRC violations, viewing the design netlist, viewing fault coverage data, identifying and analyzing faults, debugging pattern simulation mismatches, viewing diagnosis report files, and accessing the DFTVisualizer reference documentation. The following figure shows the initial state of the Task Manager.

Figure 4-22. Task Manager Window



Description

The context in which you are using DFTVisualizer determines which buttons are visible and active. Potentially, the Task Manager window has the following task buttons:

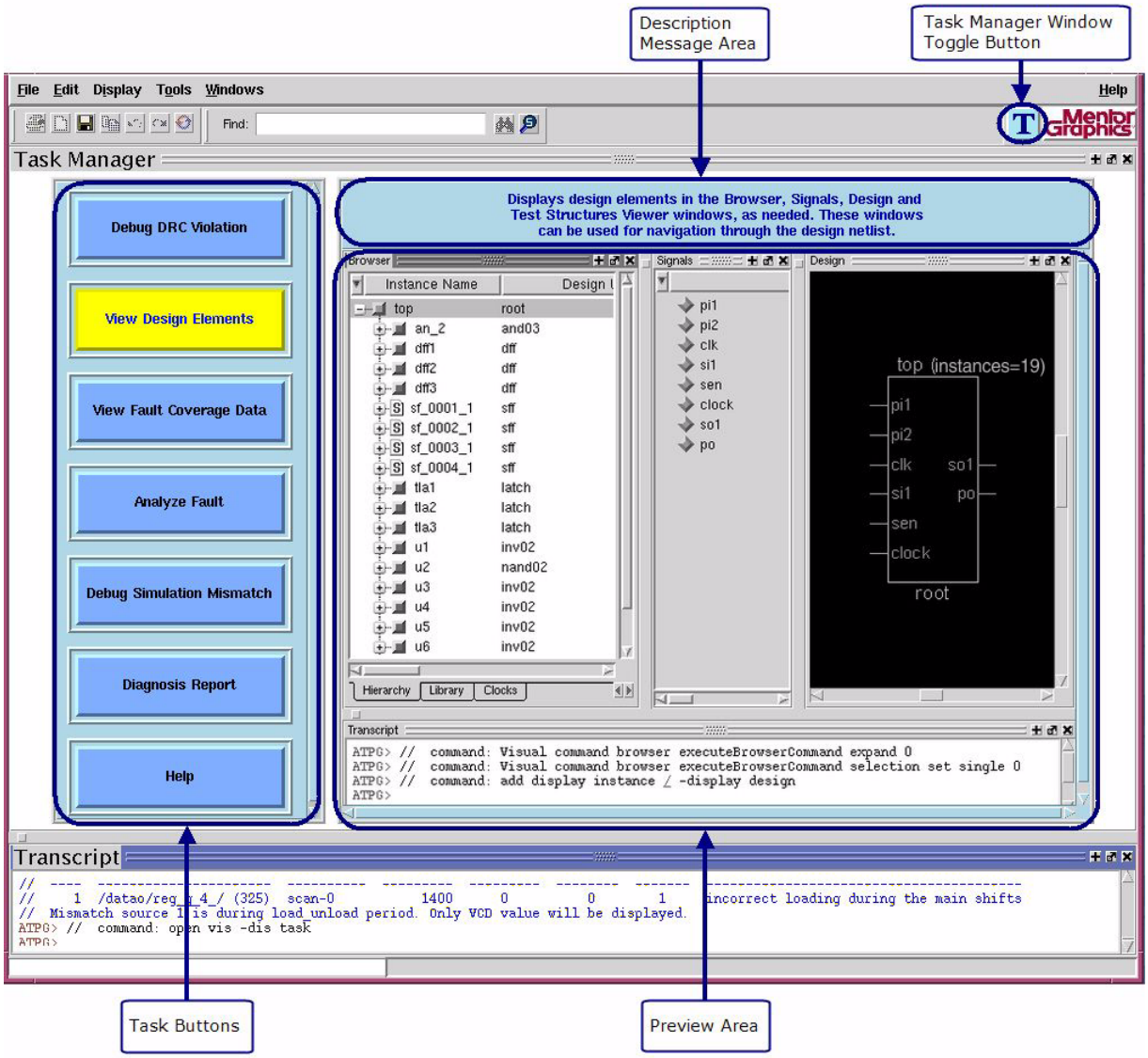
- **Debug DRC Violation** — Analyzes DRC violations, reports the violations to the session transcript, and graphically displays relevant gates and simulation data.
- **View Design Elements** — Displays the design hierarchy and schematics in the Browser window enabling you to navigate visually through the design.
- **View Fault Coverage Data** — Displays test coverage, fault classification, and DRC statistics for hierarchical design blocks.
- **Analyze Faults** — Identifies why a specified fault is not detected.

- **Debug Simulation Mismatches** — Displays and highlights design, simulation, and test pattern data for the selected simulation mismatch.
- **Diagnosis Report** — Displays a dialog box that enables you to choose and view a diagnosis report file.
- **Help** — Provides access to the DFTVisualizer reference documentation.

Usage Notes

Click the T button in DFTVisualizer to toggle on and off the Task Manager window as shown in Figure 4-23. Hovering over a task button displays a description of the task and, in the preview area, an example image of the window(s) that will open when you click the button.

Figure 4-23. Task Manager View Design Elements Task Highlighted



The current context determines which task buttons appear and whether they are active.

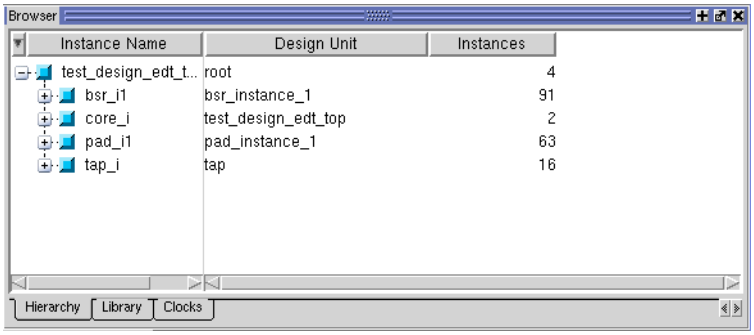
- To use the **Debug DRC Violation** button, the tool must have checked the design rules and reported a violation. See "Analysis of a DRC Violation" and "Design Rule Checking" for more information.
- To use the **Analyze Fault** button, fault simulation results must be available for the current pattern set. For instructions on specifying input for fault analysis, see "Analyzing a Fault and Displaying its Location" or refer to the analyze_fault command in the *Tessent Shell Reference Manual*.
- To use the **Debug Simulation Mismatch** button, a mismatch must exist based on tool analysis and the results of pattern simulation in Questa or a third-party simulator. For information on debugging simulation mismatches, see "Automatically Analyzing Simulation Mismatches" in the *Tessent Scan and ATPG User's Manual.*
- To use the **Diagnosis Report** button, you must be running Tessent Diagnosis.

Browser Window

To access: Choose **Windows > Browser**. By default, the Hierarchy Browser window is active as shown in Figure 4-24.

The Browser window provides access to the Hierarchy Browser, Library Browser, and the Clocks Browser windows.

Figure 4-24. Browser Tabbed Window



Description

These windows provide the following capabilities:

- Hierarchy Browser— Enables you to navigate through the design hierarchy and view coverage, fault, and DRC statistics. DFTVisualizer displays faults in the Hierarchy Browser based on whether the set_fault_mode setting specifies to show faults as collapsed or uncollapsed.
- Library Browser — Displays statistics for the ATPG library models used in a design. DFTVisualizer displays data in the Library Browser based on whether the set_fault_mode setting specifies to show faults as *collapsed* or *uncollapsed*. When specified as:

  - Collapsed — The tool does not include equivalent faults in the fault lists.
  - Uncollapsed — The tool includes equivalent faults in the fault lists. This is the default mode at tool invocation.

- Clocks Browser — Displays all of the clocks in the design and their attributes in a descending order of test coverage. Clocks and their attributes are displayed from all modes. The Clocks Browser allows you to navigate the design hierarchy and view the faults for each individual clock and analyze the distribution of faults between clock domains.

The Browser window uses the following icons to represent different instance types:

Table 4-6. Browser Window Instance Type Icons

Symbol	Means the Instance is a...
	Clock
	Submodule (instance of a Verilog module)
	Netlist/Verilog primitive
	Library level instance
	Library model (Library tab only)
	Graybox, parent
	Graybox, instances in netlist
	Graybox, instances not in netlist
	Library level instance containing one or more RAMs/ROMs
	Library level instance containing one or more scan cells
	Blackboxed instance
	Primitive

Usage Notes for the Browser Window

- Add additional data columns to the Hierarchy, Library, and Clocks Browsers using the **Data** pulldown menu and/or the buttons on the toolbar as described in Table 4-7. Menu options are disabled (greyed-out) if corresponding data is unavailable.

Table 4-7. Browser Window Data Menu Choices

Button	Data Menu	Description
	Gates	Total number of library level instances, including submodules.
	Primitives	Total number of primitives, including submodules.
	Total Faults	Total number of faults, including submodules.
	Undetected Faults	Sum of all undetected faults (UO, UC, AU, PT).
	Faults > Add All | Delete All	Number of faults for each available class; adds a column for each class.
N/A	Faults > <type>	Number of faults for a specific class. Each class can be expanded to display columns for each of its subclasses. For information on fault subclasses, see the set_relevant_coverage command in the *Tessent Shell Reference Manual*.
	DRC > Add All | Delete All	Number of DRC violations; adds one column for each violation that exists in the design.
N/A	DRC > <type>	Number of violation occurrences for a specific DRC rule.
	Coverage Data > Test Coverage	Test coverage for each submodule. In the Library Browser, test coverage data includes two subcolumns: • Max: highest test coverage achieved for any instance of the library model. • Avg: average test coverage achieved for all instances of the library model.
	Coverage Data > Relevant Test Coverage	Test coverage for each submodule after untestable faults have been added/deleted from test coverage calculations. For more information, see the set_relevant_coverage and report_statistics commands in the *Tessent Shell Reference Manual*.
	Coverage Data > Test Coverage Loss	Test coverage loss is the undetected faults in an instance displayed as a percentage of the testable faults in the entire design.

Table 4-7. Browser Window Data Menu Choices

Button	Data Menu	Description
	Coverage Data > Fault Coverage	Fault coverage for each submodule. In the Library Browser, fault coverage data includes two subcolumns: • Max: highest fault coverage achieved for any instance of the library model. • Avg: average fault coverage achieved for all instances of the library model.
	Coverage Data > ATPG Effectiveness	ATPG effectiveness for each submodule.
	Clock Attributes	Clock attributes such as off state, constraints, and internal/external clock.
	N/A	Optimized NCPs (Named Capture Procedures) display in the Wave window. To see unoptimized NCPs, click the UOP icon. Note: This icon is only available after you have selected a NCP to view using the **Data > Named Capture** menu; only NCPs defined in the test procedure file are available from this menu.
	N/A	Unoptimized NCPs display in the Wave window. To see optimized named capture procedures, click the OP icon. Note: This icon is only available after you have selected a NCP to view using the **Data > Named Capture** menu.
	Clock Attributes	Clock attributes such as off state, constraints, and internal/external clock.
	N/A	Optimized NCPs (Named Capture Procedures) display in the Wave window. To see unoptimized NCPs, click the UOP icon. Note: This icon is only available after you have selected a NCP to view using the **Data > Named Capture** menu; only NCPs defined in the test procedure file are available from this menu.
	N/A	Unoptimized NCPs display in the Wave window. To see optimized named capture procedures, click the OP icon. Note: This icon is only available after you have selected a NCP to view using the **Data > Named Capture** menu.

- View subclasses of faults by clicking ⇔ in the column header of the fault type. A new column is added for each subclass. For undetected fault classes (AU, UC, and UO), percentage numbers displayed in the column represent the percentage of the design not tested as a result of the fault.
- Report or write all faults for a particular instance by selecting the instance, clicking the right mouse button, and selecting **Commands > Report_Faults | Write_Faults** from the popup menu. Note, you can report/write unlisted faults for graybox instances separately by clicking on the graybox instance, or you can select the parent instance and report/write all listed and unlisted faults together.
- Use **File > Save As** or click  to save a text, comma separated value (CSV), or graphical version (screen capture) of the displayed contents of the Browser window to a file.
- Statistics data displays only for submodules, not library level instances or primitives. Be aware that when you display data, there is some processing time overhead (progress of which you can check in the progress bar).
- If the data in a column becomes invalid due to the state of the tool, that column becomes unavailable and the **Refresh Data** menu item on the **Data** menu is enabled. For example, if test coverage data is displayed and you create additional patterns, the test coverage column will no longer be available. Pushing the refresh button regenerates the data. The data will always match what is reported by commands like report_statistics and report_drc_rules.
- To make the display easier to read when viewing statistics, you can group all library level instances into the artificial level of hierarchy called $$Gates$$. To enable instance grouping, select the **Display > Group Instances > On** menu item.
- The Signals Window works together with the Browser Window, allowing you to browse and select ports and wires for the instance selected in the Browser.
- Copy an object's hierarchical pathname into a buffer which you can then paste into another location by selecting the object(s) and choosing "<object_name(s)> (click to copy)" from the popup menu. For more information, see "Copying and Pasting Object Names in the Design."

Usage Notes for the Hierarchy Browser

From DFTVisualizer, choose **Windows > Browser**. By default, the Hierarchy Browser displays the following columns:

- Instance Name — The name of the instance in the netlist. Instances of primitives include their gate ID. Instance types are shown with the icons listed in Table 4-6.
- Design Unit — What the instance is an instantiation of (Verilog module, ATPG library model, or primitive type).

- Instances — Number of instances at this level of hierarchy. Does not include the number of instances within each submodule.

In the Hierarchy Browser:

- Click the plus sign (+) next to an instance to display the gates and primitives for that individual instance.
- Double-click on any instance to automatically expand/collapse it.
- Click the right mouse button and use the **View In** popup menu option to add an object to another window.
- Click the right mouse button over an object in hierarchy and use the **View in Text Editor** popup menu option to view the Verilog definition for the instance in a Text Editor.
- View details of the AU fault subclasses PC and TC by clicking the  icon in the header of the AU column or by selecting **Data > Faults > AU.PC | AU.TC** from the pulldown menu. An AU.PC or AU.TC column is added with details displayed as subcolumns.
- Use **File > Save As** or click  to save a text, comma separated value (CSV), or graphical version (screen capture) of the Hierarchy Browser contents to a file.

Usage Notes for the Library Browser

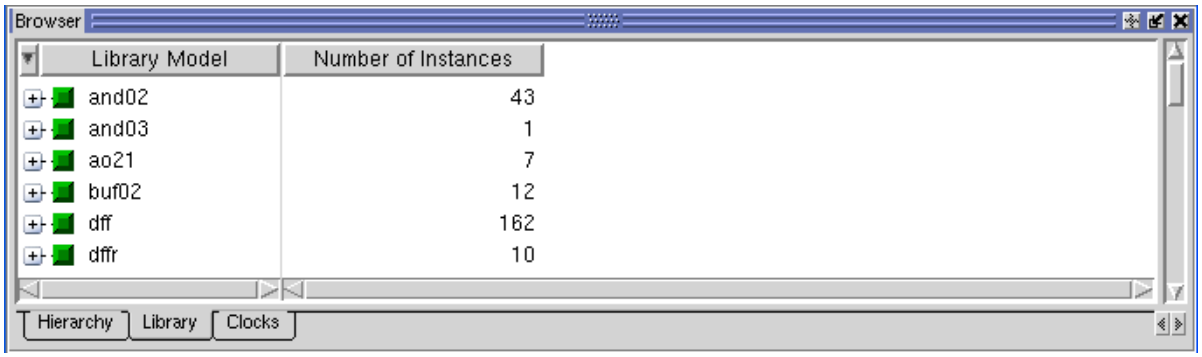
From DFTVisualizer, choose **Windows > Browser** and click the **Library** tab. By default, the Library Browser displays the following columns:

- Library Model — ATPG library models used in the design. By default, consolidated data for each ATPG model displays. Instance types are shown with the icons listed in Table 4-6.
- Number of Instances — Number of instances at this level of hierarchy. Does not include the number of instances within each submodule.

In the Library Browser:

- Click on the (+) next to a library model to display statistics for the individual instances of the model in the design.
- Click the right mouse button over an object and use the **View In** popup menu option to add an object to another window.
- Click the right mouse button over an object and use the **View in > Text Editor** (Definition) popup menu option to view the Verilog definition for the instance in a Text Editor.
- Use **File > Save As** or click  to save a text, comma separated value (CSV), or graphical version (screen capture) of the displayed contents of the Library Browser to a file.

Figure 4-25. Browser Window with Library Tab Active



Usage Notes for the Clock Browser

From DFTVisualizer, choose **Windows > Browser** and click the **Clocks** tab. By default, the Clocks Browser displays the following columns:

- Clock — The name of a clock in the design. Instance types are shown with the icons listed in Table 4-6.
- Attributes — The attributes of each individual clock as reported by the report_clocks command. Specifically, the subcolumns Off State, Constraint, and Internal are displayed.
- Faults — The total faults for each clock domain and the percentage of all faults in the design.

Note

If a fault is in multiple clock domains, the fault is attributed to each clock domain. Because of this, it is possible that the sum of all fault percentages (faults in a clock domain as a percent of the total faults in the design) can exceed 100%.

- Test Coverage — Percentage of all testable faults detected by a pattern set for that particular clock.

In the Clocks Browser:

- Click on the (+) next to a clock to display statistics of the individual instances for each clock in the design. Only instances with faults in the expanded clock domain are displayed.
- Click the right mouse popup menu and use the **View In** option to add an object to another window.
- Click the right mouse button over an object in the clock's hierarchy and use the **View in Text Editor** popup menu option to view the Verilog definition of the instance in a Text Editor.

- Use **File > Save As** or click  to save a text, comma separated value (CSV), or graphical version (screen capture) of the displayed contents of the Clocks Browser to a file.

For more information on using the **Clocks** Browser, see section "Performing Clock Domain Analysis in the Browser".

Related Topics

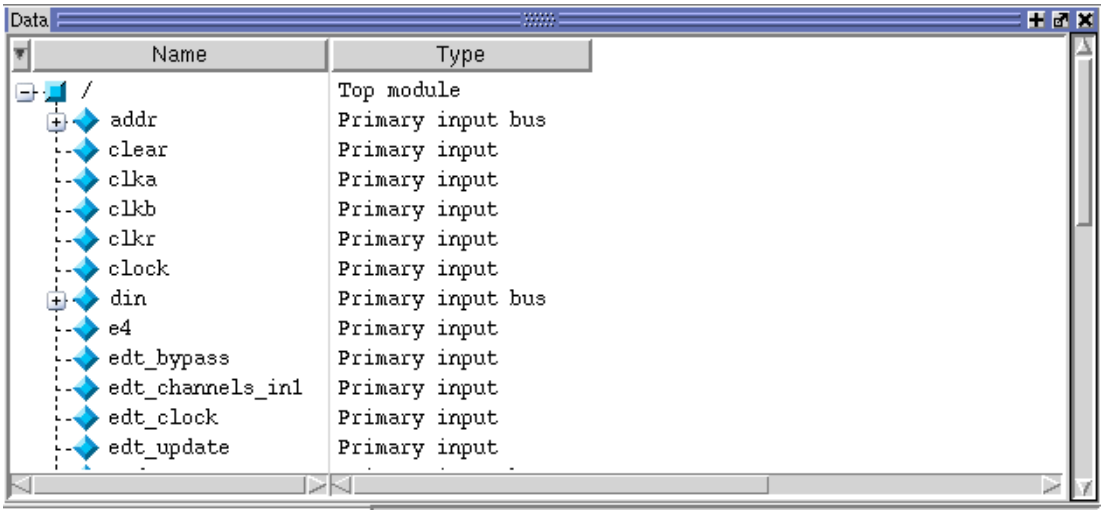
Data Window
Debug Window
Design Window
Signals Window
write_window_contents command

Data Window

To access: Choose the **Windows > Data** menu item or click the  icon.

The Data window provides a tabular presentation of the report_gates command output that allows you to see data for multiple instances and multiple data sets at the same time.

Figure 4-26. Data Window

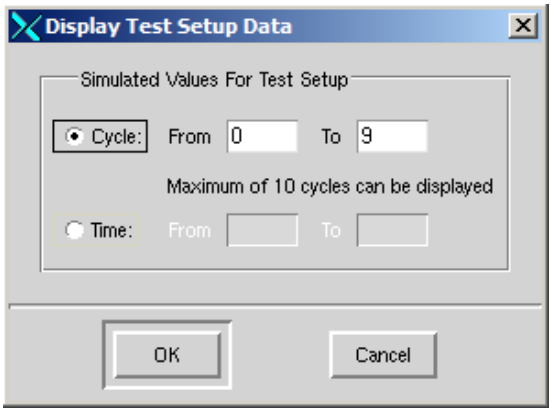


Usage Notes

- Add any type of instance (submodule, library level instance, primitive) or signal (submodule port, instance port, wire) to the Data window using any of the following methods:
  - Select one or more instances in another window. Then place the cursor over one of the selections, press the Control-left mouse button, and drag into the display area of the Data window and release.
  - Select an instance from another window and use the **View In** option on the right mouse button menu.

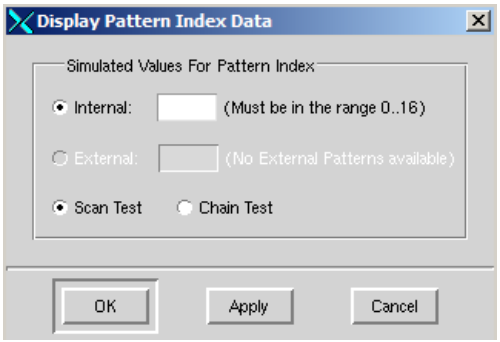
  - Use a command that adds named instances to the display (add_display_instances for example).
  - Click the **Add Instances** icon on the tool bar. The Make Additions to the Display dialog box displays. You can use any number of asterisks (*) or question marks(?) as wildcard characters to specify the string enabling you to match many pathnames in the design.

- Delete instances or signals from the Data window by first selecting them and then using any of the following methods:
  - Select the **Display > Delete** menu item.
  - Select **Delete** from the right mouse button popup menu in the window's display area.
  - Click the **Delete Selected** icon or the Delete All icon.
  - Press the Delete key.
- Display data in the Data window by using the **Data** pulldown menu, the corresponding buttons on the tool bar, or a command that adds data to the display (add_display_data for example).
- Display NCP data in the Data window by using the **Data > Capture Procedure** pulldown menu item, or by executing the **add_display_data capture_procedure** command from the transcript window.
- Remove data from the Data window by moving the cursor over the column header you want to delete and selecting the column to delete from the right mouse button popup menu.
- Re-add data you have removed from the Data window by moving the cursor over any column header and selecting the column to re-add from the right mouse button popup menu.
- View the data for cycles that are out of view in a long test_setup procedure by using the **Display > Test_Setup** menu item (or **Test_Setup** from the right mouse popup menu in the window's display area) to display the Display Test Setup Data dialog box.



Tip: To maintain the usefulness of the displayed data, the tool intentionally displays a maximum of 10 cycles in the Data window regardless of the number of cycles entered into the From/To fields. If you need to see more than 10 cycles, you should view the data in the Wave window.

- Display scan test patterns or chain test patterns by using the **Data > Pattern Index** menu item to display the Display Pattern Index Data dialog box. Note, the Chain Test option is only available after you have saved the scan test patterns.



- Open the Test_end procedure in a text editor by selecting **Data > Test-End** to display the *drc test_end* column; then, click the left mouse button on the column header.
- Display stable value data by selecting one of the **Data > Stable** menu items. This action executes the set_gate_report command with one of the STABLE_After_setup | STABLE_Capture | STABLE_Load_unload | STABLE_Shift options enabled or disabled. You can also access stable value data by clicking the following toolbar icons: Stable After Test_Setup  , Stable During Capture  , Stable During Load_Unload  , and Stable During Shift  .
- Copy an object's hierarchical pathname into a buffer which you can then paste into another location by selecting the object(s) and choosing "<object_name(s)> (click to copy)" from the popup menu. For more information, see "Copying and Pasting Object Names in the Design."
- If the Data window is detached from and hidden behind the main window, click the  icon to bring the window back to the front.
- Use **File > Save As** or click  to save a text, comma separated value (CSV), or graphical version (screen capture) of the displayed contents of the Data window to a file. If you specify Text format, the contents of the Data window including the column headers are output to a *.txt* file.

Related Topics

Debug Window
Design Window

Debug Window

To access: Choose the **Windows > Debug** menu item or click the  icon.

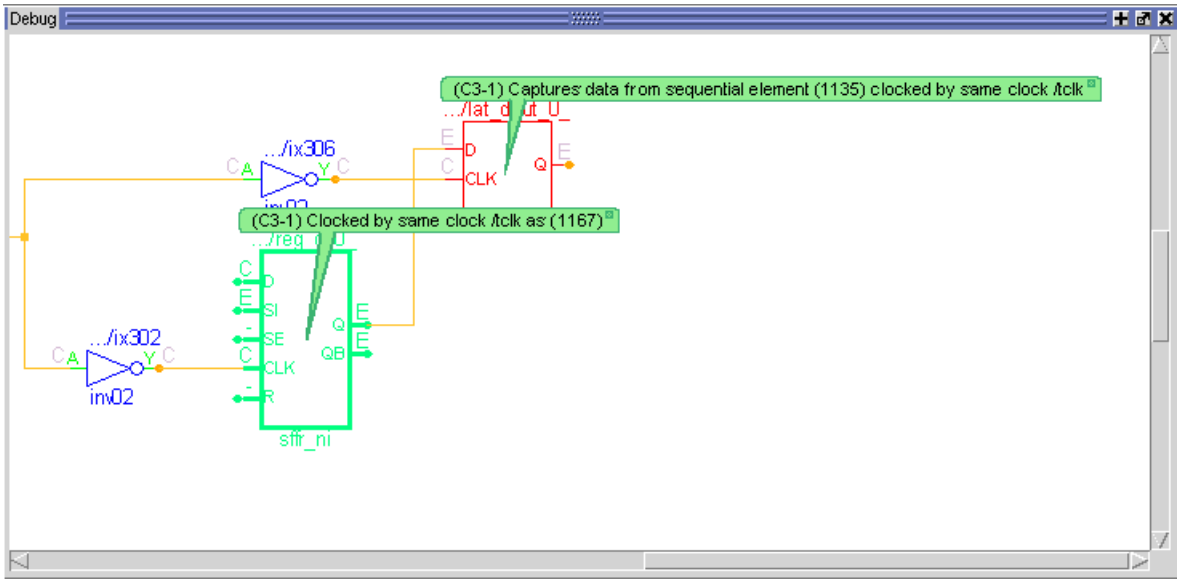
The Debug window displays a schematic representation of the *flattened* design.

You can refer to Model Flattening in the *Tessent Scan and ATPG User's Manual* for more information about flattening.

Note

In FlexTest, the Debug window is only available when FlexTest is in DRC mode.

Figure 4-27. Debug Window



Description

The Debug window contains the following objects:

Objects	Location	Description
Progress Bar	Status bar	Shows the percentage of the design that has been loaded.
Selected Instance	Status bar	Shows the type of the selected instance (submodule or design level gate) and its path and name.
Number of Instances Displayed (Selected)	Status bar	Shows the number of instances in the display and, of those, the number selected.

Objects	Location	Description
Gate Level Setting	Status bar	Indicates the gate level setting. The schematic is presented at design level by default but can be shown at primitive level (controlled by **Display > Gate Level or Ctrl + L**). See "Setting the Level of Gate Data" for more information.
Sheet Number Entry Box	Status bar	Enables you to navigate to a specific sheet by entering the new sheet number in the entry-box. Also displays the current sheet number and the total sheet count.
Reporting Status	Status bar	Shows the current gate report setting.
Solid-filled Object	Schematic area	Indicates a primitive.
Unfilled Instance	Schematic area	Indicates a library instance that can be expanded in place (by double-clicking) to show its primitives.
Callout	Schematic area	Indicates an expandable marker  on the schematic associated with a specific instance. The marker expands into a text box that displays information about the object that helps in debugging issues such as DRC violations and simulation mismatches. For information about possible callout messages, see "Debugging DRC Violations in DFTVisualizer."
Off Page Connector	Schematic area	Enables you to trace between sheets when the schematic is split into multiple sheets. Double-click the connector on the net at the very right or very left of the schematic.

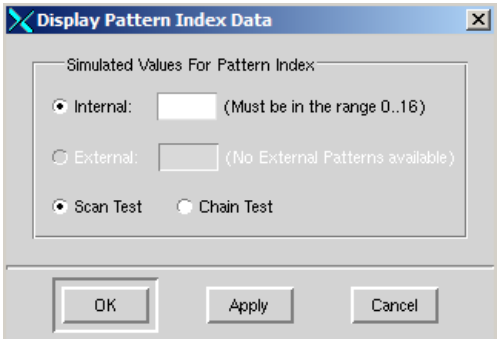
Usage Notes

- Add instances to the Debug window using any of the methods described in "Adding Instances to a Display Window" on page 49. One or more instances may also be added by the following menu selections:
  - **Tools > Analyze DRC** — Automatically adds instances associated with a particular DRC violation. See "Analysis of a DRC Violation" on page 81 for more information.
  - **Tools > Analyze Fault** — You can have the instance where an analyzed fault is located automatically added to the Debug window. See "Analyzing a Fault and Displaying its Location" on page 92 for more information.
- Select objects in the Debug window using any of the methods described in "Selecting Multiple Objects in the Debug and Design Windows" on page 51.
- View the library primitives of a specific instance in the Debug window from design level by double-clicking on the instance. The instance expands to show the library primitives and is surrounded by a bounding box. Note, the progress bar in the lower-left

of the Debug window indicates the progress of the expand operation. Double-click on the bounding box, to return to the instance view. Selecting **Display > Gate Level > Primitive** displays the primitives of the whole schematic sheet.

- Toggle the display of debugging information utilizing callout markers  on schematic objects as follows:
  - View the information in a callout (without expanding it) by hovering the mouse over the callout marker.
  - Expand a callout by clicking the callout marker using the left mouse button.
  - Move a callout box by pressing the Shift key and using the right mouse button to drag the box to a new location.
  - Close a callout by clicking the  in the upper right corner of the callout box using the left mouse button.
- View additional data in the Debug window by using the following options:
  - View Test_end data by selecting **Data > Test-End**. This action executes the set_gate_report command with the *test_end* option enabled or disabled.
  - Toggle the display of false paths on or off by clicking the Timing Exceptions  icon or selecting **Data > Timing Exceptions On | Off**. These actions execute the set_gate_report command with the *-timing_exceptions* option enabled or disabled.
  - Display faults by entering the add_faults command.
  - Display timing-aware data by selecting **Data > Delay Data**.
  - Display K19 and K22 simulation data by selecting **Data > K19 | K22**. This action executes the set_gate_report command with the K19 or K22 option enabled.
  - Display Named Capture Procedure simulation values by selecting **Data > Capture Procedure** or by executing the set_gate_report capture_procedure command from the transcript window; both methods report the value on each pin implied by the forced and conditional assignments defined in the specified named capture procedure (NCP).
  - Display simulation values for a specific simulation context by selecting **Data > Simulation Context** or by executing the “set_gate_report simulation_context” command from the transcript window; both methods report the simulated values for the current simulation context.
  - Display stable value data by selecting one of the **Data > Stable** menu items. This action executes the set_gate_report command with one of the STABLE_After_setup | STABLE_Capture | STABLE_Load_unload | STABLE_Shift options enabled or disabled. You can also access stable value data by clicking the following toolbar icons: Stable After Test_Setup  , Stable During Capture  , Stable During Load_Unload , and Stable During Shift  .

- Display scan test patterns or chain test patterns, by selecting **Data > Pattern Index** to display the Display Pattern Index Data dialog box. Note, the Chain Test option is only available after you have saved the scan test patterns.



- Display additional types of available data by selecting the **Data** pulldown menu, the corresponding buttons on the tool bar, or the set_gate_report command.

- Copy an object's hierarchical pathname into a buffer which you can then paste into another location by selecting the object(s) and choosing "<object_name(s)> (click to copy)" from the popup menu. For more information, see "Copying and Pasting Object Names in the Design."

- Access information about navigating schematics in the Debug window by referring to "Tracing Signal Paths on a Schematic" on page 59. In addition to tracing methods, this section also covers:

  - Compaction of Buffers and Inverters in Traced Circuitry

  - Annotation of Schematic Data in the Debug Window

- Create "what if" scenarios when debugging test coverage and other issues by executing the report_test_stimulus command for pins you select on a displayed schematic. See "Determining Test Stimulus" on page 93 for more information.

- To help improve performance, the schematic is automatically divided into multiple sheets when too many instances are visible. To navigate to a specific sheet enter its number in the sheet number entry box. To continue a trace to another sheet, double-click the marker on the net at the very right or very left of the schematic.

- In Tessent Diagnosis, use the **Tools > Diagnosis Report** menu item to display diagnosis suspects on a schematic. Refer to "Displaying Suspects in the Schematic View" in the *Tessent Diagnosis User's Manual* for complete information.

- If the Debug window is detached from and hidden behind the main window, click the  icon to bring the window back to the front.

Related Topics

Data Window

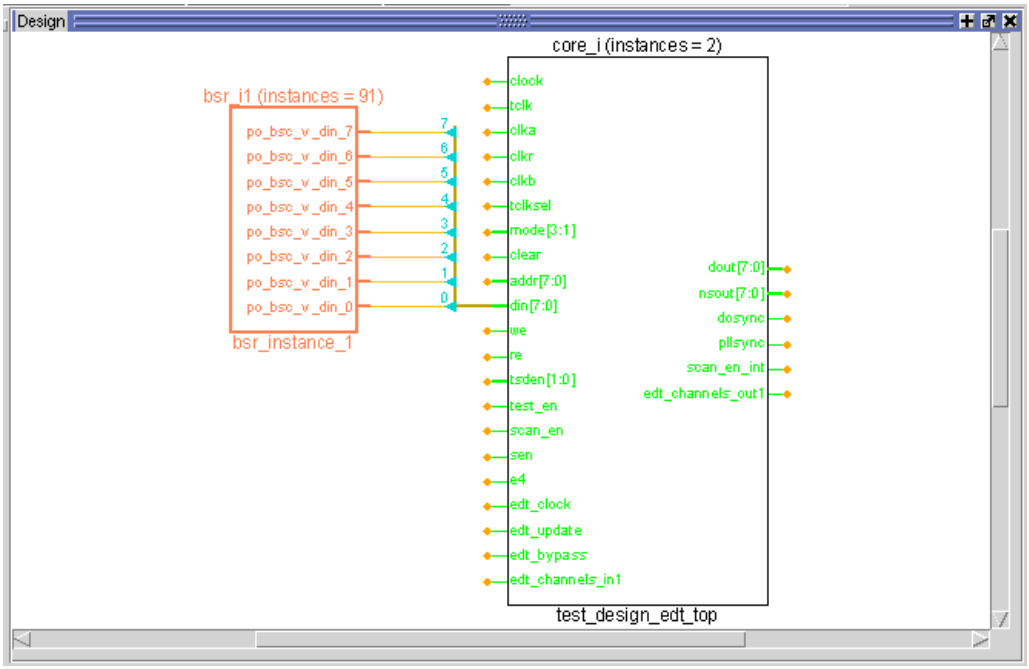
Design Window

Design Window

To access: Choose the **Windows > Design** menu item or click the  icon.

The Design window displays a hierarchical schematic of the design as described in the input netlist. The display includes net names and hierarchical ports and displays down to library level instances. The display shows the connections of individual bus bits as shown in the following figure. The window also displays at every hierarchical boundary the number of instances inside each hierarchical instance as shown in the following figure.

Figure 4-28. Design Window



Description

The Design window contains the following contents:

Window Area	Description
Selected Instance	Shows the type of a selected instance (submodule or design level gate) and its name.
Number of Instances Displayed (Selected)	Shows the number of instances being displayed (and how many of that number are currently selected).
Net Bundling Status	Shows whether net bundling (displaying nets connected between the same instances as single thick lines) is enabled. Controlled by **Display > Net Bundle** menu item.

Window Area	Description
Sheet Number Entry Box	Allows you to navigate to a specific sheet by entering that number in the entry-box. Also displays which sheet you are looking at and the total sheet count.

Usage Notes

- Add instances to the Design window using any of the methods described in "Adding Instances to a Display Window" on page 49.
- Select objects in the Design window using any of the methods described in "Selecting Multiple Objects in the Debug and Design Windows" on page 51.
- Expand an instance in the Design window by double-clicking on the instance. The instance expands to show the lower-level instances and is surrounded by a bounding box. Double-click on the bounding box, to return to the instance view.
- Copy an object's hierarchical pathname into a buffer which you can then paste into another location by selecting the object(s) and choosing "<object_name(s)> (click to copy)" from the popup menu. For more information, see "Copying and Pasting Object Names in the Design."
- Access information about navigating a schematic in the Design window by referring to:
  - "Tracing Signal Paths on a Schematic" on page 59
  - "Signal Path Tracing in the Design Window" on page 62
  - The example under "Step 2 - Find the Source of Problem Signals" on page 82
  - "Getting Oriented in a Large Design" on page 94
- To help improve performance, the schematic is automatically divided into multiple sheets when too many instances are visible. To navigate to a specific sheet enter its number in the sheet number entry box. To continue a trace to another sheet, double-click the marker on the net at the very right or very left of the schematic.
- If the Design window is detached from and hidden behind the main window, click the  icon to bring the window back to the front.

Related Topics

Data Window
Debug Window

Configuration Data Window

To access:

- From DFTVisualizer, choose **Windows > Configuration Data** from the pulldown menu or click the  icon.
- From the command line within a tool session, enter the following command:

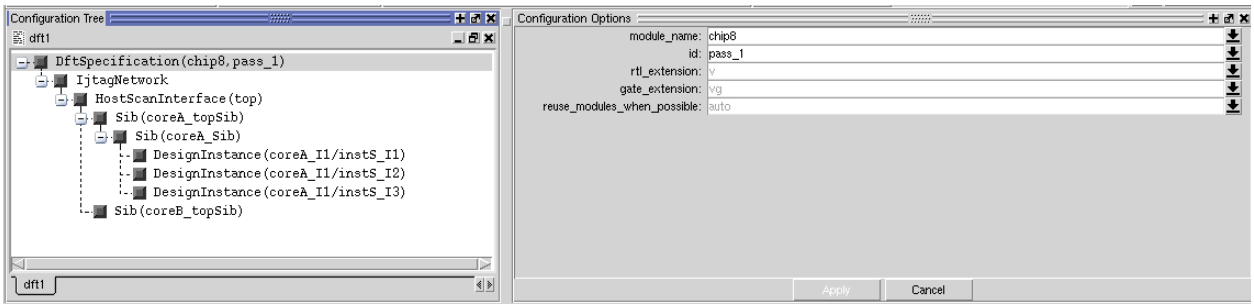
  open_visualizer -display configuration_data

- To automatically load existing configuration data from the command line within a tool session, enter the following command:

  read_config_data <configuration_file_name>
  display_configuration

The Configuration Data window provides you with a graphical interface to view specifications and to modify the configuration tree elements and their options.

Figure 4-29. Configuration Data Window



Description

The Configuration Data window provide the following capabilities:

- Configuration Tree — Enables navigation and modification of the entire specification hierarchy. Displays a tab for each defined specification. You perform most operations in this panel by clicking the right mouse button and selecting from the available menu options as described in Table 4-8.
- Configuration Options — Displays options for the element selected in the Configuration Tree panel. The content of the Configuration Options tab is specific to the element selected in the Configuration Tree panel. The options displayed in this panel are defined for the element type in the DftSpecification. For example, the options displayed in this panel for the Sib are defined in the Sib wrapper of the DftSpecification; the options available for the interface of the Sib are defined in the Sib/Interface wrapper of the DftSpecification.

The Configuration Data window uses the following toolbar icons:

Table 4-8. Configuration Data Window Icons

Icon	Description
	Validates the current specification and reports any errors. This icon skips Dft element generation and insertion. This action is equivalent to issuing the “process_dft_specification -validate_only” command.
	Executes the current specification by generating the Dft components such as the RTL and the ICL description, and inserting them into the design. This action is equivalent to issuing the process_dft_specification command.

Usage Notes for the Configuration Tree

You can access the following options in the Configuration Tree by clicking the right mouse button and selecting a menu item:

Menu Item	Description
Move Up | Move Down	Moves the selected element up or down within the configuration tree. You can also move the location of the element by clicking the left mouse button on the element, dragging it to the desired position within the configuration tree, and releasing the mouse button.
Cut | Copy | Paste	Cuts or copies a selected element, or pastes a previously cut or copied element. This is a time-saving feature that allows you to create new elements faster by using existing elements as a starting point. Once you copy and paste a new element in the Configuration Tree, you can customize it in the Configuration Options panel.
Expand | Collapse	Expands or collapses the selected tree hierarchy.
Add | Delete	Creates a new element or deletes existing ones. The **Add** sub-menu is a context-sensitive list of elements that can be legally added under the selected element. For example, several elements such as a SIB, TDR, ScanMux, or design element can be added to an existing SIB but only Inputs can be added to a ScanMux. The tool only lists the valid elements based on the selected element.
Config Options	Displays the Configuration Options panel if it is not visible.
Close Tab	Closes the current tab in the Configuration Tree. Closing the tab does not delete the specification or in any way modify it in tool memory; it simply removes it from display.

Usage Notes for the Configuration Options

- The options displayed in the Configuration Options panel are defined for the element type in the DftSpecification. For example, the options displayed in the Configuration Options panel for the Sib are defined in the Sib wrapper of the DftSpecification; the options available for the interface of the Sib are defined in the Sib/Interface wrapper of the DftSpecification.
- Specify the parameters of the newly added element in the Configuration Options panel as follows:
  - When entering the values of the element's listed options, use the  icon to select from pre-defined values for that element.
  - Click Interface [+] to define parameters of the element's interface.
  - Click DataInPorts [+] and DataOutPorts [+] to define the number of ports on the element, their names, and connections.
  - Click Attributes [+] to define arbitrary attributes on the element's ICL module.
- In the Configuration Options panel:
  - Default option values are shown in gray text.
  - User-defined option values display in black text.
  - Changes you make to the options are not saved until you click the **Apply** button. If you make changes and do not click the **Apply** button before leaving the panel, the tool displays a dialog box to confirm whether you want to save or discard the changes.

Related Topics

Modifying the Contents of the Configuration Data Window

Adding a Multiplexer to a SIB Example

Adding a Test Data Register to a SIB Example

Global Search Window

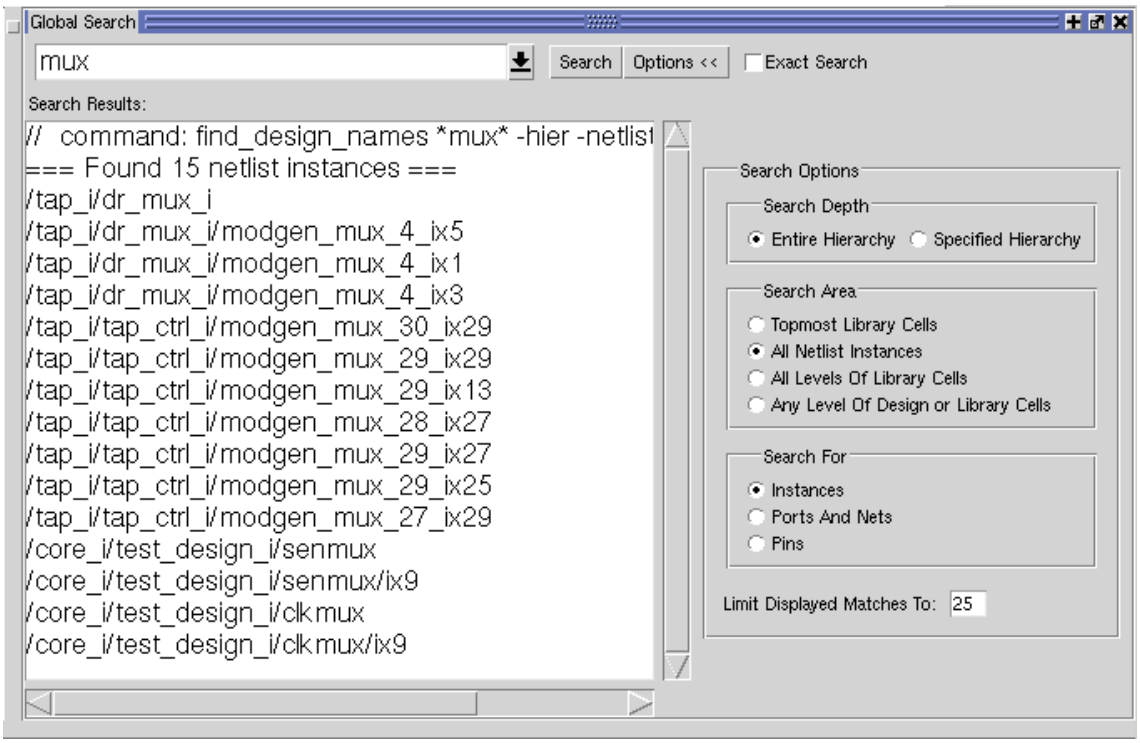
To access: Choose **Windows > Global Search** or click the Search icon  .

Use the Global Search window to search for any instance, net, or pin in the design.

Note

You can use the Find icon  on the menu bar to specify a search that is limited to the active window.

Figure 4-30. Global Search Window



Usage Notes

- Enter a string in the Global Search text entry box to search for any occurrence of that string. No wildcards are needed. The tool returns every instance of that string in any pathname of the active design as shown in Figure 4-30. Alternatively, you can enable the Exact Search field to disable the use of implicit wildcards and only search for the exact string.
- Click the icon next to the search field to display a search history.
- Click the **Options** button to toggle the display of additional search parameters as shown on the right of Figure 4-30.
- Click on a result to copy it to the copy/paste buffer or drag-n-drop (using Control-left click). For more information, see “Viewing Instances in Other Windows” on page 54.

Note

You can cancel the search operation by clicking **Cancel** in the Global Search dialog box. The **Cancel** button displays when the search operation starts to display results; you may not see the button unless the Limit Displayed Matches To: field is set to a large number.

- Select an object(s) and choose the **View In >** menu item from the popup menu to view objects in one or more other windows. You can select multiple objects by holding down the Shift key while clicking on the objects you want to select; when you release the Shift

key, the set of objects you clicked on remains selected and available to View in another window.

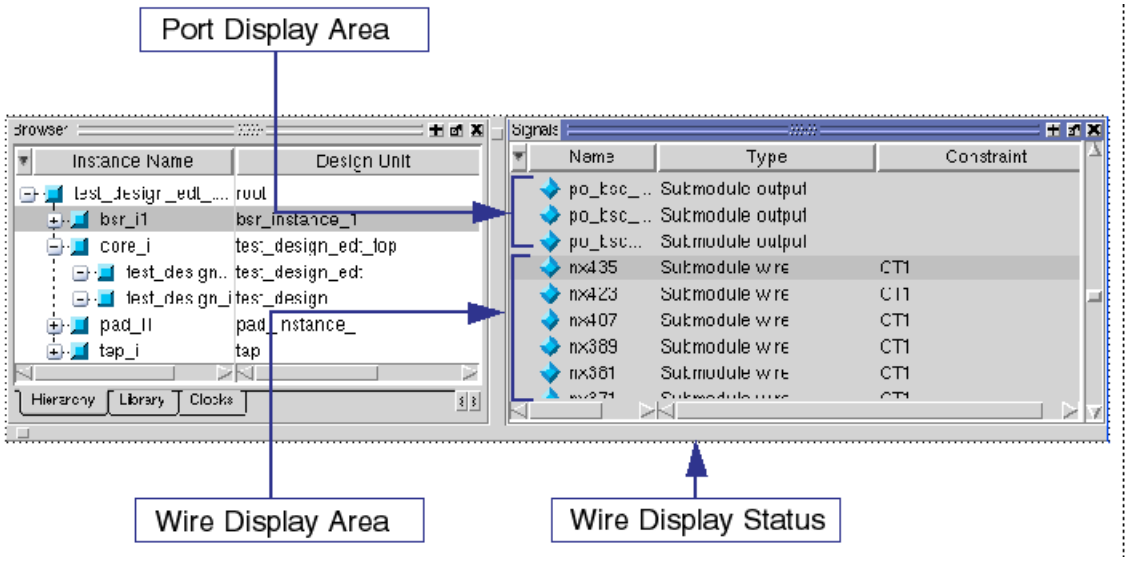
- If the Global Search window is detached from and hidden behind the main window, click the icon to bring the window back to the front.
- Use **File > Save As** or click to save a text, comma separated value (CSV), or graphical version (screen capture) of the Global Search window contents to a file.

Signals Window

To access: Choose **Windows > Signals**.

The Signals window lets you view ports and signals for instances selected in the Browser window.

Figure 4-31. Signals Window



Description

The Signals window contains the following contents:

Window Area	Description
Port Display Area	Shows the name, type of each port, and constraints of the instance selected in the Browser.
Wire Display Area	Shows the names of wires inside the selected instance (if it is a submodule and wire display is enabled) in the Browser.
Wire Display Status	Shows whether the display of internal wires is enabled for the Signal window (**Display > Display Nets > On**).

Usage Notes

- Display signals in alphabetical order by clicking the Name column header; to view signals sorted by type, click the Type column header.
- Display wires (nets) that are inside a submodule selected in the **Hierarchy** tab of the Browser window by choosing the **Display > Display Nets > On** menu item. This option is valid for submodules only.
- Show the names of individual bus pins by clicking the plus sign (+) next to the bus's name. To remove the names of individual bus pins, click the minus sign (-) next to the bus's name.
- The Signals window has the same right mouse popup menu options as the Browser window.

Related Topics

Browser Window

Test Structures Window

To access:

- From DFTVisualizer, choose **Windows > Test Structures** from the DFTVisualizer pulldown menu or click the  icon.
- From the command line within a tool session, enter the following command:

```
open_visualizer -display test_structures
```

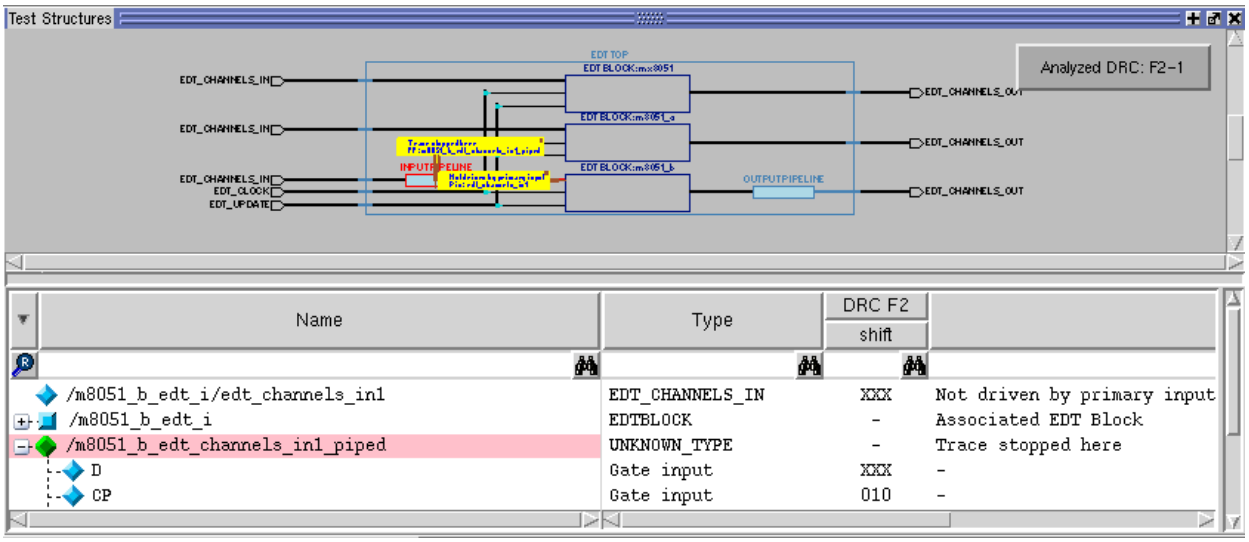
You can use the Test Structures window to do the following:

- Browse a virtual graphical representation of the EDT logic as shown in Figure 4-32. To display actual net and pin mapping information graphically, you must add the EDT component to the Debug or Design window.
- Display textual information about components within the EDT logic.
- Debug DRC violations related to the EDT logic.

Note

You must first run set_edt_finder on a design to gather EDT logic information before you can view it in the Test Structures window. You can then view EDT logic in the Test Structures window at any time prior to test pattern generation; EDT Finder data is not preserved during ATPG.

Figure 4-32. Test Structures Window



Usage Notes

- Double-click graphic blocks to descend down and display internal components.
- Click on graphic objects to display information about the object in the text pane.
- Click callout boxes on graphic objects to toggle the display of DRC analysis information. Press the Shift key and use the right mouse button to move the callout box markers.
- Click on the plus sign preceding a device in the text pane to expand text to include device pin information.
- Double-click on a device in the text pane to display it in the Design window.
- Search/Filter the contents of the text pane as follows: Click in the top of a text pane column, type a term to search/filter on, and click on the binoculars. The contents of the column is filtered so the specified term displays at the top. Click the magnifying glass at the top of the first column to clear all filter entries.
- Copy an object's hierarchical pathname into a buffer which you can then paste into another location, by selecting the object(s) and choosing "<object_name(s)> (click to copy)" from the popup menu. For more information, see "Copying and Pasting Object Names in the Design."
- If the Test Structures window is detached from and hidden behind the main window, click the  icon to bring the window back to the front.

Related Topics

Tessent TestKompress User's Manual

Text Editor Window

To access, do one of the following.

- Choose **Windows > Text Editor** from the DFTVisualizer pulldown menu.
- Execute the following command from the tool command line:

  open_visualizer -display text_editor

- Click the right mouse button (RMB) on an instance in any of the following DFTVisualizer windows: Debug, Design, Data, Wave, Browser, or Global Search. Choose one of the **View in Text Editor > Defining Text | Instantiating Text** menu items.
- Click on a hyperlink (pink underlined text) in the Transcript window.

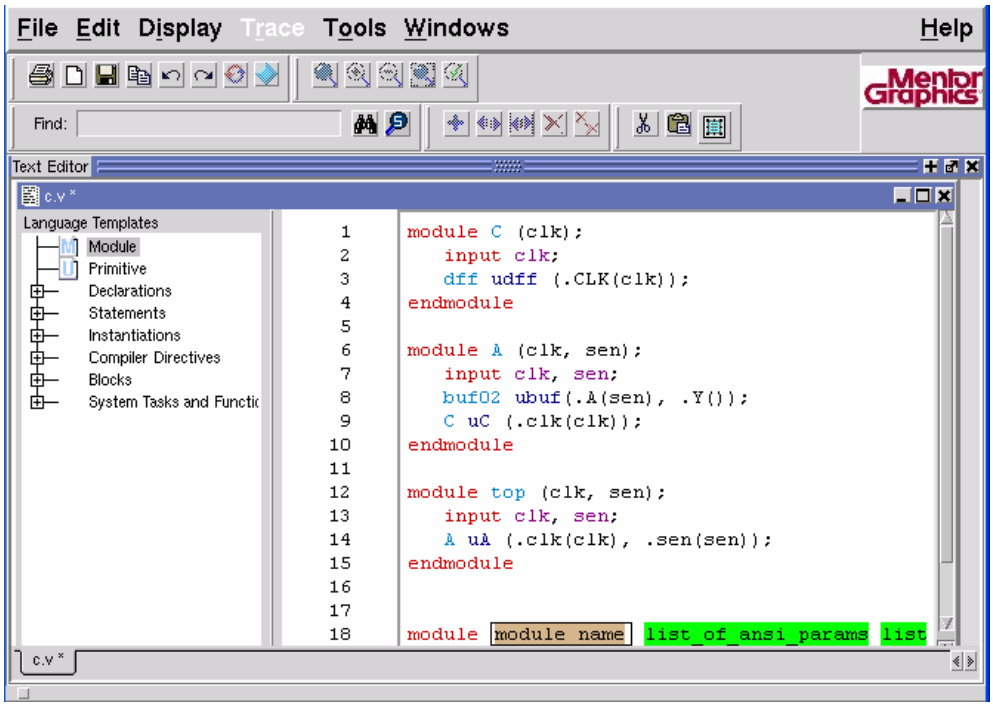
Use the Text Editor window to do the following:

- View currently loaded design files: netlists, ATPG library, test procedure files, startup files, and dofiles.
- View the definition and instantiation of instances in a Verilog netlist or ATPG library.
- Create, edit, and save test procedure files, dofiles, and startup files.
- Troubleshoot test procedure files.
- Locate and debug test procedure file errors during DRC.
- Use built-in Verilog, VHDL, and test procedure file templates for making on-the-fly changes.

Note

DFTVisualizer supports all of these operations for compressed netlist and ATPG library files with any of the following file extensions: “.Z”, “.gz”, “.gzip”, and “.zip”.

Figure 4-33. Text Editor Window



Usage Notes

The Text Editor contains standard text editor functions. You access these functions from the **File** and **Edit** menus and by clicking the right mouse button (RMB) to display the popup menu.

You can specify the type of syntax highlighting to be applied to the contents of the Text Editor by selecting the **Display > Syntax Highlighting** menu and choosing from the following syntaxes: Verilog, VHDL, ATPG Library, Test Procedure, or Dofile. By default, the Text Editor highlights text using Verilog syntax.

The Text Editor also provides:

- Access to any design files currently loaded in DFTVisualizer. To view currently loaded Verilog netlist files, dofiles (including startup files), the ATPG library, and the test procedure file, choose **File > Open > Current Design Files** or click the  icon and select the file you want to open.
  - To open all currently loaded design files, choose **All**. Be aware that if the tool is invoked on a flat model, the netlist, ATPG library, and test procedure file are not opened. If a dofile or startup file is specified, it is opened using this option.
  - To open any subset of the currently loaded design files, choose from the individual files listed under the submenus: **Netlist | ATPG Library | Dofiles**.
- Display options such as window positioning options, font sizes, and line numbers. To set display options, choose the **Display > Windows | Font | Line Number** menu items.

- Search and replace operations using ASCII text or regular expressions. To search and/or replace, choose **Edit > Search | Replace**. To use regular expressions, click **More** and enable the Regular Expression option.
- Support for **vi** and **emacs** text editor key bindings. To access, choose **Display > Key Stroke Mode**.
- Support for viewing and modifying compressed netlist and ATPG library files. To open a ZIP file, choose **File > Open > Other** or click the  icon, select the compressed file to open from the Open File dialog box, and click **OK**. Each compressed file you open displays as a tab in the Text Editor.

Note

DFTVisualizer supports all of these operations for compressed netlist and ATPG library files with any of the following file extensions: ".Z", ".gz", ".gzip", and ".zip".

- Cross-highlighting between a selected instance and its description. To display a selected instance in the Debug, Design, Data, Wave, Browser, or Global Search window, choose one of the **View In Text Editor > Defining Text | Instantiating Text** menu items:
  - **Defining Text** — displays the Verilog module definition for gate instances or the ATPG library model for library instances and primitives.
  - **Instantiating Text** — displays the Verilog instantiation in the netlist.

  If the tool is invoked on a flat model, the **View in Text Editor** option is disabled.

Note

Some designs contain multiple definitions of library models or Verilog modules in the ATPG library and netlist files. The Text Editor will only display the definition that is in use by the tool; this is determined by the priority used when the library and Verilog files are parsed at tool invocation.

- Templates for DFT-specific operations. To access the templates, choose **Display > Show Template** and click a specific template item to insert the template into the currently-opened file.

Transcript Window

To access:

- Choose **Windows > Transcript** from the DFTVisualizer pulldown menu.
- Execute the following command from the tool command line:

  open_visualizer -display transcript

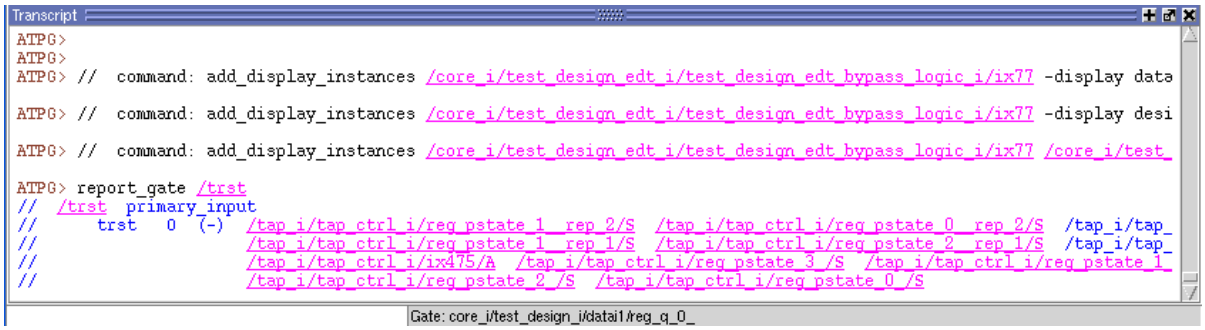
Note

The Transcript window opens by default when DFTVisualizer invokes.

Use the Transcript window to do the following:

- Enter and execute tool commands.
- View color-coded log file information.
- View instances reported by tool in schematic windows.
- Open documentation for reported DRC violations.
- View notes, warnings, and errors applicable to the current session.

Figure 4-34. Transcript Window



Usage Notes

- All tool commands can be executed in the Transcript window. Tool responses are displayed in the Transcript window as well as in the shell window. Tool responses are color-coded for better readability and easy identification of important messages.
- A dofile of commands executed in the Transcript window can be created. To write a dofile of the commands shown in the Transcript window, choose the **File > Create Dofile** menu item. Specify the name of the dofile to create in the Selection field of the Create Dofile dialog box. All of the commands output by the **Display > Filter > Command** menu item are saved to the dofile with the exception of comment characters "// command:" which are omitted.

Note

The dofile is saved to the path specified in the Selection field. If you do not specify an absolute path, the dofile is saved in the current directory.

- The contents of the Transcript window can be filtered. To filter the contents of the Transcript window, choose one of the **Display > Filter** menu items described in Table 4-9.

Table 4-9. Transcript Window Contents

Filter	Description
Error	Only errors are displayed in the Transcript window. Errors display in red font.
Warning	Only warnings are displayed in the Transcript window. Warnings display in green font.
Command	Only commands are displayed in the Transcript window. Commands display in black font.
Show All	All errors, warnings, and commands are displayed in the Transcript window and each displays in their respective font color. Report output is also displayed in blue font.

- You can quickly view the contents of the Transcript window by clicking and holding down the middle mouse button while moving the mouse up and down to view the desired text.

Related Topics

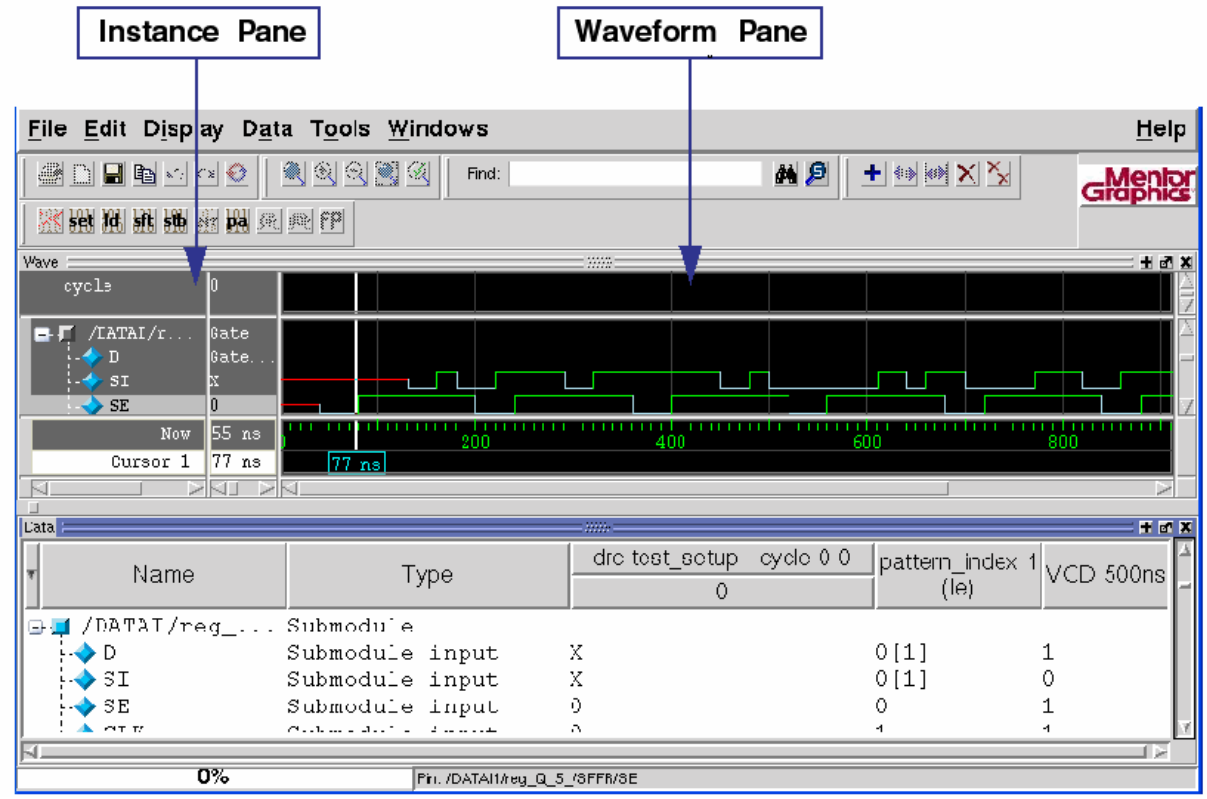
write_visualizer_dofile command

Wave Window

To access: Choose the **Windows** > **Wave** menu item or click the  icon.

The Wave window displays a waveform representation of the simulation results for the test_setup procedure, test_end, VCD, and named capture procedures.

Figure 4-35. Wave Window



Description

The Wave window contains the following contents:

Window Area	Description
Instance Pane	Shows the same instances as the Data window.
Waveform Pane	Shows a waveform representation of the same test_setup procedure data shown in the Data window.

Usage Notes

- To use the Wave window, first add instances and the test_setup column to the Data window using the **Data** > **Test-Setup** menu item; then, open the Wave window.
- To specify to display a specific time range in the Wave window, activate the Wave window and select **Data** > **VCD** from the pulldown menu.

- To view Test_end data in the Wave window, select the **Data > Test-End** menu item.
- To view Named Capture Procedure (NCP) data in the Wave window, select the **Data > Named Capture** menu item which executes the report_procedures command for the specified NCP.
- To copy an object's hierarchical pathname into a buffer which you can then paste into another location, select the object(s) and choose "<object_name(s)> (click to copy)" from the popup menu. For more information, see "Copying and Pasting Object Names in the Design."
- If the Wave window is detached from and hidden behind the main window, click the  icon to bring the window back to the front.

Tip: The Wave window works together with Data window, automatically showing the same instances and test_setup data as the Data window, but showing the data as a waveform with timing information.

Related Topics

Data Window

DFTVisualizer Command Quick Reference

The following table provides a brief summary of tool commands that are specifically for operating DFTVisualizer.

Table 4-10. Command Summary

Command	Description
add_browser_data	Adds data columns to the active tab of the Browser window of DFTVisualizer.
add_display_data	Adds data columns to the Data window of DFTVisualizer.
add_display_instances	Displays instances in DFTVisualizer and enables you to trace visually through the design using the Debug or Design window.
analyze_drc_violation	Generates a netlist of the portion of the design involved with the specified rule violation number.
close_visualizer	Closes the DFTVisualizer window.
delete_browser_data	Removes data columns from the Browser window of DFTVisualizer.
delete_display_data	Removes a data column from the Data window of DFTVisualizer.
delete_display_instances	Removes the specified instances from a DFTVisualizer display window.
display_diagnosis_report	Opens a diagnosis report in which you can click symptoms and suspect locations to add related gates to the DFTVisualizer Debug window.
mark_display_instances	Changes the color of the specified instances in the Debug/Design window.
open_visualizer	Opens the DFTVisualizer main window.
read_visualizer_preferences	Reads a DFTVisualizer preferences file and sets current preferences as described in the file.
report_display_instances	Lists netlist information for specified instances displayed in the Debug window.
select_display_instances	Selects the specified objects in the Debug and/or Design window.
set_visualizer_logging	Writes the commands entered into the Transcript window to the file specified by the *enhanced_dofile* argument.

Table 4-10. Command Summary (cont.)

Command	Description
set_visualizer_preferences	Controls a subset of DFTVisualizer preferences for the Debug, and Design, Browser, and Data windows.
unmark_display_instances	Removes color highlighting and/or marking from instances in the Debug window.
unselect_display_instances	Unselects the specified objects in the Debug and/or Design window.
write_visualizer_dofile	Writes a dofile containing commands needed to recreate current instance and data displays.
write_visualizer_preferences	Writes the current DFTVisualizer preference settings to a file.
write_window_contents	Saves a screen capture of a DFTVisualizer window.

Chapter 5
Simulation Contexts

Tessent Shell provides simulation contexts for design analysis and introspection.

Simulation Context Overview

Simulation contexts are useful for design analysis and introspection. Tessent Shell provides commands for creating and managing simulation contexts. From a particular user-defined simulation context, you can use these commands to apply stimulus forces to specified gate_pins, run simulation for a specific number of cycles, and then introspect gate_pins for simulation values. This gives you the capability to create “simulation scratch pads” for rapid investigation of good-machine behavior of specific portions of your design.

The following sections describe the commands and attributes that support simulation contexts:

- Commands for Managing Simulation Contexts
- Commands for Managing Stimulus and Simulating within Simulation Contexts
- Commands for Introspection and Analysis within Simulation Contexts
- Attributes for gate_pin Objects

Commands for Managing Simulation Contexts

The following commands are for creating and managing user-defined simulation contexts, as well as for using predefined simulation contexts:

- add_simulation_context — Creates a new user-defined simulation context.
- copy_simulation_context — Copies the simulation values and forces from one simulation context to another.
- delete_simulation_contexts — Deletes one or more user-defined simulation contexts.
- get_current_simulation_context — Returns the name of the current simulation context.
- get_simulation_context_list — Returns the available simulation contexts in a Tcl list.
- report_simulation_contexts — Lists the available simulation contexts and indicates the current simulation context.
- set_current_simulation_context — Sets the current simulation context.

Commands for Managing Stimulus and Simulating within Simulation Contexts

The following commands are for managing stimulus (simulation forces and clock pulses) and running limited simulations within a simulation context:

- add_simulation_force — Forces one or more gate_pin objects to the specified value.
- delete_simulation_forces — Removes forces from one or more gate_pin objects.
- report_simulation_forces — Lists the active forces on the specified gate_pin objects.
- simulate_clock_pulses — Pulses one or more clocks within the current simulation context.
- simulate_forces — Simulates the queued forces in the current simulation context.

Commands for Introspection and Analysis within Simulation Contexts

The following commands are for introspection and analysis within simulation contexts, which includes examining simulation results:

- get_current_simulation_context — Returns the name of the current simulation context.
- get_simulation_context_list — Returns the available simulation contexts in a Tcl list.
- get_simulation_value_list — Returns the simulation values on the specified gate_pin objects.
- report_simulation_contexts — Lists the available simulation contexts and indicates the current simulation context.
- report_simulation_forces — Lists the active forces on the specified gate_pin objects.
- set_gate_report — Specifies the information displayed by the report_gates command. This command allows reporting of simulated values in the current simulation context.
- trace_flat_model — Traces the flat model within the current simulation context. This command always operates within the current simulation context.

Attributes for gate_pin Objects

Simulation contexts use the following attributes for gate_pin objects:

simulation_force_value	String that contains the value specified with the add_simulation_force command for the gate_pin objects that are forced in the current context. For gate_pins that are not forced, this attribute contains an empty string, which is the default value.
simulation_value	String that contains the simulation value on a gate_pin.

For a complete list of attributes, refer to the "Data Models" chapter in the *Tessent Shell Reference Manual*.

Simulation Context Data in DFTVisualizer

DFTVisualizer can display the simulation context values in the Debug window after you issue the “set_gate_report simulation_context” command or select the **Data > Simulation Context** menu item.

For more information on displaying simulation data, see the “Debug Window.” For information on setting the number of simulation characters displayed, see the “Schematics Preferences Dialog Box.”

Introspection and Analysis Using Simulation Contexts

Simulation contexts allow you to perform various types of design analysis and introspection. The four predefined simulation contexts are: stable_after_setup, stable_load_unload, stable_shift, and stable_capture. After setting a simulation context, you can introspect gate_pins for simulation values based on that specific simulation context. For example, the trace_flat_model command can do design tracing based on values specified for the current simulation context.

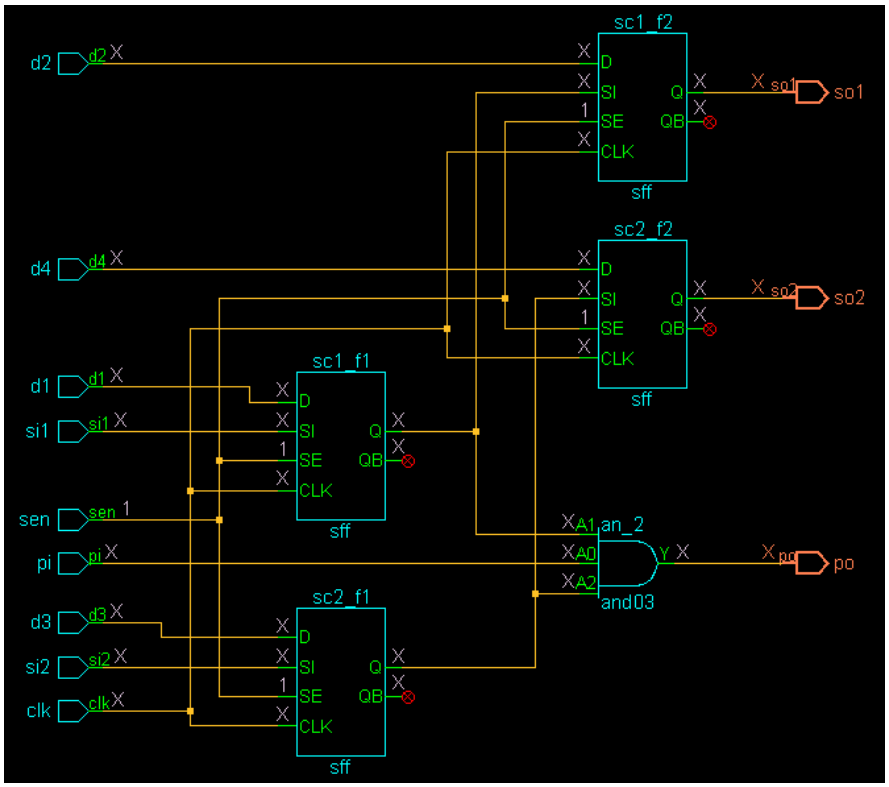
In order to use any of the following techniques in this chapter, you must be:

- Operating in the analysis system mode of Tessent Shell.
- Operating on a flattened design, and have read in the test procedure file (if available). At this point, the values specified in the setup, shift, capture, and load_unload procedures are in place in the design when you set the corresponding simulation context as current.

Example

For example, if you want to trace the design based on the values specified in the shift procedure, use the “set_current_simulation_context stable_shift” command. When you use this command with a small design containing two 2-cell scan chains, the values on the gate_pins display in the DFTVisualizer Debug window as shown in Figure 5-1.

Figure 5-1. DFTVisualizer Debug Window (gate level)



In this case, no values are forced except for sen (scan enable), which is set to 1. Note that the four possible simulation values are 0, 1, X, and Z.

In addition to viewing values in the Debug window, you can get the current simulation values of gate_pins using any of the following techniques:

- Use the get_simulation_value_list command and specify the desired gate_pin object(s):

  ANALYSIS> **set_current_simulation_context stable_shift**
  ANALYSIS> **get_simulation_value_list sc2_f1/D**
  X

- Check the value of the simulation_value attribute for the gate_pins:

  ANALYSIS> **get_attribute_value_list [get_gate_pin sc2_f1/D] -name simulation_value**
  X

- Issue the command “set_gate_report simulation_context,” after which you can use the report_gates command and the Debug window to display the simulated values from the current simulation context:

  ANALYSIS> **set_gate_report simulation_context**
  ANALYSIS> **report_gates sc2_f1**

```
//  /sc2_f1  sff
//       D      I  (X)  /d3
//       SI     I  (X)  /si2
//       SE     I  (1)  /sen
//       CLK    I  (X)  /clk
//       Q      O  (X)  /an_2/A2  /sc2_f2/SI
//       QB     O  (X)
```

You can use simulation context functionality for different types of analysis. For example, by default the trace_flat_model command uses the stable_after_setup values as a simulation background. You can now change it to stable_capture, for example, if you want to trace based on sensitized paths during capture. Another example is to copy an existing simulation context to a new one, force some simulation value changes, then evaluate the results in the circuit after simulating the forces and/or clock pulses.

Chapter 6
Test Procedure File

Test procedure files specify how the scan circuitry within a design operates. The scan circuitry operation is specified using previously-defined scan clocks and other control signals. In order to operate the scan circuitry in your design, you must define the scan circuitry and provide a test procedure file to describe its operation. The design rules checking (DRC) process, which occurs when you exit setup mode, performs extensive checking to ensure the scan circuitry operates correctly.

Test Procedure File Creation

You insert scan circuitry and create test procedure files for ATPG operations using the “patterns -scan” context of Tessent Shell. If your design already contains scan circuitry, you will need to create a test procedure file to describe its operation either by hand or with Tessent Shell.

You can specify a test procedure file in setup mode using the add_scan_groups command. The tools can also read in procedure files by using the read_procfile command or the write_patterns command when not in setup mode. When you load more than one test procedure file, the tool merges the timing and procedure data.

You can also have the **stil2mgc** tool translate a STIL Procedure File (SPF) into a dofile and test procedure file. This tool produces a dofile that defines clocks, scan chains, scan groups, and pin constraints. This tool also creates test procedure files with a timeplate and the following standard scan procedures: test_setup, load_unload, and shift. For more information about **stil2mgc**, refer to “Notes About Using the stil2mgc Tool” on page 238.

The following subsections describe the syntax and rules of test procedure files, give examples for the various types of scan architectures, and outline the checking that determines whether the circuitry is operating correctly.

Test Procedure File Syntax

The following section describes the syntax of the test procedure file.

Syntax Conventions

The following syntax conventions are used in this chapter:

- **Bold** — Indicates a keyword. Enter the keyword exactly as shown.
- *Italic* — Indicates lexical elements such as identifiers, strings, or numbers. Replace the italicized word with the appropriate name or integer.
- | — A vertical bar or pipe character indicates a logical “OR” as in “select foo OR foo_not”.
- [ ] — Square brackets indicate optional elements. Do not include the brackets.
- … — An ellipsis indicates a repeatable item or set.

Reserved Characters

If you have a pin or pathname that uses a reserved punctuation character, you must enclose that name in double quotes. See Table 6-1 for a list of reserved punctuation characters.

For example, the following statement is illegal because it uses the exclamation point outside of double quotes.

```
force /inst_my_adder_1/xclk_header!x1!x1/op1[9] 1
```

The signal name contains a reserved punctuation character, the exclamation point (!), so it must be enclosed inside double quotes. The correct syntax would be:

```
force "/inst_my_adder_1/xclk_header!x1!x1/op1[9]" 1
```

Table 6-1. Reserved Punctuation Characters

Name	Character
Ampersand/AND	&
Caret/Circumflex/XOR	^
Comma	,
Equals	=
Exclamation mark	!
Left/Opening brace	{

Table 6-1. Reserved Punctuation Characters

Name	Character
Left/Opening parenthesis	(
Right/Closing brace	}
Right/Closing parenthesis	)
Semicolon	;
Vertical bar/OR	|

Throughout this chapter, value = 0, 1, X, or Z.

Using Tcl in the Test Procedure File

Procedure files support the Tcl conditional statements “if”, “else”, and “elseif” using the following syntax:

```
if { tcl_expr }  {
  procedure file statements
}
elseif { tcl_expr } {
  procedure file statements
}
else {
  procedure file statements
}
```

Where a “tcl_expr” is any Boolean Tcl expression that uses Tcl variables, dofile variables, or environment variables. Just as when doing variable substitution in the procedure file, other Tcl statements and defining Tcl variables are not supported. All variables must be defined in the dofile or from the shell as environment variables.

The body of these Tcl conditional statements should contain only legal procedure file syntax, not any other Tcl statements. The Tcl conditional statements are treated as preprocessor statements in the procedure file parser. They are not preserved in the tool after parsing is finished; only the procedure file code selected by the evaluation of the Tcl “if” expression is stored in the tool. Therefore, when using write_procfile to write out the procedure file, none of the Tcl conditional statements are present, and the procedure file code not used is also not present. For more information refer to “Using the Tessent Tcl Interface” on page 241.

Introductory Test Procedure File Example

The following is an example of a simple test procedure file.

```
// Comments use "//" characters
//
// Set the base time increment for use in all timeplates
set time scale 1.0 ns;

// Define the strobe time for the measure statements
set strobe_window time 1;
```

```
// This design uses a single timeplate, named “tp1”, for all
// vectors.

timeplate tp1 =
    force_pi 0;
    measure_po 1;
    pulse CLK0_7 2 1;
    pulse CLK8_15 2 1;
    period 4;
end;

// The shift and load_unload procedures define how the design
// must be configured to allow shifting data through the scan
// chains. The procedures define the timeplate that will be
// used and the scan group that it will reference.

procedure shift =
    scan_group grp1;
    timeplate tp1;
    cycle =
        force_sci;
        measure_sco;
        pulse CLK8_15;
        pulse CLK0_7;
    end;
end;

procedure load_unload =
    scan_group grp1;
    timeplate tp1;
    cycle=
        force CLEAR 0;
        force CLK0_7 0;
        force CLK8_15 0;
        force scen1 1;
    end;
    apply shift 8;
end;

// The capture procedure is a "non-scan" procedure. This
// procedure describes the timeplate that will be used for the
// capture cycle. It also defines the number of cycles that
// will be used in the capture cycle. In this example there is
// just one cycle.

procedure capture =
timeplate tp1;
    cycle =
        force_pi;
        measure_po;
        pulse_capture_clock;
    end;
end;
```

Test Procedure File Structure

The test procedure file consists of the following structural elements. For more information about each element, refer to the page listed below:

```
#include "<file_name>"; // page 159
[set_statement ...]     // page 159
[alias_definition ...]  // page 162
[timing_variables ...]  // page 164
timeplate_definition [timeplate_definition] // page 167
always_block            // optional block defintion page 171
procedure_definition [procedure_definition] // page 172
```

#include Statement

The "#include" statement specifies that the tool read test procedure data from a specified file.

The following rules apply to #include statements and files:

- The "#include" statement can occur anywhere in the file, and multiple "#include" statements can occur in one file. For example:

  ```
  #include "foo.proc";
  ```

- The file name to be included must be enclosed in double quotes, and the statement must be followed by a semicolon.
- All timeplates and procedure rules apply to the statements placed in #include files.
- Included files can use the "#include" statement to include other files, up to a maximum include depth of 512. If you later use the write_procfile command write out procedure data, the "#include" statements are not preserved, and the tool writes all procedure data to a single file.

Set Statement

The Set statements define specific parameters used throughout the test procedure file.

The following statements are available:

set time scale *tscale*;

Defines the time scale and unit. The "set time scale" statement must be at the beginning of the procedure file, before any timeplate or procedure definition. If you do not specify the time scale, the default value is 1 ns.

The tool applies the time scale and unit to the test procedure file and timeplates. The *tscale* you specify can be any real number. Time values in the timeplate, however, must be integers, representing whole time scale units. If you find you are specifying fractional times in the

timeplate, you must reduce the time scale unit so you can specify integer time values in the timeplate. For example, the following would result in a syntax error:

```
set time scale 1 ns ;
set strobe_window time 1 ;

 timeplate fast_clk_tp =
    force_pi 0 ;
    measure_po 0.500 ;
    pulse CLKA 0.750 1.50 ;
    pulse CLKB 0.750 1.50 ;
    period 3.000 ;
 end ;
```

To correct the syntax, you could change the time scale to picoseconds, and adjust the time value to meet the scale as follows:

```
set time scale 10 ps ;
set strobe_window time 1 ;

 timeplate fast_clk_tp =
    force_pi 0 ;
    measure_po 50 ;
    pulse CLKA 75 150 ;
    pulse CLKB 75 150 ;
    period 300 ;
 end ;
```

The units supported are ms, us, ns, ps, and fs.

The tool translates the time scale in the procedure file into a Verilog `timescale directive in the Verilog testbench when writing patterns in Verilog format.

If the time scale number you specify in the test procedure file is 1 or larger, the resulting Verilog `timescale directive has the same time unit (resolution) and time precision. For example, “set time scale 1 ns ;” would result in this Verilog directive:

```
`timescale 1ns / 1ns
```

If you want the testbench to have smaller precision than resolution, there are several ways to designate this:

- Specify a time scale number of less than 1 in the procedure file. For example, “set time scale 0.5 ns ;” produces this Verilog directive:

  ```
  `timescale 1ns / 100ps
  ```

- Add *non-zero* significant bits to the time scale in the procedure file. For example, “set time scale 10.05 ns ;” produces this Verilog directive:

  ```
  `timescale 1ns / 10ps
  ```

- Add trailing zeros as significant bits for an asynchronous clock period or pattern_set period (when creating IJTAG patterns). For example, an “add_clocks -free_running -period 10.00ns” command produces this Verilog directive:

```
`timescale 1ns / 10ps
```

- Use the SIM_PRECISION parameter file keyword. For example, “SIM_PRECISION 0.5ns;” produces this Verilog directive:

```
`timescale 1ns / 100ps
```

The precision in the Verilog testbench can originate from any of the previous sources, and the tool uses the smallest specified precision when writing out the Verilog testbench.

The resolution in the Verilog testbench originates from the procedure file. When you use multiple procedure files, the various “set time scale” statements can specify different values, and the tool uses the smallest specified resolution when writing the Verilog testbench.

set strobe_window time *window_width*;

Defines the strobe-window width. The “set strobe_window time” statement must be at the beginning of the procedure file, before any timeplate or procedure definition. If you do not specify the strobe_window time, it will default to the maximum allowable size. For example, if you look at a timeplate, and if there is a 10 ns window between the measure_po event and the next event (or end of timeplate), then that is what the strobe window will be. If there are multiple timeplates, then the smallest strobe window from the timeplates is the maximum allowable strobe window.

Some tester formats measure primary outputs (POs) at the exact time that you specify with the measure_po statement in the timeplate. However, other tester formats, such as STIL, require that output measurements occur during a specified window of time (*window_width*). For WGL, this statement changes the strobe window in the output file.

A strobe window can only stretch from the measure_po time to the end of the cycle or the next force or pulse event. For example, if you issue a measure_po at time 10 and the rising edge of a pulse at time 30, the strobe window can only be a maximum of 20. Strobe_window lets you know that, starting at the measure_po time, the primary output should be stable for the time specified by the strobe window.

Note

Strobe_window only affects the following formats: STIL, TSTL2, and WGL.

set default_timeplate *timeplate_name*;

Specifies a timeplate that can be used for any procedure definition that does not explicitly specify a timeplate. The referenced *timeplate_name* must be defined prior to the Set Default_timeplate statement in the procedure file.

set autoforce off;

An optional statement that controls the behavior for automatically adding force events. If included, this statement must appear at the beginning of the procedure file, prior to any procedures.

By default (without this statement), if a constrained pin is not forced to the constrained value in the test_setup procedure, a force event is automatically added to the first cycle of the test_setup procedure. If the test_setup procedure starts with a call to a subprocedure, then the force event is added to the first cycle following the subprocedure. If a force event already exists in the test_setup procedure for a constrained pin, then no additional force event will be added for that pin.

When you include the “set autoforce off” statement, if constrained pins are not forced to constrained values in the test_setup procedure, the tool adds one cycle to the end of the test_setup procedure and adds a force event to the new cycle for each constrained pin that is not forced to the constrained value in the test_setup procedure.

Including the “set autoforce off” statement also prevents the tool from forcing clock pins to the inactive state at the beginning of the load_unload procedure.

The “set autoforce off;” statement also turns off auto forcing Z values on bidis in the load_unload procedure—see Load_Unload (Required).

Alias Definition

The Alias definition groups multiple signal names or cell paths into a single alias name. Signal Alias statements are useful in procedures or timeplates where multiple signals need to be assigned to the same value at the same time.

Cell Alias statements are used to group cell paths into a single alias name. You must define aliases before using them. The definition can occur at any place in the procedure file outside of a timeplate or procedure definition.

Note

When saving STIL2005, CTL, or Structural_STIL patterns, all aliases defined in the procedure file will be defined as SignalGroups in the resulting STIL file.

There is a predefined alias available for specifying all bidirectional pins. The “_ALL_BIDI” keyword may be useful for forcing all bidirectional pins to a specified value without having to identify each individual pin. For example:

```
force _ALL_BIDI Z;
```

In using a cell Alias statement to group cell paths from condition statements into a single alias name, it is possible to override a condition statement in a named capture procedure with a subsequent condition statement that occurs in the same place (global condition, or local to a

specific cycle). A condition statement can only override a previous condition if the first condition is specified using an alias name, and if the second condition is specified without using an Alias statement.

Tip: When using multiple named capture procedures where each procedure requires many condition statements, it is helpful to group cells into a common name and apply the condition statement once to the entire group of cells, and then override specific cells that need a different value than what was applied to the group. This frees you from having to enter numerous condition statements for each named capture procedure, while only a handful of the cells require different values for each procedure.

The Alias definition has the following format:

```
alias alias_name = pin_name [, pin_name ...];
```

or

```
alias alias_name = cell_name [, cell_name ...];
```

- ***alias_name***

  A string that specifies the name of the alias.

- ***pin_name***

  A repeatable string that specifies the pin name to associate with the alias name.

- ***cell_name***

  A repeatable string that specifies the cell name to associate with the alias name.

Alias Examples

This example groups two signal names into a single alias name.

```
alias my_group = T, U;
```

This next example shows how a named capture procedure should look when using an Alias statement for condition cells. The example sets each of four cells to a value of 0, and then the fourth cell (`/inst_3/blockb/reg_2/Q`) is overridden with a value of 1.

```
alias cond_cells = "/inst_0/blocka/reg_1/Q", "/inst_1/blocka/reg_1/Q",
                   "/inst_2/blocka/reg_1/Q", "/inst_3/blockb/reg_2/Q";
timeplate tp1 =
   force_pi 0;
   measure_po 10;
   pulse ref_clk 50 50;
   period 100;
end;
procedure capture capture1 =
```

```
      timeplate tp1;
      condition cond_cells 0;
      condition /inst_3/blockb/reg_2/Q 1;
                              // overrides condition in previous statement
      cycle =
         force scan_en 0;
         force ctrl_a 1;
         force_pi;
         pulse ref_clk;
      end;
      cycle =
         force_pi;
         measure_po;
         pulse ref_clk;
      end;
   end;
```

This example shows how to define a user-defined bus in a procedure file:

```
alias wdata = "D[0]", "D[1]", "D[2]", "D[3]";
```

And this shows how to assign a value to that user-defined bus:

```
procedure sub_procedure subpro1 =
    scan_group grp1 ;
    timeplate shift_tp ;
    cycle =
         force wdata 0010 ;
        pulse ref_clk ;
    end;
end;
```

Timing Variables

Two timing variable block definitions allow a procedure file to express timing using variables and equations, and to have this equation-based timing preserved in the tool and reproduced in the correct syntax in pattern output files.

Test data languages such as WGL and STIL have the ability to express time values in the timing blocks as numerical values or as equations based on variables. Using equation-based timing allows one value to be specified for a global attribute, such as the test cycle period, while other values are derived from this using equations.

The two timing block definitions are called “timing_variables” and “variables”. In the “timing_variables” block, variables can be defined and assigned timing values. These values will be expressed in the time scale which is already specified by the Set Time Scale statement. The “timing_variables” block must be defined before the timeplate definitions.

The “variables” block is used to define variables that are not time values and have no units associated with them. These variables can only be assigned integer numbers, and can be used as scaling multipliers in the timing equations.

The variables in the “timing_variables” block can also be assigned timing equations instead of time values. These equations are simple mathematical equations which can use either timing values or previously defined *variables* or *timing variables* as operands.

Note

The event statements in the timeplate definition block accept timing values and timing variables.

When saving patterns in the Verilog, WGL, and STIL supported formats, the waveform tables in these formats will be written using the equations and variables, and the variables will be defined in the appropriate definition blocks which exist in each format. When saving patterns in formats that don’t support equation-based timing, the equations will be computed and the timing information will be specified as the resulting numeric values in the pattern file. Setting the ALL_FLATTEN_TIMING parameter file keyword to 1 will cause Verilog, WGL, and STIL outputs to compute the timing equations and use only the resulting numeric values in the output files. Any equation that does not compute to an integer value will be rounded to the nearest integer value.

The “timing_variables” block has the following syntax:

```
variables =
   variable_name = integer;
   [variable_name = integer; ...]
end;

timing_variables =
   variable_name = time_or_equation;
   [variable_name = time_or_equation; ...]
end;
```

Note that *time_or_equation* can either be an integer time value or a time equation. A time equation is expressed using operators and operands. An operator is one of +, -, *, or /. An operand can be a time value or a variable name (time or scaling variable). The multiplication and division operators (* and /) take precedence over the addition and subtraction operators (+ and -). You can use parenthesis to group operations for precedence.

Note

In the timeplate definitions, any place where a time value can be used, a timing variable is also allowed. A scaling variable from the “variables” block cannot be used in a timeplate definition. These can only be used in time equations.

Variable names can be any identifier except for reserved keywords used in the procedure file syntax (such as “period” and “force_pi”). The variable names must conform to the rules that apply to all identifiers used in the procedure file (alpha numeric string, starting with an alpha character, and no reserved punctuation marks). If reserved characters or reserved words are used in a variable name, the name must be enclosed in quotes.

Equation-based Timing Example

The following is a partial example of using equation-based timing.

```
set time scale 1.0 ns;
variables =
   v_scale = 1;
end;

timing_variables =
   t_period = 100;
   t_force = 0;
   t_meas = ((t_period * 0.1) * v_scale );
   t_rise = ((t_period / 2) * v_scale );
   t_width = ((t_period * 0.2) * v_scale);
end;

timeplate tp1 =
   force_pi t_force;
   measure_po t_meas;
   pulse ref_clk t_rise t_width;
   period t_period;
end;
```

This is how the timing example above would be represented in the STIL output:

```
Spec STUCK_spec {
   Category STUCK_cat {
      v_scale = '1';
      t_period = '100ns';
      t_force = '0ns';
      t_meas = '(t_period*0.1)*v_scale';
      t_rise = '(t_period/2)*v_scale';
      t_width = '(t_period*0.2)*v_scale;
   }
}

Timing STUCK_timing {
   WaveformTable tset_tp1 {
      Period 't_period';
      Waveforms {
         input_time_gen_0 { 01 { 't_force' D/U; }}
         input_time_gen_1 { 01 { '0ns' D; 't_rise' D/U;
                                    't_rise+t_width' D;}}
         _po_ { LHX { '0ns' X; 't_meas' l/h/x; 't_rise' X;}}
      }
   }
}
```

This is how the timing example would be represented in the WGL output:

```
equationsheet STUCK_sheet
   exprset STUCK_set
      v_scale := 1.0;
      t_period := 100nS;
      t_force := 0nS;
      t_meas := (t_period * 0.1) * v_scale;
```

```
      t_rise := (t_period / 2) * v_scale;
      t_width := (t_period * 0.2) * v_scale;
      _tp1_fall_1 := t_rise + t_width;
   end
end

timeplate tp1 period t_period
   "input_a" := input [t_force:S];
   ...
   "output_z" := output[0nS:X, t_meas:Q, t_rise:X];
   ...
   "refclk" := input[0nS:D, t_rise:S, _tp1_fall_1:D];
end
```

Timeplate Definition

The timeplate definition describes a single tester cycle and specifies where in that cycle all event edges are placed.

You must define all timeplates before they are referenced. A procedure file must have at least one timeplate definition. All clocks must be defined in the timeplate definition. The timeplate definition has the following format:

```
timeplate timeplate_name =
   timeplate_statement
   [timeplate_statement ...]
   period time;
   end;
```

The following list contains available timeplate_statement statements. The timeplate definition should contain at least the force_pi and measure_po statements. You are not required to include pulse statements for the clocks. But if you do not “pulse” any of the clocks, the tool uses two cycles to pulse a clock, resulting in larger patterns.

Note that the tool uses the “pulse_clock” statement rather than individual “pulse” statements when generating default procedures.

```
timeplate_statement:
   offstate pin_name off_state;
   force_pi time;
   bidi_force_pi time;
   measure_po time;
   bidi_measure_po time;
   force pin_name time;
   measure pin_name time;
   pulse pin_name time width [, time width];
   pulse_clock time width;
```

Note

In “timeplate_statement” definitions, you can use timing variables instead of time values. For more information, refer to “Timing Variables” on page 164.

The following is a list of statements in the timeplate definition:

- ***timeplate_name***

  A string that specifies the name of the timeplate.

- **offstate *pin_name off_state***

  A literal and double string that specifies the inactive, off-state value (0 or 1) for a specific named primary input pin that will be pulsed within this timeplate but is not defined as a clock pin by the add_clocks command. The complex timeplates are most useful in the shift procedure where a non-clock pin must be pulsed while still maintaining a single cycle in the shift procedure.

  This statement must occur before all other timeplate_statement statements. This statement is required for any pin that is not defined as a clock pin by the add_clocks command but will be pulsed within this timeplate.

Note

An “offstate” statement does not automatically force *pin_name* to its off state at time 0. For that to occur, you must force or pulse *pin_name* appropriately in a procedure.

- **force_pi *time***

  A literal and integer pair that specifies the force time for all primary inputs.

- **bidi_force_pi *time***

  A literal and integer pair that specifies the force time for all bidirectional pins. This statement allows the bi-directional pins to be forced after applying the tri-state control signal, so the system avoids bus contention. This statement overrides “force_pi” and “measure_po”.

- **measure_po *time***

  A literal and integer pair that specifies the time at which the tool measures (or strobes) the primary outputs.

- **bidi_measure_po *time***

  A literal and integer pair that specifies the time at which the tool measures (or strobes) the bidirectional pins. This statement overrides “force_pi” and “measure_po”.

- **force *pin_name time***

  A literal, string, and integer that specifies the force time for a specific named pin.

Note

This force time overrides the force time specified in force_pi for this specific pin.

- **measure *pin_name time***

  A literal, string, and integer that specifies the measure time for a specific named pin. You can use a “measure” statement in timeplates only to specify a measure time for a pin.

Note

This measure time overrides the measure time specified in measure_po for this specific pin.

- **pulse *pin_name time width*** [, *time width*]…

  A literal, string, and repeatable integer set that specifies the pulse timing for a specific named pin.

  pin_name — String that refers to a pin in the design. Valid pins must meet one of the following conditions:

  Defined as a clock pin using the add_clocks command.

  Not defined as a clock pin, but has a pulse signal and an offstate specified by the “offstate” statement.

  time — Integer that defines the offset from time 0 to the leading edge of the pulse.

  width — Integer that defines the width of the pulse.

  To define a multiple-pulse waveform, include multiple time and width pairs separated by a comma.

  All pulses must occur within the tester cycle period. You define this period using the “period” keyword.

Note

Multiple pulses are only supported for the following output formats: Verilog, WGL, STIL, STIL2005, CTL, FJTDL, MITDL, and TSTL2. Additionally, the TSTL2 output format does not support more than two pulses.

For MITDL format there is restriction that multiple pulse timing must be a cyclical repetition of the first pulse. Consequently, multi-pulse and double-pulse timing in the procedure file only works in the MITDL output without an error if the timing fits the restrictions of the MITDL syntax.

- **pulse_clock *time width***

  A literal and integer set that specifies the pulse timing for all signals defined as clocks, unless another statement such as a “force” or “pulse” exists for a particular clock signal. This is similar to the force_pi statement, which specifies the timing for ports that are not explicitly overridden by a force statement for those specific ports.

The pulse_clock statement is useful for ensuring that any added clocks (especially internal clocks) automatically have defined timing, which helps with automation.

The pulse_clock statement has the following options:

time — Integer that defines the offset from time 0 to the leading edge of the pulse.

width — Integer that defines the width of the pulse.

All pulses must occur within the tester cycle period. You define this period using the "period" keyword.

For example:

```
timing_variables =
   tester_period = 10;
   strobe_1 = (0.96 * tester_period);
   t_time = (0.25 * tester_period);
   t_width = (0.5 * tester_period);
end;

timeplate tessent_ijtag =
   force_pi 0 ;
   measure_po strobe_1;
   force tck 0;
   pulse_clock t_time t_width;
   period tester_period;
end;
```

- **period *time***

  A literal and integer pair that defines the period of a tester cycle. This statement ensures that the cycle contains sufficient time, after the last force event, for the circuit to stabilize. The time you specify should be greater than or equal to the final event time.

Example 1

```
timeplate tp1 =
   force_pi 0;
   pulse T 30 30;
   pulse R 30 30;
   measure_po 90;
   period 100;
end;
```

Example 2

The following example shows a shift procedure that pulses b_clk with an off-state value of 0. The timeplate tp_shift defines the off-state for pin b_clk. The b_clk pin is not declared as a clock in the ATPG tool.

```
timeplate tp_shift =
   offstate b_clk 0;
   force_pi 0;
   measure_po 10;
   pulse clk 50 30;
   pulse b_clk 140 50;
```

```
   period 200;
end;

procedure shift =
   timeplate tp_shift;
   cycle =
      force_sci;
      measure_sco;
      pulse clk;
      pulse b_clk;
   end;
end;
```

Example 3

In the following example, the pin b_clk is not declared as a clock in the ATPG tool. However, in the shift procedure, the user needs the pin to be pulsed twice with an offstate of 0.

```
timeplate tp_shift =
   offstate b_clk 0;
   force_pi 0;
   measure_po 10;
   pulse clk 50 30;
   pulse b_clk 40 50, 140 50;
   period 200;
end;

procedure shift =
   timeplate tp_shift;
   cycle =
      force_sci;
      measure_sco;
      pulse clk;
      pulse b_clk;
   end;
end;
```

Always Block

This optional block definition specifies events that happen in all cycles of all procedures. Because the always block specifies events for all cycles, it will be used with all timeplates and does not require a timeplate to be referenced in the block. Also, any signal that is pulsed in the always block must have a pulse waveform in all timeplate definitions.

If you defined any free-running clocks using the add_clocks command, an always block is automatically created in the procedure file, if one does not already exist, and a pulse statement added for each clock. Similarly, if you pulse a clock signal in the always block, the signal is automatically defined as a free-running clock. For more information, refer to the add_clocks description in the *Tessent Shell Reference Manual*.

Note

Free-running clocks are not automatically pulsed in a named capture procedure. The clocks must be pulsed explicitly.

All events specified in the always block will be subject to rules checks that apply to each procedure. In other words, the events in the always block will be added to each cycle of each procedure, and all DRC rules still apply to these events.

When saving patterns that preserve the structure of the procedures as macros (such as the CTL pattern file, or structural STIL pattern file), the events in the always block will be placed in the cycles of each procedure. The always block will not be present in the structural pattern file as a macro or procedure.

Always Block Syntax

The always block has the following syntax.

```
always =
  always_statement ;
  [always_statement ; ... ]
end ;
```

The *always_statement* is defined as one of the following.

```
pulse pin_name ;
force pin_name value ;
```

Always Block Example

The following is a partial example of an always block.

```
set time scale 1,0 ns ;

timeplate tp1 =
  force_pi 0 ;
  measure_po 10 ;
  pulse ref_clk 20 20, 60 20 ;
  pulse shift_clk 50 20 ;
  period 100 ;
end ;
always =
  pulse ref_clk ;
end ;

procedure shift =
  timeplate tp1 ;
  cycle =
    force_sci ;
    measure_sco ;
    pulse shift_clk ;
  end ;
end ;
```

Procedure Definition

The procedure definition is the heart of the procedure file. The procedure defines precisely how the scan circuitry operates.

All procedure definitions contain one or more *cycle* definitions. Each cycle definition in the procedure specifies a vector; each statement in the cycle specifies which events occur in that vector. The timeplate being used specifies any timing associated with that vector. The following is a list of rules for writing procedure definitions:

- If more than one timeplate is defined, you can assign a specific timeplate for each procedure definition or for each cycle within the procedure definitions. You must assign a timeplate at some point within a procedure definition.
- You must group all procedure statements, except scan_group, timeplate, and apply, into cycle statements.
- You cannot specify time values in cycle statements.
- The order of events within a cycle definition does not matter. The assigned timeplate specifies the order.
- Within a procedure definition, you can specify a scan group.
- Each scan group needs a unique test procedure file. You associate the test procedure file with the scan group when you specify the Add Scan Group command.
- Text following "//" is a comment and is ignored.
- You can include blank lines.
- You define a procedure type for a particular scan group (with the exception of the seq_transparent and clock procedures) only once in a test procedure file.
- You can only have a single test_setup procedure, even if you define multiple scan groups for your design.

The procedure definition has the following general format, but note that certain statements are restricted to certain procedures.

```
procedure procedure_type [proc_name] =       // page 174
   [scan_group scan_group_name;]             // page 174
   proc_statement [proc_statement ...]
end;

   proc_statement:
      [timeplate timeplate_name;]            // page 175
      cycle =
         cycle_statement [cycle_statement ...]
      end;
      apply proc_name #times;                // page 176

            cycle_statement:
               force_pi;
               bidi_force_pi;
               force_sci;
               force_sci_equiv;
               measure_po;
               bidi_measure_po;
```

```
measure_sco;
restore_pi;
restore_bidi;
bidi_force_off;
pulse_capture_clock;
pulse_read_clock;
pulse_write_clock;
force pin_name value;
expect pin_name value;
condition cell_name value;
measure pin_name;
initialize instance_name [value];
pulse pin_name;
timeplate timeplate_name;    // page 175
annotate "quoted string";    // page 175
```

- ***procedure_type***

  A string that specifies the type of procedure that follows. The following list contains valid procedures types:

  - test_setup
  - shift
  - load_unload
  - shadow_control
  - master_observe
  - shadow_observe
  - capture
  - clock_po
  - ram_sequential
  - ram_passthru
  - clock_sequential
  - init_force
  - seq_transparent
  - test_end
  - clock
  - sequential
  - skew_load
  - sub_procedure

  For more information, refer to "The Procedures" on page 187.

- *proc_name*

  An optional string that specifies the user-defined name of the procedure. Since you can specify multiple seq_transparent and clock procedures in a test procedure file, these procedure types require explicit procedure names, *proc_name*, for each procedure that you define.

- **scan_group *scan_group_name***

  A literal and string pair that specifies a scan group within a scan procedure. Since some of the scan procedures are scan group specific, you can specify scan groups within scan procedures. This makes it possible to define the scan procedures (shift, load_unload) for multiple scan groups within the same procedure file. You can then specify this file on the add_scan_groups command for each scan group in this file. If you use the read_procfile command to read a procedure file, you must include this statement. However, if you use the add_scan_groups command, this statement is optional since the group is specified on the command line. When the tool writes out a procedure file, it produces the scan_group statement.

Note

The scan_group_name argument is case-sensitive if the netlist used is case-sensitive.

- **timeplate *timeplate_name***

  A literal and string pair that specifies the name of the timeplate the procedure uses.

  A timeplate statement at the beginning of the procedure, outside of the cycle definitions, is the timeplate used by the entire procedure, if no other timeplates are referenced.

  A timeplate statement within a cycle is the timeplate used for that cycle and all other subsequent cycles until another timeplate statement is encountered. For more information about timeplates, refer to "Timeplate Definition" on page 167.

- **annotate "*quoted string*";**

  A literal and string pair that reports the Verilog testbench annotations during simulation.

  This keyword can be used to create the annotate statement that can occur in the beginning of a cycle along with the cycle timeplate statement, before any event statements as follows:

```
CYCLE =
  [ TIMEPLATE tp_name ; ]
  [ ANNOTATE "quoted string" ; ]
  event_statement;
  …
END ;
```

  The annotate statement is optional and must always use a quoted string. All procedures can be annotated, including sub procedures.

  The following is an example of an annotate statement used in a test_setup procedure, and how this will appear in a STIL pattern file.

```
Procedure test_setup =
   timeplate tp1;
   cycle =
      annotate "first cycle in test_setup" ;
      force reset 1;
      force clock 0;
   end;
   cycle =
      annotate "next annotation" ;
      force reset 0;
   end;
…
```

  This is a segment of the resulting STIL pattern file:

```
W tset_tp1;
V { _pi_ = 0X11XXX;
   _po_ = XXXX;
```

```
}
Ann {* Begin chain test *}
Ann {* first cycle in test_setup *}
V { …
}
Ann {* next annotation *}
V { …
}
```

- **label *“quoted_string”;***

  A literal and string pair for including pattern labels in saved patterns. As with the annotation statement, you can have one label statement per cycle in a procedure definition. The quoted_string becomes a pattern label for the vector that corresponds to that procedure cycle.

  Only the STIL and WGL pattern formats support a pattern label statement. For pattern file formats that don't support a pattern label, the label is present as an annotation statement that has the string “label:” added at the beginning of the label string.

  For the simulation testbench, the label is also present as an annotation that has the string “label:” added at the beginning, and the annotation is echoed when the patterns are simulated. You can use the existing parameter file keyword SIM_ANNOTATE_QUIET to turn off echoing the annotations and labels while simulating.

  Each pattern label is a unique identifier, with its vector count appended to the end of the label string.

  This statement can be used at the start of any cycle, just like an annotation statement. A cycle cannot contain both a label and an annotation statement.

  The following example shows how to use the label statement within the test_setup procedure:

```
   procedure test_setup =
      timeplate my_tp;
      cycle =
         force my_sig 0;
      end;
      cycle =
         label "end of test_setup" ;
         force my_sig 1;
      end;
   end;
The previous example produces the following STIL vectors:
V { _pi_ = …;
}
"end of test_setup_1": V { _pi_ = …;
}
```

- **apply *proc_name #times***

  A literal and double-argument string that tells the tool to apply the specified procedure the specified number of times. You must use the apply shift statement at least once in the

load_unload procedure. For the apply shift statement, you should enter a proper *#times* parameter, otherwise you will get a warning message.

If required, you must enter the apply shadow_control statement immediately after the apply shift procedure statement, and you must set the *#times* argument to 1. The apply statement is only valid outside of the cycle blocks because it specifies another group of cycles within another procedure to be added at that point.

- **cycle_statement**

  The following list describes valid cycle_statement keywords. Cycle_statements cannot contain time values.

  - force_pi

    A literal that specifies for the tool to force all primary inputs.

  - bidi_force_pi

    A literal that specifies for the tool to force all bidirectional pins.

  - force_sci

    A literal that specifies for the tool, in the shift procedure, to place values on the scan chain inputs, thus implementing scan cell controllability.

  - force_sci_equiv

    A literal that acts the same as the force_sci statement, except that it also forces all pins equivalent to the scan input pins. Using this statement places the complement value on the associated differential pin of a scan input during scan loading. This statement is necessary because the test procedures do not consider pin equivalence relationships (those specified with Add Pin Equivalence).

  - measure_po

    A literal that specifies for the tool to measure or strobe the primary outputs.

  - bidi_measure_po

    A literal that specifies for the tool to measure or strobe the bidirectional pins.

  - measure_sco

    A literal that specifies for the tool, in the shift procedure, to measure scan output values, thus implementing scan cell observability. In End Measure Mode (refer to “Creating Test Procedure Files for End Measure Mode” on page 226), measure_sco is also used in the load_unload procedure.

  - restore_pi

A literal that returns primary inputs to their original states (prior to this procedure's execution). You use the restore_pi statement at the end of a seq_transparent procedure.

- restore_bidi

  A literal that returns bidirectional pins to their original states (prior to this procedure's execution). You use the restore_bidi statement at the end of a "clock" procedure.

- bidi_force_off

  A literal that specifies for the tool to force all unconstrained bidirectional pins off.

- pulse_capture_clock

  A literal that specifies for the tool to pulse the capture clock.

- pulse_read_clock

  A literal that specifies for the tool to pulse the RAM read clock.

- pulse_write_clock

  A literal that specifies for the tool to pulse the RAM write clock.

- force pin_name value

  A literal and double string that forces the specified value of 0, 1, X, or Z on the specified pin. The pin names you specify must be valid pin pathnames for primary inputs.

- expect name value

  A literal and double string that causes the tool to expect the specified value of 0, 1, X, or Z on the specified internal pin or port. You can use "expect" statements only in the test_setup and test_end procedure. Internal pins are only checked by DRC and are not compared in the testbench. For ports, the tool validates the values in both the testbench and in tester pattern formats.

- condition cell_name value

  A literal and double string that you use at the beginning of a seq_transparent procedure to identify the necessary scan cell states (conditions) to establish transparency in non-scan cells. You identify the scan cell by the pin pathname associated with the output of its state element. The path from the defined pin to the scan cell must only contain buffers and inverters. The value argument sets the value at the specified pin pathname, which may be inverted relative to the associated scan cell value.

- measure pin_name

A literal and string pair that specifies for the tool to measure the value of the named pin. You can use a “measure” statement in the capture procedure only to specify a measure on a pin in a different cycle than the measure_po event.

- initialize instance_name value

  A literal and string pair that initializes the named memory element to the value given. This statement is particularly useful for initializing the finite state machine in the TAP controller of boundary scan circuitry, when the TAP does not contain the TRST signal. Once set to a binary state, the TCK and TMS pins can place the finite state machine in a desired state. If not set, these pins remain at X.

  If you do not specify a value, the tool chooses a random value to assign to all latches and flip-flops with the specified instance name.

- pulse pin_name

  A literal and string pair that specifies for the tool to pulse the named clock pin.

- observe_method value

  A literal and string pair set to a value of master, slave, or shadow, to specify for a specific observe method to be defined for each named capture procedure.

The following example shows how to use a cycle_statement to force scan inputs and measure scan outputs:

```
procedure shift =
   scan_group grp1;
   timeplate tp1;
   cycle =
      force_sci;
      measure_sco;
      pulse T;
   end;
end;
```

Clock Control Definition

This section provides information for creating clock control definitions manually in the test procedure file.

For complete information about when you use this definition, refer to “Support for Internal Clock Control” in the *Tessent Scan and ATPG User’s Manual*.

ATPG Restrictions

The following restrictions apply to ATPG when clock control definitions are enabled:

- Clock_PO patterns are disabled.

- In undefined cycles, the internal clock is assumed to be off, even if the source clock pulses.
- Source clocks are pulsed regardless of clock restrictions. Any false paths should be explicitly defined with DC or the add_false_paths command.
- External clocks without clock control definitions are controlled through top-level pins.
- Clock control definitions applied to a clock defined as equivalent also applies to all associated equivalent clocks.
- Timeplate definitions apply only to external clocks.
- If you use the set_clock_restriction -same_clocks_between_loads command, you must use one of the following definitions to pulse the controlled clock:
  - {ATPG_SEQUENCE, END}
  - {ATPG_SEQUENCE <N> <M>, END} with N starting from 0. The generated test pattern includes M+1 capture cycles between the scan loading operation and the scan unloading operation.
  - <ATPG_CYCLE <i>, END} with i=0 defined

  If none of these statements are present a warning displays during ATPG about clock controls that cannot be applied due to clock restrictions.

Rules for Clock Control Definitions

The following rules apply to the clock control definitions in the test procedure file:

- Global control conditions and source clocks defined for equivalent clocks must be the same.
- When a clock is forced off, it cannot be used as a source in the same definition.
- A FORCE statement cannot force a clock pin to an on state.
- Clock control can not be applied to a free running clock.
- Condition statements cannot be applied to non-scan state elements.
- You must specify a condition to turn on the internal clock; otherwise, it is assumed to be off.
- When multiple sequence clock control definitions are defined for the same clock, they must use mutually exclusive pulse conditions as follows:
  - The clock control condition to pulse a clock in sequence mode must be mutually exclusive with the clock control condition for the same clock in per-cycle mode.
  - The condition to pulse a clock in sequence mode must be mutually exclusive with the condition to pulse the same clock by using another sequence mode.

Keywords

The following is a list of keywords used in clock control statement:

- **ATPG_CYCLE *cycle_number***

  A literal and integer that specifies a test pattern capture cycle to map the clock control to. The specified capture cycle values start from 0, which corresponds to the first capture cycle after scan loading.

  Multiple ATPG_CYCLE definitions can be declared to pulse the internal clock at the same capture cycle with different sets of conditions.

  Use a FORCE statement to turn the clock off, or the clock continues to pulse when the conditions are satisfied.

  The specified and actual capture cycles may to differ—see "Capture Cycle Determination" in the *Tessent Scan and ATPG User's Manual*.

- **ATPG_SEQUENCE *N M***

  A literal and an integer with a value between ***N*** and ***M*** that specifies a set of capture cycles for clock pulsing.

  Define the condition to pulse the clock continuously from capture cycle N (N>=0) to capture cycle M (M>=N) right after scan loading. If N is greater than 0, the clock is automatically set to off state from the first capture cycle right after scan loading to the capture cycle N -1. When the generated test pattern includes more than M capture cycles after scan loading, the clock is set to off state from the M +1 capture cycle to the last capture cycle.

  Multiple ATPG_SEQUENCE definitions can be declared as necessary.

  Use a FORCE statement to turn the clock off, or the clock continues to pulse when the conditions are satisfied.

  The specified and actual capture cycles may to differ—see "Capture Cycle Determination" in the *Tessent Scan and ATPG User's Manual*.

- **CLOCK_CONTROL *pin_pathname***

  A literal and string value that specifies the pin pathname of the PI for the internal clock. The specified pin must be an existing clock. Internal clocks must also be defined with the add_clocks -internal command.

- **CONDITION *cell_name value***

  An optional literal and double string that specifies necessary scan cell states (conditions). The scan cell is specified by the pin pathname associated with the output of its state element. The value argument specifies the value loaded into the scan cell at the end of shift.

The specified and actual capture cycles may to differ—see "Capture Cycle Determination" in the *Tessent Scan and ATPG User's Manual*.

- **FORCE *pin_pathname value***

  A literal and double string that forces a value of 0, 1, or Z on a specified pin. The specified pin names must be valid pin pathnames for primary inputs. This keyword is used to force necessary pins off during capture cycles when the controlled clock is pulsed.

- **SOURCE_CLOCK *pin_pathname*...**

  A literal and repeatable string that specifies one or more source clocks to drive the internal clock logic to pulse in the specified capture cycle(s). If no source clock is specified, the source clock is assumed to be a free running clock that pulses in every capture cycle.

- **END**

  Required literal the specifies the end of an ATPG_CYCLE or ATPG_SEQUENCE block, or at the end of the clock control definition.

Global Condition Statements

Global conditions are CONDITION statements that are accessible in every scope of a clock control definition. You can use global conditions to define default conditions within a clock control definition.

For example, you can specify a CONDITION statement for all ATPG_CYCLE/ATPG_SEQUENCE blocks within a definition. Then you can define a CONDITION statement within individual ATPG_CYCLE/ATPG_SEQUENCE blocks to override the global CONDITION variable when necessary.

The following example uses local conditions (italicized) to define some of the control bits necessary for each scan cell to pulse the clock. The global conditions define other conditions that must be satisfied for all clock cycles.

```
CLOCK_CONTROL /clk_ctrl/int_clk1 =
   SOURCE_CLOCK ref_clk;
   CONDITION /clk_ctrl/enable_1/q 0;
   CONDITION /clk_ctrl/enable_2/q 0;
   ATPG_CYCLE 0 =
      CONDITION /clk_ctrl/F0/q 1;
   END;
   ATPG_CYCLE 1 =
      CONDITION /clk_ctrl/F1/q 1;
   END;
   ATPG_CYCLE 2 =
      CONDITION /clk_ctrl/F2/q 1;
   END;
   ATPG_CYCLE 3 =
      CONDITION /clk_ctrl/F3/q 1;
   END
```

```
END;
```

The previous example is equivalent to the following:

```
CLOCK_CONTROL /clk_ctrl/int_clk1 =
   SOURCE_CLOCK ref_clk;
   ATPG_CYCLE 0 =
      CONDITION /clk_ctrl/F0/q 1;
      CONDITION /clk_ctrl/enable_1/q 0;
      CONDITION /clk_ctrl/enable_2/q 0;
   END;
   ATPG_CYCLE 1 =
      CONDITION /clk_ctrl/F1/q 1;
      CONDITION /clk_ctrl/enable_1/q 0;
      CONDITION /clk_ctrl/enable_2/q 0;
   END;
   ATPG_CYCLE 2 =
      CONDITION /clk_ctrl/F2/q 1;
      CONDITION /clk_ctrl/enable_1/q 0;
      CONDITION /clk_ctrl/enable_2/q 0;
   END;
   ATPG_CYCLE 3 =
      CONDITION /clk_ctrl/F3/q 1;
      CONDITION /clk_ctrl/enable_1/q 0;
      CONDITION /clk_ctrl/enable_2/q 0;
   END
END;
```

The previous example demonstrates the importance of ensuring that global conditions do not conflict with local conditions. To further illustrate this point, consider the following incorrect definition of global conditions:

```
// Example of incorrect definition of global conditions
CLOCK_CONTROL /clk_ctrl/int_clk1 =
SOURCE_CLOCK ref_clk;
   CONDITION /clk_ctrl/F0/q 0;
   ATPG_CYCLE 0 =
      CONDITION /clk_ctrl/F0/q 1;
   END;
   ATPG_CYCLE 1 =
      CONDITION /clk_ctrl/F1/q 1;
   END;
   ATPG_CYCLE 2 =
      CONDITION /clk_ctrl/F2/q 1;
   END;
   ATPG_CYCLE 3 =
      CONDITION /clk_ctrl/F3/q 1;
   END
END;
```

On the surface, it may seem correct that the condition in ATPG_CYCLE 0 will override the global condition while the other cycles can still be satisfied. However, after the global condition is expanded to all cycles, the clock control definition looks like this:

```
// Example of incorrect definition of global conditions
```

```
CLOCK_CONTROL /clk_ctrl/int_clk1 =
SOURCE_CLOCK ref_clk;
   ATPG_CYCLE 0 =
      CONDITION /clk_ctrl/F0/q 1;
   END;
   ATPG_CYCLE 1 =
      CONDITION /clk_ctrl/F1/q 1;
      CONDITION /clk_ctrl/F0/q 0;
   END;
   ATPG_CYCLE 2 =
      CONDITION /clk_ctrl/F2/q 1;
      CONDITION /clk_ctrl/F0/q 0;
   END;
   ATPG_CYCLE 3 =
      CONDITION /clk_ctrl/F3/q 1;
      CONDITION /clk_ctrl/F0/q 0;
   END
END;
```

You can now see that it would not be possible to pulse the clock in ATPG_CYCLE 0 while also pulsing it in any other cycle. The tool can load only one value into /clk_ctrl/F0, so it can either pulse the clock in cycle 0 by loading a 1 or pulse it in another cycle by loading a 0.

Per-Cycle Clock Control Definition Example

The following example defines per-cycle clock control for two internal clocks (/top/core1/clk1 and /top/core1/clk2):

```
CLOCK_CONTROL /top/core1/clk1 =
   ATPG_CYCLE 0 =
      CONDITION /pll_ctl/cell_0/Q 1;
   END;
   ATPG_CYCLE 1 =
      CONDITION /pll_ctl/cell_1/Q 1;
      CONDITION /pll_ctl/cell_4/Q 0;
//both conditions must be satisfied for clock to pulse in
//capture cycle 1
   END;
END;
CLOCK_CONTROL /top/core1/clk2 =
   ATPG_CYCLE 0 =
      CONDITION /pll_ctl/cell_2/Q 1;
   END;
   ATPG_CYCLE 1 =
      CONDITION /pll_ctl/cell_3/Q 1;
      CONDITION /pll_ctl/cell_5/Q 1;
   END;
END;
```

Sequence Clock Control Definition Example

The following example defines sequence clock control for two internal clocks (/top/core/clk1 and /top/core1/clk2) derived from the source clock clk_src:

```
CLOCK_CONTROL /top/core1/clk1 =
   SOURCE_CLOCK clk_src;
   ATPG_SEQUENCE 0 1 =
```

```
// Pulses 2 consectutive cycles if the scan cell
// is loaded with 1, and the source clock is pulsed.
      CONDITION /pll_ctl/cell_1/Q 1;
   END;
END;
CLOCK_CONTROL /top/core1/clk2 =
   SOURCE_CLOCK clk_src;
   ATPG_SEQUENCE 0 1 =
      CONDITION /pll_ctl/cell_2/Q 1;
   END;
END;
```

The following example pulses clock /top/core1/clk1 unconditionally in every capture cycle between scan loading:

```
CLOCK_CONTROL /top/core1/clk1 =
   ATPG_SEQUENCE =
// empty body
   END;
END;
```

The following example defines a multi-sequence clock control definition:

```
CLOCK_CONTROL /top/core/clk1_int =
   SOURCE_CLOCK /clk1;
   ATPG_SEQUENCE 0 2 =
      CONDITION /pll/ctl_1/Q 1;
      FORCE ENABLE_1 1;
   END;
   ATPG_SEQUENCE 3 4 =
      CONDITION /pll/ctl_1/Q 0;
      FORCE ENABLE_1 1;
   END;
END;
```

Exclusive conditions ensure that only one sequence block is applied per capture cycle (otherwise, no sequence is applied). If no cycle numbers are specified for sequence clock control, the clock pulses in every capture cycle when conditions are loaded.

Multiple Sets of Conditions for the Same Cycle Example

The following example shows that if a clock can be pulsed in a particular cycle or sequence of cycles when there are multiple sets of conditions where any one set can activate the clock for that cycle, the same cycle can be defined multiple times:

```
CLOCK_CONTROL /top/core1/clk1 =
  ATPG_CYCLE 0 =
    CONDITION /pll_ctl/cell_1/Q 1;
  END;
  ATPG_CYCLE 0 =
    CONDITION /pll_ctl/cell_2/Q 1;
  END;
END;
```

The previous example shows that /top/core1/clk1 can be pulsed in ATPG_CYCLE 0 when any set of the specified conditions are met. This demonstrates the case where loading a '1' into either /pll_ctl/cell_1 or /pll_ctl_cell_2 pulses the clock in cycle 0.

Similarly, the following example defines multiple sets of conditions for the same sequence of cycles, which can overlap. The sequence of cycles must have mutually exclusive conditions to ensure conditions for each ATPG_SEQUENCE can be satisfied without conflicting with other sequences.

```
CLOCK_CONTROL /top/core1/clk1 =
  ATPG_SEQUENCE 0 2 =
    CONDITION /pll_ctl/cell_1/Q 1;
    CONDITION /pll_ctl/cell_2/Q 0;
    CONDITION /pll_ctl/cell_3/Q 0;
  END;
  ATPG_SEQUENCE 0 3 =
    CONDITION /pll_ctl/cell_1/Q 0;
    CONDITION /pll_ctl/cell_2/Q 1;
    CONDITION /pll_ctl/cell_3/Q 0;
  END;
  ATPG_SEQUENCE 1 4 =
    CONDITION /pll_ctl/cell_1/Q 0;
    CONDITION /pll_ctl/cell_2/Q 0;
    CONDITION /pll_ctl/cell_3/Q 1;
  END;
END;
```

Source Clocks with Different Frequencies Example

The following example defines source clocks that have different frequencies when using clock control definitions:

```
timeplate _default_WFT_ =
    force_pi 0 ;
    measure_po 40 ;
    pulse clk1 45 10;
    pulse ref_clock 15 5, 40 5, 65 5, 90 5;
    pulse clocks_02/my_controller/U2/Z 45 10;
    pulse clocks_03/my_controller/U2/Z 45 10;
    pulse clocks_04/my_controller/U2/Z 45 10;
    period 100 ;
 end;

procedure capture =
    timeplate _default_WFT_;
    cycle =
        force_pi ;
        measure_po ;
        pulse_capture_clock ;
    end;
 end;
```

In this example, for one pulse of clk1, there are 4 pulses of ref_clock, specifically the ref_clock frequency is 4 times the frequency of clk1.

The Procedures

The following sections describe the test procedures that comprise a test procedure file. The procedures can be categorized based on their type as shown in the following table.

Table 6-2. Procedure Categories

Procedure Type	Procedure Name
Scan and Clock Procedures	• "Test_Setup (Optional)" on page 189 • "Shift (Required)" on page 192 • "Alternate Shift Procedure (Optional)" on page 194 • "Load_Unload (Required)" on page 196 • "Shadow_Control (Optional)" on page 199 • "Master_Observe (Sometimes Required)" on page 200 • "Shadow_Observe (Optional)" on page 201 • "Seq_Transparent (Optional)" on page 202 • "Clock (Optional)" on page 204 • "Skew_Load (Optional)" on page 205 • "Clock_run (Optional)" on page 206
Non-Scan Procedures	• "Capture Procedures (Optional)" on page 208 • "Clock_po (Optional)" on page 216 • "Ram_sequential (Optional)" on page 217 • "Ram_passthru (Optional)" on page 218 • "Clock_sequential (Optional)" on page 219 • "Init_force (Optional)" on page 220 • "Test_end (Optional, all ATPG tools)" on page 221 • "Sub_procedure" on page 223

Scan and Clock Procedures

The scan and clock-related procedures inform the tool how to operate the scan chain and pulse clocks.

Non-Scan Procedures

The non-scan procedures can represent any type of pattern that the tool produces. You can use the non-scan procedure to specify in which cycles of the procedure "potential events" happen. A potential event is an event that the ATPG engine may or may not have created to cover a certain fault.

To avoid DRC violations, each non-scan procedure must contain the proper statements in the correct order with the timing from the timeplate. The statements in a non-scan procedure can be spread over any number of cycles using a different timeplate for each cycle if needed.

A basic pattern consists of loading the scan chains, a default capture procedure, followed by unloading the scan chains; however, you do not specify the loading and unloading of scan chains in non-scan procedures. The following shows the basic pattern for non-scan procedures.

Basic Pattern

Force primary inputs
Measure primary outputs
Pulse capture clock

Example Timeplates

All example procedures shown in this section use one of the following two timeplates, unless otherwise stated:

```
timeplate tp1 =
   force_pi 0;
   measure_po 10;
   pulse scan_clk 30 10;
   pulse sys_clk 30 10;
   period 50;
end;

timeplate tp2 =
   force_pi 0;
   measure_po 10;
   pulse scan_mclk 15 10;
   pulse scan_sclk 30 10;
   period 50;
end;
```

Test_Setup (Optional)

This optional procedure, which can only contain force, pulse, init, and expect event statements, sets non-scan elements to the desired states for the load_unload procedure. You may use this procedure only once for all scan groups, and it appears only once at the beginning of the test pattern set.

This procedure is particularly useful for initializing boundary scan circuitry. For an example using this procedure to set up boundary scan circuitry, refer to "Generating Patterns for a Boundary Scan Circuit" in the *Tessent Scan and ATPG User's Manual*.

Bidirectional Scan Out Pins

The value of all bidirectional scan out pins must be forced to the Z state (indicating it is operating in "output" mode) to properly sensitize the scan chain. When reading in a test procedure file, the tool automatically adds force events to the beginning of the load_unload procedure to force all bidi pins to Z.

Bidi pins that are clocks or constrained pins are not forced to a Z, as they were already forced to the off-state or the constrained values. Bidi scan-in pins are also not forced to a Z. Any bidi pin already forced later in the load_unload procedure will not be forced to a Z. Any bidi forced to a specific value in the test_setup procedure will instead be forced to this value instead of a Z.

Like previous automatic force values, these can be disabled by putting the "set autoforce off;" statement at the beginning of the procedure file.

Pin Constraints

If you use the add_input_constraints command to set pin constraints, be aware this command only forces pins during capture. To constrain these pins during test_setup, you should include the same pin constraints in the test_setup procedure. This will ensure the pins are in the same state for loading the first pattern as for loading all subsequent patterns.

If you do not properly constrain the pins prior to the end of the test_setup procedure, the tool automatically constrains them by inserting a cycle statement in the test_setup procedure. However, this automatic handling may not insert the events with the timing you want. Also, the automatic handling is not included in DRC.

If you have defined input constraints but have not provided a test_setup procedure, the tool will automatically generate a test_setup procedure to force those pins to their constrained values.

You can use both the write_procfile and the report_procedures commands to see the contents of the test_setup procedure the tool has generated. The write_procfile command writes existing procedure and timing data to a specified file. The report_procedures command writes the information to the screen.

MBISTArchitect

For MBISTArchitect, a test_setup procedure can force, expect, and/or pulse any necessary signals during a test, which enables you to apply initialization cycles prior to any read or write

operation, or configure a bidirectional port as an input/output port by controlling the bidirectional port enable signal(s).

Example 1

The following is an example using a sub_procedure. In this example, the signal named C will retain its value of 1 during the test unless it is forced to a different value in a later cycle, by another procedure, or it is overwritten by WGL patterns.

```
procedure sub_procedure initialize =
     template soc_timeplate ;
     cycle =
          force C 1;
     end;
end;
```

The following example shows how to apply the previous sub_procedure. For more information, see “Sub_procedure” on page 223.

```
procedure test_setup =
     timeplate soc_timeplate;
     cycle =
          force test_en 1; // force test_en 1
          force chip_en 0; // force chip_en to 0
     end;
     apply initialize 10 ; // force C to 1 for 10 cycles
end;
```

Example 2

The following example shows a way to apply initialization cycles to a memory. The RST signal is active for the first 128 cycles, then it is deactivated in the next cycle (cycle 129).

```
procedure sub_procedure reset_mem =
     timeplate soc_timeplate ;
     cycle =
          force RST 1;
     end;
end;
procedure test_setup =
     timeplate soc_timeplate;
     apply reset_mem 128;
     cycle =
          force RST 0; // deactivate RST
     end;
end;
```

Example 3

The following example shows a way to use an expect statement in a test_setup procedure. The output signal (DFT) is expected to 1 in the first cycle and X in the remaining cycle. Please note that “expect” statements do not work the same as a force or pulse statement. When none is present, it is assumed to mean do not measure.

```
procedure test_setup =
     timeplate soc_timeplate;
```

```
     cycle =
          expect DFT 1 ;
     end;
end;
```

Example 4

This example shows a way to start pulsing a clock in a test_setup procedure. The SYSCLK starts pulsing at cycle number 2 until the end of test.

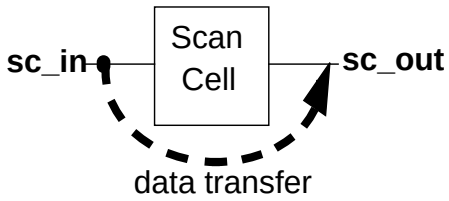
```
timeplate soc_timeplate =
     force pi;
     measure_po 90;
     pulse SYSCLK 50 50;
     period 100;
end;

procedure test_setup =
     timeplate soc_timeplate;
     cycle =
          force RST_L 0;
     end;
     cycle =
          pulse SYSCLK;
     end;
end;
```

Shift (Required)

This required procedure describes how to shift data one position down the scan chain by forcing the scan input, toggling the clock(s), and strobing the scan output. The following figure shows the data flow process for the shift procedure.

Figure 6-1. Shift Procedure



Within this procedure, you must use the force_sci, or force_sci_equiv, and the measure_sco event statements. You can also use the force and pulse event statements. A shift procedure can contain more than one cycle, although not all pattern formats can support multiple cycles and parallel load. Pattern formats that do not support multiple cycles are any parallel format other than STIL and Verilog. If you use write_patterns to write out one of these other parallel formats with a multi-cycle shift procedure, the command generates an AG11 error.

The times at which the timeplate used by the shift procedure applies the force_sci and measure_sco commands must allow proper operation of the load_unload procedure. The measure_sco will occur at the measure_po time specified in the timeplate. The force_sci will occur at the force_pi time specified in the timeplate.

The following are examples of the shift procedure for both mux-DFF and LSSD architectures.

Mux-DFF Example

```
procedure shift =
   timeplate tp1;
   cycle =
      // force scan chain input
      force_sci;
      // measure scan chain output
      measure_sco;
      // pulse the scan clock
      pulse scan_clk;
   end;
end;
```

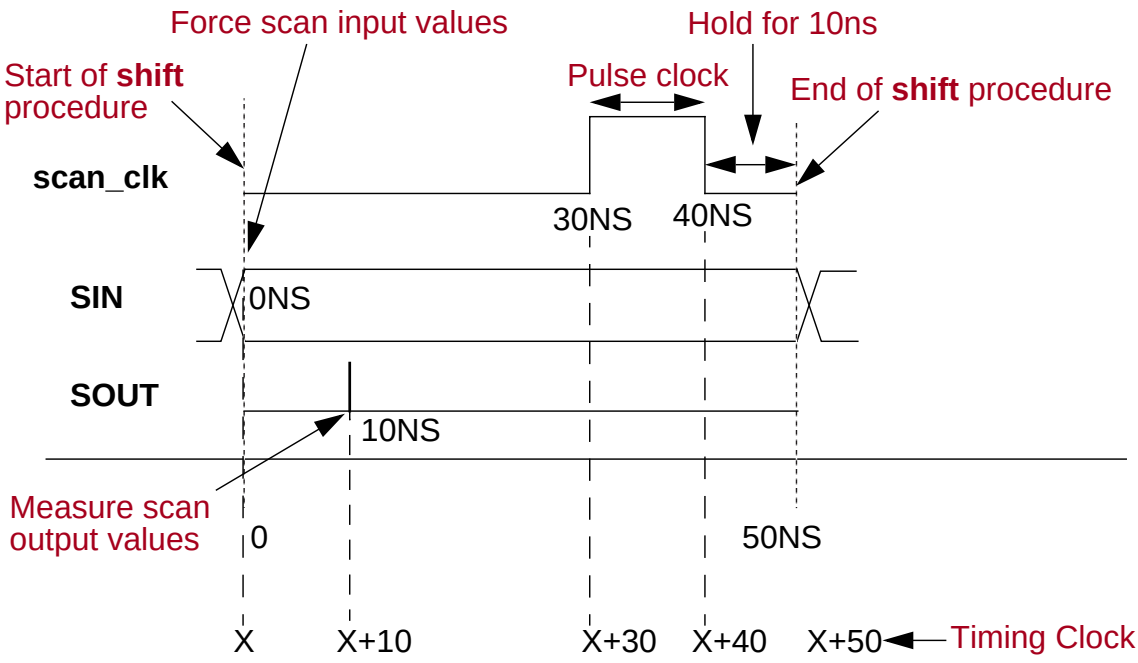
LSSD Example

```
procedure shift =
   timeplate tp2;
   cycle =
      // force scan chain input
      force_sci;
      // measure scan chain output
      measure_sco;
```

```
        // pulse master clock
        pulse scan_mclk;
        // pulse slave clock
        pulse scan_sclk;
    end;
end;
```

Figure 6-2 graphically displays the waveforms for the clock pin, the scan-in pin, and the scan-out pin derived from the Mux-DFF shift procedure example. This timing diagram shows one scan chain shift cycle, assuming the time unit is 1ns.

Figure 6-2. Timing Diagram for Shift Procedure



The procedure contains four scan events: forces scan input values at 0ns, strobes (or measures) scan output values at 10ns, pulses the scan clock scan_clk (turning it on at 30ns and off at 40ns), and holds the state of the last event until the procedure finishes at 50ns.

A timing clock monitors when each significant event occurs. If the timing clock is at X when the shift procedure begins, the timing clock assigns those four events with time values X, X+10, X+30, and X+40. When the shift procedure finishes, the timing clock advances to X+50. The shift cycle ending time becomes the starting time for the next shift cycle.

Alternate Shift Procedure (Optional)

When using on-chip clock generators, such as programmable PLLs, it is sometimes necessary to change values on input (control) signals to the clock generator a cycle or two before the change in generated clocking schemes is realized. When the shift clocks for a scan chain are also provided by the on-chip clock generator, it is sometimes not possible to reprogram the clock generator near the end of the scan chain shifting in order to stop the shift clock and prepare for the capture clocks. To accomplish this you might want to use an alternative shift procedure.

Alternate shift procedures have names, as described in the following paragraph. Alternate shift procedures can only be used for single shifts (a pre shift or a post shift), and there must be one un-named normal shift (*shift*) as the main shift in the required load_unload procedure. See Load_Unload (Required).

The shift procedure allows for an optional name following the shift procedure type. For each scan group, one shift procedure must be defined that has the default name of shift. For each scan group, additional alternate shift procedures can be defined as long as each has a unique name.

Each shift procedure is required to contain a force_sci or force_sci_equiv statement and a measure_sco statement.

Syntax

```
procedure shift [ procedure_name ] =
...
end ;
```

Example

The following is a partial example of how the alternate shift procedure might be used in a procedure file for a scan chain with a length of 100.

```
timeplate tp1 =
  force_pi 0;
  measure_po 10;
  pulse ref_clk 50 50;
  period 100;
end;
procedure shift =
  timeplate tp1;
  scan_group grp1;
  cycle =
    force ctrl_a 1;
    force_sci;
    measure_sco;
    pulse ref_clk;
  end;
end;
```

```
procedure shift shift_last =
  timeplate tp1;
  scan_group grp1;
  cycle =
    force ctrl_a 0;
    force_sci;
    measure_sco;
    pulse ref_clk;
  end;
end;
procedure load_unload =
  timeplate tp1;
  scan_group grp1;
  cycle =
    force ref_clk 0;
    force scan_en 1;
    force ctrl_a 1;
  end;
  apply shift 98;
  apply shift_last 1;
  apply shift_last 1;
end;
```

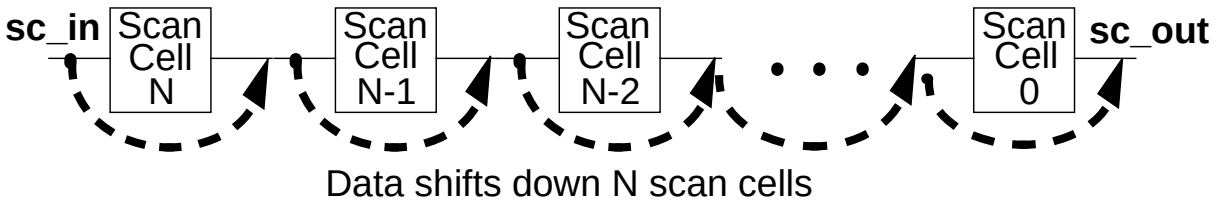
Load_Unload (Required)

This required procedure describes how to load and unload the scan chains in the scan group. To load the scan chain, you must force the circuit into the appropriate state for the start of the shift sequence. This includes forcing clocks, resets, RAM write control signals, and any other signals that need to be at their off states for scan chain loading. Also, if a reset signal is defined as a clock, and pin constrained to its off state in the dofile, it needs to again be forced to its off state in the load_unload and named capture procedures in order to avoid a P34 DRC.

Offstate for clock pins, constrained pin values, and other pins that have values forced in the test_setup procedure are automatically added as force statements to the beginning of the load_unload procedure (if not present); this helps reduce DRC failures.

Figure 6-3 shows the data flow for the load_unload procedure.

Figure 6-3. Load_Unload Procedure



If the scan out pin is bidirectional, you must force its value to the Z state (indicating it is operating in “output” mode) to properly sensitize the scan chain. If there is a scan enable signal, you must force it on to enable the scan chain prior to the shift. You then use the apply shift statement to specify the number of shift cycles (which equals the number of scan elements in the chain). If you have optionally included the shadow_control procedure (which if used, immediately follows the shift procedure), you must also include the apply command.

The following list includes the basic statements in the load_unload procedure:

Mux-DFF Example

```
procedure load_unload =
   timeplate tp1;
   cycle =
      // force clocks off
      force RST 0;
      force CLK 0;
      // activate scanning mode
      force scan_en 1;
   end;
   // shift data thru each of 7 cells
   apply shift 7;
end;
```

LSSD Example

```
procedure load_unload =
   timeplate tp2;
   cycle =
      // force all clocks off
      force RST        0;
      force CLK        0;
      force scan_sclk  0;
      force scan_mclk  0;
   end;
   // apply shift procedure 7 times
   apply shift 7;
end;
```

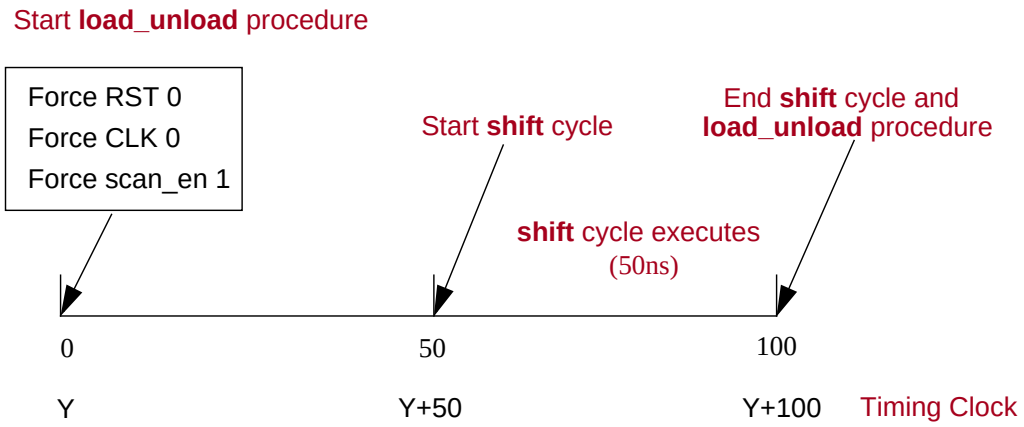
The timing for the load_unload procedure is generally straightforward. The load_unload procedure contains the apply statement. Therefore, the total time for a load_unload procedure includes the time specified by the timeplate being used plus the time required to execute the apply cycles.

For example, examine the following load_unload procedure, using the example shift procedure in the previous section.

```
procedure load_unload =
   timeplate tp1;
   cycle =
      force RST 0;
      force CLK 0;
      force scan_en 1;
   end;
   apply shift 1;
end;
```

The timeplate of the load_unload procedure specifies the period is 50ns. However, the load_unload procedure includes an apply statement that executes one shift procedure. The shift procedure requires an additional 50ns. Thus, the load_unload procedure actually requires a total time of 100ns, as shown in Figure 6-4.

Figure 6-4. Timing Diagram for Load_Unload Procedure



Within the load_unload procedure, after the completion of the cycle block, the shift procedure starts at 50ns, executes for 50ns, and ends at 100ns. Thus, the load_unload procedure also ends at 100ns.

As with the shift procedure, the timing clock determines the event times for the load_unload procedure. If the timing clock is at Y when the load_unload procedure begins, the first three events happen at time Y. When the apply cycle executes, the timing clock advances to Y+50, which is when the shift procedure begins. As mentioned previously, the shift procedure requires 50 time units. Therefore, when the apply cycle finishes, the timing clock reads Y+100.

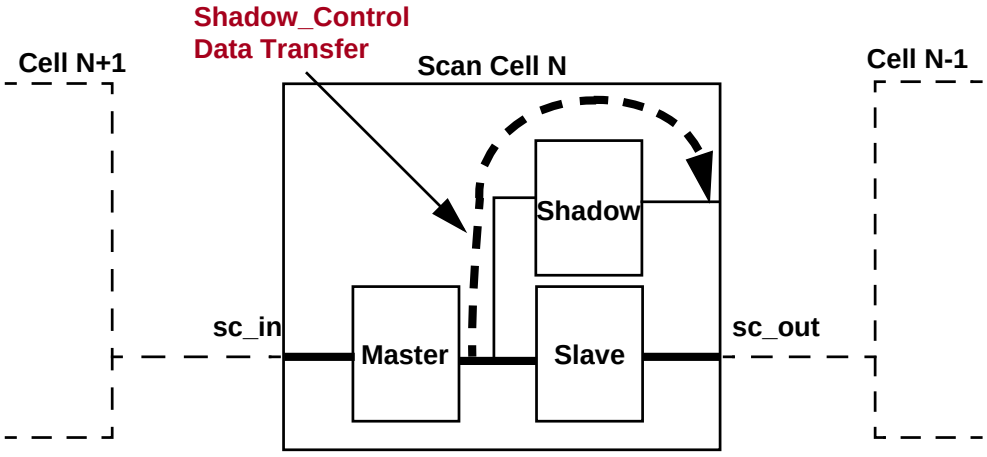
Because it is the last event in the load_unload procedure, the end of the apply cycle determines the end of the load_unload procedure.

Shadow_Control (Optional)

This optional procedure, which may only contain force and pulse event statements, describes how to load the contents of a scan cell into the associated shadow.

If you use this procedure, you must also apply the shadow_control command in the load_unload procedure. This procedure must not disturb the contents of any of the scan cells. Figure 6-5 shows the data flow for the shadow_control procedure.

Figure 6-5. Shadow_Control Procedure

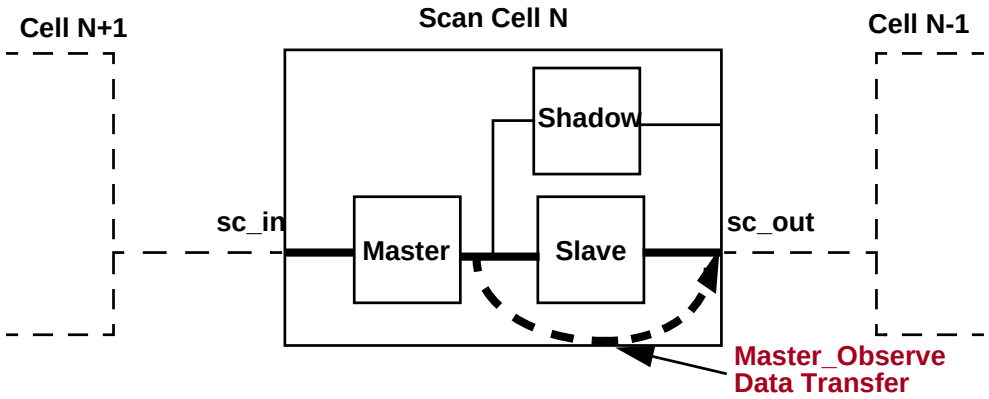


Master_Observe (Sometimes Required)

This procedure, which may only contain force and pulse event statements, describes how to place the contents of a master into the output of its scan cell, where you can observe it by using the unload operation.

Figure 6-6 shows the data flow for the master_observe procedure.

Figure 6-6. Master_Observe Procedure



You do not need to use this procedure if the master element's output *is* the output of the scan cell. The D1 rule ensures this procedure does not disturb master memory element's contents. You can override this requirement by changing the D1 rule handling. The following example shows a master_observe procedure for the LSSD architecture:

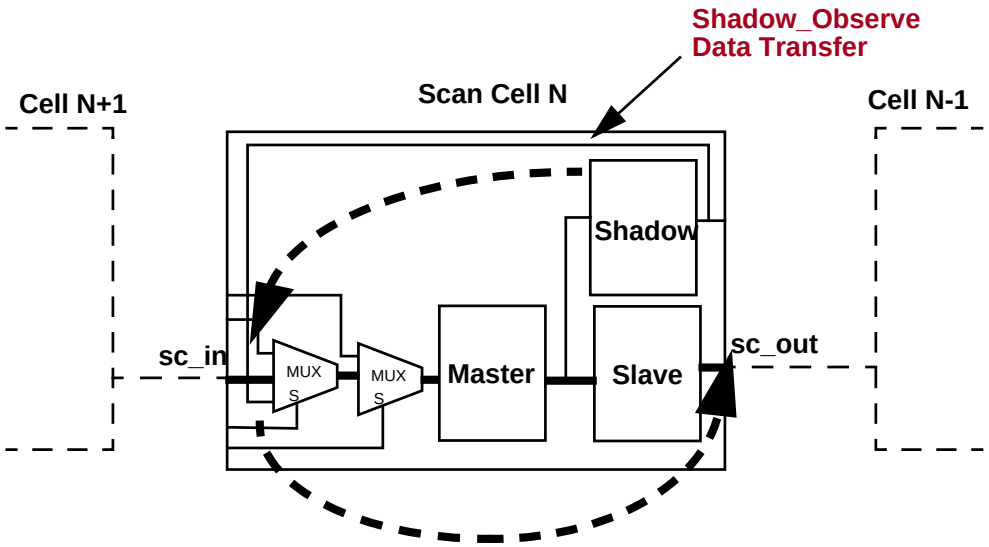
```
// LSSD architecture example
procedure master_observe =
   timeplate tp1;
   cycle =
      // Force all clocks off
      force scan_sclk 0;
      force scan_mclk 0;
      force rst       0;
      force clk       0;
      // Pulse the slave clock
      pulse scan_sclk;
   end;
end;
```

Shadow_Observe (Optional)

This optional procedure, which may only contain force and pulse event statements, describes how to place the contents of a shadow into the output of its scan cell, assuming the circuitry of the scan cell allows the transfer of data in this way. Once the data is at the scan cell output, you can observe it by applying the unload command. This procedure allows the shadow to be used as an observation point in the design.

Figure 6-7 shows the data flow of the shadow_observe procedure.

Figure 6-7. Shadow_Observe Procedure



Seq_Transparent (Optional)

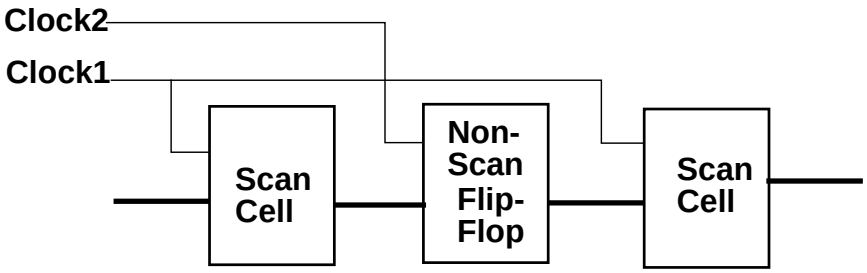
This procedure (optional for “patterns -scan” context) identifies how to make non-scan cells and RAM read ports functionally behave transparently. This procedure activates the clock inputs of non-scan cell inputs, thus pulsing data through the cells “transparently.”

All clocks must be at their off-states and constrained pins at their constrained states before applying the seq_transparent procedure. The procedure must immediately follow a force of all the primary inputs. For more information on the sequential transparent operation, refer to “Sequential Transparent Patterns” in the *Tessent Scan and ATPG User’s Manual*.

You can use multiple clock cycles to create the sequential transparent conditions. You may define up to 32 different seq_transparent procedures within a procedure file. When simulation mode is set to RAM_sequential, each force_all statement in the pattern file can use any of the possible seq_transparent procedure choices. In “patterns -scan” context, the tool treats non-scan state elements that cannot use the sequential transparent procedures as tie-X gates.

There may be occasions when you would want to use seq_transparent procedures when the design contains no scan chains. In this case, you would use the add_scan_groups command, specifying the name “dummy” for the group name; the tool would expect the specified test procedure file to contain only the timeplate and seq_transparent procedure. For more information, refer to the add_scan_groups command reference page in the *Tessent Shell Reference Manual*. Figure 6-8 shows some circuitry that could benefit from a seq_transparent procedure.

Figure 6-8. Sequential Transparent Circuitry Example



The following example shows the seq_transparent procedure for Figure 6-8.

```
timeplate tp1=
   force_pi 0;
   measure_po 0;
   pulse clock1 30 10;
   pulse clock2 30 10;
   period 50;
end;

procedure seq_transparent tran1=
   timeplate tp1;
   cycle=
      force clock1 0;
```

```
            force clock2 0;
            force reset 1;
         end;
         cycle=
            pulse clock2;
         end;
         cycle=
            restore_pi;
         end;
      end;
```

The basic stimuli necessary to create transparent behavior for the non-scan flip-flop shown in Figure 6-8 are:

- Force all clocks off
- Pulse non-scan cell clock Clock2
- Restore primary inputs to original values

In more complex situations, you may need to set primary inputs to certain values, place conditions on scan cells, pulse multiple clocks, and so on.

You can use the report_seq_transparent_procedures command to display data defined by the seq_transparent procedures. For more information, refer to the report_seq_transparent_procedures description in the *Tessent Shell Reference Manual*.

Clock (Optional)

This procedure (optional in “patterns -scan” context) provides flexible clock handling during the test procedures. Using clock procedures, instead of pulsing a single clock during a capture cycle, you can serially exercise multiple clocks and force non-clock pins that do not affect captured data.

The following example shows a clock procedure used to operate two clocks in sequence:

```
procedure clock clock_proc1 =
   timeplate tp3;
   cycle =
      pulse clk1;    // Pulse first clock
   end;
   cycle =
      pulse clk2;    // Pulse second clock
   end;
end;
```

Clock procedures must abide by the following rules:

- The procedure must activate at least one clock.
- If you define multiple clock procedures, only one of these procedures can activate a specific clock.
- The procedure events cannot violate pin constraints or equivalence conditions.
- The procedure can only force non-clock pins, if they do not affect data captured into state elements, whose clocks may activate later in the procedure.
- Multiple clocks that activate serially cannot logically interact.
- The procedure must follow all standard rules for both clock and non-clock pin usage.
- Each clock procedure must have a unique name.
- If a state element can change state during the procedure, the element must be stable when all clocks are off and pins are constrained.
- *Transparent_capture* cells are stable state elements that can capture data during the procedure and whose new data can affect other state elements later in the procedure. Design rules D10 and D11 ensure that these cells do not connect to state elements that capture old data or propagate data to primary outputs.
- The procedure must set all bidirectional pins to their input mode prior to executing the restore_bidis statement.

If a clock procedure contains a restore_bidis statement, the tool cannot use sequential ATPG. This may cause a problem if you set the tool for multiple clock compression because multiple clock compression uses sequential ATPG.

Skew_Load (Optional)

This optional procedure propagates the output value of the preceding scan cell into the master memory element of the current cell without changing the slave, for all scan cells. Using only force and pulse event statements, this procedure defines how to apply an additional pulse of the master shift clock after the scan chains are loaded.

Figure 6-9 shows the data flow of the skew_load procedure.

Figure 6-9. Skew_Load Procedure

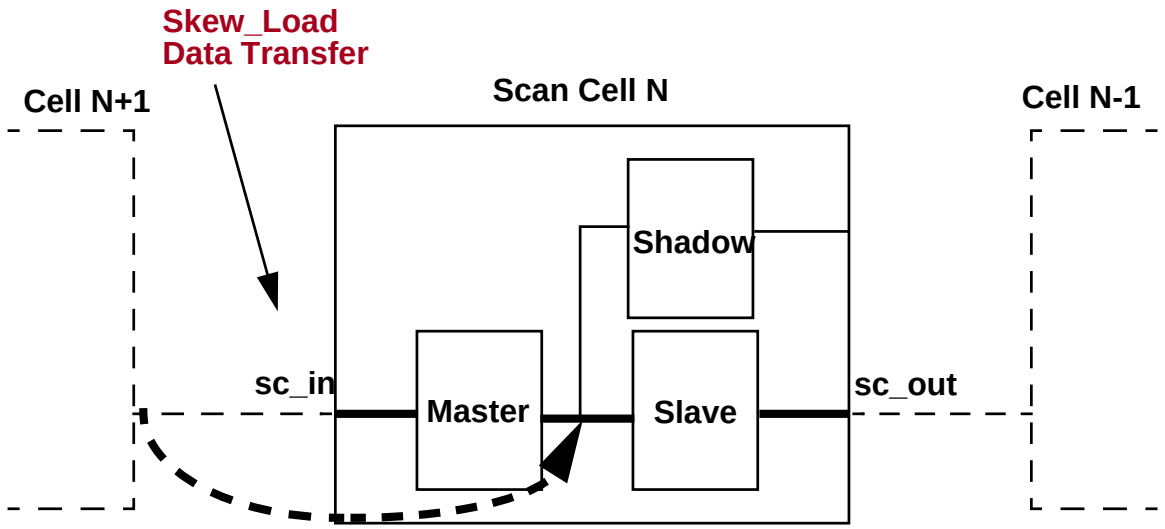
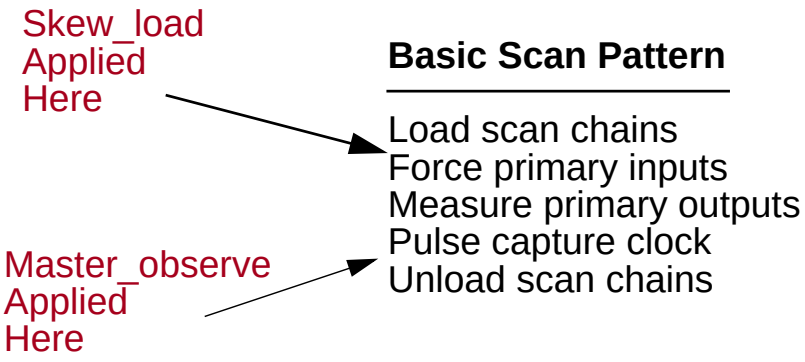


Figure 6-10 shows where you apply the skew_load procedure and the master_observe procedure within the basic scan pattern events.

Figure 6-10. Skew_load applied within Pattern



Clock_run (Optional)

For every controller, or concurrent controller group, you can write a clock_run procedure, if needed. The clock_run procedure has both an internal mode as well as an external mode.

You can specify only one clock_run procedure per controller or concurrent group; however, you do not need to specify a separate procedure for each controller instance. The same procedure can be used for multiple controllers. You need to specify a separate procedure for a controller instance only if it maps to a different set of internal clocks.

In case of controllers running concurrently, and some of these controllers clocks are driven by PLL internal clocks, the clock_run procedure is required per concurrent group. It is not required for every BIST controller participating in the group to have its clock driven by a PLL internal clock. For some controllers, their clocks can be driven by a PLL reference clock or even by a system clock.

The tool relies on you to control the PLL control signal. This can be achieved by forcing the PLL control signal to a proper value in a test_setup procedure and in external mode of clock_run procedure as well (it depends on the PLL model behavior).

A clock_run procedure has to have a N-to-1 or 1-to-N ratio between internal and external cycles; that is, either the internal mode has to have only one cycle, or the external mode has to have only one cycle. You cannot have, for example, two external cycles and three internal cycles.

Example

In this example, consider a PLL model that has two clocks: a reference clock and an internal clock. Based on the PLL control signal:

- The internal clock is 2X faster than the reference clock when PLL control is 0.
- The internal clock is 4X faster than the reference clock when PLL control is 1.

If you wanted a PLL internal clock to drive the BIST controller clock, use the following MBISTArchitect commands in your BIST insertion dofile to define the clocks and make the proper connection.

```
add clocks 0 PLL/int_clk
add clocks 0 topPLLclk
add pin mapping /PLL/int_clk    /controller/bist_clk
```

The PLL control is set to 0 using a test_setup procedure as follows:

```
procedure test_setup =
timeplate soc_timeplate;
   cycle =
    force PLL_Control 0 ;
    pulse topPLLclk;
  end;
end;
```

The following snippet has a clock_run procedure that describes the PLL model behavior assuming that the PLL_Control is set to 0 as described in the previous test_setup procedure.

```
timeplate timeplate_internal =
   force_pi 0 ;
   measure_po 90 ;
      pulse PLL/int_clk 5 50 ; // speed is 2X
   period 100 ;
end ;
timeplate timeplate_external =
      force_pi 0 ;
   measure_po 180 ;
   pulse topPLLclk 5 100 ;
   period 200 ;
end ;
procedure clock_run pll_clk=
   mode internal =
      timeplate timeplate_internal ;
         cycle =
            pulse PLL/int_clk ;
         end ;
         cycle =
            pulse PLL/int_clk ;
         end ;
   end ;
      mode external =
         timeplate timeplate_external ;
         cycle =
            pulse topPLLclk ; // connected to reference clock
         end ;
      end ;
end ;
```

Capture Procedures (Optional)

There are three types of capture procedures. These procedures are optional in the "patterns -scan" context.

- The *default capture procedure* is an optional capture procedure, without a name, that provides information on how the series of capture events are broken into cycles and which timeplates these cycles use. The default capture procedure is defined in the procfile as part of the scan group definition or internally derived by the tool when you do not define one. If you need to create or edit a default capture procedure, see "Clock_po (Optional)" in this chapter.
- The *named capture procedure* is an optional capture procedure, with a name, that is used to define explicit clock cycles. You can create multiple named capture procedures, each with a unique name, using the create_capture_procedures command. If you need to manually create or edit named capture procedures, see "Rules for Creating and Editing Named Capture Procedures" in this chapter. For information on using named capture procedures to create at-speed test patterns, see "At-Speed Test Using Named Capture Procedures" in the *Tessent Scan and ATPG User's Manual*.
- The *external_capture procedure* is an optional capture procedure used for all capture cycles between each scan load, even when the pattern is a multi-load pattern. External_capture procedures are used by the "set_external_capture_options -capture_procedure" command. External_capture procedures that are used with this command have several restrictions:
  - The procedure can only have one force_pi statement and no measure_po statements. This is because in order to use the "set_external_capture_options -capture_procedure" switch, the patterns to be saved must be hold_pi and mask_po patterns. The statements in the capture procedure must match up to this.
  - Unlike named capture procedures, the external_capture procedure cannot have any load_cycles as it is meant to be used between each scan load of a pattern. The external_capture cannot contain any events on internal signals.
  - When using the -capture_procedure switch with set_external_capture_options, all clocks in the design that are internally connected (don't trace to just a cut point), must be controlled. In addition, the clocks must either be constrained, free_running, controlled by a clock_control definition, or the clock must have an event in the external_capture procedure.

Rules for Creating and Editing a Default Capture Procedure

- The default procedure may only contain force_pi, measure_po, pulse_capture_clock, bidir_force, bidi_force_pi, bidi_force_off, and bidi_measure_po event statements that represent the non-scan activity for a normal pattern. There is no overlap between the capture procedure and the existing clock procedure.
- Use the pulse_capture_clock statement in the default capture procedure to indicate in which cycle one or more capture clocks should be pulsed.

- Do not specify any complex clocking that needs to be described for capture clocks or other clocks in the default capture procedure; specify it in the clock procedure or by using a named capture procedure.
- Do not specify any type of pin or ATPG constraint in the default capture procedure. For example, specifying that a certain pin is to be held at a certain state in the default capture procedure does not restrict the ATPG engine from applying different values to that pin. However, you can use the bidi_force and bidi_force_pi statements in the default capture procedure to force all bidirectional pins off in one cycle and force the ATPG values on the bidirectional pins in the next cycle.

Rules for Creating and Editing Named Capture Procedures

- A named capture procedure may only contain force_pi, measure_po, observe_method, pulse (named clock), and condition statements.
- If you use mode definitions, all cycles in a procedure must be defined within mode definitions. Use the keyword “mode” with two mode blocks: “internal” and “external”. Use the mode_internal definition to describe what happens on the internal side of the on-chip PLL. Use the mode_external definition to describe what happens on the external side of the on-chip PLL.
- All events in a named capture procedure that use modes must be duplicated in both modes. The only difference is that the internal mode uses only internal clocks and the external mode uses only external clocks. The number of cycles and timeplates used can be different as long as the total period of both modes is the same.
- Signal events used in both internal and external modes must happen at the same time. Examples of these events are force_pi, measure_po, and other signal forces, but also include clocks that can be used in both modes.
    - If a measure_po statement is used, it can only appear in the last cycle of the internal mode and must occur before the last clock pulse. If no measure_po statement is used, the tool issues a warning that the primary outputs is not be observed.
    - The cumulative time from the start of the first cycle to the measure_po must be the same in both modes.
    - The external mode cannot pulse any internal clocks or force any internal control signals.
    - A force_pi statement needs to appear in the first cycle of both modes and occur before the first pulse of a clock.
    - If an external clock goes to the PLL and to other internal circuitry, a C2 DRC violation is issued.
    - At-speed cycles need to be continuous; that is, a named capture procedure cannot have more than one at-speed clocking subsequence.

- All defined real clocks (excluding internal clocks) must be forced to off state first in the mode_internal definition.

For more information, see "Defining Internal and External Modes" in the *Tessent Scan and ATPG User's Manual.*

- Do not use the pulse_capture_clock statement in a named capture procedure. The clocks used are explicitly pulsed.
- If you want to specify the internal conditions that need to be met at certain scan cells in order to enable a clock sequence, use the condition statement at the beginning of the cycle statement in the named capture procedure.
- If you want to define a specific observe method for each named capture procedure, use the observe_method statement in the named capture procedure; otherwise, the ATPG engine automatically selects master, slave, or shadow observation.

Note

The write_patterns command allows you to save internal or external clock patterns. Internal clock patterns can be used to simulate the DUT without having the PLL modeled, while the external patterns only exercise the PLL external clocks and control signals. Internal patterns are the default for ASCII and binary formats, and external patterns are the default for tester formats.

- If you generate patterns using a named capture procedure that has both internal and external modes and you save them in STIL or WGL format, you must use the write_patterns command's "internal" option to read them back into the tool (for example, to use in diagnosis). For more information, see the description of the -Mode_internal and -Mode_external switches for the write_patterns command in the *Tessent Shell Reference Manual*.

DRC rules W20 through W36 check named capture procedures. If a DRC error prevents use of a capture procedure, the run aborts.

Example of a Named Capture Procedure

Following is an example of a named capture procedure:

```
procedure capture pll_1 =
   observe_method slave;
   mode internal =
   timeplate tp1_in;
   cycle slow =
      condition /dut/cell1 0;
      force clk_cntr 1;
      force_pi;
      force pll_clk 0; //external clock must be forced to off state
   end;
   cycle =
      pulse /patha/int_clk;
   end;
```

```
      cycle =
         force clk_cntr 0;
         measure_po;
         pulse /patha/int_clk;
      end;

      mode external =
         timeplate tp1_ex;
         cycle =
            force clk_cntr 1;
            force_pi:
            measure_po;
            pulse pll_clk;
         end;
         cycle =
            force clk_cntr 0;
         end;
      end;
   end;
```

Slow and Load Types in the Cycle Statement

Optionally, you can add a “slow” or a “load” type to the cycle definition. For example:

```
cycle slow =
...
end;
```

- The slow cycle indicates that at-speed faults cannot be launched or captured. The tool must know which at-speed cycles are slow in order to get accurate at-speed fault coverage simulation numbers; therefore, be sure to include “slow” when defining cycles that are not at-speed cycles in an at-speed capture procedure.

Note

At-speed cycles need to be continuous; that is, a named capture procedure cannot have more than one at-speed clocking subsequence.

- The load cycle indicates that the cycle is always preceded by an extra scan load. The first cycle in a named capture procedure is always a load (with or without the load type designation), so you typically apply “load” to subsequent cycles. An at-speed launch cycle can be a load cycle; however, none of the cycles that follow in the at-speed sequence, up to and including the capture cycle, can be load cycles.

Note

To get extra loads, you must enable the tool’s multiple load and clock sequential capabilities by issuing the set_pattern_type command with “-multiple load on” and “-sequential <2 or greater>”. For more information, see “Multiple Load Patterns” in the *Tessent Scan and ATPG User’s Manual*.

The following example illustrates the “slow” and “load” attributes:

```
procedure capture multi_load_example =
   timeplate tp1;
   // first cycle is always a load, with or without load type designation
   cycle slow =
      force_pi;
      force wr_enable 1;
      pulse int_clk1;
   end;
   cycle slow load =
      pulse int_clk1;
   end;
   cycle =
      force re_enable 1;
      pulse int_clk1; // launch clock
   end;
   cycle =
      pulse int_clk1; // capture clock
   end;
end; // end of capture procedure
```

launch_capture_pair Statement

Optionally, you can add one or more “launch_capture_pair” statements to the beginning of a named capture procedure. This statement defines legal at-speed launch and capture points in non-adjacent cycles. If you do not use the launch_capture_pair statement, the tool will launch and capture only in adjacent cycles. If at least one launch and capture clock pair is defined, the launch and capture points are derived from the defined launch and capture clock pairs.

Note

This statement is only supported when using a named capture procedure to perform test generation.

The syntax of the launch_capture_pair statement is as follows:

```
launch_capture_pair <launch_clock_pin_name> <capture_clock_pin_name>;
```

Where:

- **launch_clock_pin_name** is the clock used to launch the transition.
- **capture_clock_pin_name** is the clock used to capture the transition.

The launch clock cycle is used to check the transition condition. The capture clock cycle is used to capture the transition fault effect. The cycles between the launch clock and capture clock must be at-speed cycles. They cannot include any slow cycles between them. The faults to be tested by the named capture procedure with the defined launch and capture clock pair are the faults that can be launched by the launch clock and captured by the capture clock defined in the launch_capture_pair statement.

The following is an example of the launch_capture_pair statement:

```
procedure capture c1_c1 =
   launch_capture_pair c1 c1;

   cycle = // cycle 1
      force_pi;
      force c1 0;
      force c2 0;
      force c3 0;
      pulse c1;
   end;

   cycle = // cycle 2
      pulse c2;
   end;

   cycle = // cycle 3
      pulse c1;
      pulse c3;
   end;
end; // end of capture procedure
```

In this example, a valid launch can happen in cycle 1. A valid capture can happen in cycle 3 only with c1 as the capture clock. A launch in cycle 1 and a capture in cycle 2 is not used for fault detection. The faults to be tested by this named capture procedure are the faults that can be launched and captured by clock c1.

Example PLL Model and Named Capture Procedure

The following is an example of a PLL model and named capture procedure. In this example, consider a PLL model that has two clocks: a reference clock and an internal clock. Based on the PLL control signal, you know the following:

- The internal clock is 2X faster than the reference clock when the PLL control is 0.
- The internal clock is 4X faster than the reference clock when the PLL control is 1.

If you wanted a PLL internal clock to drive the BIST controller clock, the following example command is loaded at the BIST insertion to make the proper connection.

```
add pin mapping /PLL/int_clk    /controller/bist_clk
```

The following is a named capture procedure that describes the PLL model behavior, assuming your PLL control is 0 (for example, the internal clock speed 2X faster than the reference clock).

```
timeplate timeplate_internal =
   force_pi 0 ;
   measure_po 40 ;
    pulse PLL/int_clk 5 25 ; // 2X
   period 50;
end;
timeplate timeplate_external =
      force_pi 0 ;
   measure_po 90 ;
```

```
   pulse topPLLclk 5 50;
   period 100 ;
end;
procedure capture PLL_1 =
  mode internal =
    timeplate timeplate_internal;
    cycle =
      pulse PLL/int_clk ;
end;
cycle =
      pulse PLL/int_clk ;
    end;
end;
  mode external =
    timeplate timeplate_external ;
    cycle =
      pulse topPLLclk ; // connected to referecne clock
    end;
  end;
end;
```

The sequence of events in capture procedures must pass DRC. DRC results influence the ATPG kernel and how the ATPG engine generates test patterns. The basic pattern for the capture procedure is as follows:

Capture Procedure Pattern

Force primary inputs
Measure primary outputs
Pulse capture clock

Viewing Optimized Named Capture Procedures (NCPs)

Use this procedure when creating or debugging NCPs in order to view optimized information for an individual NCP. For more information on the optimizing process, see the report_capture_procedures command in the *Tessent Shell Reference Manual*.

Prerequisites

- DFTVisualizer is invoked and the Wave window is open and active.
- At least one NCP is defined in the test procedure file.

Procedure

1. Choose **Windows > Data** to open the Data window.
2. Choose **Windows > Wave** window to open the Wave window.
3. Select the **Data > Named Capture** menu item and choose the NCP you want to view.
4. Click the  icon to view the optimized NCPs; click the  icon to return to viewing unoptimized NCPs.

Clock_po (Optional)

This procedure (optional in “patterns -scan” context), which can contain only force_pi, measure_po, bidi_force_pi, and bidi_force_off event statements, represents the non-scan activity for a clock PO pattern. Use this procedure instead of the capture procedure.

Note that with this procedure, you must use a timeplate that does not pulse the clocks.

The following shows the pattern for the clock_po procedure pattern.

Clock_po Procedure Pattern

Force primary inputs (including clocks)
Measure primary outputs

Ram_sequential (Optional)

This procedure (optional for “patterns -scan” context), which may only contain force_pi, pulse_write_clock, pulse_read_clock, bidi_force_pi, and bidi_force_off event statements, represents the RAM sequential events in a RAM sequential pattern. Use this procedure with the capture procedure.

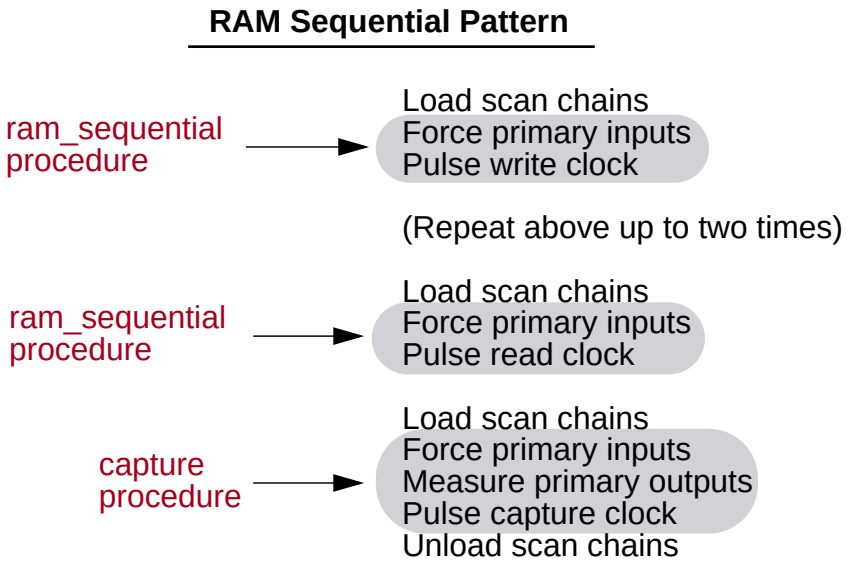
The following illustrates the basic ram_sequential procedure pattern format.

Ram_sequential Procedure Pattern

Force primary inputs
Pulse write clock
and/or Pulse read clock

Figure 6-11 shows an entire RAM sequential pattern, which illustrates where the ram_sequential and capture procedures are used.

Figure 6-11. Full Ram Sequential Pattern



Ram_passthru (Optional)

This procedure (optional in “patterns -scan” context), which may only contain force_pi, measure_po, pulse_write_clock, pulse_capture_clock, bidi_force_pi, bidi_force_off, and bidi_measure_po event statements, represents the non-scan activity for a RAM passthrough pattern. Use this procedure instead of the capture procedure.

The following shows the ram_passthru procedure pattern.

Ram_passthru Procedure Pattern

Force primary inputs
Pulse write clock
Pulse read clock
Measure primary outputs
Pulse capture clock

Clock_sequential (Optional)

This procedure (optional for “patterns -scan” context), which may only contain force_pi, pulse_write_clock, pulse_read_clock, pulse_capture_clock, bidi_force_pi, and bidi_force_off event statements, represents the clock sequential events in a clock sequential pattern. Use this procedure with the capture procedure.

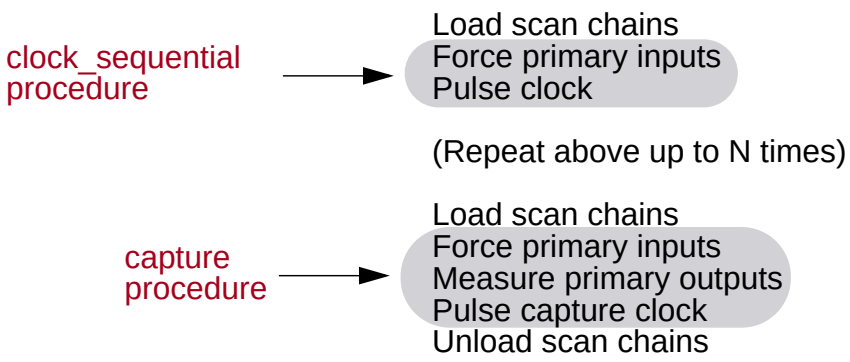
The following shows the clock_sequential procedure pattern.

Clock_sequential Procedure Pattern

Force primary inputs
Pulse write clock
and/or Pulse read clock
and/or Pulse capture clock

Figure 6-12 shows an entire clock sequential pattern, which illustrates where the clock_sequential and capture procedures are used.

Figure 6-12. Full Clock Sequential Pattern



Init_force (Optional)

This procedure (optional for “patterns -scan” context), which may only contain force_pi event statements, represents the force cycle that is used in an ATPG pattern that targets a transition fault. The transition must be launched off of the last scan chain shift. This procedure is used when the fault type is set to transition fault and either the depth is set to 2 or less or the ATPG engines fail to find a sequential pattern that can cover this transition fault. Use this procedure with the capture procedure.

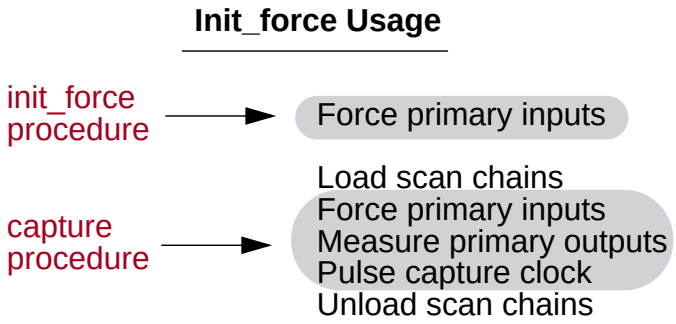
The following illustrates the format of the init_force procedure pattern.

Init_force Procedure Pattern

Force primary inputs

Figure 6-13 shows the pattern which uses the init_force procedure.

Figure 6-13. Init_force Procedure Usage



Test_end (Optional, all ATPG tools)

This optional procedure is used to add a sequence of events to the end of a test pattern set.

The test_end procedure may only contain force and pulse event statements (see the following exception), and can only be defined once for all scan groups. When saving patterns, the test_end procedure will be applied to the end of each pattern set saved.

The following shows the general pattern for the test_end procedure pattern.

Test_end Procedure Pattern
Force primary inputs Pulse clocks

Example 1: Using test_end in a Procedure File

The following is a partial example of how the test_end procedure might be used in a procedure file:

```
timeplate tp1 =
  force_pi 0;
  measure_po 10;
  pulse ref_clk 50 50;
  period 100;
end;

procedure test_end =
  timeplate tp1;
  cycle =
    force ctrl_a 1;
    force tms 0;
    pulse ref_clk;
  end;
end;
```

Example 2: Test_end Procedure with Timeplate

The following is an example test_end procedure with its corresponding timeplate:

```
timeplate tp4 =
    force_pi 0;
    pulse TCK 10 10;
    measure_po 30;
    period 40;
end;

procedure test_end =
    timeplate tp4;
    cycle =
        // TMS = 1, change to select-DR state
        force TDI 1;
        force TMS 1;
        pulse TCK;
    end;
```

```
        cycle =
            // TMS = 0, change to capture-DR state

    ...
        cycle =
            // Scan out signature (MISR has length of 4)
            force TDI 1;
            force TMS 0;
            pulse TCK;
        end;
        cycle =
            force TDI 1;
            force TMS 0;
            pulse TCK ;
        end;
    ...
    end;
```

Sub_procedure

This procedure eliminates the need to insert duplicate actions within a procedure. Once you have defined a sub_procedure, you can specify this procedure within other procedures using the apply statement.

You can also set the tool to reissue the sub_procedure as many times as needed by specifying the repeat_count. Because the repeat_count is required when using apply sub_procedure, you must enter a minimum of 1 for this parameter.

Sub_procedure Looping

Sub_procedure looping is used to reduce the size of pattern files. The default behavior of the sub_procedure is to use “loops” or “repeats” in all applicable pattern formats to repeat the contents of the sub_procedure N times, where N is greater than 1.

Disabling Sub_procedure Looping

If you want the event data in a sub_procedure to be expanded and represented as N sets of vectors in the pattern file, where N is the number of times the sub_procedure is applied, use the “ALL_NO_LOOP **1”** parameter file keyword to disable the use of “loop” or “repeat” statements.

For example, if the test_setup procedure has the following statement:

```
apply pulse_bclock 1000;
```

The vector data for the sub_procedure “pulse_bclock” would be expanded to be 1000 vectors. The default for the ALL_NO_LOOP keyword is off (0).

Sub_procedure Definition Format

The sub_procedure definition has the following format.

```
procedure sub_procedure my_subprocedure =
   timeplate tp1;
   cycle =
      force_pi;
      measure_po;
   end;
end;
```

Using the Sub_procedure in a Procedure

The following is an example of how to use the sub_procedure in a procedure.

```
procedure shift =
   scan_group grp1;
   timeplate tp1;
   apply my_subprocedure 4;
   cycle =
      force_sci;
      measure_sco;
      pulse T;
```

```
    end;
end;
```

Note

You must first define a sub_procedure before using it in a procedure. Next, you can apply a sub_procedure within any procedure type. Also, you cannot use a sub_procedure within the “cycle =” and “end;” statements.

Additional Support for Test Procedure Files

This section describes additional support for procedure files.

Environment Variables in Procedure Files

You can reference any environment variable and “dofile variable” in the procedure file.

For example, if a dofile contains the following:

```
$MY_CLOCK = clock1
add_clocks 0 $MY_CLOCK
```

Where:

$MY_CLOCK is a dofile variable. You can extend the use of this variable to the procedure file as shown in the following example:

```
timeplate tp1 =
...
pulse $MY_CLOCK 10 10
...
end;
```

Merging Procedure Files

It is possible to specify more than one procedure file for a design. You can specify a procedure file with the add_scan_groups command or with the read_procfile command. You need to supply (to the ATPG tool) a minimum set of information in the procedure file with the add_scan_groups command. You must supply all event information for the scan procedures.

However, after leaving setup mode, it is possible to specify non-scan procedures, timeplates, and new timing for the scan procedures by reading in an additional procedure file with the read_procfile command. Specifying new information for the same design, from more than one procedure file, is known as “merging the procedure files.” To properly merge the information from multiple procedure files, the Vector Interfaces code follows these rules:

- All scan procedures that you will use must be specified in the procedure file that you load with the add_scan_groups command.
- If you load a procedure that contains nothing but the procedure name, a timeplate name, and an optional scan group, it is a template procedure. If a procedure already exists by

that name for that scan group (if it is a group-specific procedure), then the timeplate is mapped onto the existing procedure. If no procedure already exists with that name, the tool stores the template procedure for future use.

- If you load a new complete procedure (not a template) and a procedure already exists by that name for the specified scan group (if applicable), the new procedure overwrites the existing one.
- In both cases, when a procedure overwrites an existing one, or if a new timeplate is mapped to an old procedure, the tool checks the procedures to make sure that the sequence of events in the new procedure does not differ from the old procedure.

Default Information Provided by the Tool

When you issue the write_patterns command, the tool checks to make sure that all procedures and timeplates needed to save the patterns in the specified format are present.

If there are any missing non-scan procedures, the tool creates default procedures and issues a warning. For example, in cases where there are ram_sequential patterns that need to be saved and no ram_sequential procedure was supplied, the tool automatically creates a default procedure.

For any procedures that are created or that do not have a timeplate specified, the default timeplate is mapped to these procedures, if it is set. You can set the default timeplate by using the set default_timeplate statement previously described in the "Set Statement" section. If you use this statement, the timeplate specified when creating default procedures is used. If the default procedure needs to be created and no default timeplate has been set, then the first timeplate specified is used. If no timeplates are specified, a default timeplate is created as well.

Creating Test Procedure Files for End Measure Mode

Use this procedure to create test procedure files that enable end measure mode. End measure mode refers to the special handling that the Vector Interfaces code needs to move the measure to the end of the shift and capture cycle.

Prerequisites

- A test procedure file.

Procedure

1. Create a new timeplate that measures the outputs after the clock pulse.
2. Change the timeplate for the shift and load_unload to point to the new timeplate.
3. Add the measure_sco statement to the load_unload procedure.
4. Make sure all shift procedures have the measure_sco statement after the shift clock. When end measure mode is enabled, the measure_sco statement measures the next value from the output of the scan chain. The very first value for the output of the scan chain is measured by a measure_sco statement in the load_unload procedure.
5. Change the timeplate for the capture cycle by breaking it into two cycles. Move the capture clock to the second cycle of the capture procedure to allow the measure at the end. In the first cycle, the force_pi and measure_po are performed. In the second cycle, the capture clock is pulsed. When using end measure mode, a measure cannot be performed after the capture clock.

Examples

```
set time scale 1.000000 ns ;
set strobe_window time 10 ;
 timeplate gen_tp1 =
    force_pi 0 ;
    measure_po 10 ;
    pulse clk 20 10;
    pulse edt_clock 20 10;
    pulse ramclk 20 10;
    period 40 ;
 end;
// CREATE A NEW TIMEPLATE THAT MEASURES AFTER THE CLOCK PULSE
  timeplate gen_tp2 =
    force_pi 0 ;
    //    measure_po 10 ;
    pulse clk 20 10;
    pulse edt_clock 20 10;
    pulse ramclk 20 10;
    measure_po 35 ;       // <<== NEW MEASURE STATEMENT
    period 40 ;
```

```
 end;
// FOR CAPTURE SPLIT INTO TWO CYCLES
 procedure capture =
    timeplate gen tp1 ;
    cycle =
        force_pi ;
        measure_po ;
    end ;
    cycle =
        pulse_capture_clock ;
    end;
 end;
// FOR THE SHIFT AND LOAD_UNLOAD, USE THE NEW TIMEPLATE
 procedure shift =
    scan_group grp1 ;
    timeplate gen_tp2 ; //<<=== NEW TIMEPLATE
    cycle =
        force_sci ;
        force edt_update 0 ;
        measure_sco ;
        pulse clk ;
        pulse edt_clock ;
    end;
 end;
// ADD A MEASURE_SCO TO THE LOAD_UNLOAD PROCEDURE
 procedure load_unload =
    scan_group grp1 ;
    timeplate gen_tp2 ; //<<=== NEW TIMEPLATE
    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 ;
   measure_sco;     //<<=== NEW MEASURE STATEMENT
    end ;
    apply shift 39;
 end;
 procedure test_setup =
    timeplate gen_tp1 ;
    cycle =
        force edt_clock 0 ;
    end;
end;
```

Serial Register Load and Unload for LogicBIST and ATPG

This section describes using the test_setup or test_end test procedure file procedures to serially load or unload control registers in the circuit under test. Additionally, this section discusses the commands you must add to your dofile to use this functionality.

Serial register load and unload targets the following flows:

- **Logic BIST** — Used to serially unload certain registers (for example, MISRs) using the test_end procedure.
- **Low Pin Count Test** — Used to serially load registers with test information through a serial access port such as a JTAG TAP controller.

Register Load and Unload Use Models

Using serial register load and unload allows you to identify a load/unload register variable value in the test procedure file and define this variable, either static or dynamic, using commands in your Tessent dofile. By using this functionality, you can load control registers by using a type of load procedure where the data value for the register can be passed as a string of values, and the load register uses a shift mechanism to show how this string of data values is shifted into the register.

To serially load/unload register value variables, you must make modifications to your test procedure file and your Tessent dofile.

The register value variable is used in the test procedure file to load the value into a register through the data input pin, or unload a value from a register through the data output pin. The load/unload procedures support pin inversions, but you must explicitly specify this.

The registers to be loaded/unloaded can be cascaded such that one load or unload operation loads or unloads multiple values, for example PRPG/MISR values.

Static Versus Dynamic Register Variables

You can define both static and dynamic register value variables.

- Static Variable — A specific register value variable string you specify during setup mode. Once set, you cannot change this variable.
- Dynamic Variable — A register value variable string that you can define or change later, and can use tool-specific switches.

Static Variable

A static register variable is one where the value is assigned to the variable when the variable is defined (using the add_register_value), and this value then cannot be changed. This static value

is used by DRC when simulating the procedures. A static variable cannot be set using tool specific switches to link the variable to tool computed values, as these may change.

Dynamic Variable

A dynamic register value variable uses tool-specific switches and arguments to link the variable to a value computed by the tool. If you use a dynamic variable, then you must specify the register's width unless the tool knows this value at the time DRC is run (for example, PRPG size).

This method is used for defining a variable that has a tool-specific computed value that is available when patterns are saved and needs to be loaded or unloaded by the test_setup or test_end procedures. The value defaults to all X bits, and you can specify the actual value can non-Setup mode by using the set_register_value command—see "Definition of a Dynamic Register Value Variable" for complete information.

If no value is specified the first time DRC is invoked after adding a register value variable, the tool considers the variable to be dynamic for DRC and any future invocation of DRC.

Test Procedure File Modifications

In the test procedure file, the load_unload_registers Procedure and shift Keyword are used.

load_unload_registers Procedure

The **load_unload_registers** procedure is a named procedure; consequently, you can have multiple occurrences of this procedure, corresponding to multiple groups of registers that need to be loaded and/or unloaded. The load_unload_registers procedure can only be called from the following procedures:

- **test_setup**
- **test_end**

The load_unload_registers procedure can load, unload, or both. Additionally, one application of the procedure can load/unload multiple cascaded registers.

When the test procedure file explicitly calls the load_unload_registers procedure from either the test_setup or test_end procedures, the values to load or unload are passed to the procedure using the string of binary values (value hard-coded in the procedure application) or the register value variables.

shift Keyword

The load_unload_registers procedure uses the **shift** keyword to define a block of events that result in one shift operation. This is similar to the IEEE STIL syntax where the shift keyword defines a shift block within a load or unload procedure. It is analogous to embedding the shift procedure into the load_unload procedure.

Apply Statement Extension

The **apply** statement in the procedure file syntax is extended in order to accept a statement that associates a set of string values or register value variables with a particular input pin, output pin, or alias. This pin or alias must then be used within the shift statement and have a “#” character associated with it. This character is a placeholder for the values that will be loaded or unloaded using the string of values passed to the procedure, or the internally generated values associated with the value name.

Test Procedure File Syntax

The following illustrates using the load_unload_registers procedure and shift keyword:

```
procedure load_unload_registers procedure_name =
   ...
  [ cycle blocks ]
  shift =
    cycle blocks
  end ;
  [ cycle blocks ]
end ;
```

The load_unload_registers procedure must have a shift block defined, which has one or more cycles used to shift the data into the data input pin or the data out of the data output pin. The procedure can also optionally have cycles which precede or follow the shift block, to be used to put the circuit into shift mode or finish the shift mode when done, similar to how a load_unload procedure is used.

Event Statements

Within a shift block, at least one event statements in a load_unload_registers procedure must use the “#” character to denote where the shift data passed into the procedure is used.

The event statement must be either a “force” event or an “expect” event. An event statement with the “#” character can also occur in the cycles preceding or following the “shift” block in order to express pre-shifts and post-shifts for loading the register. This event statement has the following syntax:

```
<force | expect > pin_or_alias_name # ;
```

The apply statement which is currently used to call shift and sub_procedure procedures and also calls the load_unload_registers procedure. When calling these procedures, however, the number of times argument in the apply statement is replaced with one or more value assignments to the data in and/or data out pins.

```
apply procedure_name [ #times | shift_data_assignment
[ , shift_data_assignment ...] ] ;
```

A shift_data_assignment has the following syntax:

```
 identifier = < value_string | register_value_variable >
   [<value_string | register_value_variable>...]
```

where identifier is either a pin name or an alias name.

The identifier used in the shift_data_assignment must match an identifier used within the procedure in one of the event statements with the "#" character.

Register Value Strings

A register value string (value_string) is one of the following:

- A binary string of 0's, 1's or X's, where the length of the string determines the number of shifts to load the register.
- A string of a different radix as long as the Verilog syntax of identifying the radix and width are used, such as "32'h" for a 32 bit hexadecimal value.

If a register_value_variable is used in the shift_data_assignment, then a value computed by the tool for that register_value_variable (as bound within the dofile) or hard-coded in the dofile is loaded or unloaded at that time and the shift length is also provided by the tool. It is possible for more than one value_string or register_value_variable to be assigned to an identifier in one shift_data_assignment. In this case, the extra values are separated by spaces. This allows multiple shorter values to be shifted into one register group.

The typical usage is that each register_value_variable corresponds to one register being loaded, and the specification of multiple variables is used when those registers are cascaded and loaded/unload on after another.

Alias Names

It is possible to use an alias name in the procedure type for loading shift data, even if that alias name refers to multiple pins. If this is the case, the number of bits assigned by the "#" character for each shift is equal to the width of the alias being used. The length of the value string being passed to the procedure must be a multiple of the width of the alias.

Loading or unloading an alias can be used if performing parallel load/unload and no shift is required. For example, if each MISR bit is connected to a separate primary output. Even if no shifting is required, the functionality is still useful to bind the expected values to the signature computed by the tool.

If multiple shift_data_assignments are passed to a procedure, then all of them must have the same shift length such that each pin being loaded/unloaded requires the same number of cycles to load/unload all the data. No padding is performed by the tool.

If a "measure" event is used in one of these procedures to unload a register, then these measure values can be compared in the final patterns and the Verilog testbench.

Dofile Modifications

You define register value variables using commands you issue to the tool interactively or in a dofile. The value variables are subsequently referenced in the procedure file. You use the following commands to perform these operations:

- add_register_value — See "Addition of a Register Value Variable"
- set_register_value — See "Definition of a Dynamic Register Value Variable"
- delete_register_value — See "Deletion of a Register Value Variable"
- report_register_value — See "Register Value Variable Reports"

Addition of a Register Value Variable

The add_register_value command defines the register value variables. You can define the register value variables as either Static Value Variables or Dynamic Value Variables.

You can only use this command in setup mode.

You must define the register value variables in the dofile *before* the variables are used in a test procedure file to load values into the registers.

Static Value Variables

You specify a static value variable (see "Static Variable") by stating a specific value string. This value variable then has this value string as a constant value.

You must specify this value in setup mode; the value is considered to be static by default, and the value is present when simulating procedures in DRC.

```
add_register_value value_name {value_string [-Radix {Binary | Decimal |
      Octal | hexadecimal} -Width integer]} optional_arguments
```

The *value_name* is a user-specified identifier, and the *value_string* is a state string in a particular radix. The default is binary radix.

If the –Radix switch is used to change this to a different radix, then a register width must also be specified using the –Width switch.

Dynamic Value Variables

You specify a dynamic value variable using the following variation of the command:

```
add_register_value value_name value_string -Dynamic -Width integer
      optional_arguments
```

The *value_string* is optional and defaults to all X bits if not specified.

You can subsequently enter the actual value once you have exited setup mode using the set_register_value command. If no value is specified the first time DRC is invoked after adding

a register value variable, it will be considered as dynamic in this and any future invocation of DRC. See "Definition of a Dynamic Register Value Variable" for more information.

Data Pin Inversions

When using either a static or dynamic variable, you can optionally specify inversions on the input and output data pins by using the -INput_pin_inversion and -OUtput_pin_inversion switches, respectively.

These are optional switches you use to specify that the data value in this variable should be inverted before it is loaded into the register through the load_unload_register procedure.

Definition of a Dynamic Register Value Variable

You can define a dynamic register value variable by using tool-specific switches and arguments to link the variable to a value computed by the tool. These variables are dynamic, and the width must be specified unless the width is known to the tool at the time DRC is run.

This method is used for defining a variable that has a tool-specific computed value that will be available when patterns are saved and needs to be loaded or unloaded by the test_setup or test_end procedures.

In this case, you use the add_register_value command with the -Width switch and omit the value while in setup mode. Once you have exited setup mode, you can use the set_register_value command to set the variable:

```
set_register_value value_name value_string [-RAdix {BInary | HEx | OCtal |
      Decimal}]
```

The name of the register value and its width are known prior to parsing the procedure file and running DRCs. The value will be all X bits for DRC.

This command can only be used for a register value that was defined without a value string, and the width of the value string must match the width specified in the add_register_value command.

If the value overflows the width specified, an error is issued.

By default, the bits extracted by force/measure "#" in the procedure are from MSB to LSB. If the value specified should be shifted in/out in the opposite order, with the LSB bits applied/measured first, use the "-LSB_shifted_first" optional switch.

Deletion of a Register Value Variable

The delete_register_value command deletes all or a specified register value variable. You can only use this command in setup mode.

Register Value Variable Reports

The report_register_value command provides a detailed report of the register value variables to stdout or, optionally, a file.

Serial Load and Unload DRC Rules

The following existing DRC rules are applied to the procedures and syntax in the section "Procedure Examples."

- P13 and P54
- P66
- W5

The P1 "syntax error" message will be used to catch many potential issues, such as specifying a "#times" value for applying a load_unload_registers procedure instead of the shift_data_assignment.

P13 and P54

The P13 and P54 rules are used to check the shift assignment statements when calling a load_unload_registers procedure.

If the shift assignment uses a value string that is not properly formatted or contains illegal characters for that radix, then a P13 is issued.

```
Error:  Invalid state value state string. (P13)
```

If the shift assignment references an undefined register value variable name, a P54 is issued.

```
Error:  Undefined identifier identifier_string referenced by signal_name.
(P54)
```

P66

The P66 rule is used to check for missing statements within a load_unload_registers procedure.

For example, if no Shift block is specified, or if an event statement using the "#' character is not present for each shift_assignment passed to the load_unload_registers procedure, then a P66 is issued.

```
Error:  Procedure procedure_name is missing required statement_string
statement. (P66)
```

The following examples illustrate the types of P66 DRC errors you could encounter.

Example 1

In the following example, the tool issues this error if there is no shift block within the load_unload registers procedure:

```
Error:  Procedure procedure_name is missing required shift statement.
(P66)
```

Example 2

In the following example, the tool issues this error if there are no events in the load_unload_register procedure that use the “#’ substitute character.

```
Error:  Procedure procedure_name is missing required substitute event
statement. (P66)
```

Example 3

In the following example, the tool issues this error if an apply statement that uses a load_unload_registers procedure has a shift data assignment that uses a signal that does not appear in the load_unload_registers procedure with the substitute character “#’.

```
Error:  Procedure procedure_name is missing event using shift assign
signal_name statement. (P66)
```

W5

The W5 DRC error is used to flag any extra events or statements in a load_unload_registers procedure, or any events that are not legal.

For example, if an event type other than Force or Expect is used with the “#’ substitute character, then a W05 rule will be issued. If the load_unload_registers procedure contains more than one shift block, this rule will be issued. If there is a Force or Expect statement using a “#’ character for a signal name that is not being passed to the procedure as a shift assignment, this rule will be issued. The W05 rule is used when shift assignments for a particular Apply statement do not all have the same length.

```
Error: Procedure procedure_name has an illegal event statement or event
order (event_statement_string) (W5)
```

Example 1

The following error is issued if an event in the load_unload_register procedure uses the “#’ substitute character, but no shift data for this signal is passed into the procedure when it is called “extra shift block”:

```
<force | measure> signal_name # without matching shift assign data
```

Example 2

The following error is issued if there is more than one shift block in the load_unload_registers procedure.

```
“extra shift block”
```

Example 3

The following error is issued if more than one event of the same type in the shift block uses the same signal name with the “#’ substitute character.

```
“too many events using shift assign signal_name”
```

Example 4

The following error is issued for an apply statement that uses a load_unload_registers procedure but has no shift data assignments in the apply statement.

```
"apply with no shift data assignment"
```

Example 5

The following error is issued for an apply statement that uses a load_unload_registers procedure and has more than one shift assignment, however, the target of the shift assignments are aliases and they are not the same width.

```
"unmatched shift assign signal width"
```

Example 6

This is issued for an apply statement that uses a load_unload_registers procedure and has more than one shift assignment and the assignments have different shift lengths.

```
"unmatched shift lengths"
```

Procedure Examples

The definition of the load_unload_registers procedure would look as follows. Notice how this procedure uses the "shift =" statement and the "force tdi #" statement to denote the shifting and where the string of data is applied, one bit at a time.

```
procedure load_unload_registers load_prpg1 =
  timeplate tp1 ;
     cycle =
       force prpg_select 1 ;
       ...  // setup tap controller to load prpg
     end;
     shift =
       cycle =
         force tdi # ;
         pulse tck ;
       end;
     end;
   end;
```

This procedure could then be applied in the test_setup procedure to initialize the PRPG, specifying the string of bits to apply to "tdi" during shifting.

```
procedure test_setup =
  timeplate tp1 ;
     cycle =
       force clk1 0 ;
       force clk2 1 ;
       force tck 0 ;
       ...
     end;
     apply load_prpg1 tdi = 000000000000000000000001 ;
     cycle =
```

```
        ...
      end;
    end;
```

For loading a register with the shift length during test_setup, using the values computed by the tool, the procedure definition would look the same, but how the procedure is called is slightly different. The following example both loads and unloads the register, as this same procedure could be used in a test_end procedure to unload the shift length value. The dofile for this example contains the following command:

add_register_value length_val -shift_length -width 16

The procedure file would contain the following:

```
procedure load_unload_registers load_unload_length =
  timeplate tp1 ;
  cycle =
    force clk1 0 ;
    ...
  end;
  shift =
    cycle =
      force tdi # ;
      expect tdo # ;
      pulse tck ;
    end;
  end;
end;
procedure test_setup =
  timeplate tp1 ;
  cycle =
    ...
  end;
  apply load_unload_length tdi = length_val, tdo = XXXXXXXXXXXXXXXX;
  cycle =
    ...
  end;
end;
```

This next example uses an alias to group three tdi signals into one alias, and also adds a post shift cycle to the load_unload_registers procedure. When this procedure is called, the data passed to it will be consumed three bits at a time, with each bit being shifted into tdi1, tdi2, and tdi3 in order. The total number of shifts applied in the “shift” block will be the total length of the value string divided by three, and then minus one for the post shift. Each shift will consumes three bits, and the final cycle adds a post shift which will assign the last three bits. The length of the value string being passed to this procedure must be a multiple of three.

```
alias TDI_GRP = tdi1, tdi2, tdi3 ;
procedure load_unload_registers load_reg1 =
  timeplate tp1 ;
     cycle =
       force clk1 0 ;
       ...
     end;
```

```
    shift =
      cycle =
        force TDI_GRP # ;
        pulse tck ;
      end;
    end;
    cycle =
      force TDI_GRP # ;
      pulse tck;
    end;
  end;
```

This final example shows how to use a dofile commands to setup a user-defined register value that will be used to store total number of scan patterns when the final patterns are saved.

```
add_register_value scan_pat_count -pattern_count scan_test -width 24
```

Notes About Using the stil2mgc Tool

You can use the **stil2mgc** tool to create a dofile and procedure file. The **stil2mgc** tool reads a STIL Procedure File (SPF) and then creates a dofile and test procedure file.

The dofile defines clocks, scan chains, scan groups, and pin constraints. The test procedure file contains a timeplate and the following standard scan procedures: test_setup, load_unload, and shift. For more information about this tool, refer to the stil2mgc command description in the *Tessent Shell Reference Manual*.

Extracting Strobe Timing Information from STIL (SPF)

In the STIL WaveformTable, strobe window and edge strobe are indicated by which event characters are used for the measure timing. Using the ‘H’ and ‘L’ characters indicates that the measure is an edge strobe, while using the ‘h’ and ‘l’ character indicates a window strobe with the length of the strobe window being determined by the following X event in the event list for that WaveformCharacter.

If the window strobe events (‘h’ or ‘l’) are used for a measure event, then the strobe window is calculated taking into account the strobe window indicated by these events. The shortest strobe window calculated is the one used for the procedure file.

If “-edge_strobe_processing on” is specified to the tool, and the edge strobe events (‘H’ or ‘L’) are used for a measure event, then the strobe window is set to 0 in the procedure file to indicate edge strobe timing.

If the “-edge_strobe_processing” option is not used or is set to off, and edge strobe events (‘H’ and ‘L’) are used for a measure event, then the existing behavior is maintained where the strobe window is calculated based on the next force event or the period of the Waveform table. The strobe window is not set to 0.

Handling the STIL ClockStructures block

When parsing STIL Procedure Files (SPF), stil2mgc recognizes the Synopsys-defined ClockStructures block and uses this information to create clock_control definitions. The Latency statement within the ClockStructures block is used to control the "set_external_capture_options" statement in the generated dofile.

Test Procedure File Commands and Output Formats

The test procedure file is supported by a set of commands and a set of output formats.

Test Procedure File Tool Commands

The following table provides a summary of the tool commands that support the use of test procedure files. For a detailed description of each command, refer to the corresponding command reference page in the "Command Dictionary" section of the *Tessent Shell Reference Manual*.

Table 6-3. Procedure File Tool Command Summary

Command	Description
add_scan_groups	Adds a scan group using the scan procedures in the named procedure file.
read_procfile	Reads a new procedure file in non-setup mode. Merges new procedure and timing data with existing data loaded from previous procedure files.
write_procfile	Writes out existing procedure and timing data as the named procedure file.
write_patterns	Loads a cycle before saving patterns and merges the new data with the existing data.
report_procedures	Reports (displays) a named procedure to the screen. The -All switch displays all procedures to the screen.
report_timeplate	Reports (displays) a named timeplate to the screen. The -All switch displays all timeplates to the screen.

Test Procedure File Output Formats

The test procedure file format supports the following output formats:

- Fujitsu TDL
- Mitsubishi TDL
- STIL
- TI TDL

- WGL
- TSTL2
- VERILOG

The -PROcfile switch causes the write_patterns command to get its timing information from the procedure file. For more information, refer to the write_patterns command in the *Tessent Shell Reference Manual*.

Appendix A
Using the Tessent Tcl Interface

The Tessent Shell tool provides a Tcl-based command interface. The following sections explains how to use the Tcl interface:

General Tcl Guidelines in Tessent Shell

The Tcl interface provides basic Tcl capabilities within Tessent Shell, either at the command line or in dofiles. The Tcl interface supports Tcl constructs, such as variables, command substitution, flow control, and procedures. You can embed Tcl constructs in tool commands and embed tool commands within Tcl constructs the same as any Tcl command.

If Tcl procedures are available in separate files, you can source these files from within the tool or dofiles. You can also place Tcl procedures in a *.tool_startup* file so that they are available for use. Tcl file input/output is also supported.

You should be aware of the following guidelines and behaviors when using Tcl in Tessent Shell:

- You can use Tcl variables interchangeably with legacy Tessent tool variables and environment variables.
- You should use Tcl syntax for setting and referencing variables, including using $env(ENVARNAME) for accessing environment variables.

  For example, if the value of environment variable “foo” equals “mode1” ($foo = mode1), you can compare this value to a Tcl variable value using the following syntax:

  ```
  set bar mode2
  if {$env(foo) == $bar} {puts Match} else {puts "No match"}
  ```

  If you want the output of any tool command returned as a Tcl result, you must use the return_output command. The following example uses return_output in a dofile:

  ```
  set resultA [return_output {report_gate /gate3} -tee ]
  // /gate3  nor02
  //      A0        I  /gate2/Y
  //      A1        I  /ff2/Q
  //      Y           O  /ff5/D  ff4/D  ff3/D /gate6b/E
  puts $resultA
  //  /gate3  nor02
  //      A0        I  /gate2/Y
  //      A1        I  /ff2/Q
  //      Y         O  /ff5/D  ff4/D  ff3/D /gate6b/E
  ```

Note

Use Tcl namespaces to avoid creating procedures that conflict with existing tool or Tcl commands.

- When processing comments within a dofile, the tool does not write comments preceded with “//” characters to the transcript. However, the tool does write comments that are preceded with a pound sign (#) (the Tcl comment delimiter) to the transcript.
- If a variable can contain a command string or be empty, attempting to execute by referencing $variable results in an error if the variable is empty.

- When a tool command error occurs, the command interpreter prints the error message and does not return anything.
- You can use the catch_output command to issue a specified tool command line and prevent command errors from aborting an enclosing dofile or Tcl proc. The catch_output command can optionally capture the output of a command or the returned result.
- When a command error occurs nested inside a Tcl construct or Tcl proc, you can obtain additional information about the error by issuing the following command:

  > **set errorInfo**

  This command prints the value of the $errorInfo Tcl variable, which may contain a traceback of the nested Tcl calls so that you can determine the root cause of the error.

Difference Between the Dofile Command and the Tcl Source Command

You normally use the tool's dofile command to execute a file of tool commands. The Tcl "source" command also executes a file of commands, but you should only use it to load Tcl procs, set Tcl variables, or do other strictly Tcl commands.

You should use the dofile command to execute a command file containing tool commands for the following reasons:

- The dofile command is affected by the set_dofile_abort command which provides you with the ability to specify whether the tool aborts when an error condition is detected. The Tcl "source" command is not affected by the set_dofile_abort command.
- The dofile command transcripts the commands to the shell and logfile. The Tcl "source" command always stops execution if any command in the file encounters an error.

Example 1

In this example, the add_scan_chains command defines every scan chain in the design. This command is sometimes used hundreds of times and makes the dofile very long. Using Tcl, you can shorten the dofile substantially by using a loop construct as shown here:

```
for {set xx 1} {$xx < 257} {incr xx} {
      add_scan_chains int chain$xx group1 /top/edt_si$xx /top/edt_so$xx
}
```

For the values of xx from 1 up to 256, the tool executes a separate add_scan_chains command for each value. The transcript and logfile will contain all 256 add_scan_chains commands (preceded with "// subcommand: ").

Example 2

The following example controls the flow of creating test patterns and uses a variable to execute different commands based on the variable's value:

```
if {$mode == stuck} {
    set_fault_type stuck
    . . .
} elseif {$mode == transition} {
    set_fault_type transition -no_shift_launch
    set_pattern_type -sequential 2
    . . .
}
```

Example 3

This example places the output from a report command in a variable for subsequent processing:

> **set chain_report [return_output {report_scan_chain}]**

> **puts $chain_report**

chain = chain1 group = grp1 input = /scan_in1 output = /scan_out1 length = unknown

. . .

chain = chain256 group = grp1 input = /scan_in256 output = /scan_out256 length = unknown

Guidelines for Modifying Existing Dofiles for Use with Tcl

When using an existing dofile with the Tcl interface, you should evaluate the dofile for issues that could cause incorrect evaluation by the Tessent Tcl interpreter. The following table provides guidance for correcting common dofile issues.

The most common issues you can run across with an existing dofile is accounting for Tcl special characters. For more information, refer to Table A-2 on page 246, which provides a list of typically-used Tcl special characters.

Table A-1. Common Dofile Issues and Solutions

Dofile Issue	Solution
Stopping dofile execution at a specific point	Use the native Tcl "error" command to stop execution of a dofile at any point. The "error" command requires a message string which the tool outputs. But to avoid any message you can use an empty string: **error ""**

Table A-1. Common Dofile Issues and Solutions

Dofile Issue	Solution
Escaping Tcl special characters	Use the following techniques to escape Tcl special characters: • Double quotes (" ") group tokens but allow $variable and [command] evaluations. • Braces ({}) also group tokens and disable $variable and [command] evaluations. • Brackets ([ ]) implement command substitution and are used to nest or embed commands. • A backslash (\) escapes the next character. Use this to tame a Tcl special character such as $, [, {, or ".
Using dollar signs in pathnames	A dollar sign ($) specifies variable substitution. In some netlists, pathnames (for example, *foo/pin$p7*) can contain the dollar sign. When using pathnames with dollar signs, enclose the pathname with braces ({ }) to prevent the tool from substituting the value as shown in the following example: report_gates {foo/pin$p7}
Escaping quotation marks	In Tcl, quotation marks (" ") instruct the Tcl interpreter to treat the enclosed words as a single argument. For example: **puts " Hello World "** If embedded quotes are required, you must use backslashes (\ ) to escape the embedded double quotes. For example: **puts " Hello \"World\" "** Otherwise, the tool issues an error message.
Using brackets	Brackets ([ ]) implement command substitution and are used to nest or embed commands. A command and its arguments enclosed in square brackets is evaluated and its result inserted in place in the enclosing command. For example: **echo [return_ouput {report_edt_connections -all_blocks}] > edt_connections.txt**
Optional single quotes	Optional single quotes (' ') are no longer valid for Tessent commands. For example, the following produces an error: **set_design_sources '-v MODB.v -v MODC.v'** Correct this by using no quotes, double quotes, or braces: **set_design_sources -v MODB.v -v MODC.v** **set_design_sources "-v MODB.v -v MODC.v"** **set_design_sources {-v MODB.v -v MODC.v}**

Table A-1. Common Dofile Issues and Solutions

Dofile Issue	Solution

Environment variables	You should use Tcl syntax for setting and referencing variables, including using $env(ENVARNAME) for accessing environment variables. For example, if the value of environment variable “foo” equals “mode1” ($foo = mode1), you can compare this value to a Tcl variable value using the following syntax:
	`set bar mode2`
	`if {$env(foo) == $bar}`
	`  {puts "Match"}`
	`else`
	`  {puts "No match"}`

Special Tcl Characters

In Tcl scripts, you often see characters used for special purposes. For a complete list of special characters, you should consult a Tcl resource.

Table A-2 lists and describes the more common special characters you can encounter when reading the examples in this manual. See the additional resources described in “Tcl Resources” on page 249.

Table A-2. Common Tcl Characters

Character	Description
;	The semicolon terminates the previous command, allowing you to place more than one command on the same line.
\	Used at the end of a line, the backslash continues a command on the following line.
\$	The backslash with other special characters, like a dollar sign, instructs the Tcl interpreter to treat the character literally.
\n	The backslash with the letter “n” instructs the Tcl interpreter to create a new line.
$	The dollar sign in front of a variable name instructs the Tcl interpreter to access the value stored in the variable.
[ ]	Square brackets group a command and its arguments, instructing the Tcl interpreter to treat everything within the brackets as a single syntactical object. You use square brackets to write nested commands. For example: **set chain_report [report_scan_chains -subchains -verbose]**

Table A-2. Common Tcl Characters (cont.)

Character	Description
{ }	Curly braces instruct the Tcl interpreter to treat the enclosed words as a single string. The Tcl interpreter accepts the string as is, without performing any variable substitution or evaluation. For example, to create a string that contains special characters such as $ or \: **set my_string {This book costs $25.98.}**
" "	Quotes instruct the Tcl interpreter to treat the enclosed words as a single string. However, when the Tcl interpreter encounters variables or commands within string in quotes, it evaluates the variables and commands to generate a string. For example, to create a string that displays a final cost calculated by adding two numbers: **set my_string “This book costs \$[expr $price + $tax]”**

Table A-2. Common Tcl Characters (cont.)

Character	Description

#	The pound sign (#) indicates a comment and directs the Tcl compiler to not evaluate the rest of the line. When using the pound sign, you must use it where a command starts, and at the beginning of a command, not within a command.
	
	• **Tricky Point 1:** Evaluate does not equal parse. Despite the pound sign, the comment below gives an error because Tcl detects an open lexical clause.
	
	`# if (some condition) {`
	`if { new text condition } {`
	`...`
	`}`
	
	• **Tricky Point 2:** The apparent beginning of a line is not always the beginning of a command:
	
	`# This is a comment`
	
	In the following code snippet, the line beginning with "# -type" is not a comment, because the line right above it has a line continuation character (\). In fact, the "#" confuses the Tcl interpreter, resulting in an error when it attempts to create the tk_messageBox.
	
	`tk_messageBox -message "The diagnosis report was successfully written."\`
	`# -type ok`
	
	`# and this is also a comment. This one will span \`
	`multiple lines, even without the # at the beginning \`
	`of the second and third lines.`
	
	In general, it is good practice to begin all comment lines with a #.
	
	• **Tricky Point 3:** The beginning of a command is not always at the beginning of a line:
	
	Usually, you begin new commands at the beginning of a line. That is, the first non-space character is the first character of the command name. However, you can combine multiple commands into one line using the semicolon ";" to designate the end of the previous command:
	
	`set myname "John Doe" ; set this_string "next command"`
	`set yourname "Ted Smith" ; # this is a comment`

Using Custom Tcl Packages in Tessent Shell

Tessent Shell supports several standard techniques for adding your own Tcl packages with the Tcl “package require” command, which you can issue from Tessent Shell.

You can use any of the following ways to specify directory locations of Tcl packages. Any directory that contains a Tcl package must also contain a *pkgIndex.tcl* file within its hierarchy. The following methods are listed in order of the precedence in effect if there is more than one package with the same name:

- Default location for Tessent Shell Tcl packages. You can place your package underneath a directory named *tessent_plugin/packages* that is located at the top of your Tessent install tree. When Tessent Shell finds a package in this directory upon invocation, it appends the directory path to *tessent_plugin/packages* to the auto_path Tcl global variable, if it exists.
- TESSENT_PLUGIN_PATH environment variable. You can set this variable to a colon-separated list of directory paths that contain Tcl packages in a *packages* subdirectory. When Tessent Shell finds a package, it appends the directory path to *packages* to the auto_path Tcl global variable, if it exists.
- auto_path Tcl global variable. You can issue the following command in Tessent Shell to specify a package location:

  > **lappend auto_path** pathToTclPackageDir

- TCLLIBPATH environment variable. You can set this variable to a space-separated list of directory paths that contain Tcl packages. Note that a *packages* subdirectory is not required under any of the paths.

Note

Tessent Shell ignores the TCL_LIBRARY environment variable.

Tcl Resources

The following website is a place to start in your search for the reference material that works best for you. It is not an endorsement of any book or website.

http://www.tcl.tk/

Appendix B
Transitioning from the Classic Point Tools

This appendix provides information to help you make the optional transition from the classic Tessent point tools to Tessent Shell. The point tool application commands are forward-compatible, so your dofiles work properly in Tessent Shell with only minor modifications. The main differences involved in using Tessent Shell is that you have to set the context before doing anything else, and then you must load the netlist and library.

Note that the classic Tessent point tools are available in the current release, so all of your existing dofiles and startup scripts continue to work as before. However, Mentor Graphics recommends that you start planning your transition now because each release of Tessent Shell contains additional features not available in the classic point tools, and future releases of Tessent Shell will continue to add new features that are not available in the classic point tools.

For information about specific point tools refer to the following sections:

Transitioning from the Classic FastScan Point Tool

Beginning with the v2012.3 release, you can optionally transition from the classic FastScan point tool to Tessent Shell. Tessent Shell provides all of the commands available in the classic FastScan point tools (invoked with the “fastscan” shell command) plus additional features.

Note that the classic Tessent point tools are available in the current release, so all of your existing dofiles and startup scripts continue to work as before. However, Mentor Graphics recommends that you start planning your transition now because each release of Tessent Shell contains additional features not available in the classic point tools, and future releases of Tessent Shell will continue to add new features that are not available in the classic point tools.

If you are already familiar with classic FastScan, the main differences involved in using Tessent Shell is that you have to set the context before doing anything else, and then you must load the Verilog netlist and library:

```
set_context patterns -scan
read_verilog netlist_name
read_cell_library library_name
set_current_design
```

Another difference with using Tessent Shell is that several system modes used with classic FastScan (atpg, good, fault) have been replaced with a single system mode (called analysis). This means that "set_system_mode atpg" invocations (as well as transitions to good or fault modes) are replaced by "set_system_mode analysis." To facilitate your transition to Tessent Shell, the set_system_mode command continues to accept the atpg/good/fault arguments, but the tool actually switches to analysis mode.

Also, it is highly recommended for any simulation to replace the old "set_pattern_source external" and "run" commands with the new read_patterns and simulate_patterns commands. Unlike the "run" command, which has slightly different behavior in the good/fault/atpg system modes, the simulate_patterns command allows you to explicitly control which pattern set to simulate and which patterns (if any) to store in the internal pattern set.

Transitioning from the Classic TestKompress Point Tool for IP Creation and Test Pattern Generation

Beginning with the v2012.3 release, you can optionally transition from the classic TestKompress point tool to Tessent Shell. Tessent Shell provides all of the commands available in the classic point tool (invoked with the "testkompress" shell command) for EDT IP creation and test pattern generation plus additional features.

Note that the classic Tessent point tools are available in the current release, so all of your existing dofiles and startup scripts continue to work as before. However, Mentor Graphics recommends that you start planning your transition now because each release of Tessent Shell contains additional features not available in the classic point tools, and future releases of Tessent Shell will continue to add new features that are not available in the classic point tools.

The following list describes how Tessent Shell is different from the classic TestKompress tool for creating EDT IP:

- Before issuing any commands, set the context to "dft -edt":

  ```
  set_context dft -edt
  ```

  Once you have set the context, all commands in setup mode of classic TestKompress are available in the setup mode of Tessent Shell. All commands available in atpg mode of

classic TestKompress (IP creation phase) are available in analysis mode of Tessent Shell.

- Use the existing write_edt_files command to generate all EDT files, including RTL, dofiles, the test procedure file, and synthesis and timing verification scripts. You can issue this command only when in analysis mode. Note that the write_edt_files command does not write out the EDT IP netlist, so you must do this with the write_design command.
- For internal IP location flow-based EDT insertion with classic TestKompress, the tool writes out all EDT files, inserts EDT IP into the design, and automatically changes to insertion mode. You then have the ability to further edit the netlist, but you must explicitly save changes with the write_design command.

  Unlike classic TestKompress, Tessent Shell does not automatically exit after executing a write_edt_files command, so you must explicitly exit when ready. This is so that after IP insertion into the design, you can perform further design editing before writing out the design, or configure how to write out the design using the write_design command. Also note that Tessent Shell does not write out the netlist as part of the write_edt_files command, so you must write out the netlist using the write_design command.
- The classic TestKompress command write_edt_files has a switch named "-insertion tk." In Tessent Shell, this switch is renamed "-insertion ts."

The following list describes how Tessent Shell is different from the classic TestKompress tool for generating patterns:

- Before generating compressed patterns, set the context to "patterns -scan":

  set_context patterns -scan

  Once you've set the context, all commands in the setup mode of classic TestKompress are available in setup mode of Tessent Shell. And all commands available in the atpg mode of classic TestKompress (Test Pattern Generation phase) are available in the analysis mode of Tessent Shell.

Transitioning from the Classic DFTAdvisor Tool

Beginning with the v2012.3 release, you can optionally transition from the classic DFTAdvisor point tool to Tessent Shell. Tessent Shell and Tessent Scan provide all of the commands available in the classic DFTAdvisor tool (invoked with the "dftadvisor" shell command), plus additional features.

Note that the classic Tessent point tools are available in the current release, so all of your existing dofiles and startup scripts continue to work as before. However, Mentor Graphics recommends that you start planning your transition now because each release of Tessent Shell

contains additional features not available in the classic point tools, and future releases of Tessent Shell will continue to add new features that are not available in the classic point tools.

The following lists the differences between the classic DFTAdvisor tool and Tessent Scan:

- You no longer need to use the “run” command. Analysis that was previously done using the “run” command is now done automatically after DRC or as part of the analyze_wrapper_cells and insert_test_logic commands.
- After issuing insert_test_logic in analysis mode, the tool updates the design and then changes to insertion mode. This allows you to optionally perform further design editing before writing out the netlist with the write_design command.
- There is no longer a system mode called Dft. Use analysis mode instead.
- Before you issue any of the classic DFTAdvisor commands, you must first set the context:

  set_context dft -scan

Transitioning from the Classic Diagnosis Tool

Beginning with the v2012.3 release, you can optionally transition from the classic Diagnosis point tool to Tessent Shell. Tessent Shell provides all of the commands available in the classic Diagnosis tool (invoked with the “tessent -diagnosis” shell command), plus additional features.

Note that the classic Tessent point tools is available in the current release, so all of your existing dofiles and startup scripts continue to work as before. However, Mentor Graphics recommends that you start planning your transition now because each release of Tessent Shell contains additional features not available in the classic point tools, and future releases of Tessent Shell will continue to add new features that are not available in the classic point tools.

After invoking Tessent Shell with the “tessent -shell” command, at a minimum, you must perform the following operations in sequence to enable scan diagnosis in Tessent Shell:

1. Set the context before doing anything else:

   set_context patterns -scan_diagnosis

2. Read the flat model of your design:

   read_flat_model *flat_model_name*

   Previously, you specified the flattened design with the “tessent -diagnosis” command. Now you must use the read_flat_model command.

Appendix C
Getting Help

There are several ways to get help when setting up and using Tessent software tools. Depending on your need, help is available from documentation, online command help, and Mentor Graphics Support.

Documentation

The Tessent software tree includes a complete set of documentation and help files in PDF format. Although you can view this documentation with any PDF reader, if you are viewing documentation on a Linux file server, you must use only Adobe® Reader® versions 8 or 9, and you must set one of these versions as the default using the MGC_PDF_READER variable in your *mgc_doc_options.ini* file.

For more information, refer to "Specifying Documentation System Defaults" in the *Managing Mentor Graphics Tessent Software* manual.

You can download a free copy of the latest Adobe Reader from this location:

http://get.adobe.com/reader

You can access the documentation in the following ways:

- **Shell Command** — On Linux platforms, enter **mgcdocs** at the shell prompt or invoke a Tessent tool with the -Manual invocation switch. This option is available only with Tessent Shell and the following classic point tools: Tessent FastScan, Tessent TestKompress, Tessent Diagnosis, and DFTAdvisor.
- **File System** — Access the Tessent bookcase directly from your file system, without invoking a Tessent tool. From your product installation, invoke Adobe Reader on the following file:

  ```
  $MGC_DFT/doc/pdfdocs/_bk_tessent.pdf
  ```

- **Application Online Help** — You can get contextual online help within most Tessent tools by using the "help -manual" tool command:

  ```
  > help dofile -manual
  ```

  This command opens the appropriate reference manual at the "dofile" command description.

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, refer to this page:

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

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 (each an "Order"), 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 those documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order or presented in any electronic portal or automated order management system, whether or not required to be electronically accepted, will not be effective unless agreed in writing and physically signed 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. 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. Mentor Graphics may choose, at its sole discretion, not to release Beta Code 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' trade secret and 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 Products or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, 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/supportterms.

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. 140201, Part No. 258976