SIEMENS EDA

Tessent™ IJTAG User's Manual

Software Version 2024.2
Document Revision 34

SIEMENS

About Siemens Digital Industries Software

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

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

Revision History ISO-26262

Revision	Changes	Status/Date
34	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Jun 2024
33	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Mar 2024
32	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Dec 2023
31	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Aug 2023

Author: In-house procedures and working practices require multiple authors for documents. All associated authors for each topic within this document are tracked within the Siemens documentation source. For specific topic authors, contact Siemens Digital Industries Software documentation department.

Revision History: Released documents include a revision history of up to four revisions. For earlier revision history, refer to earlier releases of documentation on Support Center.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Table of Contents

Third-Party Information

List of Figures

List of Tables

Chapter 1
Introduction to Tessent IJTAG

Tessent IJTAG is Siemens EDA implementation of the IEEE 1687-2014 (IJTAG) standard.

It includes the following primary aspects:

- **Hardware Rules** — For IEEE 1687 instruments including port functions, timing, and connection rules.

- **Instrument Connectivity Language (ICL)** — Describes isolated nodes and partial or complete networks. This enables retargeting pin/register read/writes to scan commands.

- **Procedural Description Language (PDL)** — Describes instrument usage at a given level and facilitates automatic retargeting to any higher level.

Figure 1-1 illustrates an example of a high-level IJTAG implementation.

Figure 1-1. IJTAG High-Level Architecture
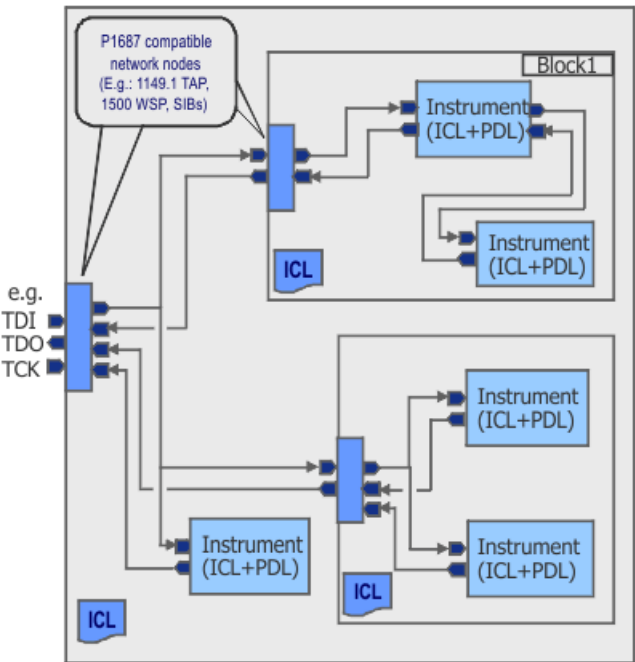
Tessent IJTAG Flow

The Tessent IJTAG usage flow and the constituent phases that make up the flow are described in this section.

This flow is illustrated in Figure 1-2. It assumes that there is an ICL description for each of the instruments as well as for the interconnect network of the instruments. Tessent IJTAG can compute the interconnect ICL from the Verilog design description. Please see Chapter 4, "ICL Extraction."

Figure 1-2. Tessent IJTAG Flow
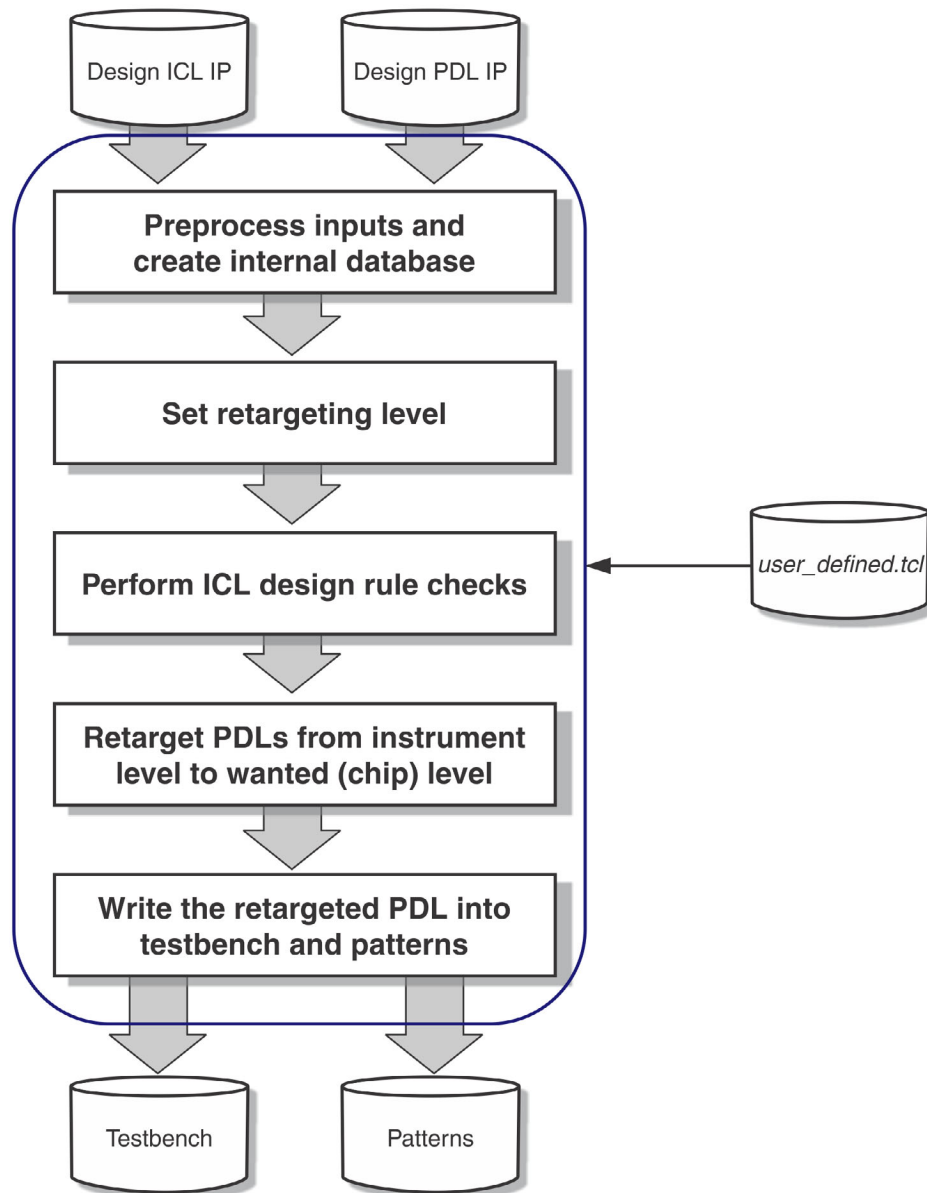
Table 1-1 provides a high-level overview of the flow and detailed description of each step of the flow.

Table 1-1. Tessent IJTAG Flow

Flow Step	Description
Tessent Shell and Tessent IJTAG	Tessent Shell is the platform you use to perform the Tessent IJTAG operations.
	Tessent Shell is a Tcl shell environment and design data model that provides a unified Tcl command set with data model interaction, including ICL introspection.
	Refer to the *Tessent Shell User's Manual* and *Tessent Shell Reference Manual* for complete information.
IJTAG Flow Inputs	In the initial phase of the flow, you must set the IJTAG context and subsequently read in the PDL and ICL descriptions of your design using native Tessent Shell commands.
	The tool loads this information into the internal ICL database. See "Read ICL Files" and "Read PDL Files."
Pattern Retargeting Level	After Tessent Shell has read in the ICL descriptions, you have access to IJTAG-specific commands such as iProc, iCall, iNote, iRead, iApply, and so on.
	Before proceeding, you must specify the ICL hierarchy level to which the PDL commands should be retargeted. See "Set the Retargeting Level."
Design Rule Checking	Tessent IJTAG automatically performs ICL design rule checking on the set ICL hierarchy level, down to the instrument level.
	Every instrument is checked for consistency between its objects and ports. You must correct any DRC violations before proceeding to the next phases of the flow—refer to "Design Rule Checks".
ICL Introspection	Tessent IJTAG provides a robust Tcl-based command set to perform ICL introspections for retrieving and reporting information from the ICL, PDL, and pattern sets.
	For more information, see "IJTAG Introspection."
PDL Command Retargeting	Tessent IJTAG performs automatic PDL retargeting based on the ICL hierarchy level you specified.
	The tool can retarget the PDL commands from instrument level up to chip level. See "A Typical PDL Retargeting Flow."
Patterns and Testbench	The final phase of the Tessent IJTAG flow is writing out the retargeted PDL, testbench, and patterns.
	For more information, see "Write PDL, Pattern, and Testbench Files."

Table 1-1. Tessent IJTAG Flow (cont.)

Flow Step	Description
ICL Extraction	An optional step of the Tessent IJTAG flow is the extraction of interconnection information of the various IJTAG building blocks from the flat design netlist. See "ICL Extraction" for complete information.

ICL and PDL Limitations

The IEEE 1687-2014 standard describes some ICL and PDL features that Tessent IJTAG does not yet support.

The unsupported ICL features are:

- ICL NameSpace and UseNameSpace. ICL modules can only be placed into the global ICL namespace. Each ICL module name must be unique at this global level.

- Call back data registers. The ReadCallBack and WriteCallBack properties cannot be used within a DataRegister definition.

- The AccessTogether property of the Alias construct.

- The AllowBroadcastingOnScanInterface property of the Instance construct.

- Backslashes in ICL strings are interpreted as escaping indicators only if the next character is a backslash or double quotation marks. If the next character is something else, the backslash is interpreted as an ordinary character of the string.

The unsupported PDL features are:

- PDL level-1 commands are meaningless when generating traditional manufacturing pattern files. Those commands are only relevant in an interactive silicon debug session context.

- The -together option of the iApply command.

- The -LastReadValue and -LastMiscompareValue options of the iState command.

- The iScan command does not support the following:

  o ICL black boxes.

  o The -stable option.

  o The use of ScanInterfaces on a module other than the current design.

  o The use of iScan in an iProc or iTopProc that is subject to iMerge.

License Usage/Requirements

Stand-alone verification of non-Tessent instruments requires an IJTAG license regardless of where you use the instruments. With the IJTAG license, you can create ICL and PDL for the instrument and validate it using commands such as iWrite, iRead, and iApply. With these ICL and PDL commands, you can generate testbenches and simulate those testbenches with the Verilog model of the instrument.

If you use at least one Tessent signed instrument (MemoryBIST, LogicBIST, TestKompress, or MissionMode) at your current design or at any lower hierarchical level, then any one of those respective licenses is sufficient to create an ICL network, including connecting your non-Tessent instruments. In all other cases, you need either an IJTAG or a MissionMode license. Similarly, you can use any license of IJTAG, MemoryBIST, LogicBIST, TestKompress, or MissionMode, to write into any non-Tessent instrument in the IJTAG network as long as there is at least one Tessent instrument connected to the IJTAG network. In all other cases, you need either an IJTAG or a MissionMode license.

In general, setting up or using any Tessent IP/instrument through the IJTAG network does not require an IJTAG license, but it may require an IJTAG license based on selected features. The Two-Pin Serial Port (TPSP) controller is one example. Usually, the tool uses the respective product license.

The following functionality requires an IJTAG license:

- DftSpecification processing of the Two-Pin Serial Port and pattern generation through that interface.

- Pattern generation with the FastIJTAG feature (specifying the "set_ijtag_retargeting_options -extra_control_setup_hold_cycles" or "set_ijtag_retargeting_options -extra_tms_setup_hold_cycles" command with a value greater than 0).

- ICL visualization.

- Pattern generation that includes reading data (iRead) from an unsigned third-party instrument.

- DftSpecification processing without a major product license (IJTAG, MemoryBIST, LogicBIST, TestKompress, or MissionMode). The tool prioritizes a major product license.

- ICL extraction without a major product license (IJTAG, MemoryBIST, LogicBIST, TestKompress, or MissionMode). The tool prioritizes a major product license.

Chapter 2
ICL and PDL Modeling

This chapter provides some insight into ICL and PDL without reproducing the entire IEEE 1687 document but provides enough information so you can understand the remainder of this manual..

Additionally, the "IJTAG Examples" chapter later in this document provides more examples. These example test cases are also a good source of information for learning about ICL, PDL, and how to use Tessent IJTAG.

This chapter includes the following topics:

ICL Instrument Description

The ICL code below is a complete description of an instrument, named *tdr1*. It has a scan in port named *si*, and a scan out port named *so*. There are four enable ports: *en*, *se*, *ce*, and *ue*. Finally, the test clock port is named *tck*. Observe that each port is defined through a keyword.

```
Module tdr1 {

    ScanInPort       si;
    ScanOutPort      so      {  Source R[0];   }
    SelectPort       en;
    ShiftEnPort      se;
    CaptureEnPort    ce;
    UpdateEnPort     ue;
    TCKPort          tck;

    ScanRegister R[7:0] {
       ScanInSource  si;
     }
}
```

With these keywords come a direction (input or output), but, more importantly, a semantic and timing. For example, the port name "se" is the shift enable port. For a human, these semantics enable a good level of understanding of the intention of the ports, their usage, and eventually, the operation of the instrument. From a tool point of view, these semantics enable very thorough design rule checks. For example, the scan in port *si* may not be driven by a data output port of another instrument. Instead, it must be connected to a scan output port.

IEEE 1687 has an explicit timing of events defined through the standard. For example, to be able to shift into the scan input port, the enable signal *se* has to be active high, the scan data has to arrive at *si*, and meet setup and hold time requirements around the rising edge of TCK. The speed of the clock and the exact timing of these events is up to the application tool and the implementation in hardware. There is no method of defining the period of a clock in ICL or PDL.

See IJTAG Network Performance Optimization for how to maximize the frequency of the IJTAG network test clock.

After the listing of the ports, this example shows an 8-bit scan register named *R*. The keywords in the attribute section of the scan register declaration are the ICL code between the brackets ({}). These keywords provide further information about the scan register. The example also declares that the scan data for *R* comes directly from the scan input port *si*. Again, the standard defines that the shift direction in *R* is from the left to the right. Accordingly, *si* is connected to R[7], and as shown in the attribute part of the ScanOutPort, so is connected to R[0]. If the register was declared as R[1:8], then it would shift from R[1] to R[8]. Understand that these connections are implicit per the definition of the standard. They do not need to be modeled and cannot be changed.

How to Build an ICL Netlist

This section describes how to build an ICL netlist.

Assume you want to create an ICL description as shown in Figure 2-1. You have ICL module definitions for all instruments, TDR1, TDR2, SIB, and TAP. For a clearer drawing, you color-code connections.

Figure 2-1. Example ICL Description
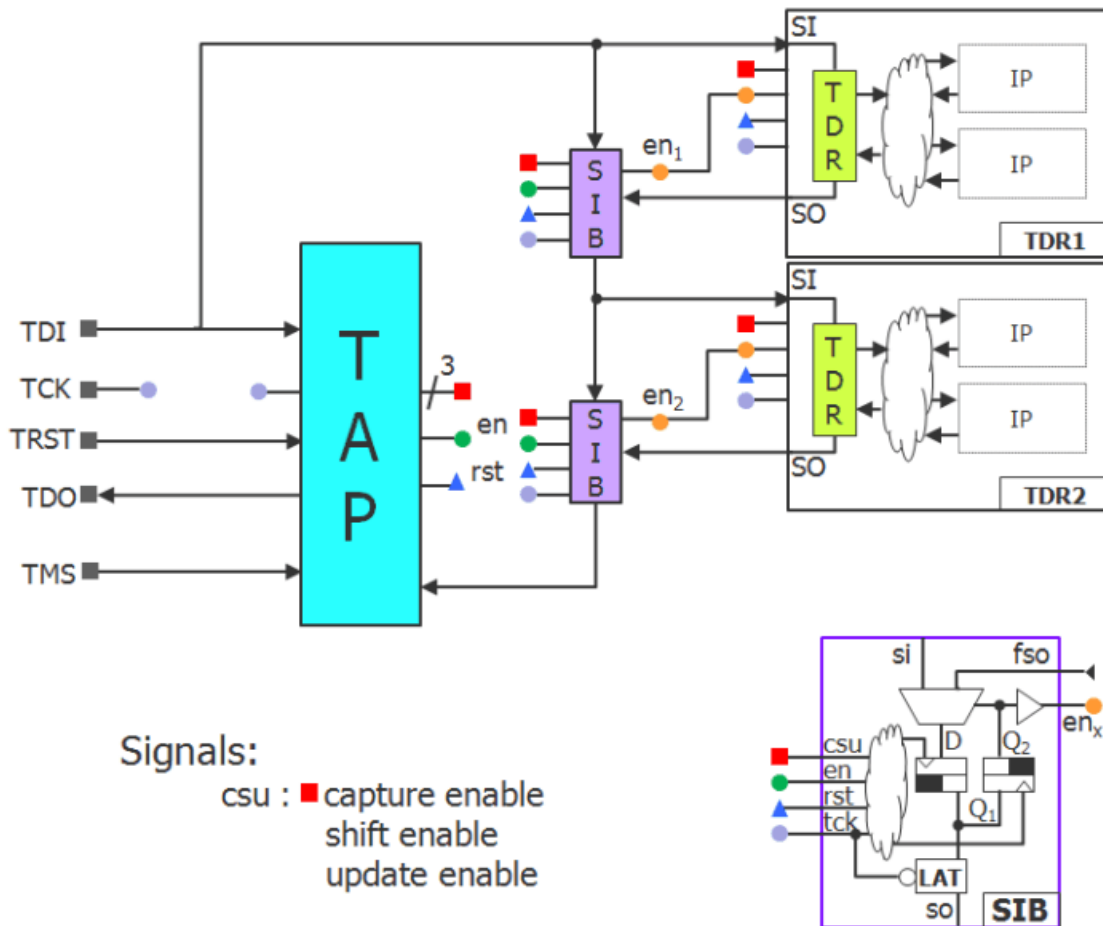
Next, instantiate each instrument, connect them, and create a top-level ICL description named
chip. The name of the top ICL module must match the module name of the corresponding
design module described in Verilog or VHDL. All ports found in the top ICL module must exist

in the design module though the design module is likely to have extra, non-IJTAG ports. The following example shows the top ICL module:

```
Module chip {

    TCKPort       tck;
    ScanInPort    tdi;
    ScanOutPort   tdo  { Source MyTap.tdo; }
    TMSPort       tms;
    TRSTPort      trst;

Instance MyTap Of tap1    { InputPort tck  = tck ;
                            InputPort tdi  = tdi ;
                            InputPort tms  = tms ;
                            InputPort trst = trst ;
                            InputPort fso  = MySib2.so ; // return scan path
                          }

Instance MySib1 Of sib1   { InputPort si   = tdi ;
                            InputPort se   = MyTap.se ; // shift enable
                            InputPort ce   = MyTap.ce ; // capture enable
                            InputPort ue   = MyTap.ue ; // update enable
                            InputPort en   = MyTap.tdrEn1 ;//select enable
                            InputPort tck  = tck ;      // test clock
                            InputPort fso  = MyTdr1.so ; //MyTdr1, scan out
                          }

Instance MyTdr1 Of tdr1   { InputPort si   = tdi ;
                            InputPort se   = MyTap.se ; // shift enable
                            InputPort ce   = MyTap.ce ; // capture enable
                            InputPort ue   = MyTap.ue ; // update enable
                            InputPort en   = MySib1.ten ; // select enable
                          }

Instance MySib2 Of sib1   { InputPort si   = MySib1.so ;
                            InputPort se   = MyTap.se ; // shift enable
                            InputPort ce   = MyTap.ce ; // capture enable
                            InputPort ue   = MyTap.ue ; // update enable
                            InputPort en   = MyTap.tdrEn1 ;//select enable
                            InputPort tck  = tck ;      // test clock
                            InputPort fso  = MyTdr2.so ;// MyTdr2, scan out
                          }

Instance MyTdr2 Of tdr2   { InputPort si   = MySib1.so ;
                            InputPort se   = MyTap.se ; // shift enable
                            InputPort ce   = MyTap.ce ; // capture enable
                            InputPort ue   = MyTap.ue ; // update enable
                            InputPort en   = MySib2.ten ; // select enable
                          }
}
```

You must define the port connections of an instrument and declare each input port connection. Consequently, the connection list of each instrument instantiation only lists input ports and the ports driving them. For example, the port *so* of instance *MySib1* drives the input port *si* of the instance *MyTdr2* of the instrument *tdr2*.

How to Model Global Reset, Local Reset, and Embedded TAPs

This section describes the different aspects of the modeling of the IJTAG concepts "Global Reset" and "Local Reset" and the modeling of embedded TAP controllers.

Reset signals can be intercepted or gated to trigger the reset of registers only in certain parts of the design. Other registers can be prevented from getting their reset pulse, while the rest of the circuit is forced into its reset state. TAP controllers can be parked either in the "reset" state or in the "idle" state.

All those topics are related to the following three types of signals in the ICL description: reset signals, trst signals and tms signals. These signals can be either explicitly connected in the ICL description, or implicitly connected by the tool.

For the sake of completeness, the next pages show the set of rules that determine the implied connectivity of reset signals, trst signals, tms signals and registers with ResetValue specification. If certain special features are required, like a Local Reset or the isolation of an embedded TAP, these signals usually have to be connected explicitly in the ICL description. But for the understanding of the modeled behavior of an IJTAG network during a Global Reset or a Local Reset, it is important to know which connections between the different parts are implied by the tool.

Rules for the Implied Connections of Ports of Type ResetPort and ToResetPort

If a ResetPort of an ICL module is not explicitly connected using the "InputPort" statement in the instantiation of this module, the tool connects the ResetPort according to the following rules:

- If there is exactly one ResetPort in the parent module, the instance ResetPort is connected to this parent module port.

- If there is no ResetPort in the parent module but in total exactly one ToResetPort in the instances within the same parent module, the instance ResetPort is connected to this instance output port.

- If there are neither ResetPorts in the parent module nor ToResetPorts in the instances within the same parent module, the ResetPort is connected to the "Global Reset", a virtual signal that is active only in case of an iReset (synchronous as well as asynchronous).

- If none of the above is true, the source of the ResetPort is ambiguous, which triggers the DRC violation ICL124 ("ambiguous source of instance input port"). If the handling of this DRC is downgraded to warning, the ResetPort is connected to the "Global Reset" as if there was no suitable reset signal source.

If a ToResetPort of an ICL module is not explicitly connected using the "Source" property of the ToResetPort specification, the tool connects the ToResetPort according to the following rules:

- If there is exactly one ResetPort in the same module, the ToResetPort is connected to this port.

- If there is no ResetPort in the same module but in total exactly one ToResetPort in the instances within the module, the ToResetPort is connected to this instance output port.

- If there are neither ResetPorts in the same module nor ToResetPorts in the instances within the module, the ToResetPort is connected to the "Global Reset", a virtual signal that is active only in case of an iReset (synchronous as well as asynchronous).

- If none of the above is true, the source of the ToResetPort is ambiguous, which triggers the DRC violation ICL123 ("ambiguous source of output port"). If the handling of this DRC is downgraded to warning, the ToResetPort is connected to the "Global Reset" as if there was no suitable reset signal source.

For the implied connections of ResetPorts and ToResetPorts, the ActivePolarity of the ports does not have to match. Tessent Shell automatically inserts inverted connections if necessary.

Rules for the Implied Connections of Ports of Type TRSTPort and ToTRSTPort

If a TRSTPort of an ICL module is not explicitly connected using the "InputPort" statement in the instantiation of this module, the tool connects the TRSTPort according to the following rules:

- If there is exactly one TRSTPort in the parent module, the instance TRSTPort is connected to this parent module port.

- If there is no TRSTPort in the parent module, but in total exactly one ToTRSTPort in the instances within the same parent module, the instance TRSTPort is connected to this instance output port.

- If there are neither TRSTPorts in the parent module nor ToTRSTPorts in the instances within the same parent module, the TRSTPort is connected to "Asynchronous Global Reset", a virtual signal that is active only in case of an asynchronous iReset (iReset without -sync switch).

- If none of the above holds, the source of the TRSTPort is ambiguous, which triggers the DRC violation ICL124 ("ambiguous source of instance input port"). If the handling of this DRC is downgraded to warning, the TRSTPort is connected to "Asynchronous Global Reset" as if there was no suitable trst signal source at all.

If a ToTRSTPort of an ICL module is not explicitly connected using the "Source" property of the ToTRSTPort specification, the tool connects the ToTRSTPort according to the following rules:

- If there is exactly one TRSTPort in the same module, the ToTRSTPort is connected to this port.

- If there is no TRSTPort in the same module, but in total exactly one ToTRSTPort in the instances within the module, the ToTRSTPort is connected to this instance output port.

- If there are neither TRSTPorts in the same module nor ToTRSTPorts in the instances within the module, the ToTRSTPort is connected to "Asynchronous Global Reset", a virtual signal that is active only in case of an asynchronous iReset (iReset without –sync switch).

- If none of the above holds, the source of the ToTRSTPort is ambiguous, which triggers the DRC violation ICL123 ("ambiguous source of output port"). If the handling of this DRC is downgraded to warning, the ToTRSTPort is connected to "Asynchronous Global Reset" as if there was no suitable trst signal source at all.

Rules for the Implied Connections of Ports of Type TMSPort and ToTMSPort

If a TMSPort of an ICL module is not explicitly connected using the "InputPort" statement in the instantiation of this module, the tool connects the TMSPort according to the following rules:

- If there is exactly one TMSPort in the parent module, the instance TMSPort is connected to this parent module port.

- If there is no TMSPort in the parent module but in total exactly one ToTMSPort in the instances within the same parent module, the instance TMSPort is connected to this instance output port.

- If there are neither TMSPorts in the parent module nor ToTMSPorts in the instances within the same parent module, the DRC violation ICL126 ("missing source of instance input port") is triggered. This DRC cannot be downgraded to warning.

- If none of the above holds, the source of the TMSPort is ambiguous, which triggers the DRC violation ICL124 ("ambiguous source of instance input port"). If the handling of this DRC is downgraded to warning, the TMSPort is treated as if it was directly driven by a top-level TMSPort. The associated TAP controllers cannot be parked, and they cannot be prevented from reacting to the synchronous Global Reset.

If a ToTMSPort of an ICL module is not explicitly connected using the "Source" property of the ToTMSPort specification, the tool connects the ToTMSPort according to the following rules:

- If there is exactly one TMSPort in the same module, the ToTMSPort is connected to this port.

- If there is no TMSPort in the same module, but in total exactly one ToTMSPort in the instances within the module, the ToTMSPort is connected to this instance output port.

- If there are neither TMSPorts in the same module nor ToTMSPorts in the instances within the module, the DRC violation ICL125 ("missing source of output port") is triggered. This DRC cannot be downgraded to warning.

- If none of the above holds, the source of the ToTMSPort is ambiguous, which triggers the DRC violation ICL123 ("ambiguous source of output port"). If the handling of this DRC is downgraded to warning, the ToTMSPort is treated as if it was directly driven by a top-level TMSPort. The associated TAP controllers cannot be parked, and they cannot be prevented from reacting to the synchronous Global Reset.

Rules for the Implied Connections of Registers

The ICL constructs "DataRegister" and "ScanRegister" do not have a dedicated possibility to specify the source of their reset signal. Therefore, the association between register and reset signal is always implicit.

The following rules apply to the implied reset connectivity of the registers with ResetValue specifications (registers without ResetValue specification are not affected by any reset activity):

- If there is exactly one ResetPort in the parent module, the register is connected to this parent module port.

- If there is no ResetPort in the parent module, but in total exactly one ToResetPort in the instances within the same parent module, the register is connected to this instance output port.

- If there are neither ResetPorts in the parent module nor ToResetPorts in the instances within the same parent module, the register is connected to "Global Reset", a virtual signal that is active only in case of an iReset (synchronous and asynchronous).

- If none of the above is true, the source of the ResetPort is ambiguous, which triggers the DRC violation ICL127 ("ambiguous register reset signal"). If the handling of this DRC is downgraded to warning, the register is connected to "Global Reset" as if there was no suitable reset signal source.

The lack of means to specify the register reset explicitly may result in the requirement to introduce additional ICL hierarchy levels to clarify the association between registers and ResetPorts. See Figure 2-2. To achieve the unique association between the ScanRegister RegA and the ResetPort rst1, and the unique association between ScanRegister RegB and the ResetPort rst2, you must introduce the modules blockA_1 and blockA_2.

Figure 2-2. Association Between ResetPorts and Registers
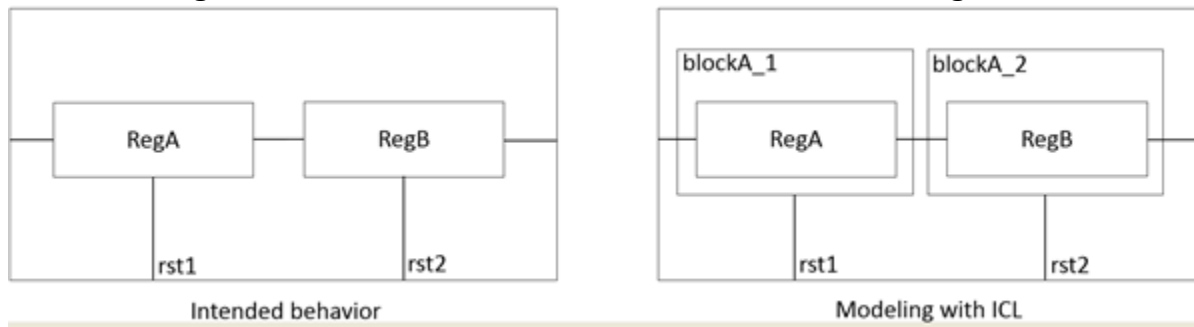
The Role of the DataMux

You must intercept reset, trst, and tms signals to introduce functionality like a Local Reset (that is, the triggering of a reset in certain parts of the circuit) or the suppression of reset activity during a Global Reset. You must this also for the isolation of TAP controllers including the possibility to park them in the "reset" state or the "idle" state. The general approach for modeling this kind of interception is the same for all three types of signals. The essential ICL element for this purpose is the DataMux. If exactly one of the data inputs of a DataMux is driven by a reset signal, trst signal, or tms signal, the DataMux becomes a "reset mux", "trst mux", or "tms mux" respectively. All the other inputs, including the select inputs, must be ordinary data signals. They must be driven by the parallel outputs of a register, by DataOutPorts or DataInPorts, by other DataMuxes, or by combinational logic modeled by the ICL construct "LogicSignal".

A DataMux that provides (through one of its inputs with an ordinary data signal) the active value of a ResetPort or ToResetPort can trigger the reset of all downstream registers. This sort of reset is called Local Reset. It happens right after the update cycle that configured the DataMux in such a way that it selects the data signal with the active reset value. The Local Reset rests until the DataMux is configured again in such a way that it presents the ordinary reset signal or an inactive reset value.

A DataMux that provides (through one of its inputs with an ordinary data signal) the inactive value of a ResetPort or ToResetPort can be used to suppress the effect of a Global Reset (that is, the reset activity that happens on behalf of an iReset command) or the effect of a higher-level Local Reset on the downstream registers.

A trst signal can be intercepted in the same way as a reset signal. This provides the possibility to either trigger an asynchronous Local Reset or to suppress the effect of an asynchronous Global Reset or an asynchronous higher-level Local Reset on the downstream registers. If a DataMux drives the value "0" to the TRSTPort of a TAP controller state machine (that is, an ICL module with a ToIRSelectPort), the TAP controller is held in the "reset" state. All registers associated with the ToResetPort of the TAP controller state machine are reset. No scan activity can happen through this TAP controller before the TRSTPort is either driven with its inactive value or with an ordinary trst signal again.

A DataMux can also be used to intercept a tms signal. If a DataMux drives the value "1" to the TMSPort of a TAP controller state machine (that is, an ICL module with a ToIRSelectPort), the TAP controller is held in the "reset" state. If a DataMux drives the value "0" to the TMSPort of a TAP controller state machine, the TAP controller is held in the idle state. In both cases, no scan activity can happen through this TAP controller before the TMSPort is driven with an ordinary tms signal again. Intercepting the tms signal of a TAP controller can be used to "park" an embedded TAP controller either in the "idle" state or in the "reset" state. In the latter case, the TAP controller is not only isolated, its registers are also reset.

Recommendations for the Design of Hierarchical Local Reset and Global Reset

Whenever it is not intended to suppress the Global Reset in certain parts of the circuit, but there is a possibility to trigger a Local Reset, the reset circuitry should be designed "hierarchically", such that the reset of a certain hierarchy level automatically triggers the reset of the modules below that level. This can be achieved by an appropriate specification of the ResetValue of the registers controlling the reset muxes. After a reset of the control registers, the attached reset muxes should present the ordinary reset signal, not one of the data signals.

Figure 2-3 on page 28 demonstrates this concept.

Figure 2-3. Hierarchical Reset
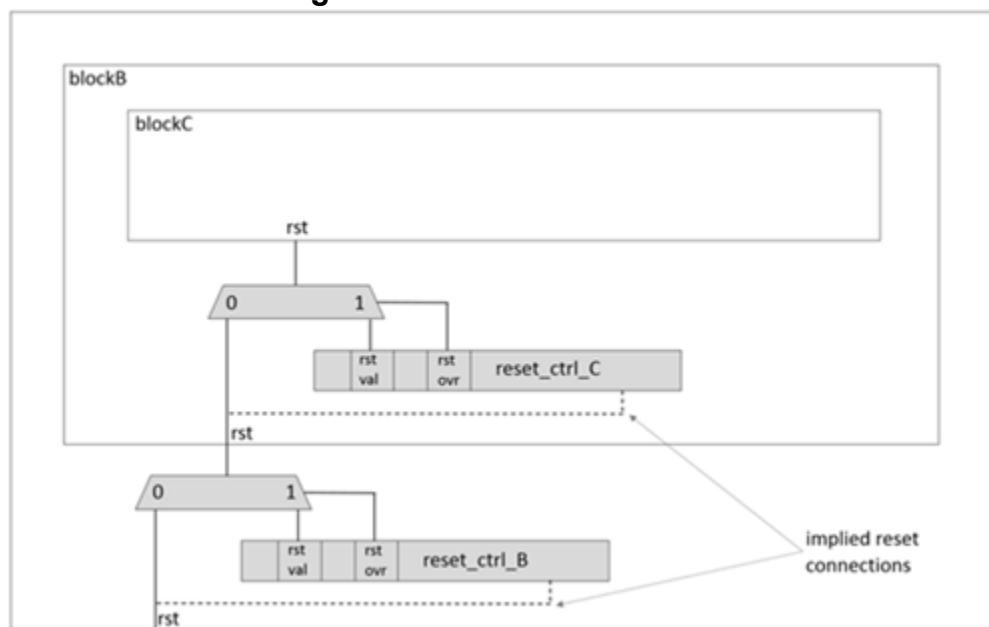
The reset_ctrl_B register can be used to trigger a Local Reset in blockB, the reset_ctrl_C register can be used to trigger a Local Reset in blockC. If the ResetValue specifications for the "rst_ovr" register bit of the two registers is set to 0, the Global Reset automatically triggers the reset in blockB, and the reset of blockB automatically triggers the reset of blockC. This is because the Global Reset resets the reset_ctrl_B register. When the "rst_ovr" bit of this register is reset to "0", the associated multiplexer immediately selects the ordinary reset signal, which is

currently active (because of the Global Reset). Therefore the active reset signal arrives at the rst input of blockB. This resets the register reset_ctrl_C. The "rst_ovr" bit of this register is reset to "0", and the associated multiplexer immediately selects the rst input of the module blockB, which is currently active. Consequently, the reset also arrives at blockC.

To achieve this desirable hierarchical behavior, it is recommended to choose the ResetValue of the control registers such that the associated multiplexers select the ordinary reset signals but not the data signals after a reset of the control registers.

_____ **Note** _____
If a Local Reset is triggered, the next scan load that shifts a value into a ScanRegister bit with the "tessent_force_rti_on_value_change" attribute set to "1" causes that scan load to end in the Run-Test/Idle (RTI) state. This includes registers not affected by the Local Reset.

Self-Clearing Local Reset

The IEEE 1687-2014 standard describes the concept of a self-clearing Local Reset. When the TDR that triggers the Local Reset reacts on this Local Reset (with a certain delay), then the reset control signals in this register can be set back to their inactive states without an additional scan load. The Local Reset is triggered immediately after the update of the TDR and becomes inactive immediately thereafter before the TAP controller is back to the "idle" state.

In the ICL, the delay cannot be modeled, and the tool automatically handles the Local Reset correctly. But it might be necessary to introduce additional hardware in the actual design description (Verilog or VHDL) to ensure the correct timing. In particular, it must be ensured that the Local Reset signal reaches all registers that are supposed to be reached before it switches itself off again.

Figure 2-4 on page 30 shows how the self-clearing Local Reset can be modeled. You need an additional level of hierarchy to describe the association between the ScanRegister and the ResetPort. ICL does not have the means to describe this association directly.

Figure 2-4. Modeling Self-clearing Local Reset
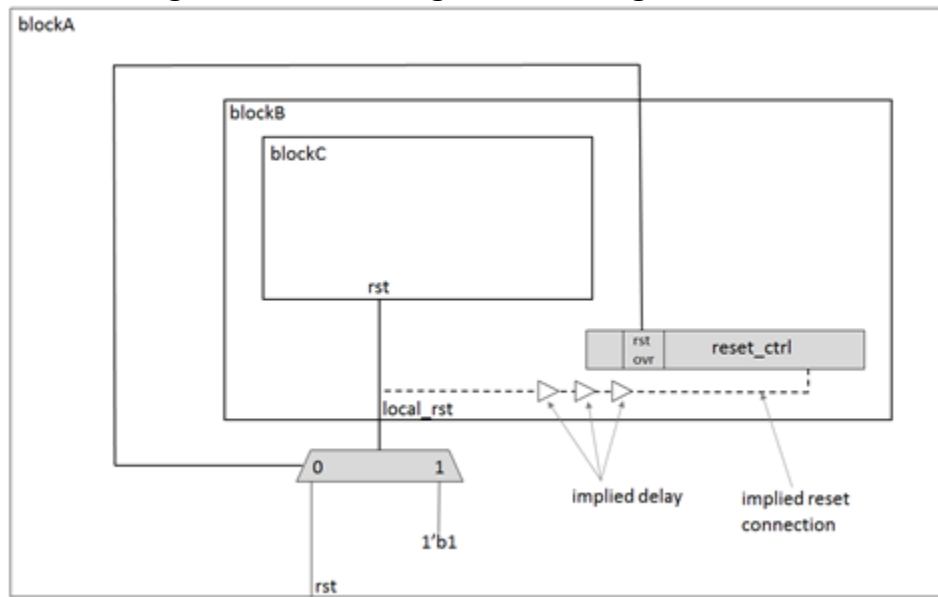
In 2-4, blockA is the top level. blockB contains all the logic that is subject to the Local Reset. blockC is just another block that makes use of the Local Reset. The rst_ovr bit in the reset_ctrl register has the ResetValue "0".

When the "reset_ctrl" register is loaded such that the "rst_ovr" bit contains the value "1", this value propagates to the DataMux in blockA and triggers a Local Reset in blockB. The reset also affects the "reset_ctrl" register, but only after resetting all registers in blockC (because of the implied delay). When the reset signal finally arrives at the "reset_ctrl" register, it sets "rst_ovr" back to "0". This switches off the Local Reset, and the "reset_ctrl" scan register is ready for ordinary scan load activity again. Any registers in blockC are usable again. The tool achieves all this with one scan load, and there is no need for dedicated "activate reset" and "disable reset" scan loads.

_____ **Note** _____
The complete hardware in this example is also subject to the ordinary Global Reset, because the inactive state of the "rst_ovr" control bit ensures that the top-level reset signal from the "rst" primary input port reaches all parts of the design.

Edge-Triggered Local Reset

In addition to the previously described reset mechanisms, a local reset can also be triggered by a rising or falling edge on a data signal. Such an edge-triggered reset is modeled through a special ICL module with the singular purpose of generating a reset signal based on a rising or falling edge.

The edge-triggered reset ICL module must have exactly three ports: One ResetPort, one ToResetPort, and one DataInPort. In addition, you must specify the

"tessent_edge_triggered_reset" attribute in the module description. The attribute can have the value "falling" or "rising". When a corresponding falling or rising edge occurs on the DataInPort of the module, the IJTAG engine automatically activates the ToResetPort of the module for one cycle. If no edge is detected, it sources the value of the ToResetPort from the ResetPort input of the module. Thus it resets a module that is connected to the ToResetPort of the edge-triggered reset module when it detects an edge as well as when the ResetPort input is active.

Figure 2-5 on page 31 and the example that follows it shows how a falling edge-triggered local reset can be modeled in ICL.

Figure 2-5. Edge-Triggered Local Reset
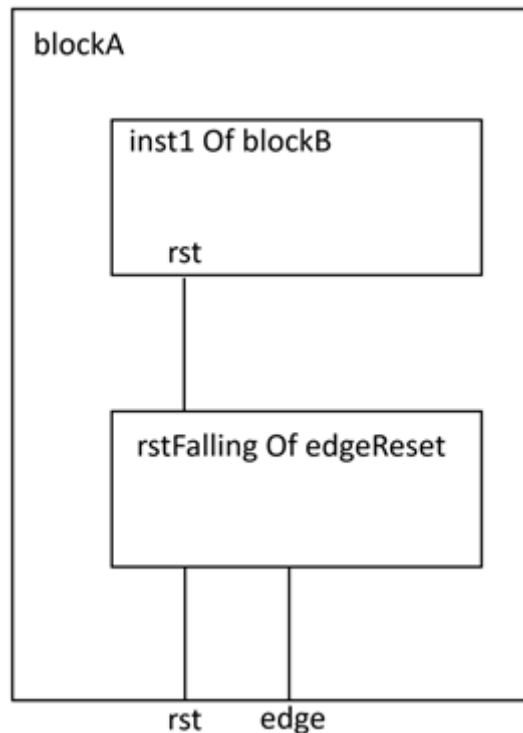
```
Module blockA {
    […]
    DataInPort edge;
    ResetPort rst;
    Instance rstFalling Of edgeReset {
        InputPort rst = rst;
        InputPort edge = edge;
    }

    Instance inst1 Of blockB {
        InputPort rst = rstFalling.toRst;
    }
}
Module edgeReset {
    ResetPort rst;
    DataInPort edge;
    ToResetPort toRst;
    Attribute tessent_edge_triggered_reset = "falling";
}
```

In Figure 2-5 on page 31, blockB has the ResetPort "rst" that is connected to the instance rstFalling of the edge-triggered reset module edgeReset. As such, the reset of blockB is triggered whenever the "rst" input of blockA is active and whenever a falling edge is detected on the "edge" input of blockA.

The following general rules apply to the edge-triggered reset module:

- The ResetPort and ToResetPort must have the same ActivePolarity.

- The decoding logic of the data signal that drives the DataInPort must not contain any unconnected parts. All of its inputs must ultimately be driven by DataRegisters, ScanRegisters, or top-level DataInPorts.

Synchronous Reset and Asynchronous Reset

The IEEE 1687 standard describes two types of reset: synchronous reset and asynchronous reset.

The command "iReset" triggers an asynchronous Global Reset, the command "iReset –sync" triggers a synchronous Global Reset.

The asynchronous Global Reset applies appropriate waveforms to the top-level ResetPorts and TRSTPorts. The effect of this reset is simulated in the retargeter considering all the explicit and implied reset connections. All internal ResetPorts, ToResetPorts, TRSTPorts, ToTRSTPorts, and registers, which are not connected to the reset circuitry (neither explicitly nor implicitly), are assumed to be driven with the active reset values.

The synchronous Global Reset applies appropriate waveforms to the top-level ResetPorts and TMSPorts to enforce the transition of the TAP state machine or machines to the state "test logic reset". The effect of this reset is simulated in the retargeter considering all the explicit and implied reset connections. All internal ResetPorts, ToResetPorts, and registers, which are not

connected to the reset circuitry (neither explicitly nor implicitly), are assumed to be driven with the active reset values. The internal TMSPorts and ToTMSPorts that are not connected to other TMS signal sources (neither explicitly nor implicitly), are assumed to be driven with the value "1" during this reset, such that the synchronous reset is also applied to the embedded TAP controllers without explicitly connected TMSPorts.

An ICL module that is meant to model the TAP state machine (that is, an ICL module with a ToIRSelectPort specification) can have a ToResetPort. In such a configuration of ports, the ToResetPort becomes active when either the TRSTPort of the module is active or the TMSPort of the module is constantly driven with the value "1" (which means that the TAP state machine arrives in the "test logic reset").

A ResetPort or a ToResetPort can be forced to react on asynchronous resets only. This can be achieved by explicitly connecting them to a TRSTPort or ToTRSTPort. A ResetPort or ToResetPort that is connected like this does not react on the synchronous reset.

A Local Reset is nearly simulated in the same way as a Global Reset. The trigger for a Local Reset is not the iReset command, but the update of a register that configures a "reset mux" or a "trst mux" such that a reset signal or a trst signal becomes active. The Local Reset does not have any effect on the internal ResetPorts, ToResetPorts, TRSTPorts, ToTRSTPorts, and registers that are not connected to the reset circuitry (neither explicitly nor implicitly).

In some cases, the asynchronous Global Reset is not possible. In the following situations, Tessent IJTAG automatically performs a synchronous Global Reset, even if the –sync switch of the iReset command is omitted:

- If the top-level TRSTPort is constrained to the inactive value (CT1 constraint)

- If the top-level TRSTPort is forced to the inactive value by iForcePort

- If one of the TAP controllers does not have a TRSTPort

The use of the iReset command has some effects that are not immediately related to the reset of the ScanRegisters, DataRegisters, and TAP state machines. These are:

- The iClock specifications are deleted

- The iClockOverride specifications are deleted

- The iOverrideScanInterface specifications are set to their default values

Requirements and Limitations

The support of Local Reset and Embedded TAP controllers is subject to a few special requirements and limitations:

- **Dedicated iApply commands for Local Reset** — The retargeter must not try to resolve the iWrite and iRead requirements by means of a Local Reset, because this would have undesirable side effects on other parts of the circuit. To ensure that the ordinary iRead or

iWrite retargeting does not interfere with a Local Reset, all modifications on the register bits and primary inputs controlling the Local Reset must happen in the *last* iApply operation (scan load or parallel I/O operation). If this restriction makes the retargeting of the iApply impossible, you must separate the ordinary iRead and iWrite commands from those that are meant to trigger or terminate the Local Reset and use dedicated iApply commands for each of them. In such a situation, the following message is shown:

```
The retargeter cannot find a solution for the current PDL
constraints without toggling the reset of at least one register
in an intermediate iApply operation (iApply operation = scan load
or primary input alteration). The ICL and the PDL must be designed
such that it is not required to toggle the register reset before
the last iApply operation. Consider splitting the iApply into
several parts, for example, one iApply to apply/terminate the
Local Reset and one iApply for the remaining PDL targets.
```

- **Timing considerations** — The embedded TAP controllers with "parking" functionality and the related control signals must be designed such that the TAP controllers are automatically synchronized with the other enabled TAP controllers after they have been released from their parking state ("reset" or "idle"). This requirement can only be fulfilled if the timing of the involved signals is carefully designed.

  For example: Assume that an embedded TAP controller has been parked by means of an interception of its TMS signal in the *reset* state. This parking usually ends on the falling edge of TCK, when the control register for the isolation of the embedded TAP (that is, the register that controls the interception of the TMS signal of the embedded TAP) is updated by a higher-level TAP. From this moment on, the embedded TAP controller must receive the TMS signal again, and the state machine of the embedded TAP and the state machine of the other TAPs must be running. While the other TAPs are currently in the Update-DR state, the embedded TAP that has just been woken up is still in the reset state. There is only one TAP state transition left before the retargeter assumes that all TAP controllers are in IDLE state again. Therefore, it is crucial that the interception of the TMS ends before the next TAP state transition is applied, such that the next state transition with TMS=0 takes the embedded TAP from reset to idle, while at the same time the other TAPs are taken from Update-DR to idle. If the signal for the termination of the TMS interception arrives too late, the embedded TAP stays in the "reset" state and the TAP controllers get out of sync.

ScanInterfaces and Associations Between Ports and ScanInterfaces

This section explains the behavior of the tool regarding ScanInterface creation, inference, checking, and introspection.

An ICL ScanInterface describes the assembly of ports that provide scan access to the IJTAG network or a subset of the IJTAG network.

You can explicitly specify ScanInterfaces or the tool can infer them. It can also infer the association between ports and explicitly specified ScanInterfaces. The conditions for the automated inference of ScanInterfaces and their ports are complex.

How the tool handles ScanInterfaces depends on whether the ICL module in question is associated with the current design or not. The rules for ICL modules associated with the current design are stricter. The tool must know which ScanInterface control ports of the current design (CaptureEnPort, ShiftEnPort, UpdateEnPort, SelectPort, or TMSPort) are associated with the individual scan ports (ScanInPort or ScanOutPort) to create the stimulus data for the control ports correctly.

In this section, the term "current design" refers to the ICL module that is associated with the current design. The term "internal module" refers to an ICL module that is part of the current ICL elaboration but not associated with the current design.

> **Note**
> The IEEE 1687-2014 standard uses the term "handoff module" for an ICL module that parties exchange and need stricter checks. Tessent IJTAG only treats the current design as an ICL "handoff module".

All mechanisms that infer ScanInterfaces or associations between ports and ScanInterfaces do not affect the content of files you create using the write_icl command. The internal data model changes only, which affects the behavior of the PDL retargeting and the result of ICL introspection.

Modules With Explicitly Specified ScanInterfaces

This section explains the design rule checks and the additional inferred associations between ports and ScanInterfaces for modules with explicitly specified ScanInterfaces.

DRCs for Explicitly Specified ScanInterfaces

In a module with explicitly specified ScanInterfaces, the ports and the ScanInterfaces are subject to various checks. This section describes them.

Checking ICL Module Ports

If the current design contains more than one ScanInPort or ScanOutPort, all its scan ports (ScanInPorts and ScanOutPorts) must be members of explicitly specified ScanInterfaces. This is true irrespective of the number of existing ScanInterfaces and the number of scan ports already members of ScanInterfaces. For example, if some scan ports lack membership in a ScanInterface but other ScanInterface specifications are already in place, the tool does not try to infer ScanInterfaces for the "unattended" scan ports. The related DRC is ICL103.

You can downgrade the ICL103 severity level to "Warning". However, an ICL103 violation prevents IJTAG retargeting, independent of its severity level.

For internal modules, no such restriction exists. There can be a mixture of scan ports that are members of ScanInterfaces and scan ports that are not members of ScanInterfaces, as long as the existing ScanInterface specifications are valid. As for the current design, the tool does not try to infer the presence of additional ScanInterfaces if there is at least one explicitly specified ScanInterface.

Checking ScanInterface Members

The tool checks the consistency and standard compliance of Port and Chain specifications of explicitly specified ScanInterfaces using several DRCs, but these are the most important:

- **ICL70** — Invalid mixture of ports that are specific for client, host, client-TAP, or host-TAP ScanInterfaces.

- **ICL101** — ScanInterface without control ports.

- **ICL105** — Invalid combinations of ports that are not covered by one of the other DRCs.

- **ICL106** — Invalid ports in the "Chain" construct of the ScanInterface.

- **ICL107** — "Chain" construct in a host or host-TAP ScanInterface.

- **ICL108** — Missing associations between scan ports and ScanInterface chains.

- **ICL109** — DefaultLoadValue outside of Chain specification.

- **ICL110** — Chain specifications with overlapping ports.

Refer to the related descriptions in the *Tessent Shell Reference Manual* for more details.

Inferred Associations Between Ports and Explicitly Specified ScanInterfaces

The following section shows the rules that determine the implied association between ICL ports and ICL ScanInterfaces.

These additional associations between ICL ports and ICL ScanInterfaces affect the introspection of the icl_port, icl_pin, icl_scan_interface, and icl_scan_interface_of_module objects. However, they do not appear in the files you create using the write_icl command.

_____ **Note** _____

The tool does not consider ICL ports with the "ir" or "ir_dr" function_modifier and does not automatically assign them to a ScanInterface if there are several ICL ports with the same port function as the port in question.

In the following, the phrase "all <port_function>" refers to all ports of type <port_function> except the ones with the "ir" or "ir_dr" function_modifier if there are several ICL ports of type <port_function>. Otherwise, it refers to the unique port of type <port_function>, independently of the function_modifier.

The capture and update ports have two conditions that indicate the tool should not interfere with the explicit ScanInterface specifications. The tool suppresses implied associations between ICL ports and ICL ScanInterfaces when both of the following conditions are true:

- The ScanInterface contains an explicitly specified ShiftEnPort or ToShiftEnPort.

- All ports with the port_function in question (CaptureEnPort, UpdateEnPort, ToCaptureEnPort, or ToUpdateEnPort) are already part of other explicitly specified ScanInterfaces.

The tool associates all CaptureEnPorts of an ICL module with a ScanInterface if all of the following conditions are true:

- All CaptureEnPorts of the ICL module have distinct function_modifier attribute values.

- The ScanInterface has been identified as a client ScanInterface.

- There are no CaptureEnPorts in the original specification of the ScanInterface.

- The implied association of the CaptureEnPorts with the ScanInterface is not suppressed by the tool.

The tool associates all ShiftEnPorts of an ICL module with a ScanInterface if all of the following conditions are true:

- All ShiftEnPorts of the ICL module have distinct function_modifier attribute values.

- The ScanInterface has been identified as a client ScanInterface.

- There are no ShiftEnPorts in the original specification of the ScanInterface.

The tool associates all UpdateEnPorts of an ICL module with a ScanInterface if all of the following conditions are true:

- All UpdateEnPorts of the ICL module have distinct function_modifier attribute values.

- The ScanInterface has been identified as a client ScanInterface.

- There are no UpdateEnPorts in the original specification of the ScanInterface.

- The implied association of the UpdateEnPorts with the ScanInterface is not suppressed by the tool.

The tool associates all ToCaptureEnPorts of an ICL module with a ScanInterface if all of the following conditions are true:

- All ToCaptureEnPorts of the ICL module have distinct function_modifier attribute values.

- The ScanInterface has been identified as a host ScanInterface.

- There are no ToCaptureEnPorts in the original specification of the ScanInterface.

- The implied association of the ToCaptureEnPorts with the ScanInterface is not suppressed by the tool.

The tool associates all ToShiftEnPorts of an ICL module with a ScanInterface if all of the following conditions are true:

- All ToShiftEnPorts of the ICL module have distinct function_modifier attribute values.

- The ScanInterface has been identified as a host ScanInterface.

- There are no ToShiftEnPorts in the original specification of the ScanInterface.

The tool associates all ToUpdateEnPorts of an ICL module with a ScanInterface if all of the following conditions are true:

- All ToUpdateEnPorts of the ICL module have distinct function_modifier attribute values.

- The ScanInterface has been identified as a host ScanInterface.

- There are no ToUpdateEnPorts in the original specification of the ScanInterface.

- The implied association of the ToUpdateEnPorts with the ScanInterface is not suppressed by the tool.

The tool associates a TCKPort of an ICL module with a ScanInterface if all of the following conditions are true:

- The port is the only TCKPort in the ICL Module.

- The ScanInterface has been identified as a client or client-TAP ScanInterface.

- There is no TCKPort in the original specification of the ScanInterface.

The tool associates a ToTCKPort of an ICL module with a ScanInterface if all of the following conditions are true:

- The port is the only ToTCKPort in the ICL Module.

- The ScanInterface has been identified as a host or host-TAP ScanInterface.

- There is no ToTCKPort in the original specification of the ScanInterface.

The tool associates a ResetPort of an ICL module with a ScanInterface if all of the following conditions are true:

- The port is the only ResetPort in the ICL Module.

- The ScanInterface has been identified as a client ScanInterface.

- There is no ResetPort in the original specification of the ScanInterface.

The tool associates a ToResetPort of an ICL module with a ScanInterface if all of the following conditions are true:

- The port is the only ToResetPort in the ICL Module.

- The ScanInterface has been identified as a host ScanInterface.

- There is no ToResetPort in the original specification of the ScanInterface.

The tool associates a TRSTPort of an ICL module with a ScanInterface if all of the following conditions are true:

- The port is the only TRSTPort in the ICL Module.

- The ScanInterface has been identified as a client-TAP ScanInterface.

- There is no TRSTPort in the original specification of the ScanInterface.

The tool associates a ToTRSTPort of an ICL module with a ScanInterface if all of the following conditions are true:

- The port is the only ToTRSTPort in the ICL Module.

- The ScanInterface has been identified as a host-TAP ScanInterface.

- There is no ToTRSTPort in the original specification of the ScanInterface.

Modules Without Explicitly Specified ScanInterfaces

This section explains the design rule checks and the creation of anonymous ScanInterfaces for modules without explicitly specified ScanInterfaces.

Checking ICL Module Ports

The combination of the ICL ports in the ICL module must enable the unambiguous creation of one anonymous single-chain ScanInterface based on the rules described in this section.

_____ **Note** _____

These rules apply when the current design does not contain any explicit ScanInterface specifications, but it does contain at least one scan port (ScanInPort or ScanOutPort).

These checks are the minimal requirements for inferring an anonymous ScanInterface in the current design. The actual strategy to infer a ScanInterface (including additional ports like CaptureEnPort or UpdateEnPort) is described in "Anonymous ScanInterface Creation" on page 42.

If the tool cannot infer a required ScanInterface for the current design, it reports an ICL71 violation.

You can downgrade ICL71 to a "Warning" severity level. However, an ICL71 violation prevents IJTAG retargeting, independent of the violation's severity level.

For internal modules with scan ports but without explicit ScanInterface specifications, the possibility to infer an anonymous ScanInterface is not mandatory. The tool attempts to infer a ScanInterface as described in Anonymous ScanInterface Creation, but it does not report if it fails to do so.

- The tool ignores ScanInPorts directly feeding into OneHotScanGroups during the checks described in this section. The tool treats them as inactive inputs of the OneHotScanGroups later during IJTAG retargeting.

- There can only be one ScanInPort and one ScanOutPort.

  o The tool is working to infer one anonymous single-chain ScanInterface. The presence of multiple ScanInPorts or ScanOutPorts requires one of the following: multiple ScanInterfaces, multiple chains in one ScanInterface, or it is ambiguous which ports to use for the ScanInterface.

- At most, one of the following port types can appear in the ICL module: ShiftEnPort, ToShiftEnPort, TMSPort, or ToTMSPort.

  o These port_functions are necessarily associated with ScanInterfaces. They unambiguously determine the type of the ScanInterface. The ScanInterface type is ambiguous if there is more than one of these port functions.

- If there are no ports of type ShiftEnPort, ToShiftEnPort, TMSPort, or ToTMSPort, then at most, one of the following port types can appear in the ICL module: SelectPort or ToSelectPort.

  o SelectPorts and ToSelectPorts are not necessarily associated with ScanInterfaces (that is why they do not appear in the list of port_functions in the previous check).

  o However, without ShiftEnPort, ToShiftEnPort, TMSPort, and ToTMSPort, the only possibility to form a valid ScanInterface is having either a SelectPort or a ToSelectPort as a control signal for the ScanInterface.

  o If there are both SelectPort and ToSelectPort, the type of the ScanInterface is ambiguous.

- One of the following port types must appear in the ICL module: ShiftEnPort, ToShiftEnPort, TMSPort, ToTMSPort, SelectPort, or ToSelectPort.

  o It is not possible to form a valid ScanInterface without any of these ports.

  o A client ScanInterface must have a ShiftEnPort, SelectPort, or both.

  o A host ScanInterface must have a ToShiftEnPort, ToSelectPort, or both.

  o A client-TAP ScanInterface must have a TMSPort.

  o A host-TAP ScanInterface must have a ToTMSPort.

- If the previous checks pass and indicate a client or client-TAP ScanInterface, the ICL module must contain a ScanInPort and a ScanOutPort. Only host or host-TAP ScanInterfaces can lack one of the scan ports.

- The ShiftEnPort, ToShiftEnPort, TMSPort, ToTMSPort, SelectPort or ToSelectPort must be unique. Ultimately, this determines the type of the ScanInterface.

  o If this is the ShiftEnPort, ToShiftEnPort, TMSPort, or ToTMSPort, the tool ignores the SelectPorts and ToSelectPorts of the ICL Module.

  o If there are no ports of type ShiftEnPort, ToShiftEnPort, TMSPort, or ToTMSPort, then the SelectPort or ToSelectPort must be unique.

Anonymous ScanInterface Creation

The creation of anonymous ScanInterfaces can occur in the current design and internal modules. In the current design, the tool can create an anonymous ScanInterface if the scan ports exist. In

internal modules, the tool silently skips the creation of the ScanInterfaces if it is impossible or ambiguous.

The tool assigns the name "unnamed" to the anonymous ScanInterface. The tool can perform introspection on the anonymous ScanInterface like any other ScanInterface. However, they do not appear in the files you create using the write_icl command.

When the tool determines the presence and uniqueness of an anonymous client ScanInterface from the rules in "Checking ICL Module Ports" on page 41, it creates the ScanInterface with the following members:

- A unique ScanInPort and the unique ScanOutPort.

- A SelectPort if it exists and it is unique.

- All CaptureEnPorts of the module if they have distinct function_modifier attribute values.

- All ShiftEnPorts of the module if they have distinct function_modifier attribute values.

- All UpdateEnPorts of the module if they have distinct function_modifier attribute values.

- A ResetPort if it exists and it is unique.

- A TCKPort if it exists and it is unique.

When the tool determines the presence and uniqueness of an anonymous host ScanInterface from the rules in "Checking ICL Module Ports", it creates the ScanInterface with the following members:

- A unique ScanInPort, the unique ScanOutPort, or both.

- A ToSelectPort if it exists and it is unique, and there are no ToShiftEnPorts in the middle.

- All ToCaptureEnPorts of the module if they have distinct function_modifier attribute values.

- All ToShiftEnPorts of the module if they have distinct function_modifier attribute values.

- All ToUpdateEnPorts of the module if they have distinct function_modifier attribute values.

- A ToResetPort if it exists and it is unique.

- A ToTCKPort if it exists and it is unique.

When the tool deduces the presence and uniqueness of an anonymous client-TAP ScanInterface from the rules in "Checking ICL Module Ports", it creates the ScanInterface with the following members:

- A unique ScanInPort and the unique ScanOutPort.

- A unique TMSPort.

- A TRSTPort if it exists and it is unique.

- A TCKPort if it exists and it is unique.

When the tool deduces the presence and uniqueness of an anonymous host-TAP ScanInterface from the rules in "Checking ICL Module Ports", it creates the ScanInterface with the following members:

- A unique ScanInPort, the unique ScanOutPort, or both.

- A unique ToTMSPort.

- A ToTRSTPort if it exists and it is unique.

- A ToTCKPort if it exists and it is unique.

How to Define an iProc

Most of your ICL instruments have PDL that describes, for example, how the instrument is supposed to be tested or operated.

The PDL is described at the IO-boundary of the instrument. It is then up to Tessent IJTAG to retarget these PDL commands to the chosen ICL hierarchy level, referred to as the current_design and defined using the set_current_design command. Here are two simple examples of the usage of 'iProc'.

```
iProcsForModule tdr1

iProc write_data { value } {
    iNote "Writing the value '$value' to register R"
    iWrite R $value
    iApply
}

iProc read_data { value } {
    iNote "Reading the expect value '$value' from register R"
    iRead R $value
    iApply
}
```

The first observation is that PDL requires that an iProc is bound to one ICL module. This binding is accomplished with the PDL keyword iProcsForModule. All PDL iProcs following

the iProcsForModule keyword are bound to the specified module. The range of the binding is up to the next usage of iProcsForModule and is not linked in any way to the file in which it was specified.

How to Call an iProc

By using the binding of an iProc to an ICL module, you make the iProc available to each instance of the module.

Assume that an instance *MyTdr* of *tdr1* of the example above is located in an instance named *Block1* that is instantiated in an ICL module instance of *Core3*. The full hierarchical ICL instance path is therefore *Core3.Block1.MyTdr*. Because each instance of module *tdr1* has access to the iProc *write_data* you can invoke the iProc by calling it as follows:

    iCall Core3.Block1.MyTdr.write_data 0xff

In more general terms, the iCall *effective_icl_instance_path* to an iProc is a concatenation of the iProc instance path (the path of the iProc being processed), the ICL hierarchy prefix defined through the iPrefix command, and the specified ICL path in the first argument of the iCall command. Refer to the iCall in the *Tessent Shell Reference Manual* for a full description of the command.

Chapter 3
A Typical PDL Retargeting Flow

This chapter describes how to perform the basic PDL retargeting flow with Tessent Shell.

The Basic PDL Retargeting Flow

The main steps of the PDL retargeting flow consist of invoking Tessent Shell and setting the context and system modes, reading the ICL and PDL files, setting the retargeting level, defining clocks and timing, checking design rules, creating patterns sets, and writing the pattern files.

Figure 3-1 illustrates the main steps of the basic flow.

Figure 3-1. PDL Retargeting Flow
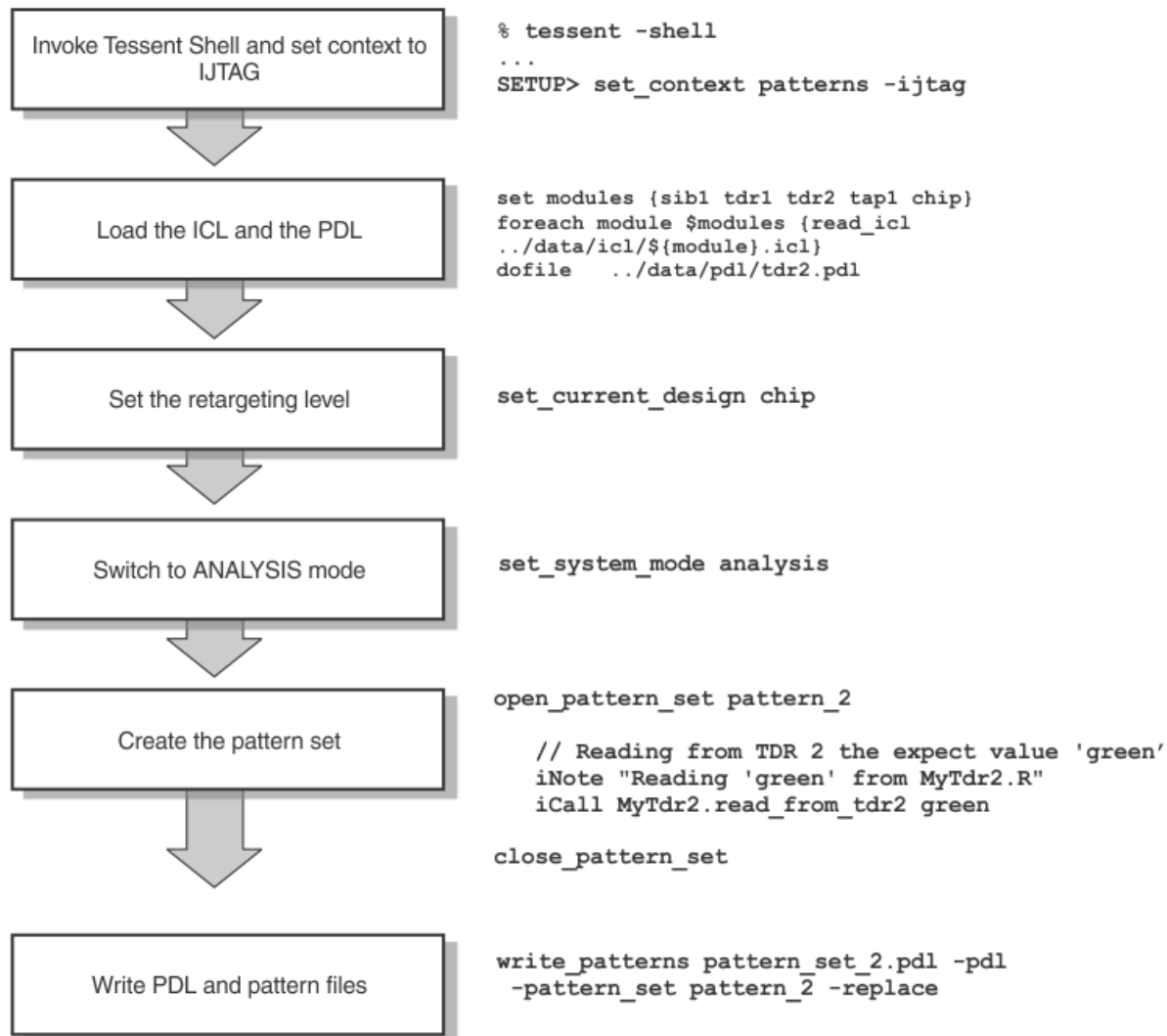
The following sections define the flow and discuss each step separately. The sections describe the necessary steps for an ICL flow that uses only ICL and PDL files. Consequently, the patterns Tessent IJTAG computes can contain only ports defined in the top-level ICL module as defined using the set_current_design command. If you need to include ports that are outside of the ICL description, you must also read in at least the top-level Verilog description of your netlist. This enables you, for example, to define the speed of a non-ICL system clock, add input

constraints outside of the ICL, or in general have all ports of the topmost Verilog module automatically included in the retargeted PDL testbench. These optional steps are discussed in the section "Optional Elements of a PDL Retargeting Flow". For these reasons, it is recommended to also read in the top netlist module, as described in the section "Set the Retargeting Level".

If you do not have a top-level ICL file for your design, Tessent IJTAG can compute one for you using the Verilog gate-level netlist description.

Tessent IJTAG functionality is implemented in Tessent Shell. Many of the commands used in Tessent IJTAG are the same commands you know from ATPG, like the add_clocks or set_procfile_name commands. This document makes frequent references to commands in Tessent Shell as needed for the understanding of the flow and usage.

Refer to the *Tessent Shell Reference Manual* for a full description of the commands.

Invoke Tessent Shell

You invoke Tessent Shell from a command line shell.

Use the following syntax:

```
% tessent -shell
```

After invocation, the tool is in setup mode. Refer to "Tool Invocation" in the *Tessent Shell User's Manual* for additional invocation options.

Set the IJTAG Context and System Modes

After Tessent Shell loads, you must tell the tool what you intend to do next.

Do this using the set_context command.

For PDL retargeting, you use the options (sub-context) of ijtag pattern generation as follows:

set_context patterns -ijtag

Running this command moves Tessent Shell to setup mode for IJTAG command retargeting. At this point, you declare the files to read and other setup parameters, as explained in the following section.

Read ICL Files

Tessent IJTAG builds the ICL hierarchy by reading the ICL module definitions and ICL instantiations. This is the only mandatory step in setup mode.

There is only one command for reading in any ICL file:

read_icl filename

For example, the ICL includes three files: a top-level ICL module in the local directory in the file named *./top.icl*, a module of a sib in a file located in a provided ICL library, *${ICL_Library_Path}/sib.icl*, and an instrument module description in *${ICL_Library_Path}/ instr_1.icl*. The following line reads all three files:

read_icl ./top.icl ${ICL_Library_Path}/sib.icl ${ICL_Library_Path}/instr_1.icl

The Tessent IJTAG tool automatically determines the ICL hierarchy described in these files. You do not need to specify the modules in any particular order. Of course, using more than one read_icl command is possible. The following alternative example is equivalent to the one above:

set icl_module_ list { sib instr_1 }

foreach icl_module $icl_module_list { read_icl ${ICL_Library_Path}/${icl_module}.icl }

read_icl ./top.icl

Tessent IJTAG does not require that the ICL files have the *.icl* filename extension, however, it is recommended. Using this naming convention, you can easily read all ICL files of a particular directory, for example:

read_icl ${ICL_Library_Path}/*.icl

Refer to the read_icl command in the *Tessent Shell Reference Manual* for a full description.

Read PDL Files

Strictly speaking, PDL files are not mandatory to read. You can create pattern sets using direct iRead and iWrite operations to the instruments. However, this is uncommon.

More common is that most of your instruments have PDL files associated with them. These PDL files provide iProcs bound to the instrument you can iCall later during PDL command retargeting.

In the Tessent IJTAG tool within Tessent Shell, PDL command files are considered Tcl dofiles because they describe actions that the tool must perform, just like any other Tcl dofile. You can therefore use the normal dofile command:

dofile PDL_filename[.gz]

Tessent IJTAG does not require that the PDL files have the .pdl filename extension, however, it is recommended.

There are two types of PDL files:

- A PDL file that wraps all PDL commands other than iProc and iProcsForModule inside of iProcs

- A PDL file that contains PDL commands other than iProc and iProcsForModule outside of iProcs

While loading the PDL dofiles, Tessent Shell runs the commands in the dofile. You can therefore mix Tcl commands, Tcl procs, and PDL iProcs in the same file. Tcl procs and PDL iProcs are registered and can be called later. Other commands are run immediately, just like any other dofile. Therefore, the second type of PDL file is not allowed in setup mode. You can only run it inside an open pattern set.

Consider avoiding the second type because using this type of PDL is not portable. It must be written with a particular top ICL module in mind, results in hard to follow PDL commands, and may be error-prone in its usage. Encapsulating all PDL reading and writing commands inside of iProcs enables the PDL retargeting later in the flow to use only iCalls to meaningful instances and iProc names.

PDL files of the first type can be called in both setup and analysis mode, and inside an open pattern set.

Set the Retargeting Level

Once the ICL is read, you must tell the tool to which level in the hierarchy the PDL commands should be retargeted.

Tessent IJTAG uses the same command set_current_design, also used for setting the topmost netlist level for a Verilog netlist description.

>   **set_current_design [module name]**

The module name can be omitted if there is no ambiguity about which module is the top-most module and if you want to retarget the PDL to this level. Running the set_current_design triggers among others, the creation of the ICL hierarchy based on the read ICL modules and ICL connectivity. Any error, for example, a misconnection that violates IJTAG semantic rules, is flagged as a design rule violation at that point.

Usually, the current design is set to the top-level ICL module. We recommend to also load the topmost Verilog module, at least as a black box. This enables subsequent optional adding of non-ICL clocks and input constraints to non-ICL input ports. Even if your ICL modeling does not need non-ICL clocks or input constraints, your Verilog testbench from the ICL flow is still difficult to simulate against the Verilog/RTL level netlist description because it contains only the ICL-known ports. Any non-ICL port in your top-most Verilog/RTL module is not part of your ICL testbench. Loading the netlist into Tessent IJTAG makes all ports known to the tool. It then automatically adds all ports to the ICL testbench.

Therefore, Siemens EDA recommends the following:

>   **read_verilog <topmost Verilog file name> -interface_only**

>   **set_current_design [module name]**

See read_verilog and set_current_design in the *Tessent Shell Reference Manual*.

Define Clocks and Timing

To define the timing of your patterns or testbench computed from the retargeted PDL, you must understand the relationship and dependencies between the default behavior and default timeplates built into Tessent IJTAG, any user-specified timeplates, the add_clocks command, and the open_pattern_set command.

For more information, see add_clocks and open_pattern_set in the *Tessent Shell Reference Manual*.

Test Clock

In the simplest PDL retargeting flow, you do not need to declare clocks and timeplates to the tool. The tool knows the ICL test clock through the ICL port function "TCKPort" in the topmost ICL module.

See IJTAG Network Performance Optimization for how to maximize the frequency of the IJTAG network test clock.

Tessent IJTAG knows several default timeplates, depending on the off-state of the test clock and other timing properties. Hence, there is no need for a user-defined timeplate. See the open_pattern_set command for a detailed description of these defaults.

By default, the tool assumes the following properties for the test clock and relative timing of pins:

- The test clock has an off-state of 0

- The test clock period is 100ns

- The relative timing of the ports is force off PI, measure PO, pulse off the test clock

If the test clock is declared with an off-state of 1 through the add_clocks command, the default changes to the following:

- The test clock has an off-state of 1 (as you have declared)

- The test clock period is 100ns

- The relative timing of the ports is force off PI, pulse off the test clock, measure PO, leading clock edge

The exact timing of all events at the ports is automatically determined for you. It depends on the off state of the test clock and the open_pattern_set -tck_ratio switch as explained in detail in the

open_pattern_set command. If you are satisfied with these built-in, default timeplates and when events like the measure of the PO happen, you do not need to define a timeplate.

If you just want to change the period of the test clock, you can do this easily at the moment you describe the PDL to retarget as part of the open_pattern_set command. However, you may want to use a timeplate if you want to change the exact timing of events, for example when the PO is measured. The only time a custom timeplate is mandatory is when you want to define more than two edges of system clocks.

Test Clock Example

Assume you want to use a 200ns test clock named tck, with an off-state of 1. You also do not want to use a timeplate. Because you want to use the off-state of 1, but the Tessent IJTAG tool's default is the off-state of 0 for the test clock, you have to use the add_clocks command first. You can then use options to open_pattern_set to change the default 100ns timing of the test clock to 200ns.

Together, this looks like the following:

add_clocks 1 tck

set_system_mode analysis

open_pattern_set pat1 -tester_period 200ns -tck_ratio 1

You must specify the -tck_ratio as 1, which means that the tck period is equal to the tester period. If you do not specify the -tck_ratio, it is computed automatically to create the TCK period specified with the "set_ijtag_retargeting_options -tck_period" command (the default TCK period is 100 ns). In this example, this results in a TCK ratio of 2. Alternatively, you can also change the TCK period length to 200 ns ("set_ijtag_retargeting_options -tck_period 200ns").

Synchronous System Clock

System clocks are declared in ICL using the ICL port function 'ClockPort'.

All system clocks declared this way are by definition of the IEEE 1687 standard pulse-always clocks. Hence, you must use the add_clocks command with the -pulse_always option.

System clocks declared to Tessent IJTAG in this way are considered synchronous clocks. This means they are synchronous to the tester clock, defined with the -tester_period option of the open_pattern_set command.

Synchronous System Clock Example

Let us continue the test clock example from the Test Clock Example section. In addition to the 200ns period test clock, you now also have two synchronous 50ns system clocks. One system clock, the one named sclk0, has an off state of 0, whereas the other, sclk1, has an off state of 1.

System clocks require the add_clocks command. You can then use open_pattern_set options to define the timing of the test clock versus the system clocks. You must set the tester period to the speed of the system clock, then use the tck ratio to time the test clock relative to the system clock speed, as shown here:

> **add_clocks 1 tck**
>
> **add_clocks 0 sclk0 -pulse_always**
>
> **add_clocks 1 sclk1 -pulse_always**
>
> **set_system_mode analysis**
>
> **open_pattern_set pat1 -tester_period 50ns -tck_ratio 4**

Asynchronous System Clock

Asynchronous system clocks are declared to the tool in a very similar way as synchronous system clocks.

You need to use the -period option of the add_clocks command to declare the period of these asynchronous system clocks.

For example, the following line declares a 10 ns asynchronous system clock:

> **add_clocks SysClk -period 10ns**

Asynchronous System Clock Example

Assume that sclk0 and sclk1 were not synchronous clocks of 50 ns period, but asynchronous clocks of 30 ns and 70 ns period, respectively. The example would now look like this:

> **add_clocks 1 tck**
>
> **add_clocks 0 sclk0 -period 30ns**
>
> **add_clocks 1 sclk1 -period 70ns**
>
> **set_system_mode analysis**
>
> **open_pattern_set pat1 -tester_period 200ns -tck_ratio 1**

Because the asynchronous clocks are independent of the tester period, you can use the -tester_period option in combination with the -tck_ratio option to define the period of the test clock. If you do not specify the tck_ratio, it is automatically computed such that the product of tester period and TCK ratio is at least as long as the TCK period specified with the "set_ijtag_retargeting_options -tck_period" command. Because the default tck_period is 100 ns, the tck_ratio would be automatically computed as 1 in this example.

As a second example, assume that only sclk0 is a 30 ns asynchronous clock. Because sclk1 is a synchronous clock relative to the tester period (50 ns again), the example now looks like this:

> **add_clocks 1 tck**

add_clocks 0 sclk0 -period 30ns

add_clocks 1 sclk -pulse_always

set_system_mode analysis

open_pattern_set pat1 -tester_period 50ns -tck_ratio 4

Observe that this is the same set of options of the open_pattern_set command, as used in the synchronous clock example above. In other words, asynchronous clocks add only the -period option to the add_clocks command, but have no other influence on the flow. Only synchronous clocks related to the test clock period must be considered when using the open_pattern_set options.

Refer to the *Tessent Shell Reference Manual* for additional information on the add_clocks and open_pattern_set Tcl commands and the PDL command iClock.

Design Rule Checks

During the change from setup to analysis mode the final set of design rules are checked and test procedures, if declared, are evaluated.

You do this by calling the Tessent Shell command and option

set_system_mode analysis

This switches from setup mode to analysis mode, in which you can create patterns through PDL command retargeting.

Tessent IJTAG is very verbose in its error messages. Usually, it lists the ICL or PDL filename, the line number in error, the offending keyword if applicable, and a very verbose text explaining the error and often also how to fix it. The example below shows an error in the tdr1 ICL file. The data width used in an alias statement is incorrect between the left and right side of the alias statement.

```
// Error: Expression width conflict found in ICL module 'tdr2' Enum
// definition 'PermissibleValues'. The Enum definition 'PermissibleValues'
// has a width of 8 whereas the value '7'b1111111' has a width of 7. The
// actual expression must have a width that is the same as the width of the
// target. Found on or near line 21 of file '../data/icl/tdr2.icl'.
// ICL semantic rule ICL52
```

No error can be waived. All errors must be fixed before Tessent IJTAG can enter the analysis mode.

Create Pattern Sets

Once in analysis mode, there are only two things to do, creating patterns and writing patterns. Tessent IJTAG encapsulates the PDL commands inside of named pattern sets.

To do this:

> **open_pattern_set pattern_set_name [options]**
>
> **<iCall of previously defined iProcs, iRead, iWrite, … >…**
>
> **close_pattern_set [options]**

The name of the pattern set is mandatory and must be unique among all used pattern sets. This name is used to reference the patterns in several later commands, for example, reporting or writing the pattern set to disk.

You can declare multiple pattern sets, but only one can be open at a time. There is no option to append to a pattern set once it is closed.

The order of multiple pattern set definitions is not important, because Tessent IJTAG runs the PDL command iReset at the beginning of each pattern set. The effect of iReset depends on the reset-value definition defined for an instrument and its components. All registers having a specified ResetValue in ICL are expected to be in their reset state. All other registers are assumed to have an unknown value.

Because an iReset is performed at the beginning of each pattern set, the starting state of the ICL netlist is identical from pattern set to pattern set. Therefore, the pattern sets do not depend on each other. They can be defined and saved in any order. Patterns sets can also be skipped when saving.

Sometimes this automatic iReset at the beginning of a pattern set is undesirable. An example of this is a complex PDL-based setup of a PLL, followed by one or several PDL pattern sets, requiring the PLL. You may require that the PLL remains locked and does not get reset. However, if you suppress this iReset, the current pattern set depends on the state reached at the end of the previous pattern set. Therefore, this reset option cannot be used in the very first pattern set. Further, you must take great care to save and run both pattern sets in the proper order.

The following example shows how to open a pattern set suppressing the initial iReset:

> **open_pattern_set pat1 -no_initial_ireset**

Please consult the *Tessent Shell Reference Manual* for complete information. The open_pattern_set command reference has several examples showing how to achieve a certain timing behavior of the retargeted PDL.

Retargeting is done at every iApply. Once the retargeting has completed successfully, you can open another pattern set or save the retargeted PDL into one or several pattern formats. All pattern sets remain available until deleted or until the tool terminates.

To report all currently available pattern sets use the report_pattern_sets command:

> **report_pattern_sets [options]**

In the following example, a pattern set "test1" is created using default options (timeplate, tester period, tck ratio, initial iReset, active scan interfaces, network end state, TAP start state, TAP end state):

```
set_context patterns -ijtag
read_icl chip.icl
source chip.pdl
set_current_design chip
set_system_mode analysis
open_pattern_set test1
iCall mytest1
close_pattern_set
report_pattern_sets
```

This produces the following report:

Figure 3-2. First Pattern Set

```
// name| timeplate | tester | tck   | tester | initial | active scan |network  | TAP start|TAP end|saved
//     |           | period | ratio | cycles | iReset  | interfaces  |end state| state    |state  |
// ----+-----------+--------+-------+--------+---------+-------------+---------+----------+-------+-----
// test1| (default) | 100ns  | 1     | 369    | yes     | (all)       |keep     | any      | IDLE  | no
```

If you now create a second pattern set "test2" with the different options specified here:

```
open_pattern_set test2 -tester_period 200ns -tck_ratio 4 -no_initial_ireset
iCall mytest2
close_pattern_set -network_end_state reset
write_patterns test2.stil -stil -pattern_sets test2
report_pattern_sets
```

you get the following report listing both pattern sets:

Figure 3-3. First and Second Pattern Sets

```
// name| timeplate | tester | tck   | tester | initial | active scan |network  | TAP start|TAP end|saved
//     |           | period | ratio | cycles | iReset  | interfaces  |end state| state    |state  |
// ----+-----------+--------+-------+--------+---------+-------------+---------+----------+-------+-----
// test1| (default) | 100ns  | 1     | 369    | yes     | (all)       |keep     | any      | IDLE  | no
// test2| (default) | 200ns  | 4     | 1256   | no      | (all)       |reset    | IDLE     | IDLE  | yes
```

The report contains the following columns:

name — Name of the pattern set. First argument of the open_pattern_set command.

timeplate — The timeplate used for this pattern set. Argument of the -timeplate switch of the open_pattern_set command.

tester period — The tester period. Either the argument of the -tester_period switch of the open_pattern_set command or derived from the timeplate. If there is no timeplate and no -tester_period switch, the tester period is chosen such that the TCK period equals the product of tester period and TCK ratio.

tck ratio — The TCK ratio. Number of tester cycles for one TCK cycle. Argument of the -tck_ratio switch of the open_pattern_set command. If the -tck_ratio switch is not specified, the

tck_ratio is computed such that the product of tester period and TCK ratio is at least as long as the TCK period specified with the "set_ijtag_retargeting_options -tck_period" command.

tester cycles — The overall number of tester cycles of this pattern set. This is calculated when the pattern set is closed.

initial iReset — The pattern set contains an automatically added iReset. Default is yes. This can be switched off by means of the -no_initial_ireset switch of the open_pattern_set command.

active_scan_interfaces — The names of the active scan interfaces. Argument of the -active_scan_interfaces switch of the open_pattern_set command.

network endstate — The state to which the ICL scan network is forced when the pattern set is closed. This is the argument of the -network_end_state switch of the close_pattern_set command. This can be either "keep" (the state is not changed) or "initial" (the state as it has been at the beginning of the pattern set) or "reset" (the state as it is after reset).

TAP start state — The expected TAP start state. The possible values are: "IDLE", "DRPAUSE", "IRPAUSE" or "any".

TAP end state — The established TAP end state. The possible values are: "IDLE", "DRPAUSE" or "IRPAUSE".

saved — Whether the pattern set has been saved by the write_patterns command.

_____ **Note** _____
If a pattern set has an initial iReset, then the column "TAP start state" contains the word "any" instead of the name of a particular TAP state, because the TAP start state does not matter in this case.

Write PDL, Pattern, and Testbench Files

Tessent IJTAG only stores an internal representation of the retargeted PDL. When you write patterns, the retargeted PDL is translated into the requested pattern format.

You can select which of the pattern sets should be included. If no pattern set is given, all pattern sets in the sequence of their declaration are saved. The following example writes the pattern sets p1 and p3 in the STIL format to file *p13.stil*; pattern set p2 is written as a Verilog testbench into *filep2.v*; pattern set p4 is written in the SVF format to file *p4.svf*.

```
write_patterns p13.stil -stil -pattern_sets p1 p3

write_patterns p2.v -verilog -pattern_sets p2

write_patterns p4.svf -svf -pattern_sets p4
```

___ **Note** ___

The "write_patterns -svf" command has been enhanced to write retargeted SVF patterns. This includes generating SVF patterns, for example, for WTAP interfaces or other hierarchical modules. SVF files generated through this new feature are then easily read as user-defined sequences (UDS) and applied to a DUT.

In earlier Tessent IJTAG versions, SVF patterns were only generated for a top-level design with a TAP. The resulting SVF file typically included SIR (Scan Instruction Register), SDR (Scan Data Register), STATE (move FSM to specific state), and TRST (Test ReSeT) statements.

The enhanced SVF writer now generates SVF even if a TAP interface was not positively identified in the current design. In such a case, the SVF file toggles the design pins using PIO (Parallel IO) lines instead of the usual SIR, SDR, STATE, and TRST statements.

Please consult the *Tessent Shell Reference Manual* for complete information on all options of the write_patterns command.

Comments and Annotations in Tessent IJTAG

Tessent IJTAG creates various types of comments, notes, and annotations during IJTAG retargeting, IJTAG pattern file generation, and IJTAG testbench generation.

During the actual retargeting step, Tessent IJTAG processes commands such as iWrite, iRead, iApply, and stores the result in an internal format (IJTAG pattern sets). When it does this, two different types of notes come into existence: user notes and process notes. User notes are the result of the iNote command. Process notes are created by the retargeter to annotate the different steps of the retargeting process and to add useful information to the pattern sets.

When you use the command write_patterns to create a pattern file or a testbench, Tessent IJTAG converts the user notes and the process notes into the appropriate representation in the pattern file format. Besides this, the command write_patterns may convert certain content of the pattern sets into comments in the pattern file, if this content is not representable by other means of the pattern file output language.

User Notes

Origin

When Tessent IJTAG processes each iNote command during IJTAG retargeting, it converts it into a user note. When it imports and converts SVF files into IJTAG pattern sets (either by using the command import_patterns_from_svf or by using the svf_file property of the ProcedureStep wrapper in the pattern specification), it also converts the comments in the SVF file into user notes.

Pattern Set Data

The pattern set introspection command, get_pattern_set_data, provides access to the user notes in the IJTAG pattern sets. For each event of the pattern set, the lists that are associated with the keys "notes.type_list" and "notes.text_list" contain the information about the user notes and the process notes of the event. If the nth element of the type_list is "user", then the nth element of the text_list is a user note.

write_patterns -svf

Tessent IJTAG converts user notes into SVF comments, using the "exclamation point" comment style.

Example:

```
! This is a user note
```

write_patterns –pdl

Tessent IJTAG converts user notes into iNote commands in the output PDL. Example:

```
iNote "This is a user note"
```

write_patterns (all other pattern file and testbench formats)

The output-format-specific language constructs represent the user notes. Example (STIL):

```
Ann {* This is a user note *}
```

Process Notes

Most frequent process notes

The IJTAG retargeter creates process notes to report on the currently addressed PDL targets and to show human-readable details about the results of the retargeting.

Process notes per iApply

For each iApply, the tool adds the list of addressed iRead and iWrite commands as a process comment. You can specify the style of this list using the following command:

```
set_ijtag_retargeting_options -iapply_target_annotations off|dense|full
```

If you set this option to "off", it adds the list of addressed iRead and iWrite commands as process comment.

If you set the option to "dense", it groups registers and bus ports, and the values may be abbreviated.

Example:

```
//
// Targets:
//   Apply 0
//     reads:
//         block2_I1.longreg[199:0] = 00010100010001000111...00010100010001000111
//         block2_I2.shortreg[7:0]  = 11001110
//     writes:
//         block1_I1.datain = 1
//         block1_I1.cfg    = 0
//
```

If you set the option to "full", it reports each single bit separately and abbreviation does not happen.

```
//
// Targets:
//   Apply 0
//     reads:
//         block2_I2.shortreg[5] = 1
//         block2_I2.shortreg[4] = 1
//         block2_I2.shortreg[3] = 0
//         block2_I2.shortreg[2] = 0
//         block2_I2.shortreg[1] = 0
//         block2_I2.shortreg[0] = 1
//     writes:
//         block1_I1.datain = 1
//         block1_I1.cfg    = 0
//
```

If the tool internally splits an iApply into several iApply commands (this can happen, for example in case of indirect addressing schemes with DataRegisters), the number behind the "Apply" denotes the currently processed internal iApply.

Process notes per scan load

The tool augments each scan load of the result of the retargeting with process notes about the scan path, the load value and the unload value.

Example 1 (scan load through a TAP ScanInterface without Chain construct):

```
// TAP vector tdi..tdo (tap1_I1.bypass sib1_I1.SIB block2_I1.tdr.R[10:0]
//                      sib1_I2.SIB sib1_I3.SIB)
//   Loading: 0_0_10000000000_1_0
// Unloading: 0_0_000XXXXXXXX_0_0
```

Example 2 (scan load through an ordinary ScanInterface without Chain construct, inverted register):

```
// scan vector si..so (sib1_I1.SIB ~block2_I1.tdr.R[10:0]
//                      sib1_I2.SIB sib1_I3.SIB)
//   Loading: 0_10000000000_1_0
// Unloading: 0_000XXXXXXXX_0_0
```

Example 3 (scan load through an ordinary ScanInterface with four Chain statements):

```
// scan (ch3) vector si[3]..so[3] (blockA.reg3[3:0] blockB.reg3[3:0])
//   Loading: 0000_1100
// Unloading: 1100_1111
// scan (ch2) vector si[2]..so[2] (blockA.reg2[3:0] blockB.reg2[3:0])
//   Loading: 1010_0101
// Unloading: 1110_0001
// scan (ch1) vector si[1]..so[1] (blockA.reg1[3:0] blockB.reg1[3:0])
//   Loading: 1111_0000
// Unloading: 0000_0000
// scan (ch0) vector si[0]..so[0] (blockA.reg0[3:0] blockB.reg0[3:0])
//   Loading: 0101_1010
// Unloading: 0011_0011
```

Process notes per cycle or cycle group

When the IJTAG retargeter expands scan loads or TAP state transitions into test vectors, it also creates process notes to document what is going on.

You can find the following notes when scan loads through ordinary ScanInterfaces are expanded:

```
// Capture Cycle
// Shift Cycles
// Update Cycle
```

You can find the following notes when scan loads through TAP ScanInterfaces are expanded:

```
// Advance TAP controller from idle to Shift-IR
// Advance TAP controller from idle to Shift-DR
// Advance TAP controller from Pause-IR to Shift-IR
// Advance TAP controller from Pause-DR to Shift-DR
// Shift-IR
// Shift-DR
// Advance TAP controller from Shift-IR to Exit-IR
// Advance TAP controller from Shift-DR to Exit-DR
```

Comments and Annotations created during write_patterns

_____ **Note** _____

A backslash shown in this section indicates line continuation. The complete comment displays in one single line in the pattern file and the backslash does not appear in the pattern file.

write_patterns –svf

The SVF pattern file writer creates the following comments:

!svf_cmd *number*

The index of the next SVF command.

!TESSENT_PRAGMA pattern_set *name*

The name of the pattern set.

!TESSENT_PRAGMA tester_period *period*

The tester period.

!TESSENT_PRAGMA icl_checksum *checksum_and_version*

The ICL checksum and the version of the checksum computation algorithm.

If several versions of the ICL checksum computation algorithm exist, the tool performs each computation and several "!TESSENT_PRAGMA icl_checksum" notes appear in the SVF file.

!TESSENT_PRAGMA clock [-type *type*] **-pin** *pin* **[-pin_inverse** *npin*] **[-period** *period*] \

[-offstate 0|1]

The specification of the added clocks. These notes appear at the beginning of the pattern file. There is one "TESSENT_PRAGMA clock" comment per added clock.

- *type* - *type* is only specified in case of synchronous clocks.

  It is one of sync, sync2x, sync3x, and so on.

- *pin* - The name of the clock.

- *npin* - The associated port in case of a differential clock.

- *period* - The clock period. It is only specified in case of asynchronous clocks.

- -offstate - The off-state of the clock.

!TESSENT_PRAGMA expect_z *binary_value*

SVF does not have a generic possibility to represent "Detect Floating".The "!TESSENT_PRAGMA expect_z" comment is inserted before a PIO command if at least one of the ports is compared to the value Z. The positions with the digit "1" mark the positions of the ports that are compared to "Z" in the next PIO command. The order of digits in the *binary_value* matches the PIOMAP specification, the leftmost bit in the *binary_value* refers to the first port in the PIOMAP, the rightmost bit refers to the last port in the PIOMAP. The length of the binary_value must match the number of ports in the PIOMAP.

!TESSENT_PRAGMA annotation *pdl_target* **-type** *type* **-var_bits** *var_bits* **[-var_length** *length*] \

-pin *pin* **[-relative_cycles** *cycles*] **[-inversion** *inversion*] **[-persistent_compare On|Off]**

This comment refers to the next PIO, SDR, or SIR command. It is required by the pattern management to map the original iRead and iWrite targets to the ports and the values in the SVF pattern file:

- *pdl_target* — The original iRead, iComparePort, iWrite, or iForcePort target that is observed (or set) by the next SVF command (PIO, SDR, SIR).

  pdl_target can be followed by an occurrence count in parentheses.

- *type* — "read" in case of iRead and iComparePort, "write" in case of iWrite and iForcePort. Note: "TESSENT_PRAGMA annotation" comments of type "write" are currently not generated. "write" is reserved for future use.

- *var_bits* — The *pdl_target* does not contain indices or ranges, even if the original PDL target is a port or a register with several bits. It always contains only the base name, potentially followed by an occurrence count in parentheses. *var_bits* specifies which parts of the PDL target this TESSENT_PRAGMA refers to. It is a list of indices or ranges.

- *length* — If the *pdl_target* is a port or a register with several bits but the first occurrence of the PDL target is used only with a subset of this range, subsequent processing steps might need to know the complete range of the PDL target at the time of its first occurrence. The *length* parameter provides this information. It is one more than the maximum occurring index of the PDL target.

- *pin* — The top-level port through which the iRead, iComparePort, iWrite, or iForcePort target is observed (or set). For iRead targets that are observed through scan registers, *pin* is the ScanOutPort that unloads the register. For iWrite targets that are set using scan registers, *pin* is the related ScanInPort that is used to load the register.

- *cycles* — For SIR and SDR commands, the cycles in which the observations of the iRead targets happen at the ScanOutPort, or the cycles in which the values for the iWrite targets are applied to the ScanInPort. Like the *var_bits*, *cycles* is a list that can contain a mixture of indices and ranges.

- *inversion* — A binary number that specifies the inversion between the original PDL target and the value that is observed (or set) at *pin*. The width of the binary number must match the number of indices in *var_bits*. The leftmost bit of *inversion* refers to the leftmost index in *var_bits*, the rightmost bit of *inversion* refers to the rightmost index in *var_bits*.

- -persistent_compare — Set this switch to "On" if the next PIO command is the first one in which a iComparePort command becomes effective. This switch specifies pins for continuous observation. Set the switch to "Off" to stop the continuous observation.

!TESSENT_PRAGMA variable *variable* **-type** *type* **-var_bits** *var_bits* \

    [-var_length *length***] -pin** *pin* **[-relative_cycles** *cycles***] [-inversion** *inversion***] **

    [-persistent_compare On|Off]

This is required by the pattern management to map the iRead variables and iWrite variables to the ports and the values in the SVF pattern file.

write_patterns –pdl

There are no relevant comments that are created during write_patterns –pdl.

write_patterns (all other pattern file and testbench formats)

The following annotations are created in all other pattern file formats:

> **TESSENT_PRAGMA annotation pdl_target -type** *type* **-var_bits** *var_bits* \
>
> **[-var_length** *length*] **-pin** *pin* **[-relative_cycles** *cycles*] **[-inversion** *inversion*]

The description of the parameters matches the description of the SVF pattern files.

For a scan load, this kind of annotation displays only at the first ***shift*** vector of the scan load. All "TESSENT_PRAGMA annotation …" specifications belonging to the scan load are collected and inserted at this place.

The –persistent_compare switch, which exists in the SVF pattern files, does not exist for other pattern file formats, because these other formats reapply the "expect" states derived from the iComparePort commands in each single vector anyway.

> **TESSENT_PRAGMA variable** *variable* **-type** *type* **-var_bits** *var_bits* **[-var_length** *length*] \
>
> **-pin** *pin* **[-relative_cycles** *cycles*] **[-inversion** *inversion*]

The description of the parameters matches the description of the SVF pattern files.

For a scan load, this kind of annotation can only be found at the first ***shift*** vector of the scan load. All "TESSENT_PRAGMA variable …" specifications belonging to the scan load are collected and inserted at this place.

The –persistent_compare switch, which exists in the SVF pattern files, does not exist for other pattern file formats, because these other formats reapply the "expect" states derived from the iComparePort commands in each single vector, anyway.

> **TESSENT_PRAGMA bit_annotation** *pdl_target* **-type** *type* **-var_bits** *var_bits* \
>
> **[-var_length** *length*] **-pin** *pin* **[-inversion** *inversion*]

The tool creates these pragmas as an unrolled version of the "TESSENT_PRAGMA annotation". Unlike the "TESSENT_PRAGMA annotation" that displays together with the first shift vector of a scan load, the "TESSENT_PRAGMA bit_annotation" displays directly together with the vector that contains the related drive state or expect state. Therefore, the relative_cycles parameter is dispensable for this kind of pragma.

> **TESSENT_PRAGMA bit_variable** *variable* **-type** *type* **-var_bits** *var_bits* **[-var_length** *length*] \
>
> **-pin** *pin* **[-inversion** *inversion*]

The tool creates these pragmas are as an unrolled version of the "TESSENT_PRAGMA variable". Unlike the "TESSENT_PRAGMA variable" that displays together with the first shift vector of a scan load, the "TESSENT_PRAGMA bit_variable" displays directly together with the vector that contains the related drive state or expect state. Therefore, the relative_cycles parameter is dispensable for this kind of pragma.

Exit the Tool

Use the Tessent Shell exit command to exit the tool.

> **exit**

You can optionally specify the -force switch that instructs the tool to terminate even if there are unsaved patterns.

Optional Elements of a PDL Retargeting Flow

Up to this point, only the mandatory components of a basic PDL retargeting flow have been discussed. This section discusses several optional elements of the flow.

The first three elements deal with objects outside of ICL and PDL: test procedures, for example for bringing the circuit into 'ijtag test mode,' defining and using non-ICL clocks, and defining input constraints on non-ICL ports. Following this is a discussion of reporting and introspection.

Test Setup and Test End Procedures

You have already seen that timeplates can be used to define a particular clocking and pin event sequence. If there are test_setup or test_end procedures in the loaded procedure file, these procedures are considered in Tessent IJTAG as well.

Because the test procedures cannot be simulated on the ICL netlist, Tessent IJTAG can only add the cycles defined in the procedures to the saved pattern sets. The "write_patterns" command adds the cycles from the test_setup procedure at the beginning of each saved pattern file, independently of how many pattern sets are contained in the written file, and only for the formats of STIL, WGL, and Verilog. Similarly, it adds the cycles from the test_end procedure at the end of each saved pattern file.

Writing patterns in PDL format cannot contain the cycle information, because the written PDL represents only the retargeted PDL.

test_setup procedures have a second effect: all forced pins that are constant at the end of the test_setup run are regarded as input constraints. This behavior is equivalent to the Tessent ATPG tools. To force an input pin in a test procedure that is not in the top-level ICL module, you first have to load at least an interface-only version of the top-level Verilog netlist description before setting the current_design in setup mode:

```
read_verilog myTopLevelVerilogModule.v -interface_only
```

Once the Verilog netlist has been loaded in setup mode, the tool knows about the non-ICL pin. You are free to force the pin in the test procedures as described earlier.

In Tessent IJTAG, you cannot iCall an iProc from within any test procedure. This functionality is only available in the ATPG functionality in Tessent Shell, because the test_setup procedure in

ijtag context is used to enable the ICL network using arbitrary events. For example, it may be required to turn on powered down regions of the die. You can also use test_setup to program the system clock circuitry when it is not under the control of the ICL network.

How to Define and Use Clocks Outside ICL

You can declare additional clocks that do not correspond to ICL clock ports.

You do this with the add_clocks command. For example, you could have a second, non-IJTAG test clock or a reference clock outside of ICL that you need to perform the events described in your test_setup.

The timing of these clocks may be defined in a timeplate in the test procedure file. You can pulse these clocks also in the test_setup and test_end procedures.

To define either aspect of a non-ICL clock, you first have to load at least an interface-only version of the top-level Verilog netlist description:

    read_verilog myTopLevelVerilogModule.v -interface_only

Once the Verilog netlist has been loaded in setup mode, the tool knows about the non-ICL pin. Use the add_clocks command to declare the clock and its properties to the tool, including using the "-pulse_always" option of the add_clocks command. Such always-pulse non-ICL clocks are pulsed as defined during IJTAG operation.

As usual, if your clock is not always-pulse, you have to explicitly pulse clocks in the test procedures (test setup, test end). You also must have a timeplate for these clocks. Consequently, these clock events are used when processing the procedures as part of writing the patterns (other than PDL) to disk. These clocks are not pulsed during any IJTAG operation.

How to Constrain Inputs

You may want the design to be statically configured outside of the ICL and PDL in an "ijtag" test mode. The PDL retargeter knows only about top-level ports that are in the port list of the top-level ICL module. Most of the time, there are just the JTAG ports leading to and from the TAP controller. Neither ICL nor PDL have the concept of statically constrained top-level ports. In the IEEE 1687 standard, this task is left to the application tool.

There are two principal ways of achieving this static configuration. The first is through a test_setup procedure as discussed above. The second way is through the usual Tessent Shell command:

    add_input_constraints <primary_input_pin_name> [options]

In order to constrain a non-ICL input port, you first have to load at least an interface-only version of the top-level Verilog netlist description:

read_verilog myTopLevelVerilogModule.v -interface_only

Once the Verilog netlist has been loaded in setup mode, Tessent Shell knows about the non-ICL pin. Use the add_input_constraints command to declare this input pin and its properties to the tool.

Input constraints on ICL ports are not allowed, with the following exceptions:

- You can use the add_input_constraints command to constrain an ICL port of type DataInPorts with CT0 or CT1 constraints.

- You can use the add_input_constraints command to constrain an ICL port of type TRSTPort with a CT1 constraint. This enables a single ICL network description to be used for a wafer test, where a TRSTPort is available, and also for further package tests when the TRSTPort has been bonded to the package and is no longer available.

  If you place a CT1 input constraint on the TRSTPort and use iReset, the tool performs a synchronous reset using five test clock pulses, while holding the TMSPort high, followed by a test clock pulse with TMSPort low.

- You can use the add_input_constraints command to constrain an ICL port of type ClockPort. The iClock command detects constrained clock sources, including constrained differential or inverse clock sources.

  In the following example, the tool traces iClock to the constrained port ClkA and a constrained differential clock port ClkP.

```
//   command: add_clocks ClkA -period 10ns
//   command: add_clocks ClkP -period 10ns
//   command: add_clocks ClkM -period 10ns
//   command: add_input_constraints ClkA -C0
//   command: add_input_constraints ClkP -C0
//   command: add_input_constraints ClkM -C0
.
.
.
//   command: # constrained clock
//   command: catch { iClock block1_I1.raw1_I1.ClkA }
//   sub-command: iClock block1_I1.raw1_I1.ClkA

//   Error: Unable to trace the iClock 'block1_I1.raw1_I1.ClkA' to a
valid clock source. Traced the system clock to the constrained port
'ClkA'.

//   command: # constrained differential inv
//   command: catch { iClock block1_I1.raw1_I1.ClkP }
//   sub-command: iClock block1_I1.raw1_I1.ClkP
//   Error: Unable to trace the iClock 'block1_I1.raw1_I1.ClkP' to
//   a valid clock source. Traced the system clock to clock source
//   'ClkP'. This port is the representative port of a differential
//   clock, but the associated port is constrained.
```

Report Generation

All Tessent Shell reporting commands start with "report_". A report is a human-readable output from the tool to the screen or the transcript file.

With Tessent Shell, Siemens EDA also introduces introspection. In contrast to reporting, introspection creates, manipulates, operates on, or deletes information in a way suitable for scripting. All introspection commands that generate information start with 'get_'.

Using Tessent Shell, you can report information about iClocks using the following command:

report_iclock — Reports the ICL ClockPort specified by the iClock commands and their extracted sources and cumulative freqMultiplier and freqDivider values. You can only specify this when you have an opened pattern set.

Reporting all pattern sets is achieved as follows:

report_pattern_sets — Reports all pattern sets if no parameter is given. If no options are given, the command lists all patterns sets in order of declaration, not in alphabetical order. You can use options to report only pattern sets that match a certain name or to change the sorting criteria of the report.

The following lines show an example of usage in Tessent Shell:

Figure 3-4. Pattern Set Report With Two Patterns

```
//command: report_pattern_set
//
//name       | timeplate | tester | tck   | tester| initial | active scan |network   | TAP start|TAP end|saved
//           |           | period | ratio | cycles| iReset  | interfaces  |end state| state    |state  |
//  ----+-----------+--------+-------+-------+---------+-------------+---------+----------+-------+-----
// pattern_2| (default)| 100ns  | 1     | 33    | yes     | (all)       |keep     | any      | IDLE  | no
// pattern_1| Slow     | 200ns  | 1     | 46    | yes     | (all)       |keep     | IDLE     | IDLE  | no
```

In this example, two pattern sets were created using the open_pattern_set / close_pattern_set pair of commands. One pattern set is named pattern_1, the other is named pattern_2. The former uses a timeplate named "Slow", whereas the latter uses no specific timeplate (this means the timing is according to the default timing of Tessent IJTAG). Both do not change the relationship between the ICL test clock and the ATE's timing (tck ratio remains 1). The information in the column "tester_cycle" provides the number of test clock cycles required for the pattern set to run. The two columns of "initial iReset" and "network end state" give information about which options were used in the open_pattern_set and close_pattern_set commands. With these options, the state of the ICL network can be manipulated. For example, the initial reset can be suppressed or the Tessent IJTAG is asked to retain the state of the network as it was before the opening of the pattern set. For more details, see the command descriptions for the open_pattern_set and close_pattern_set commands. The final column indicates if the pattern set had already been saved to disk or not.

IJTAG Introspection

Tessent Shell provides a robust Tcl-based command set you can use to introspect design objects.

This introspection is also available for the ICL data model. For more information, see "ICL Data Model" in the *Tessent Shell Reference Manual* and "Object Specification Format" in the *Tessent Shell User's Manual*. There you find several elaborate examples of using introspection for a Verilog netlist. Here, the focus is on using introspection in the IJTAG context. The only difference is that here, you access the ICL data structures, and not the data structures representing the Verilog netlist. The concepts of introspection and collections remain the same.

Below are some examples of introspection and report generation. Refer to the *Tessent Shell Reference Manual* for a complete description of the commands used in these examples.

Example 1

Get all ICL modules and print their names.

```
puts [ get_name_list [ get_icl_modules ] ]
```

The innermost Tessent Shell introspection command "get_icl_modules" computes and returns a collection of all currently loaded ICL module objects. The command "get_name_list" computes

all names of the ICL objects given to it in the form of a collection. In this case, "get_name_list" computes a Tcl list, containing the names of all the ICL modules currently loaded.

For example, the Tcl lines above, if run in a dofile, would generate this transcript:

```
//  command: puts [ get_name_list [ get_icl_modules ] ]
sib1 tdr1 raw1 block1 tdr2 block2 block3 tap1 tap1_fsm chip
```

Example 2

Get all instances of a particular ICL module and print their names.

puts [get_name_list [get_icl_instances -of_module tdr*]]

This example shows the use of the wildcard character (*) when you specify ("filter") the module name for which you want all instances listed. In general, you can use regular expressions to define the filtering options for example:

-of_module [get_icl_modules <pattern> -regexp]

For example, the Tcl lines above, if run in a dofile, would generate this transcript:

```
//  command: puts [ get_name_list [get_icl_instances -of_module tdr* ] ]
block1_I1.tdr1 block1_I1.tdr2 block1_I2.tdr1 block1_I2.tdr2
block3_I1.tdr1 block3_I1.tdr2 block3_I1.tdr3 block3_I1.tdr4
```

Example 3

Get all instances of all ICL modules. Use looping to access each module and instance one after another. Explicitly use the attribute to get the name of the instance.

set moduleColl [get_icl_modules]

foreach_in_collection module $moduleColl {

    set instanceColl [get_icl_instances -of_module $module]

    foreach_in_collection instance $instanceColl {

        puts [get_attribute_value_list $instance -name name]

    }

}

This example demonstrates the usage of the foreach_in_collection looping and how to access elements in the collections. The innermost command, get_attribute_value_list, returns the "value" of the attribute "name" of the design object, which in this case is an ICL instance.

Note that this example only illustrates other introspection features, like the usage of attributes and collections. Combining the introspection of ICL modules and ICL instances of Examples 1 and 2, would result in a much more compact, and also faster running introspection of the same result:

puts [get_name_list [get_icl_instances -of_module [get_icl_modules]]]

Example 4

Report all attributes currently associated with a particular module.

report_attributes [get_icl_modules tdr1]

For example, the Tcl lines above, if run in a dofile, would generate this transcript:

```
Attribute Definition Report

Attributes defined for object 'tdr1':
Name            Value       Inheritance
--------------- ----------  -----------
is_created      false       -
is_modified     false       -
is_valid        true        -
master_name     tdr1        -
name            tdr1        -
object_type     icl_module  -
port_group_list             -
```

Using the get_icl_* command family is the correct way of accessing information about ICL objects. You must not use, for example, "get_modules tdr1" for an ICL module *tdr1*. The get_modules command is meant to access the design objects. Assume, there is a design module also named *tdr1*. Using "get_modules tdr1" would give you the introspection result for this *design object* and not for the ICL object as you intended.

Example 5

Introspect into the ICL port functions and get the name of the tck port of a particular module.

set modName tdr1

puts [get_name_list [get_icl_ports * -of_module $modName -function tck]]

This example generates a collection of ICL ports, filtered twofold. The first filter is the name of the module of interest, which in this example is stored in a Tcl variable. The second filter that is applied simultaneously with the first one is the function of the ICL port. The get_icl_ports command together with these two filters computes and returns a collection of ICL TCKPort port names of the module *tdr1*. See icl_port in the *Tessent Shell Reference Manual* for more information.

Example 6

Automatically report the invocation instance from within an iProc. Using the get_icl_scope command, you can get, among others, the ICL instance path of the ICL instance that calls the iProc.

iProc myTest { } {

iNote "iProc 'myTest' was called for ICL instance [get_icl_scope]"

}

Example 7

Report which ICL modules have been loaded. The key command here is report_icl_modules.
This command takes several optional switches. Without any switches, it reports only the names
of all loaded or extracted ICL modules. This is similar to the get_icl_modules command. In the
example below the reporting command is used to print the ICL module definition for a loaded
SIB module (the transcript is abbreviated).

ANALYSIS> report_icl_modules -modules sib

```
// instanced as block1.MyBlock1Sib
// instanced as chip4.sib1_I1
// instanced as chip4.sib1_I2
Module sib {
    // ICL module read from source on or near line 1
    // of file '../data/icl/sib.icl'
    ScanInPort si;
    ScanOutPort so { Source SIB; }
    ShiftEnPort se;
    […]
```

PDL Retargeting With Symbolic Variables

Tessent IJTAG offers the possibility to associate user-defined symbolic variables with ICL objects that are the targets of an iRead or iWrite command. The purpose of these variables is to identify the location within the pattern files, where a particular iRead or iWrite happens on the target objects.

When you specify symbolic variables during PDL retargeting, these variables appear as special annotations in the resulting patterns files together with the information required for mapping the original iRead or iWrite command to the associated data bits in the pattern vector.

These iWrite and iRead variables enable patching their values on the ATE by providing machine-readable annotations in the pattern files. Tessent Silicon Insight supports the patching of these variables natively, and some ATE companies can patch these variables using their native software.

Rule checks of the iWriteVar objects ensure they do not influence the IJTAG network's state because they are intended to be patched. Patching an IJTAG object that affects the network state invalidates the rest of the scan operations.

Specifying Symbolic Variables in PDL

Symbolic variables are identified within the PDL using a note with a special pragma string value.

To identify the target of an iRead or iWrite command to be tracked by a variable, use an iNote command with a special pragma string value:

> iNote "**tessent_pragma** *variable_type variable_name pdl_target_list* [*inversion_mask*]"

The tool applies the pragma to the object of the next iRead or iWrite command specified by the *pdl_target_list* of the iNote.

Use the following arguments to define symbolic variables:

- **tessent_pragma** — A case-insensitive keyword that indicates this iNote command serves a particular purpose specifically in the Tessent Shell. The tool does not process the iNote in the usual way.

- *variable_type* — A case-insensitive keyword that reflects the type of command to associate with the symbolic variable. Use the iReadVar keyword for the iRead command. Use the iWriteVar keyword for the iWrite command. Although the

recommended practice uses mixed-case to identify the associated command more readily, the keywords are case-insensitive.

For each variable of type iReadVar, there must be a matching iRead command before the next iApply. Likewise, for each variable of type iWriteVar, there must be a matching iWrite command before the next iApply. As long as the iApply has not been run, iReadVar specifications, iWriteVar specifications, iRead commands, and iWrite commands can appear in any order.

- *variable_name* — The name of the symbolic variable to register.

  The symbolic variable can consist of several bits. Like ports or registers, the name is hierarchical, which means the symbolic variable registers against a particular ICL instance.

  An effective prefix is the currently active instance path. It consists of the currently processed iCall instance path and the most recent iPrefix specification within the current iPrefix. The tool automatically prepends the effective prefix to the instance path of the symbolic variable.

  The instance path of the symbolic variable is independent of the instance paths of the *pdl_target_list* elements. The tool automatically considers the effective prefix so that the symbolic variables can be uniquely registered inside of iProcs that are called for several different instances of the same ICL module.

  o Optionally specify symbolic variables with a range. The range width must match the overall number of bits of all elements in the *pdl_target_list*.

  o Optionally specify symbolic variables without a range. The tool infers the range [width-1:0]. The width is the overall number of bits of the elements in the *pdl_target_list*. This is also true for PDL targets of width 1.

  > **Note**
  > There are no scalar symbolic variables.

  o After retargeting, a symbolic variable always has a range [*n*-1:0] where *n* is the largest index used in all iNotes. If you specify myvar[4:3] as your *variable_name* in an iNote and never define other iNotes for bits 2:0, the variable is registered with size [4:0] and bits [2:0] are not mapped to any ICL object.

  The mapping between the bits of a multi-bit symbolic variable and a multi-bit PDL target is from left to right, whether the range is ascending or descending. The leftmost bit of the symbolic variable matches the leftmost bit of the leftmost element in the *pdl_target_list*. The rightmost bit of the symbolic variable matches the rightmost bit of the rightmost element in the *pdl_target_list*.

  A symbolic variable name can only occur once in a pattern set. An exception is when different bits of the variable are written during different iApply commands. In this case, you can define variable XYZ[3:2] in one iNote and XYZ[1:0] in another.

- *pdl_target_list* — A comma-separated list of the iRead or iWrite that specifies the target associations of the symbolic variable. The list must not contain any whitespace characters.

  You must specify the individual elements of the *pdl_target_list* in the same way as they are specified in iRead or iWrite commands. Use [join $name_list ,] to define the list of elements over several lines. Use the lappend command to add one or more entries to the list. Use the previously mentioned join command to convert it into a comma-separated list with no whitespace.

- *inversion_mask* — A numerical PDL value that specifies whether the symbolic variable represents the unaltered values or the inverted values of the elements in the *pdl_target_list*. The value follows the PDL conventions for decimal, binary, or hexadecimal numbers.

  The *inversion_mask* is padded (on the left side) with zeros or truncated (on the left side) until its width matches the variable's width. Then, the leftmost bit (the most significant bit) of the *inversion_mask* is associated with the leftmost bit of the variable. The rightmost bit (the least significant bit) of the *inversion_mask* is associated with the rightmost bit of the variable. If a bit of the *inversion_mask* is set, it means that the corresponding bit of the symbolic variable represents the inverted value of the corresponding bit of the *pdl_target_list*.

Special Considerations for iWriteVars

Symbolic variables are potentially used to patch the patterns after the PDL retargeting. In other words, the patterns are potentially modified without undergoing the entire retargeting process again. For this reason, symbolic variables of type iWriteVar are checked to ensure that they do not interfere with any logic that substantially changes the ICL model's behavior. The ICL model must be in a known state at all times during the retargeting process. The list of ICL objects that cannot be the target of an iWriteVar are:

- Scan path reconfiguration, including Enable specifications of ScanOutPorts.

- Control of DataMux elements that are gating update_en or capture_en signals.

- Control of DataMux elements that are gating trst, tms, or reset signals.

- All elements that are referenced in a tessent_pipeline_enable attribute.

Excessive use of iWriteVar specifications can significantly impact the performance of the PDL retargeting. Symbolic variables of type iWriteVar make the IJTAG retargeting more difficult because the retargeter must assume that all of the logic involved in the iWriteVar retargeting is in an unknown state as soon as the associated iWrite becomes effective.

For debugging purposes, you can request a bit_annotation to be inserted into an output pattern file such as STIL to identify the ICL object for which each scanin value is targeted. Turn on this feature using the "set_ijtag_retargeting_options -annotation_parameter_values {scanin on}"

command. It has no performance impact. Use iWriteVar specifications only when you want to enable patching for those values on the tester.

Example

The following commands register the variables and associations shown in Table 3-1:

```
iNote "tessent_pragma iReadVar readvar inst1.dataout[7:5] 0b101"
iNote "tessent_pragma iWriteVar inst1.writevar[2:3] inst1.datain[1:0]"
iNote "tessent_pragma iReadVar readpo dout1,dout2[1:0],dout3 0xA"
```

Table 3-1. Variables and Associations

Variable	Type	PDL Target	Inversion
readvar[2]	iReadVar	inst1.dataout[7]	Yes
readvar[1]	iReadVar	inst1.dataout[6]	No
readvar[0]	iReadVar	inst1.dataout[5]	Yes
inst1.writevar[2]	iWriteVar	inst1.datain[1]	No
inst1.writevar[3]	iWriteVar	inst1.datain[0]	No
readpo[3]	iReadVar	dout1	Yes
readpo[2]	iReadVar	dout2[1]	No
readpo[1]	iReadVar	dout2[0]	Yes
readpo[0]	iReadVar	dout3	No

Retargeted Symbolic Variables

When you associate an iRead or iWrite command with a symbolic variable, the tool keeps track of when and how the command becomes effective in the IJTAG retargeting process. It augments the resulting pattern files (for example, STIL, WGL, Verilog testbench, or SVF) with suitable annotations containing all of the available information about the symbolic variable.

_____ **Note** _____

The annotations created as a result of the specification of symbolic variables for iRead and iWrite commands replace the "TESSENT_PRAGMA annotation" and "TESSENT_PRAGMA bit_annotation" comments that are typically created for all read values, and for write values when "set_ijtag_retargeting_options -annotation_parameter_values {scanin on}" is used.

The output in the pattern file is an annotation with the following content:

TESSENT_PRAGMA *var_keyword var_name* **-type** *type* **-var_bits** *bits* \
   [-var_length *length*] **-pin** *pin* **[-relative_cycles** *cycles*] **[-inversion** *inversion*] \
   {[-last_cycle_correction *value*] | **[-first_cycle_correction** *value*]}**

- *var_keyword* — The keyword "variable" or "bit_variable".

  The "variable" keyword appears at the start of an operation, such as scan loads or vectors. The tool collects and inserts all "TESSENT_PRAGMA variable …" annotations that belong to one scan load before the first shift vector of this scan load.

  The "bit_variable" keyword appears for the individual vectors that result from the unrolling of a scan load and for non-scan load vectors. It appears together with the vector that contains the related drive state or expected state. The -relative_cycles switch is not used in this case.

  The tool uses "variable" annotations to identify all bits of a symbolic variable. The "bit_variable" annotations help your review of the pattern file. Tool automation does not use the "bit_variable" annotations. Refer to the Example for details of how the information in the "variable" annotation is all that is required to identify all bits of a symbolic variable properly.

- *var_name* — The name of the symbolic variable without any index and prefixed by the ICL instance name against which the PDL defining the symbolic variable was called.

  The *var_name* specification does not contain indices or ranges, even if the symbolic variable is declared with several bits in the PDL where it was introduced. The *var_name* contains the base name of the symbolic variable only.

  For example, when myVar is a symbolic variable name defined in an iProc that is called against the u1.my_ip_i1 ICL instance, the effective name is u1.my_ip_i1.myVar. Refer to Figure 3-6, Figure 3-8, and Figure 3-11 for precise examples of this naming convention.

- *type* — The keyword "read" or "write" defines the type of the symbolic variable.

- **-var_bits** *bits* — Specifies which parts of the symbolic variable this TESSENT_PRAGMA refers to. *bits* is a space-separated list of indices or ranges.

  For example, "-var_bits {15 12:10 3:2 0}" means that a scan load accesses the bits 15, 12 down to 10, 3 down to 2, and 0. Typically, a single scan load accesses the entire symbolic variable. However, you can define a symbolic variable for ICL objects that cannot scan in the same scan load.

- **-var_length** *length* — Specifies the *length* of the symbolic variable. This specification is optional. When omitted, the *length* is extracted from the -var_bits *bits*.

  The *length* is the entire width of the symbolic variable, whereas *bits* lists the bits to access during the current scan load.

The indices of a symbolic variable are [*n*-1:0], where *n* is the var_length. The *bits* referenced with the -var_bits switch always refer to indices in the [*n*-1:0] range. It is possible that a scan load only accesses a subset of the bits of a symbolic variable. When all bits are not accessed in a single scan load, the *length* is specified in the annotation associated with the first scan load accessing one or more bits of the symbolic variable.

- **-pin** *pin* — The top-level port through which the symbolic variable is read from or written to. For read variables observed through scan registers, the referenced *pin* is the ScanOutPort used to unload the register. For write variables defined on scan registers, the referenced *pin* is the related ScanInPort used to load the register.

- **-relative_cycles** *cycles* — For scan loads, the *cycles* offset in which the iRead target observations happen at the ScanOutPort, or the vector offset in which the iWrite target values must be applied to the ScanInPort.

  Like -var_bits *bits*, *cycles* is a list that can contain a mixture of indices and ranges. The number of entries in the *cycles* list always matches the number of *bits*. The relative cycle is relative to the position of the annotation placed at the beginning of the scan load used to load or unload bits of a symbolic variable.

  When the pattern is written out with a tck_ratio larger than 1, you must consider the specified tck_ratio value when converting relative cycles into actual test_cycle or vector locations. The section How to Patch a Symbolic Variable in the ATE memory and the section How to Map a Miscompare on the ATE to an iReadVar describes this in detail.

- **-inversion** *inversion* — A binary number that specifies the *inversion* between the symbolic variable and the value that is observed (or set) at the associated *pin*. The width of the binary number matches the number of indices in -var_bits *bits*. The leftmost bit of inversion refers to the leftmost index in *bits*, the rightmost bit of inversion refers to the rightmost index in *bits*.

- **-last_cycle_correction** *value* — For scan loads, *value* specifies an offset in the number of tester cycles to apply to the highest "-relative_cycles" value to obtain the correct tester cycle for the corresponding iRead or iWrite action. The use of this parameter is mutually exclusive with "-first_cycle_correction".

- **-first_cycle_correction** *value* — For scan loads, *value* specifies an offset in the number of tester cycles to apply to the lowest "-relative_cycles" value to obtain the correct tester cycle for the corresponding iRead or iWrite action. The use of this parameter is mutually exclusive with "-last_cycle_correction".

How to Patch a Symbolic Variable in the ATE memory

Follow these steps to patch an ATE pattern to change the value of a symbolic variable. The Example illustrates these steps in detail.

1. Identify the currently used tck_ratio value using the "TESSENT_PRAGMA vector_set" annotation. Refer to Figure 3-11 and Figure 3-12 for examples.

2. Identify all occurrences of the "TESSENT_PRAGMA variable *var_name*" annotation for the symbolic variable to patch. Refer to Figure 3-15 for an example. For iWriteVar, it is possible that the values for some or all bits are repeated several times. The Example illustrates this with the seed_var iWriteVar.

3. Identify the vector_id associated to the annotation using the "Pattern:# Vector:# TesterCycle:#" annotation above it. Refer to Figure 3-15 for an example.

4. Extract the associated relative offset specified in the -relative_cycles switch for each bit specified in the -var_bits switch.

5. Extract the possible cycle_correction value from the "TESSENT_PRAGMA variable *var_name*" annotation. If the -first_cycle_correction switch is given, the range start must be updated. If the -last_cycle_correction switch is given, the range end must be updated.

6. Compute the vector range to patch using these equations:

```
range_start = vector_id_of_annotation +
              (relative_offset_of_bit * tck_ratio) +
              first_cycle_correction

range_end = vector_id_of_annotation +
            (relative_offset_of_bit * tck_ratio) +
            (tck_ratio - 1) + last_cycle_correction
```

The reason the patch corresponds to a range of vectors when tck_ratio is greater than one is explained in the paragraph located above Figure 3-16.

How to Map a Miscompare on the ATE to an iReadVar

Follow these steps to map a miscompare on the ATE to an iReadVar. The Example illustrates these steps in detail.

1. Identify the currently used tck_ratio value using the "TESSENT_PRAGMA vector_set" annotation. Refer to Figure 3-11 and Figure 3-12 for examples.

2. Identify all occurrences of the "TESSENT_PRAGMA variable *var_name*" annotation for the symbolic variable to map miscompares to. Refer to Figure 3-13 for an example.

3. Identify the tester cycle id associated with the annotation using the "Pattern:# Vector:# TesterCycle:#" annotation above it. Refer to Figure 3-13 for an example.

4. Extract the associated relative offset specified in the -relative_cycles switch for each bit specified in the -var_bits switch.

5. If the bit corresponds to the first bit of the variable, and if the "TESSENT_PRAGMA variable *var_name*" statement has the -first_cycle_correction switch set, extract the offset from the switch. Alternately, if the bit corresponds to the last bit of the variable, and if the "TESSENT_PRAGMA variable *var_name*" statement has the -last_cycle_correction switch set, extract the offset from the switch. In all other cases, ignore the value of the switch.

6. Compute the tester cycle id associated to each bit of the symbolic variable using this equation, where cycle_correction is the value determined by the previous step:

```
tester_cycle_id_of_annotation +
  (relative_offset_of_bit * tck_ratio) +
  cycle_correction
```

A miscompare happening on the given pin at the specific tester cycle id correspond to a miscompare of the given bit of the iReadVar. This process is clearly illustrated in the paragraphs above Figure 3-14.

Example

This example illustrates the steps to define Read and Write symbolic variables into the PDL and shows how they are mapped into the STIL file annotation. The methods used to patch a symbolic variable are explained in the How to Patch a Symbolic Variable in the ATE memory section. The methods used to map miscompares to iReadVars are explained in the How to Map a Miscompare on the ATE to an iReadVar section. This example illustrates both in detail using a pattern generated for tck_ratio of 1 and 4.

Figure 3-5 shows the ICL that describes the IJTAG element of a simple instrument. Its IJTAG ScanInterface provides access to a 10-bit scan chain composed of two scan registers: enable and cnt_int[8:0]. Two alias names, gonogo and fail_cnt[4:0], are mapped to bits of the same register. The mapping between the fail_cnt alias and the cnt_init register illustrates a more complex mapping to illustrate how the -relative_cycles switch appears when the symbolic variable bits do not come out in order. This is not typical in well-designed instruments, but it is a valid possibility for general-purpose software to handle.

Figure 3-5. ICL Description of the my_ip Instrument

```
Module my_ip {
   TCKPort ijtag_clock;
   ResetPort ijtag_reset {
     ActivePolarity 0;
   }
   CaptureEnPort ijtag_ce;
   ShiftEnPort   ijtag_se;
   UpdateEnPort  ijtag_ue;
   SelectPort    ijtag_sel;
   ScanInPort    ijtag_si;
   ScanOutPort   ijtag_so {
     Source cnt_init[0];
   }

   ScanRegister enable {
     CaptureSource enable;
     ResetValue 1'b0;
     ScanInSource ijtag_si;
   }
   ScanRegister cnt_init[8:0] {
     ScanInSource enable;
   }
   Alias gonogo = ~cnt_init[0];
   Alias fail_cnt[4:0] = cnt_init[4],cnt_init[6],
                         cnt_init[3],cnt_init[8],cnt_init[7];
}
```

Next, an iProc for the given instrument is shown in Figure 3-6. The iProc is used to run a given test on its associated instrument. It includes four special iNotes that define two iWriteVars and two iReadVars.

The first symbolic variable, called seed_var, is of type write and is defined on the cnt_init ScanRegister defined in the ICL. Because no indices are used for cnt_init, all register bits are referenced, that is cnt_init[8:0].

The second symbolic variable of type write is called enable_var and is defined on the ICL ScanRegister named enable. The last two symbolic variables are of type read and are mapped to the aliases defined in the associated ICL.

Notice the iApply command between the iWrite on the cnt_init register and the iWrite on the enable register. This iApply instructs the IJTAG retargeter to load the seed with a scan load and then load the enable with another scan load. Because both registers are part of the same IJTAG scan chain, the cnt_init register is scanned in again with the same value during the second scan load used to set the enable to 1.

This is the reason the annotation for the seed_var iWriteVar appears twice in the pattern. When you patch the seed_var value in the first scan load, you must also patch it for the second scan load. The green annotation in Figure 3-11 shows how they repeat for each instance of my_ip.

Figure 3-6. An iProc Called run_test for my_ip Instrument

```
iProcsForModule my_ip
iProc run_test {seed wait_cycles} {
  iNote "tessent_pragma iWriteVar seed_var cnt_init"
  iWrite cnt_init $seed
  iApply
  iNote "tessent_pragma iWriteVar enable_var enable"
  iWrite enable 1
  iApply
  iRunLoop $wait_cycles
  iWrite cnt_init 0
  iWrite enable 0
  iNote "tessent_pragma iReadVar status_var fail_cnt[4:0]"
  iRead fail_cnt 0
  iNote "tessent_pragma iReadVar GoNogo_var gonogo"
  iRead gonogo 1
  iApply
}
```

Figure 3-7 illustrates an ICL network containing three my_ip instrument instances. Notice how the scan path including the my_ip_i1 instance is mutually exclusive to the scan path that includes the other two instances, my_ip_i2 and my_ip_i3. This not how to build an optimal network but it illustrates the mapping of symbolic variables when they cannot all be accessed during the same scan load.

Figure 3-7. ICL Network with Three Instances of my_ip
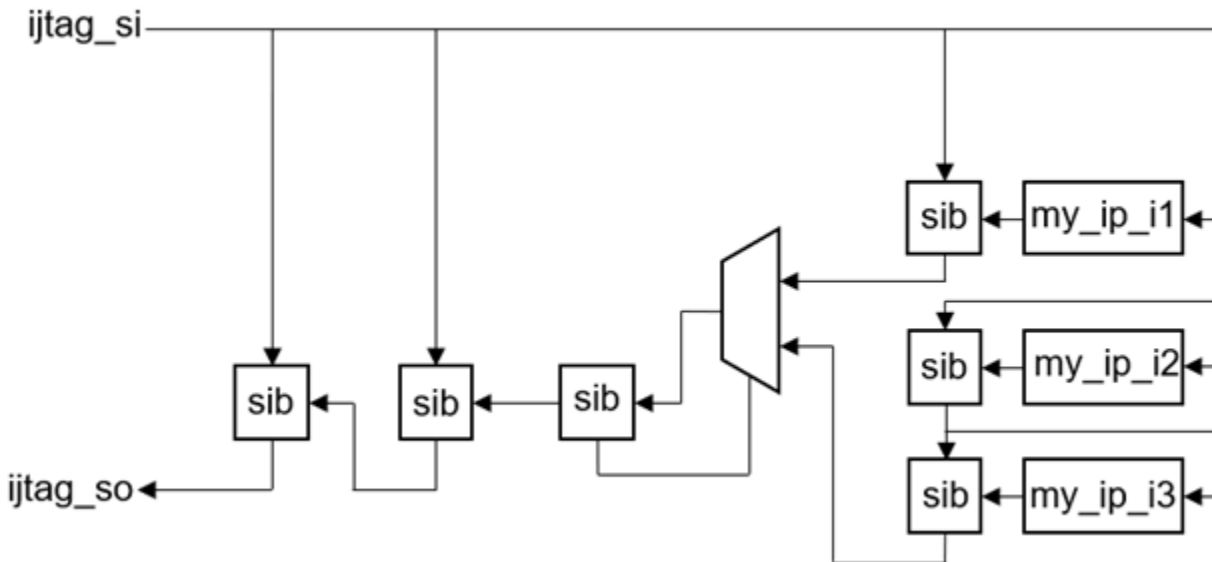
Figure 3-8 shows the PDL to concurrently run the test on all three instances of the "my_ip" instrument. Notice how the run_test iProc in Figure 3-6 is called on the three instances. For example, the my_ip_i1 instance uses the arguments 80 and 120. Each call uses different arguments. The iCall commands are within an iMerge block to instruct the IJTAG retargeter to run the tests in parallel and not in series.

The open_pattern_set command uses the -tck_ratio switch with its default value of one. The -tck_ratio switch describes how each TCK cycle is created. When this is set to one, TCK is defined as an RZ clock, and each TCK cycle consumes one vector on the ATE.

When tck_ratio is specified to *n*, TCK gets defined as an NRZ signal, and it is forced low for half of the *n*/2 vectors and force high for the other half. Each TCK cycle thus consumes *n* vectors on the ATE.

An annotation describes the specified tck_ratio value, affecting how to interpret the relative_cycles. In the case of an iWrite variable, tck_ratio affects how many vectors you must modify for a single bit of a symbolic variable.

Refer to the How to Patch a Symbolic Variable in the ATE memory section for more information about the patching process. Continue with this example for a complete illustration of the patching process in the presence of a tck_ratio > 1.

Figure 3-8. PDL to Run the Test on Three Instances of my_ip

```
open_pattern_set run_test -tck_ratio 1
  iMerge -begin
    iCall my_ip_i1.run_test 80 120
    iCall my_ip_i2.run_test 90 120
    iCall my_ip_i3.run_test 70 120
  iMerge -end
close_pattern_set
```

The IJTAG retargeter optimizes the scan loads required to achieve the requested operations. It opens the scan path to include my_ip_i1 and loads the seed value, as shown in Figure 3-9. It then opens the scan path to include my_ip_i2 and my_ip_i3 and loads their seed values, as shown in Figure 3-10. Then, as the run_test iProc of Figure 3-6 instructs, it must repeat the two operations to set the enable bit high. The fail_cnt and gonogo status bits are read from the three instruments after the specified number of TCK cycles.

Figure 3-9. Scan Path to Access my_ip_i1
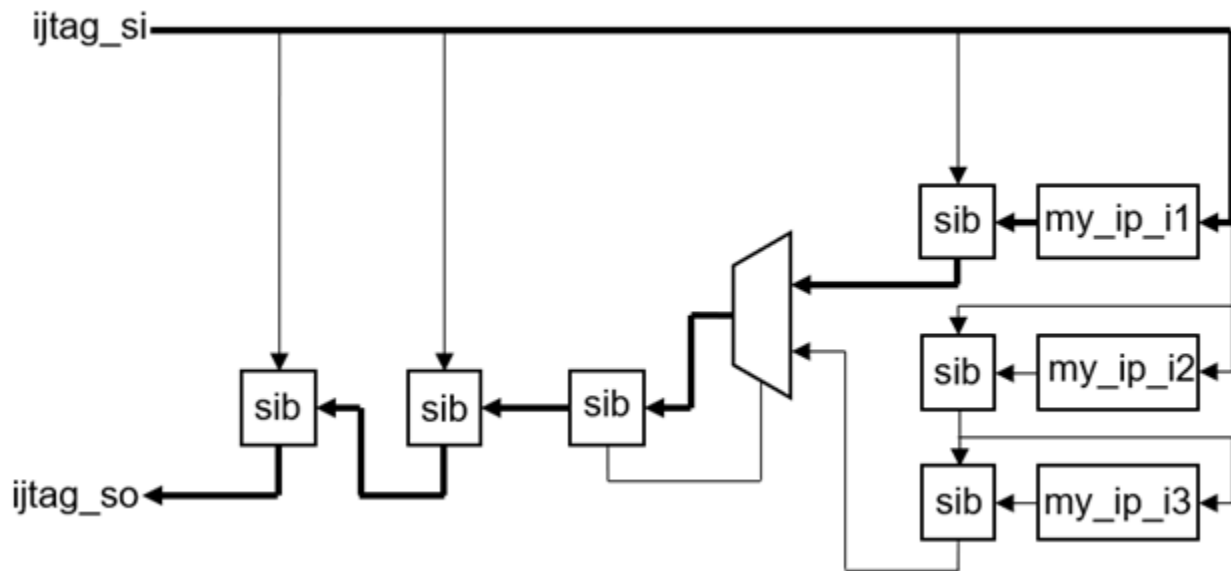
Figure 3-10. Scan Path to Access my_ip_i2 and my_ip_i3
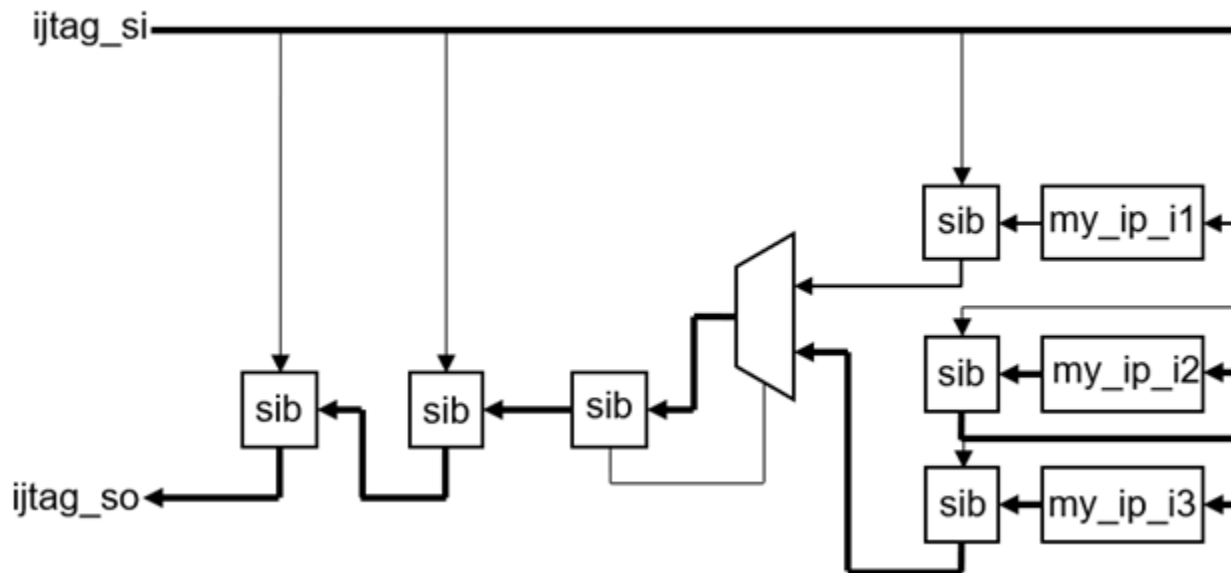
Figure 3-11 shows the relevant annotations extracted from the STIL file. Notice that each vector starts with an annotation that identifies its vector id and its tester cycle id. The vector id is used for patching the vector values. The tester cycle id is used to back map a miscompare to an iReadVar.

At the start of the pattern, there is an annotation that lists the pattern_set name with an optional -tck_ratio switch. In this example, the -tck_ratio switch is not present because it has a value of 1.

Figure 3-12 shows the same annotations when the PDL in Figure 3-8 is retargeted with a tck_ratio value of 4. Compare the annotations in both cases. The "TESSENT_PRAGMA variable" annotations are identical. The first difference is the "TESSENT_PRAGMA pattern_set" annotation. The term "-tck_ratio 4" is present to specify the tck_ratio to a value larger than one.

The second difference is that the vector id and tester cycle id annotations, which identify the location of the "TESSENT_PRAGMA variable" annotations, are now all multiplied by 4. In this example, iProc uses an iRunLoop -tck, which explains why everything got stretched by tck_ratio.

If the PDL includes -iRunLoops with either the -sck or the -time switch, the loop is not required to be a multiple of tck_ratio. In this case, the vector ids and tester cycle ids are not always required to be a multiple of tck_ratio. Regardless, they represent the location when and where the "TESSENT_PRAGMA variable" annotations reside on the ATE.

Figure 3-11. Generated "pattern_set" and "variable" Annotations (tck_ratio = 1)

```
Ann {* Pattern:0  Vector:0  TesterCycle:0 *}
Ann {* TESSENT_PRAGMA pattern_set run_test *}

Ann {* Pattern:0  Vector:16  TesterCycle:16 *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {12:4} *}

Ann {* Pattern:0  Vector:44  TesterCycle:44 *}
Ann {* TESSENT_PRAGMA variable my_ip_i2.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {13:5} *}
Ann {* TESSENT_PRAGMA variable my_ip_i3.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {24:16} *}

Ann {* Pattern:0  Vector:74  TesterCycle:74 *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.enable_var -type write -var_bits
{0} -pin ijtag_si -relative_cycles {13} *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {12:4} *}

Ann {* Pattern:0  Vector:92  TesterCycle:92 *}
Ann {* TESSENT_PRAGMA variable my_ip_i2.enable_var -type write -var_bits
{0} -pin ijtag_si -relative_cycles {14} *}
Ann {* TESSENT_PRAGMA variable my_ip_i2.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {13:5} *}
Ann {* TESSENT_PRAGMA variable my_ip_i3.enable_var -type write -var_bits
{0} -pin ijtag_si -relative_cycles {25} *}
Ann {* TESSENT_PRAGMA variable my_ip_i3.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {24:16} *}

Ann {* Pattern:0  Vector:124  TesterCycle:243 *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.status_var -type read -var_bits
{4:0} -var_length 5 -pin ijtag_so -relative_cycles {8 10 7 12:11} *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.GoNogo_var -type read -var_bits
{0} -pin ijtag_so -relative_cycles {4} -inversion 0b1 *}

Ann {* Pattern:0  Vector:142  TesterCycle:261 *}
Ann {* TESSENT_PRAGMA variable my_ip_i3.status_var -type read -var_bits
{4:0} -var_length 5 -pin ijtag_so -relative_cycles {20 22 19 24:23} *}
Ann {* TESSENT_PRAGMA variable my_ip_i3.GoNogo_var -type read -var_bits
{0} -pin ijtag_so -relative_cycles {16} -inversion 0b1 *}
Ann {* TESSENT_PRAGMA variable my_ip_i2.status_var -type read -var_bits
{4:0} -var_length 5 -pin ijtag_so -relative_cycles {9 11 8 13:12} *}
Ann {* TESSENT_PRAGMA variable my_ip_i2.GoNogo_var -type read -var_bits
{0} -pin ijtag_so -relative_cycles {5} -inversion 0b1 *}
```

Figure 3-12. Generated "pattern_set" and "variable" Annotations (tck_ratio = 4)

```
Ann {* Pattern:0  Vector:0  TesterCycle:0 *}
Ann {* TESSENT_PRAGMA pattern_set run_test -tck_ratio 4*}

Ann {* Pattern:0  Vector:64  TesterCycle:64 *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {12:4} *}

Ann {* Pattern:0  Vector:176  TesterCycle:176 *}
Ann {* TESSENT_PRAGMA variable my_ip_i2.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {13:5} *}
Ann {* TESSENT_PRAGMA variable my_ip_i3.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {24:16} *}

Ann {* Pattern:0  Vector:296  TesterCycle:296 *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.enable_var -type write -var_bits
{0} -pin ijtag_si -relative_cycles {13} *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {12:4} *}

Ann {* Pattern:0  Vector:368  TesterCycle:368 *}
Ann {* TESSENT_PRAGMA variable my_ip_i2.enable_var -type write -var_bits
{0} -pin ijtag_si -relative_cycles {14} *}
Ann {* TESSENT_PRAGMA variable my_ip_i2.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {13:5} *}
Ann {* TESSENT_PRAGMA variable my_ip_i3.enable_var -type write -var_bits
{0} -pin ijtag_si -relative_cycles {25} *}
Ann {* TESSENT_PRAGMA variable my_ip_i3.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {24:16} *}

Ann {* Pattern:0  Vector:496  TesterCycle:972 *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.status_var -type read -var_bits
{4:0} -var_length 5 -pin ijtag_so -relative_cycles {8 10 7 12:11} *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.GoNogo_var -type read -var_bits
{0} -pin ijtag_so -relative_cycles {4} -inversion 0b1 *}

Ann {* Pattern:0  Vector:568  TesterCycle:1044 *}
Ann {* TESSENT_PRAGMA variable my_ip_i3.status_var -type read -var_bits
{4:0} -var_length 5 -pin ijtag_so -relative_cycles {20 22 19 24:23} *}
Ann {* TESSENT_PRAGMA variable my_ip_i3.GoNogo_var -type read -var_bits
{0} -pin ijtag_so -relative_cycles {16} -inversion 0b1 *}
Ann {* TESSENT_PRAGMA variable my_ip_i2.status_var -type read -var_bits
{4:0} -var_length 5 -pin ijtag_so -relative_cycles {9 11 8 13:12} *}
Ann {* TESSENT_PRAGMA variable my_ip_i2.GoNogo_var -type read -var_bits
{0} -pin ijtag_so -relative_cycles {5} -inversion 0b1 *}
```

Mapping a Miscompare to a Bit of an iReadVar

When backmapping a miscompare to an iReadVar, you must identify the tester cycle id associated to each bit of the iReadVar. Refer to the section How to Map a Miscompare on the ATE to an iReadVar, which describes the steps to map a miscompare.

One example of an iReadVar annotation from Figure 3-11 and Figure 3-12 can illustrate the process both with and without a tck_ratio. Figure 3-13 shows the selected annotations. The first

example does not specify the -tck_ratio switch in its "TESSENT_PRAGMA pattern_set" annotation, which means tck_ratio is 1. The second example uses tck_ratio = 4.

Figure 3-13. iReadVar Extracted from the tck_ratio = 1 and 4 Examples

```
// With tck_ratio = 1
Ann {* TESSENT_PRAGMA pattern_set run_test *}
Ann {* Pattern:0  Vector:124  TesterCycle:243 *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.status_var -type read -var_bits
{4:0} -var_length 5 -pin ijtag_so -relative_cycles {8 10 7 12:11} *}

// With tck_ratio = 4
Ann {* TESSENT_PRAGMA pattern_set run_test -tck_ratio 4*}
Ann {* Pattern:0  Vector:496  TesterCycle:972 *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.status_var -type read -var_bits
{4:0} -var_length 5 -pin ijtag_so -relative_cycles {8 10 7 12:11} *}
```

The "TESSENT_PRAGMA variable" annotations are identical in both cases. The symbolic variable has 5 bits. Bit 4 is sampled by the vector that is 8 × tck_ratio vectors after where the TESSENT_PRAGMA variable" annotation is located.

For the case with tck_ratio = 1, to find the vector sampling bit 4 of the status_var:

- 124 + 8 = 132

For the case with tck_ratio = 4, to find the vector sampling bit 4 of the status_var:

- 496 + 8 × 4 = 528

Figure 3-14 shows the annotations and the vector present at that location in the tck_ratio = 1 and the tck_ratio = 4 cases.

Notice the "TESSENT_PRAGMA bit_variable" annotations at those locations. The tool ignores them. They help you read the STIL file. The TesterCycle value provides the tester cycle id to map a miscompare on the specified pin to the given bit of the iReadVar. The expected value for the iReadVar is extracted from the _po_ value of that vector.

Figure 3-14. Annotation at Referenced Vector for tck_ratio = 1 and 4

```
// With tck_ratio = 1
Ann {* Pattern:0  Vector:132  TesterCycle:251 *}
Ann {* TESSENT_PRAGMA bit_variable my_ip_i1.status_var -type read
-var_bits {4} -pin ijtag_so -inversion 0b0 *}
V {_pi_=1101010; _po_=L; }

// With tck_ratio = 4
Ann {* Pattern:0  Vector:528  TesterCycle:1004 *}
Ann {* TESSENT_PRAGMA bit_variable my_ip_i1.status_var -type read
-var_bits {4} -pin ijtag_so -inversion 0b0 *}
V {_po_=L;}
```

Patching a Bit of an iWriteVar

When patching a bit of an iWriteVar, you must identify the vectors associated with that bit. Refer to the section How to Patch a Symbolic Variable in the ATE memory, which describes the steps to patch a bit.

One example of an iWriteVar annotation from Figure 3-11 and Figure 3-12 illustrates the process both with and without a tck_ratio. The selected annotations are in Figure 3-15. The first example does not specify the -tck_ratio switch in its "TESSENT_PRAGMA pattern_set" annotation, which means tck_ratio is 1. The second example uses tck_ratio = 4.

Figure 3-15. iWriteVar Extracted from the tck_ratio = 1 and 4 Examples

```
// With tck_ratio = 1
Ann {* TESSENT_PRAGMA pattern_set run_test *}
Ann {* Pattern:0  Vector:16  TesterCycle:16 *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {12:4} *}

Ann {* Pattern:0  Vector:74  TesterCycle:74 *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {12:4} *}

// With tck_ratio = 4
Ann {* TESSENT_PRAGMA pattern_set run_test_ratio4 -tck_ratio 4 *}
Ann {* Pattern:0  Vector:64  TesterCycle:64 *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {12:4} *}

Ann {* Pattern:0  Vector:296  TesterCycle:296 *}
Ann {* TESSENT_PRAGMA variable my_ip_i1.seed_var -type write -var_bits
{8:0} -pin ijtag_si -relative_cycles {12:4} *}
```

The "TESSENT_PRAGMA variable" annotations are identical and are repeated in both cases. Refer to the paragraph before Figure 3-6 describing the iProc with an iApply between seed and enable iWrites. Both registers are part of the same scan chain, so the seed values are scanned in twice. You must patch the seed value each time it is scanned in.

The -var_length switch is not specified, meaning that the specified -var_bits make up the entire symbolic variable. The scanin value for bit 8 is located in the vector that is $12 \times$ tck_ratio vectors after each occurrence where the "TESSENT_PRAGMA variable" annotation is located.

For the case with tck_ratio = 1, to find the vectors loading bit 8 of the seed_var:

- $16 + 12 = 28$

- $74 + 12 = 86$

For the case with tck_ratio = 4, to find the vectors loading bit 8 of the seed_var:

- $64 + 12 \times 4 = 112$

- $64 + 12 \times 4 + (4 - 1) = 115$

- $296 + 12 \times 4 = 344$

- $296 + 12 \times 4 + (4 - 1) = 347$

Together, these evaluate into the patching vectors 112:115 and 344:347. When tck_ratio is larger than one, each scanin bit is duplicated by the tck_ratio value, so you must patch a range of vectors in that case.

Figure 3-16 shows the annotations and vectors present at those locations for the two cases (tck_ratio = 1 and tck_ratio = 4).

Notice the "TESSENT_PRAGMA bit_variable" annotation at those locations. The tool ignores them. They help you read the STIL file. The SignalGroup definition for the _pi_ signal group shows which vector bit corresponds to the ijtag_si pin.

Figure 3-16. Annotation at Referenced Vector for tck_ratio = 1 and 4

```
SignalGroups {
  _pi_ = '"ijtag_tck" + "ijtag_reset" + "ijtag_ce" + "ijtag_se" +
"ijtag_ue" + "ijtag_sel" + "ijtag_si"';
}

// With tck_ratio = 1
Ann {* Pattern:0  Vector:28  TesterCycle:28 *}
Ann {* TESSENT_PRAGMA bit_variable my_ip_i1.seed_var -type write -var_bits
{8} -pin ijtag_si -inversion 0b0 *}
V { _pi_=1101010; }

Ann {* Pattern:0  Vector:86  TesterCycle:86 *}
Ann {* TESSENT_PRAGMA bit_variable my_ip_i1.seed_var -type write -var_bits
{8} -pin ijtag_si -inversion 0b0 *}
V { _pi_=1101010; }

// With tck_ratio = 4
Ann {* Pattern:0  Vector:112  TesterCycle:112 *}
Ann {* TESSENT_PRAGMA bit_variable my_ip_i1.seed_var -type write -var_bits
{8} -pin ijtag_si -inversion 0b0 *}
V { _pi_=0101010; }
Ann {* Pattern:0  Vector:113  TesterCycle:113 *}
V { _pi_=1101010; }
Ann {* Pattern:0  Vector:114  TesterCycle:114 *}
V { _pi_=1101010; }
Ann {* Pattern:0  Vector:115  TesterCycle:115 *}
V { _pi_=0101010; }

Ann {* Pattern:0  Vector:344  TesterCycle:344 *}
Ann {* TESSENT_PRAGMA bit_variable my_ip_i1.seed_var -type write -var_bits
{8} -pin ijtag_si -inversion 0b0 *}
V { _pi_=0101010; }
Ann {* Pattern:0  Vector:345  TesterCycle:345 *}
V { _pi_=1101010; }
Ann {* Pattern:0  Vector:346  TesterCycle:346 *}
V { _pi_=1101010; }
Ann {* Pattern:0  Vector:347  TesterCycle:347 *}
V { _pi_=0101010; }
```

Symbolic Variables Specific to Boundary Scan Patterns

During the boundary scan extraction flow, the tool writes a set of annotations to the STIL pattern set to identify the boundary scan register. You can use these annotations to debug simulation failures.

The output in the pattern file are annotations that start with the following content:

TESSENT_PRAGMA boundary_scan *bscan_reg_path* -begin

- ***bscan_reg_path*** — This is the pathname of the boundary scan register. The -begin keyword marks the beginning of the register.

This annotation is followed by one or more of the following:

TESSENT_PRAGMA boundary_scan_bit -relative_cycles *cycle_num* -top_port *port_name*

- ***cycle_num*** — The position of the boundary scan bit in the boundary scan register.

- ***port_name*** — The name of the top-level port associated with the boundary scan bit.

The output ends with the following content:

TESSENT_PRAGMA boundary_scan *bscan_reg_path* -end

- ***bscan_reg_path*** — This is the path name of the boundary scan register. It matches the name given in the -begin annotation. The -end keyword marks the end of the register.

To turn this behavior off, set the annotation parameter "pragma_bscan_annotation" to off with the -annotation_parameter_values switch of the set_ijtag_retargeting_options command.

The following is an example of annotations identifying a boundary scan register consisting of six bits:

```
TESSENT_PRAGMA boundary_scan top_bscan_interFace_I.BScanReg -begin
TESSENT_PRAGMA boundary_scan_bit -relative_cycles 0 -top_port io_0
TESSENT_PRAGMA boundary_scan_bit -relative_cycles 1 -top_port io_1
TESSENT_PRAGMA boundary_scan_bit -relative_cycles 2 -top_port io_2
TESSENT_PRAGMA boundary_scan_bit -relative_cycles 3 -top_port io_3
TESSENT_PRAGMA boundary_scan_bit -relative_cycles 4 -top_port io_4
TESSENT_PRAGMA boundary_scan_bit -relative_cycles 5 -top_port io_5
TESSENT_PRAGMA boundary_scan top_bscan_interFace_I.BScanReg -end
```

iApply Retargeting in the IJTAG Engine

The IJTAG engine retargets the queued operations of an iApply command to the active boundary of the IJTAG network. Each iApply command is run and optimized separately.

The IJTAG engine attempts to find efficient ways to meet all the iApply requirements and automatically optimizes the retargeted solution based on the following optimality constraints. Each optimization step incrementally builds and improves upon the result of the prior step.

- **initial_solution step —** The first optimization step satisfies the requirements of the iApply with the fewest individual scan loads and primary input/output (PIO) operations as possible. By default, the tool attempts to solve the iApply within a 64 scan load and PIO operation limit. In the case where the iApply cannot be satisfied, the tool reports it reached the 64 operation limit and found no solution. You can increase the default limit with the "set_ijtag_retargeting_options -max_operations_per_iapply" command and switch.

- **scan_cycles step —** This step optimizes the solution to use the fewest scan cycles possible across all scan loads. The initial solution might traverse parts of the network that were not required, or those that run operations in non-ideal sequences.

  The tool attempts to reduce the number of required scan cycles, which can be costly in complex IJTAG networks, especially for large iApply blocks. Therefore, the optimization strategically divides these blocks into smaller portions, which greatly reduces computational complexity but may not always find an optimal result. You can eliminate the dividing of the scan cycle organization with the "set_ijtag_retargeting_options -divide_scan_cycle_optimization off" command and switch setting.

  _____**Note**_____

  Eliminating the dividing of the scan cycle organization may result in a significant increase in processing time for the iApply.

- **mux_select_penalty step —** The third step optimizes which input of a ScanMux on the active scan is used. This is done only if multiple inputs of the ScanMux have identical sources and lengths. See "How to Model a ScanMux Selection Preference" for additional information.

- **mux_break_scan_path step —** The final step examines all ScanMuxes in the IJTAG network, including those not in the active scan path, and attempts to select ScanMux inputs that do not break the scan path. This step is necessary to avoid deadlocks in highly complex and interdependent designs, where an unexpected ScanMux selection could result in a deadlock in an unrelated part of the network. Such a deadlock may result in unsolvable iApply blocks later on.

After completing the four optimization steps, the IJTAG engine determines all care bits in the solution, stores them in the pattern set, and is ready for the next iApply.

iApply Partitioning

Depending on the complexity of the IJTAG network and the number of operations queued for the iApply, it can be difficult for the IJTAG engine to find a solution for all constraints of an iApply at the same time. This is especially true for star topology IJTAG networks, where accessing one part of the network blocks access to other parts.

To help overcome this complexity, the tool has the capability to internally partition an iApply into multiple smaller portions that can be solved individually. This partitioning happens before any of the optimization steps described in the previous section. Each portion goes through the full sequence of optimization steps. The tool ensures that the final iApply end state satisfies all requirements of the original iApply.

By default, the partitioning is conservative and only splits an iApply if the operations of the iApply absolutely cannot be performed in the same scan load. This ensures the quality of the solution is not negatively impacted. However, the algorithm might also fail to adequately split a large iApply into smaller portions, resulting in long retargeting times. In this case, you can use the "set_ijtag_retargeting_options -physical_block_iapply_partitioning" command and switch to have the tool always partition iApply blocks based on the physical block structure of the network.

How to Run iCalls in Parallel

You can instruct the PDL retargeter to try to run PDL commands in parallel.

The PDL iMerge command instructs the PDL retargeter to try to run subsequent PDL commands in parallel. The IEEE 1687 document describes iMerge only as a desired option to a PDL retargeter. The retargeter is not required to parallelize any or all of the PDL commands. The PDL retargeter might identify that some of the PDL commands cannot be run in parallel and chooses to serialize them instead. Overall, the order of the parallel run is not determined by the standard. Instead, the order is determined by the application tool.

The following is a brief description of the commands iMerge, iTake, and iRelease. The iMerge command is used to specify the beginning and the end of a so-called "merge block", that is, a set of iCall commands that are meant to be processed in parallel in the final representation of the test patterns. The syntax is as follows:

```
iMerge -begin
    iCall [<instPath>.]<proc> [<args>…]
    iCall [<instPath>.]<proc> [<args>…]
    iCall [<instPath>.]<proc> [<args>…]
    …
iMerge -end
```

iCall is the only PDL command that is allowed in between iMerge -begin and iMerge -end.

iTake can be used inside of an iProc to reserve a "resource" (a port, a register, or an instance) for exclusive use with this iProc. None of the other parallel running iProcs is allowed to alter states or clock frequencies on the resources taken by an iProc. The reservation persists until the end of the iProc. iTake uses the following syntax:

```
iProc <name> {<args>… } {
iTake <resourceIdentifier>
…
```

- iRelease explicitly undoes a reservation done by iTake.

- iMerge can be called recursively, that is, an iProc called by an iCall command inside of a merge block can also contain a merge block.

- iProcs inherit the reservations of their callers.

- iProcs can release the reservations of their callers by using iRelease.

The implied release of all resources at the end of the iProc does not release the resources of the caller if there is (on purpose or by accident) an intersection of resources.

PDL Specialties and Exceptions

Tessent IJTAG generates an error or warning message when certain situations occur due to erroneous user input.

The following situations result in an error or warning:

- Unprocessed iWrite/iRead/iScan targets are not allowed at the end of an iProc that is called from within an iMerge block.

- Missing "iMerge -end" at the end of iProc is not allowed.

- Missing "iMerge -end" at close_pattern_set is not allowed.

- iApply -end_in_pause is not allowed in iMerge threads.

- iReset is not allowed in iMerge threads.

iMerge Conflict Reporting

In a merge block, all PDL commands between the -begin and -end of iMerge are processed in parallel as much as possible.

By default, Tessent IJTAG identifies any resource conflicts, such as two PDL commands writing to the same register, and all conflicts arising from reservations done by the iTake command. All conflicting commands are then processed serially. For the remaining iCall commands, Tessent IJTAG tries to find an optimal parallel solution.

Processing conflicting commands serially can cause the settings associated with the first processed task to be overwritten by subsequently processed tasks. In some situations, such as those caused by erroneous user input, the conflicts are unexpected and overwriting the previous settings is destructive.

To minimize the incidence of destructive overwrites, you can instruct the tool to halt processing when it detects conflicts. To do so, use the iMerge -error_on_conflict switch. This switch instructs iMerge to stop on the first event that creates a conflict and to display a detailed conflict report about the involved events and conflicts. This enables you to verify the conflicts before the tool overwrites a previous task.

In the ATPG context, the set_test_setup_icall -merge and set_test_end_icall -merge commands automatically perform conflict reporting. Refer to set_test_setup_icall and set_test_end_icall in the *Tessent Shell Reference Manual* for a full description.

> **Note**
> Conflicts cause the open pattern set to be in an undefined state. That is, some of the iMerge block's scheduled events have been processed and stored to the open pattern set while others have not been processed and stored. To obtain usable patterns, you must close the pattern set, identify and eliminate the root cause of the conflict, and re-create the pattern set from the beginning.

The iMerge conflict report consists of three parts:

- Error message with the list of conflicts. When you call iMerge as part of an open pattern set in the context patterns -ijtag, the error message is as follows:

  ```
  //  Error: iMerge conflict encountered.
  ```

  As an example, the following conflict applies to a situation in which two events cannot be merged because they write conflicting values:

  ```
  //  Event '3' tries to set the value of 'block1.R[0]' to '1' to meet
  the iApply targets, whereas event '6' tries to set the value of
  'block1.R[0]' to '0' to meet the iApply targets.
  ```

  All events are identified by a unique event ID that is simply an integer. This event ID is referred to in all three parts of the report.

- Description of the involved events. The description section begins as follows:

  ```
  //  Event:
  //     iNote:
  //     Resources:
  ```

- iMerge flow graph. See "Example of iMerge Conflict Reporting and Analysis" for a usage example.

Table 3-2 defines the terminology used in the description section of the conflict report.

Table 3-2. Conflict Report Terminology

Term	Description
Controllable entity	Primary input or scan register bit. The values applied to these entities do not depend directly on something else, but they can be freely chosen during a scan load or the application of the stimuli of the top-level ports.
Random access	In the message "A controllable entity requests random access," the tool assumes that it must apply the values 0 and 1 at least once within the same iApply.

Table 3-2. Conflict Report Terminology (cont.)

Term	Description
Resource	A port, scan register, or data register or instance that has been subject to an iTake command. The ports that have been targeted by iForcePort or iComparePort commands are also (implied) resources.
Setup command	A command that precedes an action (iApply or iRunLoop) and that determines the targets and the behavior of that action. Setup commands of an iApply include iWrite and iRead. Setup commands of an iRunLoop include iClock and iClockOverride.
Unspecific read	An iRead without specification of the expected value.

Example of iMerge Conflict Reporting and Analysis

This section provides a sample scenario in which iMerge detects an error and stops processing. By examining the conflict report and the ICL description you can identify the root cause of the problem and thus eliminate it.

Suppose you have the following PDL description. The iMerge blocks in this example reflect the actual ICL hierarchy such that you have a nested iMerge structure.

```
set_context patterns -ijtag

read_icl ../data/icl/*
source ../data/pdl/raw1.iprocs

iProcsForModule chip
iProc testAllRaw {} {
   iMerge -begin -error_on_conflict  //Specified on outermost iMerge block
      iCall block1_I1.testAllRaw
      iCall block1_I2.testAllRaw
      iCall block2_I1.testAllRaw
      iCall block3_I1.testAllRaw
   iMerge -end
}

iProcsForModule block1
iProc testAllRaw {} {
   iMerge -begin
      iCall raw1_I1.run_testa
      iCall raw1_I2.run_testa
   iMerge -end
}

iProcsForModule block2
iProc testAllRaw {} {
   iMerge -begin
      iCall raw1_I1.run_testa blue            //iMerge conflict
      iCall raw1_I2.run_testa green
   iMerge -end
}

iProcsForModule block3
iProc testAllRaw {} {
   iMerge -begin
      iCall raw1_I1.run_testa
      iCall raw1_I2.run_testa
      iCall raw1_I3.run_testa
      iCall raw1_I4.run_testa
   iMerge -end
}

set_current_design
add_clocks ClkA -period 10ns
set_system_mode analysis

open_pattern_set test1 -tester_period 100ns
   iCall testAllRaw
close_pattern_set

write_patterns pat1.stil -stil -repl
```

The resulting conflict report displays as follows:

```
//   Error: iMerge conflict encountered.
//
//   Event '43' tries to set the value of 'block2_I1.tdr.R[6]' to '1' to
meet the iApply targets, whereas event '50' tries to set the value of
'block2_I1.tdr.R[6]' to '0' to meet the iApply targets.
//   Event '43' tries to set the value of 'block2_I1.tdr.R[5]' to '0' to
meet the iApply targets, whereas event '50' tries to set the value of
'block2_I1.tdr.R[5]' to '1' to meet the iApply targets.
//
//   Event:          50, iApply
//       iNote:          Set mode to green
//       Resources:
//          INSTANCE block2_I1.raw1_I2
//       Controllable entities requesting value 0:
//          block2_I1.tdr.R[6]
//          block2_I1.tdr.R[3]
//          block2_I1.tdr.R[2]
//       Controllable entities requesting value 1:
//          block2_I1.tdr.R[5]
//       Controllable entities requesting random access:
//          block1_I1.sib1.SIB
//          block1_I1.sib2.SIB
//          block1_I2.sib1.SIB
//          ...
//       Controllable entities fixed to 0 to stabilize resources:
//          block2_I1.tdr.R[0]
//          block2_I1.tdr.R[7]
//          block2_I1.tdr.R[4]
//          block2_I1.tdr.R[1]
//       Targeted setup commands:
//          iWrite mode green
//
//   Event:          43, iApply
//       iNote:          Set mode to blue
//       Resources:
//          INSTANCE block2_I1.raw1_I1
//       Controllable entities requesting value 0:
//          block2_I1.tdr.R[5]
//          block2_I1.tdr.R[3]
//          block2_I1.tdr.R[2]
//       Controllable entities requesting value 1:
//          block2_I1.tdr.R[6]
//       Controllable entities requesting random access:
//          block1_I1.sib1.SIB
//          block1_I1.sib2.SIB
//          block1_I2.sib1.SIB
//          ...
//       Controllable entities fixed to 0 to stabilize resources:
//          block2_I1.tdr.R[0]
//          block2_I1.tdr.R[7]
//          block2_I1.tdr.R[4]
//          block2_I1.tdr.R[1]
//       Targeted setup commands:
//          iWrite mode blue
```

This is followed by the flow graph. Figure 3-17 shows the partial flow graph that applies to this example. When you examine the flow graph, you see that the iApply of event 43 is the second iApply called from within iCall raw1_I1.run_testa that in turn is called from within iCall block2_I1.testAllRaw. The iApply of event 50 is the second iApply called from within iCall raw1_I2.run_testa that in turn is also called from within iCall block2_I1.testAllRaw.

Figure 3-17. iMerge Flow Graph

```
//  iMerge Flow Graph
//  -----------------
//  0                                               Fork event of iMerge block
//  +----------+----------+----------+
//  2          |          |          |              Start processing iCall block1_I1.testAllRaw
//  3          |          |          |               Fork event of iMerge block
//  +----+.....|..........|..........|............    
//  5    |     |          |          |              Start processing iCall raw1_I1.run_testa
//  6    |     |          |          |                Queue an iApply
//  7    |     |          |          |                Queue an iApply
//  8    |     |          |          |                Queue an iApply
//  9....|.....|..........|..........|............      Queue an iRunLoop
//  10   |     |          |          |                Queue an iApply
//  11   |     |          |          |               Finished processing iCall raw1_I1.run_testa
//  |    12    |          |          |              Start processing iCall raw1_I2.run_testa
//  |    13    |          |          |                Queue an iApply
//  |.....14...|..........|..........|............      Queue an iApply
//  |    15    |          |          |                Queue an iApply
//  |    16    |          |          |                Queue an iRunLoop
//  |    17    |          |          |                Queue an iApply
//  |    18    |          |          |               Finished processing iCall raw1_I2.run_testa
//  +-----+....|..........|..........|............    
//  4          |          |          |              Join event of iMerge block
//  19         |          |          |              Finished processing iCall block1_I1.testAllRaw
//  |          20         |          |              Start processing iCall block1_I2.testAllRaw
//  |          21         |          |               Fork event of iMerge block
//  |..........+-----+....|..........|............    
//  |          23    |    |          |              Start processing iCall raw1_I1.run_testa
//  |          24    |    |          |                Queue an iApply
//  |          25    |    |          |                Queue an iApply
//  |          26    |    |          |                Queue an iApply
//  |..........27....|....|..........|............      Queue an iRunLoop
//  |          28    |    |          |                Queue an iApply
//  |          29    |    |          |               Finished processing iCall raw1_I1.run_testa
//  |          |    30    |          |              Start processing iCall raw1_I2.run_testa
//  |          |    31    |          |                Queue an iApply
//  |..........|.....32...|..........|............      Queue an iApply
//  |          |    33    |          |                Queue an iApply
//  |          |    34    |          |                Queue an iRunLoop
//  |          |    35    |          |                Queue an iApply
//  |          |    36    |          |               Finished processing iCall raw1_I2.run_testa
//  |..........+-----+....|..........|............    
//  |          22         |          |              Join event of iMerge block
//  |          37         |          |              Finished processing iCall block1_I2.testAllRaw
//  |          |          38         |              Start processing iCall block2_I1.testAllRaw
//  |          |          39         |               Fork event of iMerge block
//  |..........|..........+-----+....|............    
//  |          |          41    |    |              Start processing iCall raw1_I1.run_testa
//  |          |          42    |    |                Queue an iApply
//  |          |          43    |    |                Queue an iApply
//  |          |          44    |    |                Queue an iApply
//  |..........|..........45....|....|............      Queue an iRunLoop
//  |          |          46    |    |                Queue an iApply
//  |          |          47    |    |               Finished processing iCall raw1_I1.run_testa
//  |          |          |    48    |              Start processing iCall raw1_I2.run_testa
//  |          |          |    49    |                Queue an iApply
//  |..........|..........|.....50...|............      Queue an iApply
//  |          |          |    51    |                Queue an iApply
//  |          |          |    52    |                Queue an iRunLoop
//  |          |          |    53    |                Queue an iApply
//  |          |          |    54    |               Finished processing iCall raw1_I2.run_testa
//  |..........|..........+-----+....|............    
```

Next, examining the ICL, you can see that different DataRegisters (DR1 and DR2) drive the mode values of the instances of module raw1, but those DataRegisters have the same data source. Because iMerge does not know whether those data registers can be enabled or disabled independently of each other, it assumes a conflict to avoid potential issues.

```
Module block2 {
    ScanInPort       si1;
    ScanOutPort      so1    { Source tdr.so; }
    SelectPort       en1;
    ShiftEnPort      se;
    CaptureEnPort    ce;
    UpdateEnPort     ue;
    TCKPort          tck;
    ClockPort        ClkA;

    Instance tdr Of tdr2 {
        InputPort en = en1;
        InputPort si = si1;
        InputPort fq = 3'b0,RMux[7:0];
    }
    Instance raw1_I1 Of raw1 {
        InputPort in = DR1[7:0];
        InputPort clk = ClkA;
    }
    Instance raw1_I2 Of raw1 {
        InputPort in = DR2[7:0];
        InputPort clk = ClkA;
    }
    DataRegister DR1[7:0] {
        WriteEnSource    we1;
        WriteDataSource  tdr.td[7:0];
    }
    DataRegister DR2[7:0] {
        WriteEnSource    we2;
        WriteDataSource  tdr.td[7:0];
    }
    LogicSignal we1 {
        tdr.td[9],tdr.td[8] == 2'b10;
    }
    LogicSignal we2 {
        tdr.td[9],tdr.td[8] == 2'b11;
    }
    DataMux  RMux[7:0]  SelectedBy tdr.td[10],tdr.td[8] {
        2'b10 : raw1_I1.out;
        2'b11 : raw1_I2.out;
```

PDL Retargeting Commands

The following table contains a summary of the PDL retargeting commands available in Tessent Shell.

Table 3-3. PDL Retargeting Command Summary

Command	Description
iApply	Triggers the retargeting of all queued iRead and iWrite commands.
iCall	Calls an iProc registered against the ICL module associated with the specified *effective_icl_instance_path*.

Table 3-3. PDL Retargeting Command Summary (cont.)

Command	Description
iClock	Checks whether a controlling clock path from a valid clock source to the specified clock port exists and computes the cumulative frequency multiplier and divider values.
iClockOverride	Models how the functional clocking has been programmed.
iMerge	Encloses one or several iCall statements that may be run in parallel instead of serially.
iNote	Inserts a note or annotation in the opened pattern set.
iOverrideScanInterface	Imposes user-specified behavior on the operation of a ScanInterface.
iPrefix	Sets the iPrefix path that is used to compute the *effective_icl_instance_path* for the iCall command and other PDL commands.
iProc	Specifies a PDL procedure that can run later when referenced by the iCall command.
iProcsForModule	Specifies the ICL module that subsequent iProc commands refer to and optionally its PDL name space.
iRead	Adds a read operation to the command queue that is solved by the next iApply command.
iRelease	Releases a resource previously taken by iTake.
iReset	Adds a sequence of actions to the current pattern set that is required to set the ICL network into the reset state.
iRunLoop	Creates a vector loop of a given duration.
iTake	Takes ownership of ICL resources to prevent the retargeting software from altering their states during the processing of subsequent iApply commands or during the processing of concurrent iProcs in iMerge parallelization
iUseProcNameSpace	Within a pattern set, selects a PDL name space valid for all subsequent PDL commands.
iWrite	Adds a write operation to the command queue that is solved by the next iApply command.
add_clocks	Adds ICL system clocks. Also used for adding non-ICL clocks. Needed to define a test clock off-state other than 0, which is the default.
add_input_constraints	Constrains primary input pins to certain values during the ATPG process.
close_pattern_set	Finalizes and closes the currently open pattern_set.

Table 3-3. PDL Retargeting Command Summary (cont.)

Command	Description
delete_patterns	Deletes the pattern set currently in memory.
open_pattern_set	Opens an empty named pattern_set and makes it ready to be populated with the specified PDL commands.
read_icl	Reads ICL files into the internal ICL database.
reset_open_pattern_set	Clears the content of the currently open pattern set.
set_context	Specifies the current usage context of Tessent Shell. You must set the context before you can invoke most other commands in Tessent Shell.
set_current_design	Specifies the top level of the design or ICL module from which the data module is elaborated downward for all subsequent commands until reset by another run of this command.
set_ijtag_retargeting_options	Enables you to configure different aspects of IJTAG retargeting.
set_module_matching_options	Defines the prefixes and suffixes or regular expressions to use when matching an ICL module name to a design module name during the ICL Extraction flow.
set_system_mode	Specifies the system mode you want the tool to enter.

Introspection and Reporting Commands

The following table provides a summary of all relevant ICL and PDL introspection commands as well as all IJTAG related reporting commands available in Tessent Shell.

Commands specific to ICL extraction are listed separately in "ICL Extraction Commands" on page 130.

All Tessent Shell reporting commands start with "report_", whereas all introspection commands returning information about objects start with "get_". Commands starting with "delete_" remove objects from the tool memory.

Table 3-4. ICL Introspection and Reporting Command Summary

Command	Description
delete_icl_modules	Deletes the specified ICL modules from memory.
delete_iprocs	Deletes the specified list of iProcs attached to the ICL module that was specified by the last iProcsForModule command.

Table 3-4. ICL Introspection and Reporting Command Summary (cont.)

Command	Description
get_icl_fanins	Returns a collection of all requested objects found in the fan-in of the specified pin or port objects.
get_icl_fanins_in_module	Returns a collection of all requested objects found in the fan-in of the specified icl_pin_in_module or icl_port objects. This can be used even if the current design has not yet been set.
get_icl_fanouts	Returns a collection of all requested objects found in the fanout of the specified ICL pin, or port objects.
get_icl_fanouts_in_module	Returns a collection of all requested objects found in the fanout of the specified icl_pin_in_module or icl_port objects. This can be used even if the current design has not yet been set.
get_icl_instances	Returns a collection of all ICL instances instantiated relative to the current design that match the specified *name_patterns* list.
get_icl_modules	Returns a collection of all ICL modules that match the specified *name_patterns* list.
get_icl_module_parameter_list	Returns the list of parameters of an ICL module.
get_icl_module_parameter_value	Returns the value of the parameter on the specified module.
get_icl_objects	Returns a collection of all requested objects matching various criteria like name patterns, object types, associated ICL objects, attribute expressions, and so on.
get_icl_pins	Returns a collection of all hierarchical ICL pins instantiated relative to the current design that match the specified *name_patterns* list.
get_icl_ports	Returns a collection of all ports on a given module that match the specified *name_patterns* list.
get_icl_scan_interface_list	Returns the names of the scan interfaces in an existing ICL top module, if the ICL top module exists. If the ICL top module does not exist, it returns the names of the scan interfaces that are created in the new ICL top module by ICL extraction after the add_icl_scan_interfaces command has been used to specify the scan interfaces.
get_icl_scan_interface_port_list	Returns the names of the ports for the specified scan interface in an existing ICL top module, or the names of the ports for the specified scan interface that are created in the new ICL top module during ICL extraction.

Table 3-4. ICL Introspection and Reporting Command Summary (cont.)

Command	Description
get_icl_scope	Returns the current ICL scope, for example, the instance path name for which an iCall was issued.
get_iclock_list	Returns a list of ICL port and pin names on which you have issued an iClock command.
get_iclock_option	Returns the effective or specified source, frequency multiplier, or frequency divider values for the specified icl_port_or_pin_name.
get_ijtag_retargeting_options	Returns the value of the specified option either set by a set_ijtag_retargeting_options command or its default value.
get_iproc_argument_default	Returns the default value for the specified arg_name of the specified *proc_name* attached to the ICL module specified by the last iProcsForModule command.
get_iproc_argument_list	Returns the list of argument names for the specified iProc attached to the ICL module that was specified by the last iProcsForModule command.
get_iproc_body	Returns the body for the specified iProc attached to the ICL module that was specified by the last iProcsForModule command.
get_iproc_list	Returns a Tcl list of iProcs attached to the ICL module specified by the last iProcsForModule command.
get_open_pattern_set	Returns the name of the currently open pattern set.
get_pattern_set_data	Returns requested details of the internal representation of the specified pattern set.
report_design_sources	Reports the pathnames or file extensions previously specified with the set_design_sources command.
report_icl_modules	Reports loaded and extracted ICL modules in either ICL syntax or human-readable form.
report_iclock	Reports the ICL ClockPort specified by the iClock commands as well as their extracted sources and cumulative freqMultiplier and freqDivider values.
report_ijtag_logical_connections	Reports the logical paths that exist within the current design between the specified source and destination pins/ports, as well as all connections from or to the specified pins/ports. If no pin or port connections are specified, all logical connections are listed.

Table 3-4. ICL Introspection and Reporting Command Summary (cont.)

Command	Description
report_ijtag_retargeting_options	Reports the types and values of each option either set by a set_ijtag_retargeting_options command or their default values.
report_module_matching	With the -icl option, this command reports the identified name matching between the ICL modules and Verilog modules during the ICL Extraction flow.
report_module_matching_options	Reports the current settings defined by the set_module_matching_options command.
report_pattern_sets	Creates a human-readable report of a specified pattern set or of all pattern sets.

Chapter 4
ICL Extraction

The goal of ICL Extraction, or more precisely ICL network extraction, is the automated generation of the interconnection information of the various IJTAG building blocks (instruments, SIBs, TDRs, and so on) from the flattened netlist of a design.

The output of the extraction process is the interconnect information of the instantiated IJTAG building blocks in ICL format. You can use the extract_icl command to perform the ICL extraction. Refer to extract_icl in the *Tessent Shell Reference Manual* for a full description of the command. However, if the IJTAG network was manually inserted or using other methods, this chapter describes how you can extract the ICL. See also "Top-Down and Bottom-Up ICL Extraction Flows".

This flow is used in an environment where the ICL is available only for the IJTAG building blocks. There is no ICL for the network that connects all ICL blocks, although the Verilog gate-level design contains all these connections. It is the task of Tessent IJTAG in this flow to generate the missing ICL from the design data and netlist setup information.

Once the missing ICL has been generated, the PDL retargeting flow using this generated ICL file commences without any change.

ICL Extraction has a number of specific design rule checks, some of which are supported in Tessent Visualizer for graphical debug. These design rule checks ensure that the generated ICL is syntactically and semantically correct.

The following topics are described in this chapter:

ICL Extraction Flow

The main steps of the ICL Extraction Flow and the corresponding commands that implement the flow are described in this section.

Figure 4-1 illustrates the steps of the basic Tessent IJTAG ICL Extraction flow you perform with Tessent Shell.

Figure 4-1. Generic ICL Extraction Flow

Step	Command

Invoke Tessent Shell and set context to IJTAG	`% tessent -shell`
	`...`
	`SETUP> set_context patterns -ijtag`

Read in the design and the library	`read_verilog chip.v` `read_cell_library libraries/tessentCellLib`

Load the ICL and the PDL	`set modules {sib1 tdr1 tdr2 tap1}`
	`foreach module $modules {read_icl`
	`../data/icl/${module}.icl}`
	`dofile   ../data/pdl/tdr2.pdl`

Set the top-level module of the design	`set_current_design`
Specify the test procedure file	`set_procfile_name test_setup.proc`
Switch to ANALYSIS mode	`set_system_mode analysis`

Write out the extracted ICL	`write_icl -output_file generated_chip.icl`
	`-modules [get_single_name [get_current_design]]`
	`-hierarchical -replace`

Required Inputs for ICL Extraction

To perform ICL extraction, you must provide information about your design to Tessent Shell.

The following inputs are required:

- **Design Data** — Currently the Verilog gate-level netlist.

- **Library** — The ATPG library.

- **ICL Data** — The ICL descriptions of the IJTAG building blocks instantiated within the design. The ICL descriptions may contain special extraction attributes that direct the ICL extraction process.

Optional Inputs for ICL Extraction

In addition to the required design information listed above, you can provide certain optional inputs that can influence the ICL extraction process.

- **Test Procedure File or Input Constraints** — The purpose is to set the design into a mode that sensitizes the paths between the IJTAG building blocks.

  For example, you have a MUX that is in the path between two IJTAG building blocks, but the MUX itself is not an IJTAG building block described in an ICL file. The select input of the MUX must be set to the necessary value to sensitize the path between the connected IJTAG building blocks.

- **Extraction modifiers** – Through commands in Tessent IJTAG you can influence how the ICL extraction processes are performed. Through these commands you can, for example, instruct Tessent IJTAG to ignore a loaded ICL module, or to declare to the tool how to deal with a black box instance in your Verilog design.

Performing ICL Extraction

The following command sequence provides the basic Tessent Shell commands to perform ICL extraction.

Prerequisites

- A Verilog design netlist

- One or more cell libraries

- ICL primitives

Procedure

1. In a shell, invoke Tessent Shell:

   ```
   % tessent -shell
   ```

After invocation, the tool is in an unspecified setup mode. You must set the context before you can invoke the ICL extraction commands.

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

   set_context patterns -ijtag

3. Read in the design netlist using the read_verilog command. For example:

   read_verilog chip.v

4. Read in one or more cell libraries into the tool using the read_cell_library command as follows:

   read_cell_library ./libraries/tessentCellLib

5. Read in the ICL for the primitives using the read_icl command. For example:

   read_icl ./data/icl_primitives/sib1.icl

   Upon reading the ICL data, the tool performs ICL semantic rule checks on this data.

6. If needed, specify the acceptable prefixes and suffixes or regular expressions to use when matching an ICL module to a design using the set_module_matching_options command. For example:

   **set_module_matching_options -prefix_pattern_list {mycore_} \
       -suffix_pattern_list {_[0-9]+} -regexp**

7. Set the top-level of the design using the set_current_design command. For example:

   set_current_design chip

   The order of reading the design netlist files, library files, and ICL files is not important to the tool. However, once the top-level of the design is set, Tessent IJTAG matches the ICL module names against the design module names as the first step in the ICL extraction process. Any ICL or design file read in afterward is not considered. Use a subsequent "set_current_design" command in this case.

8. Depending on your design style, specify any additional parameters including the following commands:

   add_black_box

   add_clocks

   add_input_constraints

9. If needed, specify the test procedure file that contains the test_setup procedure using the set_procfile_name command. For example:

   set_procfile_name procedures/test_setup.proc

10. If needed, specify any additional commands or attributes that influence ICL extraction. Commands include the following:

    add_ijtag_logical_connection

---

add_icl_scan_interfaces

set_icl_scan_interface_ports

You can insert additional ICL extraction related attributes into modules, using the command:

set_attribute_value

11. Change the system mode to analysis to run the ICL extraction using the set_system_mode command as follows:

set_system_mode analysis

During the transition from setup to analysis mode, the tool performs ICL design rule checking and special ICL extraction related design rule checks that essentially validate the ICL function-aware tracing between the ICL modules. Once in analysis mode, the generated ICL module is available. You may proceed with PDL retargeting or save the generated ICL module.

12. Write the extracted ICL results to an external file using the write_icl command:

```
write_icl -output_file generated_chip.icl
    -modules [get_single_name [get_current_design]] -hierarchical -replace
```

Top-Down and Bottom-Up ICL Extraction Flows

ICL extraction is performed automatically in the "patterns -ijtag" context when the system mode is switched to analysis and no ICL module has been read in that matches the top-gate level module name.

ICL extraction is supported in both -no_rtl and -rtl contexts. In the patterns -ijtag context, the -rtl switch is inferred if you come from the dft context and use the -rtl option when setting the dft context. When you enter the patterns -ijtag context from the unspecified context, the -no_rtl option is assumed. ICL extraction makes use of quick synthesis to convert any RTL in the fan-in or fanout of ICL modules. See the description of the synthesize_before_analysis and the exclude_from_synthesis module attributes in the *Tessent Shell Reference Manual* if you are using a test_setup procedure for ICL extraction.

Tessent Shell supports the following ICL extraction flows:

- Top-Down ICL Extraction Flow — This flow generates a flat ICL description of the ICL network connecting all loaded ICL modules. The resulting set of ICL modules consists of all initially provided ICL modules plus a single, flat, extracted ICL module representing the ICL interconnect network across all design hierarchy boundaries.

- Bottom-Up ICL Extraction Flow — In this flow, you extract ICL modules one by one from the leaf level instruments to the top. Stepping through the design hierarchy, one ICL module is generated for each hierarchy step, building the ICL netlist hierarchy bottom-up to the top-level design module.

In both flows, Tessent IJTAG matches the loaded ICL modules against the loaded design modules. This matching is by name of the module, taking uniquifications and other name manipulations into account. The set_module_matching_options command enables the specifications of how the ICL and design module names should be matched.

When issuing set_current_design in setup mode, Tessent IJTAG tests if there is an ICL module name that matches the name of the chosen top-level design module under the set matching options. ICL Extraction is automatically enabled if none of the ICL modules names in the database match the name of the specified current design. If there is a matching module name, no extraction is triggered.

Once Tessent IJTAG has determined that the current flow requires ICL Extraction a list of matched ICL and Verilog module names can be reported with the 'report_module_matching -icl' command and option.

You can also introspect this decision of Tessent IJTAG with the 'get_context -extraction' command and option that returns "1" if you are in an ICL extraction flow and "0" otherwise.

The extraction of ICL itself is part of switching to the analysis mode from setup mode. The tool performs initial DRC including the application of test_setup and constraints, matches the ICL

modules, instances, and ports to the corresponding design entities, and uses a tracing-based algorithm to identify the design components that implement the ICL access network. This tracing is performed in the flat model of the design netlist.

You can provide additional data to make this tracing work correctly, for example, declared clocks, a test setup procedure, or input constraints on top-level IO ports and on internal (cut) points. You can also declare logical connections between points in the design that define where Tessent IJTAG should continue the extraction process, or define which modules to ignore during ICL extraction.

Once the tracing completes successfully, an ICL module or file can be written that represents the identified ICL access network. The ICL data generated by the extraction process is readily available for subsequent PDL retargeting. It is not required to read the generated ICL file in setup mode and once more go to analysis mode.

ICL extraction differentiates between input constraints that were applied at the current top-level of the design and constraints that were applied internally to the design. All these constraints were provided to Tessent IJTAG before ICL extraction was started. It is expected that these constraints are fulfilled also during PDL retargeting, that is, the design setup during ICL extraction is a compatible subset of the design setup during PDL retargeting.

Tessent Shell enforces this automatically by using the add_input_constraints command for all current top-level constraints listed in the attributes. Non-top-level constraints are not enforced. However, in future, Tessent IJTAG may check that the internal constraints set during ICL extraction are satisfied by the overall design setup.

Top-Down ICL Extraction Flow

The top-down flow calculates the IJTAG building block connections starting at the top module of a chip.

Depending on the options used during write_icl, the resulting ICL connection file can contain the connections and instances of all IJTAG building blocks of the complete chip.

For all practical purposes, the flow is identical to the PDL retargeting flow described in A Typical PDL Retargeting Flow. The only difference is that not all ICL modules that provide ICL interconnections were loaded.

Bottom-Up ICL Extraction Flow

The bottom-up flow calculates the IJTAG building block connections starting at the lower level modules of the design and writes the IJTAG building block connectivity within these lower-level modules to ICL connection files.

The module of interest is set through the set_current_design Tessent Shell command. It is possible to load the Verilog netlist description for the entire design, although the ICL extraction is done only for a submodule that is defined through set_current_design. The hierarchy level specified with the set_current_design command applies to both the ICL and the Verilog.

Once the new ICL connection files are extracted and saved, they are then used as input in the next step to calculate the connections for the next higher level of the design. This command sequence must be repeated for each desired design module level.

ICL Extraction Design Rule Checks

ICL extraction has additional design rule checks (DRCs) that are performed at the beginning of the extraction process and after each loaded ICL module passes its syntax and semantics checks. After the ICL module for the instrument interconnection has been extracted from the design description, the newly generated module and all loaded modules must pass the remainder of the ICL syntax and semantics checks.

The ICL extraction DRCs start with the letter 'I', followed by a number. For example, DRC I1 performs consistency checks on each design module that was mapped to an ICL module during the run of the "set_current_design" command. This includes verifying that all ports on the ICL module are present in the corresponding Verilog module. The reverse is not necessary because the Verilog design module usually has more ports than the ICL module.

A common ICL extraction DRC violation is I2. This is one of the main tracing checks. It verifies that a connection can be identified from an ICL-attributed pin to another ICL-attributed pin or a top-level port. An ICL-attributed pin is a pin of a Verilog design module instance that has been mapped to an ICL module instance and pin.

To start debugging ICL extraction DRC violations, you can use the following ICL reporting command:

> report_module_matching -icl

The report of this command is available once the set_current_design command has completed successfully, before the ICL extraction is performed at the beginning of set_system_mode analysis. The report shows which ICL and Verilog modules have been matched by name and under consideration of the prefix and suffix regular expressions (if you specify the -regexp switch) of the 'set_module_matching_options' command. Below is an example:

```
//  design module  design instance             ICL file                        ICL module
//  -------------  --------------------------  ------------------------------  --------
//  chip_JTAP      chip/JTAP_INST              ../chip_JTAP.icl                chip_JTAP
//  sib            chip/sib_top_designConfigReg ../Libraries/ijtag/sib.icl     sib
//  sib            chip/sib_piccpu_1           ../Libraries/ijtag/sib.icl     sib
//  sib            chip/sib_piccpu_2           ../Libraries/ijtag/sib.icl     sib
//  tdr1           chip/piccpu_1/tdr           ../Libraries/ijtag/tdr1_mod.icl tdr1
//  tdr1           chip/piccpu_2/tdr           ../Libraries/ijtag/tdr1_mod.icl tdr1
//  tdr1           chip/top_designConfigReg/tdr ../Libraries/ijtag/tdr1_mod.icl tdr1
```

If you find anything suspicious, please revisit the set_module_matching_options or double-check that you have loaded all ICL modules you require for the design. Start with the command report_module_matching_options to check the options set for the tool for matching the names of modules between the design and the ICL descriptions.

You can influence the ICL extraction process through commands in Tessent Shell and through ICL and design attributes. Some of these attributes can be used to resolve I2 DRC violations.

This is explained later in this chapter in the section "How to Influence the ICL Extraction Process".

Debugging DRC Violations With Tessent Visualizer

You can also debug some ICL extraction DRC violations using Tessent Visualizer. This is especially helpful for the I2 connection tracing DRC.

Prerequisites

DRC violations reported by Tessent IJTAG

Procedure

1. Open Tessent Visualizer using the open_visualizer command.

2. Use the Instance Browser and DRC Browser tabs to find the ICL extraction DRC violations of interest. Grouping the DRC violations by instance and applying appropriate filters helps expose the specific design objects associated with each violation. From the DRC Browser, right-click a violation in the table and choose "Visualize DRC" from the popup window. The instances associated with the violation are displayed in the Flat Schematic along with a description of the violation. Alternatively, you can use the command "analyze_drc_violation." at the tool prompt in the Transcript tab.

Figure 4-2. ICL Rule Violation Debug in Tessent Visualizer
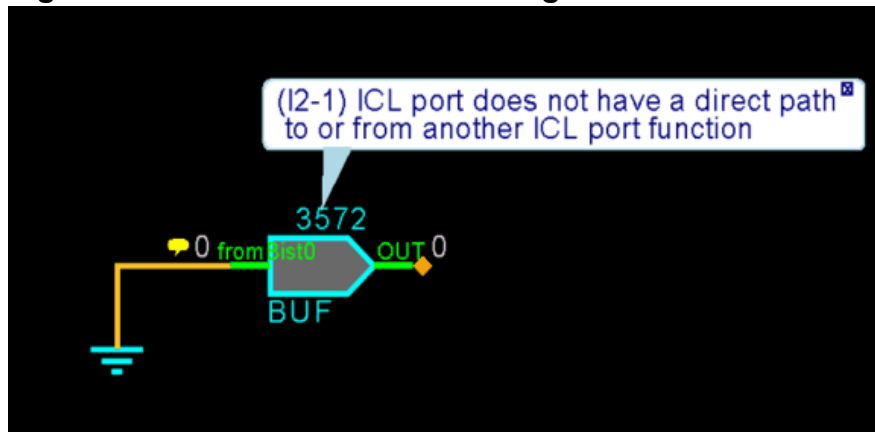
How to Influence the ICL Extraction Process

You can influence how ICL extraction is being performed through several attributes associated with ICL or design modules and instances, as well as commands of Tessent IJTAG.

Using these attributes, you can, for example, ensure that ICL-defined data ports may be tied off, preventing the tool from issuing an I2 DRC violation. Many of the commands and attributes you use and define here are translated into the ICL syntax of the generated ICL module.

How to Influence ICL Extraction Through Commands

You can use Tessent Shell commands to influence ICL extraction.

With the set_module_matching_options command, you tell how the ICL module names and the design module names can be matched, as shown earlier in the "ICL Extraction Design Rule Checks" section. In the following sections, additional commands are described through which you can influence the ICL extraction process and the resulting ICL module.

How to Map ICL Module Names and Design Module Names

The matching of ICL modules and design modules is done through the module names. Unfortunately, there are many reasons why an ICL module name and a design module name might not match exactly. Unification of names as part of the synthesis process is a major contributor to these name mismatches. Fortunately, these name modifications follow specific patterns.

Typically, synthesis adds a prefix or suffix to the design module's original name. For example, names are changed from "MyModule" to "MyModule_X1", to "MyModule_X2", and so on. Using the Tessent Shell set_module_matching_options command you can tell the tool these module name mapping patterns specifically for your design. For the example above, you would use the following:

```
set_module_matching_options -suffix_pattern_list {_X[0-9]+} -regexp
```

With this command, option, and parameter, you tell the tool to expect module name changes that add "_X" followed by a number that is at least one digit. Now assume that there is a second module name mapping pattern in your design. This second pattern may change the names as follows: "MyModule" to "MyModule_Y1", to "MyModule_Y2", and so on. Because you want to add another module matching option in addition to the first one, make sure you use the "-append" switch. Without the -append switch, the second command invocation overwrites the first one.

```
set_module_matching_options -suffix_pattern_list {_Y[0-9]+} -regexp -append
```

Now any design module name that matches either mapping pattern can be recognized and subsequently be mapped to the ICL module name "MyModule". Use the set_module_matching_options command before you use set_current_design, because part of setting the current design is creating the ICL to design the module name mapping table.

Once you have set the current design, you can learn about the ICL and design module matching the tool has identified by using the report_module_matching command, with the "-icl" option.

How to Add Top-Level Ports

As mentioned earlier, ICL Network Extraction uses tracing to identify design instances and design ports that should be included in the generated ICL description. It is possible that additional data ports, like global test mode signals, should be added to the ICL top-level module, but these ports are not reachable through tracing. To declare to the ICL Extraction functionality to include such additional top-level ports, Tessent Shell provides the command add_icl_ports. With this command, you can add ports of different types, like DataInPort, DataOutPort, ScanInPort, or ScanOutPort.

If you have add_clocks -pulse_always defined on any port of your current design, it is defined as a ClockPort in the extracted ICL even if it has not been reached by tracing from a ClockPort of an instantiated module with matching ICL description.

You can also trigger the creation of differential clocks in the new ICL top-level module. Use the add_input_constraints and add_clocks commands as shown in the following example:

```
add_input_constraints -equivalent CLKP -invert CLKN
add_clocks 0 CLKP -period 50ns
```

By default, the tool uses the port with the off-state "0" as the ordinary ClockPort in ICL ("representative port"), and uses the port with the off-state "1" as the ClockPort with the "DifferentialInvOf" property ("associated port"). Connecting the ports to other ICL ClockPorts or to internal clocks that unambiguously determine their roles in the differential group overrides the default.

How to Ignore a Design Instance During ICL Extraction

Assume now that the module matching report shows a design module instance that, for whatever reason, should be excluded from the ICL extraction process, but other instances of the same module should be considered. You can do this by setting the attribute ignore_during_icl_extraction to "true" for this instance from within Tessent Shell before you enter analysis mode. In the example below, if you want to exclude the design instance named "MyBlock4_ignore" from ICL extraction, do the following:

```
set_attribute_value [ get_instance MyBlock4_ignore ]
    -name ignore_during_icl_extraction -value true
```

Observe the usage of the get_instance command, not the get_icl_instance command.

How to Extract ICL From an Incomplete Design Description

Another group of commands that influence the ICL extraction process are related to logical connections. Through a logical connection, you can influence the design netlist tracing by connecting an arbitrary design module instance pin (source) with another design module instance pin (target). When the design netlist tracing algorithm encounters such a source during forward tracing, it continues with the designated target location, ignoring the actual design netlist structure. Similarly, during backward tracing the process continues with the designated source location when a target pin was reached. The source and the target location may also be ports.

Use logical connections to move forward with your IJTAG work, such as for incompletely defined designs. Assume, for example, a case in which you have design module black boxes on the ICL extraction path. Using logical connections you can connect the ICL relevant pins of the black boxes, enabling the ICL extraction tracing to go "through" the design black box. Another example is an incompletely defined ICL network in your existing top-level module. Using logical connections you can "patch" the missing pieces until the design is complete.

The ICL network created through the ICL extraction process uses both sets of data, the one extracted from the design description as well the declared logical connections. In case of discrepancies, you have logical connections as well as connections in the design between the same pins, the declared logical connections take precedence.

In the example below, you instruct the ICL extraction process to logically connect the instance pin U1/Y with the instance pin U2/A.

add_ijtag_logical_connection -from U1/Y -to U2/A

Figure 4-3. Logical Connection Example
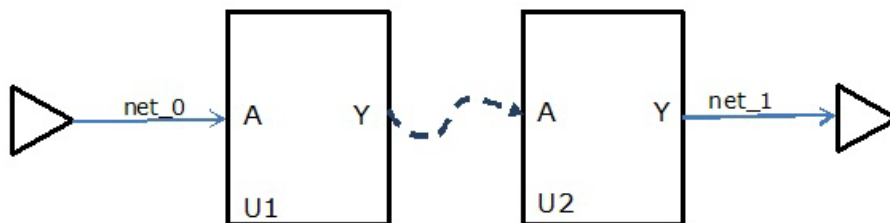
Instance pin name denotes the instance pin in the design. Therefore, it must be expressed in the usual instance-pin-path name syntax of the design description, that is, by using the slash character (/) – not the IJTAG method of using the period (.) as a hierarchy separator. This add_ijtag_logical_connection command creates a connection in the newly generated top-level ICL module, from the ICL instance pin mapped to U1/Y to the ICL instance pin mapped to U2/A. This connection is created irrespective of the actual design connections, if any.

Matching the add_ijtag_logical_connection command, there is a delete_ijtag_logical_connection command and a report_ijtag_logical_connections command. Assume a report like the following:

report_ijtag_logical_connections

```
//  IJTAG Logical  Connections:
//  From Source    To Destination
//  =============  =====================
//  /i1/i/dout[1]  /i2/i/din[1]
//  /i2/i/dout[1]  /i1/\escaped/din[45]
//  /tap1_I1/tdo   tdo
//  tdi            /pll1_I1/si1
//  tdi            /sib1_IPLL/si
//  tdi            /tap1_I1/tdi
```

Besides reporting, you can introspect the logical connections using attributes placed on the pins (ports) you used in the add_ijtag_logical_connection command. For example, following the above example, the commands

**get_attribute_value_list [get_pins /i2/i/din[1]] \
    -name ijtag_logical_hier_connection_from_src**

**get_attribute_value_list [get_ports /tdi] \
    -name ijtag_logical_hier_connection_to_dst**

compute the following results

```
{{/i1/i/dout[1]}}
```

and

```
{/pll1_I1/si1 /sib1_IPLL/si /tap1_I1/tdi}
```

respectively.

To declare which logical connection to delete usually you would use both a source and a target design module instance pin name. Deleting multiple logical connections at once is also possible. Deleting all logical connections originating from "tdi" of this example can be accomplished by using only the source option of the command:

delete_ijtag_logical_connections -from tdi

report_ijtag_logical_connections

```
//  IJTAG Logical  Connections:
//  From Source    To Destination
//  =============  =====================
//  /i1/i/dout[1]  /i2/i/din[1]
//  /i2/i/dout[1]  /i1/\escaped/din[45]
//  /tap1_I1/tdo   tdo
```

It is important to understand that the declared logical connection only influences the design tracing during ICL extraction. It does not affect the other operations that are part of the ICL extraction process. In particular, any design simulations performed during I5 checks, such as checking for blocked or controlling paths, use the unchanged design description.

How to Add Scan Interface Data to the Extracted Module

The ICL extraction process determines the port name and ICL port function for all module ports identified during the ICL extraction process. You can explicitly specify ScanInterfaces or the tool can infer them. See "ScanInterfaces and Associations Between Ports and ScanInterfaces" on page 35 for more details.

To explicitly provide this information, you can use the add_icl_scan_interfaces and set_icl_scan_interface_ports commands. For a list of introspection and extraction commands, see "ICL Introspection and Reporting Command Summary" on page 106 and "ICL Extraction Command Summary" on page 130.

Assume your newly created top-level ICL module needs the following ScanInterface syntax:

ScanInterface I1 {

Port P1 ;

Port P2 ;

}

ScanInterface I2 {

Port P1 ;

Port P4 ;

}

You would use the following commands after the design has been set, but before switching to analysis mode:

add_icl_scan_interfaces { I1 I2 }

set_icl_scan_interface_ports -name I1 -ports {P1 P2}

set_icl_scan_interface_ports -name I2 -ports {P1 P4}

Of course, all used port names must be valid ports of the ICL module that is created through ICL extraction. The created scan interface must follow all rules defined in the standard.

Cases can exist where the ScanInterfaces inferred by the tool interfere with the ScanInterfaces that you specify, or where the tool creates unwanted results in the inferred ScanInterfaces. For these cases, you can use the following command to prevent the tool from inferring ScanInterfaces during ICL extraction:

set_icl_extraction_options -extract_scan_interfaces off

You must then explicitly specify all required ScanInterfaces of the extracted ICL module with the add_icl_scan_interfaces and set_icl_scan_interface_ports commands.

Note

The default for the -extract_scan_interfaces switch of the set_icl_extraction_options command is "on". By default, the tool tries to infer ScanInterfaces for scan ports you have not specified with set_icl_scan_interface_ports commands.

How to Influence ICL Extraction Through ICL Module Attributes

This section describes how you can use Tessent Shell attributes to influence ICL extraction.

The attributes related to ICL extraction fall into one of two categories. The first category contains attributes that instruct Tessent IJTAG to verify a particular ICL network structure. The first implemented attribute of this category makes the tool validate that a certain ICL port is connected to a top-level port. Use this attribute if, for example, the port must be controlled or observed directly at top-level ports, without an intermediate data or scan register.

The second category of attributes may be used to prevent I2 DRC violations, as follows: During ICL extraction, if any port of an ICL-attributed design module instantiated on the current design cannot be traced to a port on the top level or on another ICL-attributed module, an I2 violation is issued. To bypass DRC I2 violations, an attribute called connection_rule_option can be specified in the ICL file for ICL ports to indicate that the IJTAG logic driven by or driving such ports can be unused for ICL extraction tracing. The "connection_rule_option" attribute also has an impact on the previous hierarchical tracing of the input and output cones. If defined on an input port with the attribute value "allowed_no_source", no hierarchical tracing is performed at that input port and, therefore, no synthesis is done in the input cone of that port. If defined on an output port with the attribute value "allowed_no_destination", no hierarchical tracing is performed at that input port and, therefore, no synthesis is done in the output cone of that port. If synthesis is required for those cones, it has to be manually defined by defining "synthesize_before_analysis" attributes on the synthesis-required design modules.

A typical example is a TAP controller with multiple return scan in ports from the logic. You may have only one ICL module definition in your ICL design library, and connect the TAP in different designs differently. Using these attributes enables Tessent IJTAG, during the ICL extraction phase, to waive any I2 connection issues for ports that your current design is not using, but are still declared in your ICL module definition. Nonetheless, these attributes should be used with care, because you might waive a valid design rule violation.

The allowed values for the connection_rule_option attribute are described below in Table 4-1.

Table 4-1. Values for ICL Extraction Attribute connection_rule_option

Attribute Value	Port Direction	Allowed Simulation Value (stable_after_setup)	Description
allowed_no_source	input	0/1/Z/X	The port can be floating or be driven by any logic. No synthesis happens in the input cone of this port
allowed_tied	input	0/1	The port can be tied to low/high or be driven by any logic with a simulation value of 0 or 1 in stable_after_setup simulation context
allowed_tied_low	input	0	The port can be tied to low or be driven by any logic with a simulation value of 0 in stable_after_setup simulation context
allowed_tied_high	input	1	The port can be tied to high or be driven by any logic with simulation value of 1 in stable_after_setup simulation context
allowed_no_destination	output	0/1/Z/X	The port can be open or connected to any logic regardless of simulation values. No synthesis happens in the output cone of this port
must_connect_to_top_port	input and output	N.A.	Enables an extra DRC test, validating that the port is connected to the top-level IO ports of the design

As an example, you may want to constrain a data-in port named "din" of the ICL module "block1" to high. Such a constraint in the Verilog netlist would cause a tracing violation because the data-in connection from "din" towards the inputs of the design would have been blocked. Using the correct value for the connection_rule_option attribute in the ICL module

waives this I2 violation. In this example, you must add the following ICL attribute to the ICL
port function of the ICL module definition:

Module block1 {

…

DataInPort din { Attribute connection_rule_option = "allowed_tied_high"; }

…

}

The resulting top-level ICL created by the ICL extraction is then for example:

Module top {

…

Instance block1_I1 Of block1 { InputPort din = 'b1; }

…

}

ICL Network Extraction of Parameterized Modules

ICL supports the extraction of generic or parameterized modules.

Consider the following parameterized ICL example:

```
Module bus {
    DataInPort datain[ $MSB:0 ] ;
    DataOutPort dataout { Source RegD[ $MSB:0 ] ; }
    <…>
    Parameter MSB = 5 ;
}
```

ICL enables every instance of module "bus" to either proceed with the default value of 5 for
MSB, seen below in the instance "bus_inst1", or to overwrite the parameter value during
instantiation as shown in the next instance "bus_inst2".

```
Instance bus_inst1 Of bus {
    InputPort datain[5:0] = toplevel_datain[15:10] ;
}

Instance bus_inst2 Of bus {
    Parameter MSB = 7 ;
    InputPort datain[7:0] = toplevel_datain[17:10] ;
}
```

Although both instances are derived from the same ICL module, the width of the data input port is 6 bits for bus_inst1, but 8 bits for bus_inst2.

This type of parameterized module is also known in the design space, including the possibility to overwrite the default value. The following is a (partial) example that matches the previous example:

```
module bus ( datain, dataout, <…> );
    parameter MSB = 5 ;
    input [MSB:0] datain ;
    <…>
endmodule
```

With the following instantiations:

```
bus bus_inst1 ( .datain(toplevel_datain[15:10]), <…> ) ;
```

and

```
bus #(.MSB(7)) bus_inst2 ( .datain(toplevel_datain[17:10]), <…> );
```

ICL Network Extraction recognizes these design parameters and correctly generate the corresponding ICL Network, automating the parameter overwrite for each respective instance of parameterized design modules. In the example shown above, ICL Network Extraction generates the ICL instantiations bus_inst1 and bus_inst2 from the design example instances shown above.

Currently, you can only use simple expressions in the design modules, as complex parameter expressions are not supported. Everything that is allowed in ICL is supported.

ICL Extraction Commands

The following table provides a summary of all relevant ICL commands available in Tessent Shell.

Table 4-2. ICL Extraction Command Summary

Command	Description
add_icl_ports	Specifies top design ports that are added as DataInPort or DataOutPort ports in the ICL file generated during ICL extraction.
add_icl_scan_interfaces	Defines the names of one or several ICL ScanInterface definitions. You must follow through with set_icl_scan_interface_ports, defining the ports for each of the added scan interfaces.
add_ijtag_logical_connection	Defines a logical connection between a source and a target instance pin (port). The logical connection becomes part of the generated ICL module.

Table 4-2. ICL Extraction Command Summary (cont.)

Command	Description
delete_icl_ports	Undoes the effect of the add_icl_ports command on a specified list of top-level ports.
delete_icl_scan_interfaces	Deletes previously added scan interfaces.
delete_ijtag_logical_connection	Deletes previously added logical connections.
extract_icl	Checks the ICL connectivity rules between IJTAG instances and extracts the top-level ICL module.
get_attribute_value_list	Gains access to the values of attributes associated with ICL or design objects.
get_icl_extraction_options	Provides access to the settings specified by the set_icl_extraction_options command.
get_icl_scan_interface_list	Provides introspection into existing or added ScanInterface names.
get_icl_scan_interface_port_list	Provides introspection into existing or added ports of ScanInterfaces.
get_test_end_icall_list	Returns the iCalls added to the test_end procedure by the set_test_end_icall commands. Each iCall is a list containing the iProc and its arguments.
get_test_setup_icall_list	Returns the iCalls added to the test_setup procedure by the set_test_setup_icall commands. Each iCall is a list containing the iProc and its arguments.
read_cell_library	Reads an ATPG library.
read_icl	Reads ICL files into the internal ICL database.
read_verilog	Reads the design in Verilog format.
read_vhdl	Reads the design in VHDL format.
report_design_sources	Returns the pathnames or file extensions previously specified with the set_design_sources command.
report_icl_extraction_options	Provides a human-readable report of the ICL extraction options.
report_icl_modules	Reports loaded ICL modules in either ICL syntax or human-readable form.
report_ijtag_logical_connections	Prints a report of all previously added logical connections.
report_module_matching	Reports the identified name matching between the ICL modules and Verilog modules during the ICL extraction flow when used with the -icl option.

Table 4-2. ICL Extraction Command Summary (cont.)

Command	Description
report_module_matching_options	Reports the current settings defined by the set_module_matching_options command.
set_attribute_value	Sets the value of an attribute.
set_current_design	Specifies the top level of the design for all subsequent commands until reset by another run of this command.
set_design_sources	Specifies where the tool should search for the definition of undefined modules in the list of files specified by the read_icl command.
set_icl_extraction_options	Customizes the behavior of ICL extraction.
set_icl_scan_interface_ports	Defines a list of ports to be added to a scan interface, previously added by add_icl_scan_interfaces.
set_module_matching_options	Defines the acceptable prefixes and suffixes or regular expressions to use when matching an ICL module to a design module during ICL extraction.
set_procfile_name	Specifies a new procedure file for the tool to process at a later time.
set_test_end_icall	Adds an iCall to the start of the test_end procedure.
set_test_setup_icall	Adds an iCall to the end of the test_setup procedure.
write_icl	Writes out ICL modules created or read in with the read_icl command to the specified file.

Chapter 5
IJTAG Network Insertion

The IJTAG Network Insertion functionality enables you to connect existing instruments and insert SIBs, TDRs, and ScanMuxes to create your own IJTAG network.

The IJTAG Network Insertion functionality enables you to connect the network to a TAP controller or a pre-existing TAP controller in the design. The principle of IJTAG Network Insertion is straightforward using the create_dft_specification command. The tool reads in the ICL models for the instrument in the design and inserts a SIB or TDR based on how the ICL models need to be accessed. You can edit or modify the IJTAG network to suit your design requirements if necessary.

After you complete your design edits, you can generate the ICL description of the IJTAG network using the extract_icl command. Note that the tool does not automatically perform ICL extraction after the IJTAG network insertion because you have the option to perform additional editing before extraction.

Tessent IJTAG can generate and stitch up its own TAP or it can connect to a pre-existing TAP controller. If the IJTAG network needs to connect to a pre-existing TAP controller, an ICL for that TAP controller must be provided.
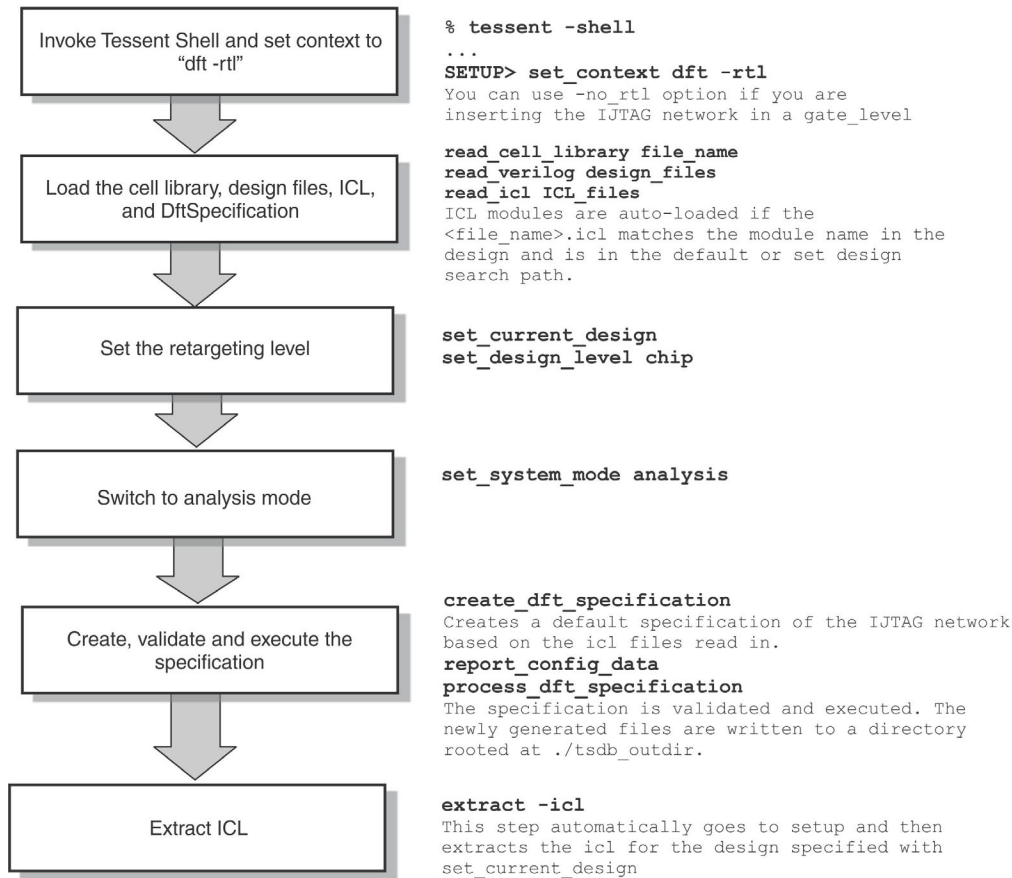
The IJTAG Network Insertion Flow

This section presents the basic IJTAG Network Insertion flow and lists the corresponding commands that implement the flow.

Figure 5-1 shows the basic IJTAG Network Insertion flow steps you perform with Tessent Shell.

Figure 5-1. IJTAG Network Insertion Flow

As Figure 5-1 shows, the IJTAG Network Insertion flow is relatively simple. Because you want to modify the design files, you have to set the tool to the dft context and then load a cell library, your design files, and the ICL for all instruments used (which can be loaded automatically for you). One create_dft_specification command instructs the tool to create the DftSpecification and the second command, process_dft_specification, runs a validation step before generating and making any edits to the design files.

As the tool processes the DftSpecification, it writes files to disk in an organized directory structure. These files include all inserted IJTAG network objects (SIBs, TDRs, and ScanMuxes) in both ICL and Verilog format and all modified design files. The IJTAG network is

automatically generated using the create_dft_specification command. However, you can always modify the created DftSpecification using editing commands or using the GUI with display_specification. As mentioned before, the ICL description of the network is not automatically generated because you may want to do further design editing. However, because all data resides in memory, you can perform the subsequent IJTAG Network Extraction step using the extract_icl command.

IJTAG Network Insertion Example

The following is an example of IJTAG Network Insertion.

```
set_context dft -no_rtl

##Read the libraries

read_cell_library ./library/adk_complete.tcelllib
read_cell_library ./library/memory.atpglib

##Read the netlist

read_verilog ./netlist/cpu_top_scan_tk.v
read_verilog ./generated/cpu_top_edt.v
read_verilog ./PLL/PLL.v -interface_only

##Read ICL and PDL files before set_current_design

read_icl ./PLL/PLL.icl
dofile ./PLL/PLL.pdl

set_current_design cpu_top

##Set design level before running set_system_mode analysis

set_design_level chip

##Specify the TAP pins using set_attribute_value

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

set_system_mode analysis

report_icl_modules

##Automatically read any ICL from the directories that verilog is picked from

create_dft_specification
report_config_data
```

##Use display_specification to edit or modify the specification or use editing commands ##if needed.

process_dft_specification

extract_icl

exit

The above example starts by setting context to dft and reading libraries.

The next step is reading in the Verilog netlist that has already been scan inserted and EDT IP inserted with the PLL module already present. For the PLL module, an ICL and PDL have been previously created and validated stand-alone. The PDL and ICLs for the PLL are read in next. The level at which the IJTAG network is inserted is specified using set_design_level. In this example, the IJTAG network is inserted at the top of the design and so the TAP pins are specified before running "set_system_mode analysis".

With create_dft_specification, the ICL for the PLL and the EDT instruments is automatically configured for insertion into an IJTAG network. This network can be reported using report_config_data. If the IJTAG network connection is desired then use process_dft_specification, otherwise use the editing commands or display_specification with the Config Data Browser to edit. The last step is extract_icl that provides the ICL for the level that was set using set_current_design.

Placement-Aware IJTAG Stitching

The placement-aware IJTAG stitching feature improves the ordering of elements in the IJTAG scan chain by using coordinates from a DEF file to create the shortest paths between the nodes.

Without any layout information, the tool alphabetically connects signoff blocks and IJTAG instruments into one IJTAG chain at each hierarchy level. An alphabetical connection may lead to layout and routing problems, but placement-aware stitching helps alleviate this problem.

Figure 5-2 compares an example IJTAG network layout with alphabetical stitching (left side) and placement-aware stitching (right side).

Figure 5-2. Placement-Aware Stitching
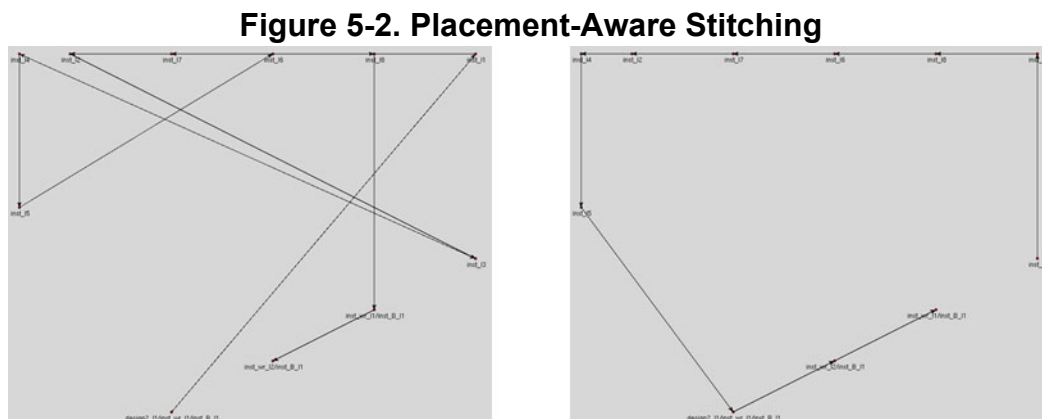
Placement-aware stitching uses coordinates from a DEF file to generate a DftSpecification wrapper that orders the IJTAG elements to create the shortest paths between the nodes. Placement-aware processing is automatic when you load a DEF file. (No special command or switch is necessary.) The DEF file should provide placement information for the scan-out pins of the IJTAG instruments. Placement-aware stitching uses the instance coordinates when coordinates of the scan-out pins are missing.

Stitching starts at the IJTAG scan-out port (ijtag_so or tdo). The tool determines which element has the closest placement and makes it the next node until all elements are in the chain. If the coordinates for the scan-out port are missing, stitching starts at the element with coordinates closest to the (0,0) point.

> **Note**
> You can also provide the node coordinates inside Tessent Shell by setting the "def_x_coordinate" and "def_y_coordinate" attributes.

Modification of the IJTAG Network Insertion Flow

In most usage cases, you can use the basic IJTAG Network Insertion flow. However, the following flow modifications are available to you, if needed.

Table 5-1. Modifications to the IJTAG Network Insertion Flow

To...	Description
Change the output directory root	By default, the process_dft_specification command writes all edited design files and generated IJTAG network object files into a sorted directory structure rooted at *./tsdb_outdir*. You can instruct the tool to use any other directory root using the set_tsdb_output_directory command; the tool creates it if it does not already exist.
Verify that the written DftSpecification is correct	You have the various options of verifying that the written DftSpecification is correct. See the options "-no_insertion" and "-validate_only" of the process_dft_specification command.
Transcript all design edits performed	The high-level process_dft_specification command runs a series of Tessent Shell editing commands, such as create_connections or create_instance. Usually, these commands are not transcripted but they may be useful when debugging. The process_dft_specification command provides you with the -transcript_insertion_commands option that adds all design editing steps performed during the processing of the DftSpecification to the transcript.
Run a DftSpecification	You can have more than one DftSpecification loaded for the current design; they differ based on a user-specified identifier. For more information, see the DftSpecification wrapper description. Selecting one or the other DftSpecification is easily done through the "id" option of the process_dft_specification command.

**Table 5-1. Modifications to the IJTAG Network Insertion Flow (cont.)**

To...	Description
Automatically run additional design edits	At the end of the DftSpecification processing, the modified design file is written to the output directory. If you want to further edit the design, the automatic writing of the design file is an unnecessary and potentially time-consuming step. The process_dft_specification command provides a method to tell the tool to first run your design editing command before writing out the final, modified design file. This file then includes both the inserted IJTAG network and your specified design edits.
Write a process_dft_specification.post_insertion Procedure	If you are writing a Tcl procedure, with the specific name of process_dft_specification.post_insertion, in memory, you load the Tcl procedure using the dofile command and target a file that contains this procedure. Alternatively, you can code the Tcl procedure in the main dofile directly. When the process_dft_specification command sees that a Tcl procedure exists, it automatically calls the Tcl procedure after all design edits specified by the DftSpecification have successfully completed, but before the write_design command is run.

How to Edit or Modify a DftSpecification

You can create a new DftSpecification and modify elements of an existing DftSpecification in one of two ways: using either the Config Data Browser window in Tessent Visualizer or an ASCII text editor.

The Config Data Browser provides you with a graphical interface that facilitates the creation and modification of specification elements as shown in Figure 5-3. The window displays a treelike representation of a specification you have defined using DftSpecification syntax. You can graphically view the hierarchy of the specification, move the placement of elements in the specification hierarchy, and create new elements as well as modify the properties for elements that are already defined.

Figure 5-3. Config Data Browser
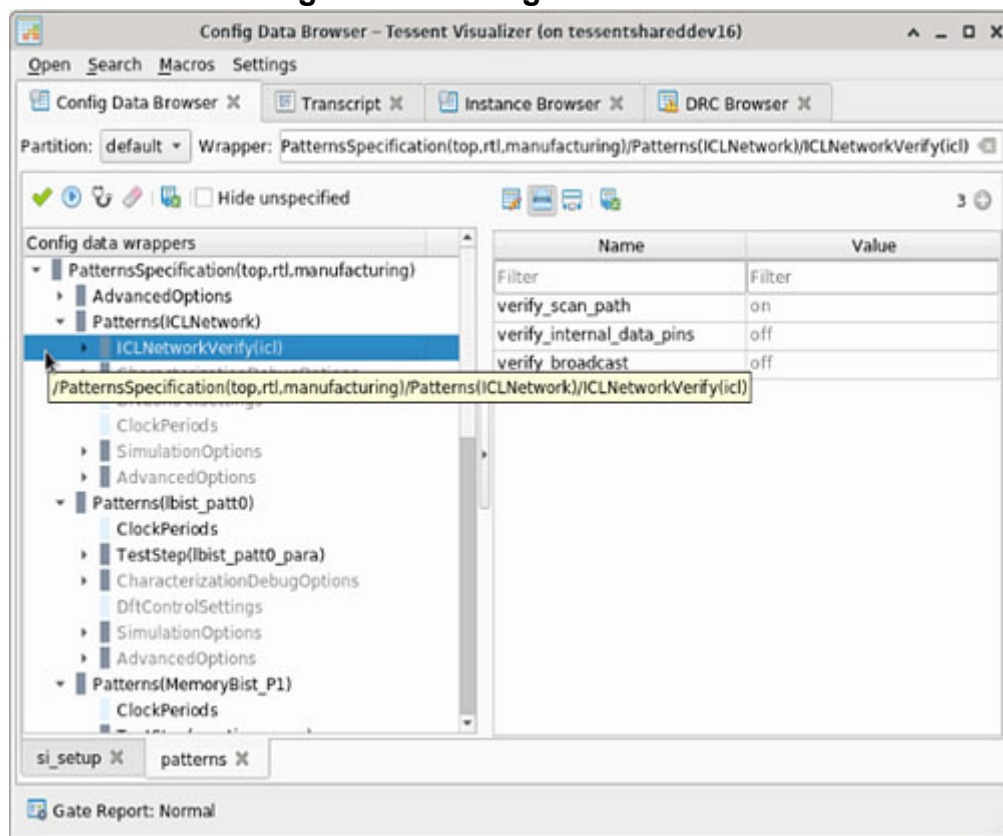
The graphical interface guides you through the IJTAG Network Insertion process by enabling you to choose only those objects that are legal at the current insertion step. When you are ready, you can also validate the IJTAG network specification before you instruct the tool to insert it into the design, and the tool highlights any errors.

You can create a new DftSpecification by using the following command:

display_specification -create

The command opens the Config Data Browser and creates the DftSpecification wrapper linked to the current design. For more information, see the display_specification command.

You can display a DftSpecification currently in memory and modify or append the specified IJTAG network by using the same command without the "-create" option. For example, the following line opens the DftSpecification for the user-provided ID "good3" for the current top-level design:

display_specification good3

For information on using the Config Data Browser, see "Config Data Browser" in the *Tessent Shell User's Manual*. For information on DftSpecification syntax and examples, see "DftSpecification" in the *Tessent Shell Reference Manual*. The grammar is completely described in the *Tessent Shell Reference Manual*.

- IJTAG Network Insertion Commands

Table 5-2 lists of all the relevant IJTAG Network Insertion extraction commands available in Tessent Shell.

Table 5-2. IJTAG Network Insertion Command Summary

Command	Description
display_specification	Displays the DftSpecification in the Config Data Browser. Enables subsequent editing of the displayed DftSpecification. Use the "-create" option to get a new, empty DftSpecification.
process_dft_specification	Runs the DftSpecification. It modifies the current design and creates design and ICL files as needed by the specification.
read_config_data	Reads a configuration file. For ICL Insertion, this configuration file is a DftSpecification.

DftSpecification Examples

This section presents examples of specific, common IJTAG Network Insertion tasks created using DftSpecification elements and syntax.

The Configuration-Based Specification chapter in the *Tessent Shell Reference Manual* documents all elements of the DftSpecification in great detail and also provides many examples. You should familiarize yourself with this chapter to understand all of the capabilities of the IJTAG Network Insertion flow.

Examples. 141

Examples

The examples in this section are based on the assumption that you are creating a DftSpecification using an ASCII text editor and not using the graphical interface provided by the Config Data Browser. However, these examples are valid with either method.

Connection of a Basic Scan Instrument to a SIB

In this example, you have an existing instrument with a single scan interface that you want to connect to a SIB, that is inserted.

Instrument

```
Module instrumentB {
   ScanInPort si;
   ScanOutPort so { Source R[0]; }
   ShiftEnPort se;
   SelectPort  sel;
   TCKPort clk;
   ScanRegister R[1:0] {
      ScanInSource si;
   }
}
```

DftSpecification

You use a SIB wrapper, identified by the ID "S3", and declare the instance path to the design instance of the instrument within the SIB wrapper. You do not need to specify "scan-in", "scan-out", or any of the other control ports because the tool retrieves this information from the ICL module description. Note that the instance path must already exist in the design because specification processing does not create the design instance; it only connects it as specified.

```
Sib(S3) {
   (design2_I1/instrumentB_I1) {
   }
}
```

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Result

The resulting IJTAG network has a SIB inserted. The SIB controls the SelectPort "sel" of the instrument instance at 'design2_I1/instrument_I1. The scan-out of the instrument is connected to the second scan-in port of the SIB; the scan-in of the instrument is connected to the same IJTAG scan chain as the first scan-in port of the SIB. Similarly, scan-, capture-, and update-control ports of the instrument are connected to the same source from which the SIB receives these control signals. Tck is connected similarly.

Connection of a Scan Instrument With More Than One ScanInterface

In this example, the instrument has two ScanInterface definitions. If you were to use the syntax of the "Connection of a Basic Scan Instrument to a SIB" example, the tool would not know which of the two scan ports to connect to the SIB. Therefore, in this case, you must also declare the name of the ScanInterface name as it is defined in the ICL file.

Instrument

```
Module instrumentC {
  ScanInPort si;
  ScanOutPort so1 { Source R1[0]; }
  ScanOutPort so2 { Source R2[0]; }
  ShiftEnPort se; SelectPort  sel1;
  SelectPort  sel2;
  TCKPort clk;

  ScanInterface P1 {Port so1; Port sel1;}
  ScanInterface P2 {Port so2; Port sel2;}
  ScanRegister R1[1:0] {
    ScanInSource si;
  }
  ScanRegister R2[1:0] {
    ScanInSource si;
  }
}
```

DftSpecification

In this example, all you want to do is connect the ports of ScanInterface P2 to the SIB "S3". The ScanInterface P1 is connected differently.

```
Sib(S3) {
  DesignInstance(design2_I1/instrumentB_I1) {
    scan_interface : P2 ;
  }
}
```

Connection of a Parallel Data Instrument

The examples "Connection of a Basic Scan Instrument to a SIB" and "Connection of a Scan Instrument With More Than One ScanInterface" showed how to connect an instrument with a ScanInterface. This example shows how to connect an instrument's parallel data ports to a TDR.

Instrument

```
Module instrumentA {
  DataInPort  INA[6:0];
  DataOutPort OUTA[7:0];
}
```

DftSpecification

You now want to connect the instrument to a TDR that is inserted as part of the specification
processing. You use the following basic connection specification:

```
Tdr(T3) {
  DataOutPorts {
    Connection(6:0) : design2_I1/instrumentA_I1/INA[6:0];
  }
  DataInPorts {
    Connection(7:0) : design2_I1/instrumentA_I1/OUTA[7:0];
  }
}
```

Result

The resulting IJTAG network contains a TDR register of eight bits. The size of the TDR is
automatically determined based on the connectivity requirements. The TDR contains eight data
input ports, seven data output ports, and all of the usual control signal ports. The seven lowest
bits of the instrument are connected to the seven data output ports of the TDR, which then lead
to the seven data input ports of the instrument. Similarly, the eight data output ports of the
instrument are connected to the eight data input ports of the TDR, from which the eight bits in
the TDR register capture the data.

Connection of a Parallel Data Instrument to the Top Level

This example shows how to connect the parallel data ports of an instrument to the data ports at
the top level. The example "Connection of a Parallel Data Instrument to a TDR" describes how
the instrument can be connected to a TDR.

Instrument

```
Module instrumentA {
  DataInPort INA[6:0];
  DataOutPort OUTA[7:0];
}
```

DftSpecification

You want to connect the instrument to the top level as part of the specification processing. You use the following basic connection specification:

```
IjtagNetwork {
  DataOutPorts {
    Connection(6:0) : design2_I1/instrumentA_I1/INA[6:0];
  }
  DataInPorts   {
    Connection(7:0) : design2_I1/instrumentA_I1/OUTA[7:0];
  }
}
```

If the top-level input ports are already connected to other parts of the design, you do not want to separate the connection. In this case, by default, if a multiplexer is needed, the tool automatically inserts a multiplexer in the direct parent instance of the connected instrument pin or pins, switching between the original connection and the newly-inserted connection described in the DftSpecification. The tool does not insert a multiplexer if the connected pin is floating or tied, or if the net connected to the pin has no fan-in (multiplexing : auto). The select input of the inserted multiplexer is connected to a newly-created top-level DataInPort.

The following example shows the use of the multiplexing parameter to force the tool to always insert a multiplexer between the top-level input port and the pins of the instrument, whether it is needed or not (multiplexing : on). Note, you can always instruct the tool to not insert any multiplexers (multiplexing : off).

```
IjtagNetwork {
  DataOutPorts{
    Connection(6:0) : design2_I1/instrumentA_I1/INA[6:0];
  }
  DataInPorts {
    Connection(7:0) : design2_I1/instrumentA_I1/OUTA[7:0];
    multiplexing : on;
  }
}
```

Connection of a Parallel Data Instrument to a TDR

This example connects an instrument's parallel data ports to a TDR that is inserted during specification processing. This example also shows the use of the multiplexing parameter.

The multiplexing parameter used in this example has the same options (auto (default), on, and off) as in the example "Connection of a Parallel Data Instrument to the Top Level", and the same type of analysis is performed to determine if a multiplexer is needed or not. The only difference is that for TDRs, the select port of the multiplexer is connected to an additional DataOutPort fed by an additional bit of the inserted TDR.

DftSpecification

```
Tdr(T3) {
  DataOutPorts{
    Connection(6:0) : design2_I1/instrumentA_I1/INA[6:0];
    multiplexing : on;
  }
  DataInPorts{
    Connection(7:0) : design2_I1/instrumentA_I1/OUTA[7:0];
  }
}
```

Result

The size of the TDR is automatically determined based on the connectivity requirements. In this example, the resulting IJTAG network contains a TDR register of nine bits: eight data input ports, eight data output ports, and all the usual control signal ports. The seven lowest bits of the TDR register are connected to the seven lowest data output ports of the TDR, which then lead to the seven data input ports of the instrument. Similarly, the eight data output ports of the instrument are connected to the eight data input ports of the TDR, from which the lowest eight bits in the TDR's register capture the data. The ninth bit in the TDR register and the eighth data out port is needed because of the multiplexing select line requirement.

Creation of a TDR With More Bits Than Needed for the Current Specification

This example builds on example "Connection of a Parallel Data Instrument to a TDR" and shows how to reserve additional bits in the TDR when the connection is unknown during specification processing.

You can use the parameter "length" to specify how many bits the TDR's register should have; you cannot specify a length that is smaller than needed to satisfy all other connection requirements.

DftSpecification

You add a TDR register of length 10.

```
Tdr(T3) {
  length : 10 ;
  DataOutPorts {
    Connection(6:0) : design2_I1/instrumentA_I1/INA[6:0];
  }
  DataInPorts {
    Connection(7:0) : design2_I1/instrumentA_I1/OUTA[7:0];
  }
}
```

Connection of an EDT Controller

The next example continues to build on the example "Creation of a TDR With More Bits Than Needed for the Current Specification." This example describes a flow in which the EDT IP is already inserted, and you want to create the IJTAG network and connect the static EDT configuration bits to a TDR in the network. (Section "IJTAG ATPG Flow Overview" on

page 151 describes an example in which an EDT IP is connected to an existing TDR as part of the EDT IP Insertion flow.)

DftSpecification

The EDT ICL module contains only DataInPort objects that statically configure the EDT IP instance. One of those control bits is "edt_bypass". In the specification, you want to connect bit 9 of the TDR "T3" to the edt bypass port "CA_bypass" in the EDT IP.

```
Tdr(T3) {
  length : 10 ;
  DataOutPorts {
    Connection(8) :   design2_I1/edtIP_I1/CA_bypass;
    Connection(6:0) : design2_I1/instrumentA_I1/INA[6:0];
  }
  DataInPorts {
    Connection(7:0) : design2_I1/instrumentA_I1/OUTA[7:0];
  }
}
```

Note that you may need to add the "set_edt_pins bypass -" command to the EDT instance's dofile to denote that the edt bypass pin is now internally connected.

Connection to a TAP Controller

This example shows how to connect the IJTAG network to an existing TAP controller. The specification describes the ports of the TAP that connect to the chip internal side (the host interface). The TAP's left-side ports (TDI, TDO, TCK, TMS, TRST) should already be connected.

DftSpecification

These two examples introduce a HostScanInterface onto which the IJTAG network is connected. A set of design instance pin pathnames are used to denote the functions that each of the design instance pins perform on the host scan interface:

```
IjtagNetwork {
  HostScanInterface(ijtag) {
    Interface {
      scan_in : main_tap/tdi;
      scan_out : main_tap/host1_scanin;
      select : main_tap/host1_select;
      capture_en : main_tap/ce;
      shift_en : main_tap/se;
      update_en : main_tap/ue;
      reset : main_tap/tlr;
      reset_polarity : active_low;
      tck  : main_tap/tck;
    }
  }
}
```

If the TAP ICL and design modules are available and loaded, the same can be accomplished more easily in a way similar to the scan instrument connection explained earlier in "Connection

of a Basic Scan Instrument to a SIB." This example assumes that the TAP ICL defines a ScanInterface named "host1" that includes the scan and outgoing controlling port functions: the ScanInPort (from the chip-side back to the TAP), ToSelectPort, ToCaptureEnPort, ToShiftEnPort, and ToUpdateEnPort.

```
IjtagNetwork {
  HostScanInterface(ijtag) {
    Interface {
      design_instance : main_tap ;
      scan_interface : host1 ;
    }
  }
}
```

Creation of a Hierarchy of SIBs

It is a common practice to define a hierarchy of SIBs. For example, you may place a first level of SIBs in front of every power domain or in front of every core. Then, within this core, you may need to add additional SIBs. This example shows how to define a set of SIBs that are connected serially just by their relative order of declaration.

DftSpecification

The sequence of SIB, TDR, and ScanMux elements in the DftSpecification, read from top to bottom, defines the order in which they appear in the IJTAG network, from the scan output towards the scan input. This means that for this example the SIB with the ID "S1" is closest to the scan output, connected to the scan input port of the TAP, the SIB with ID "S3" is connected to the TDI, and the SIB with the ID "S2" is located between the two. The first scan input pin of "S2" is connected to the scan output pin of "S3", and the scan output pin of "S2" is connected to the scan input pin of "S1". For a detailed example and schematic refer to the description of the IjtagNetwork wrapper in the *Tessent Shell Reference Manual*.

```
IjtagNetwork {
  HostScanInterface(ijtag) {
    Interface {
      design_instance : main_tap ;
      scan_interface  : host1 ;
    }
    Sib(S1) {
    }
    Sib(S2) {
    }
    Sib(S3) {
    }
  }
}
```

Next, you want to place "S2" in the ICL sub-network controlled by "S1". This is easily done by moving the SIB(S2) wrapper declaration into the wrapper of SIB(S1) as follows.

```
IjtagNetwork {
  HostScanInterface(ijtag) {
    Interface {
      design_instance : main_tap ;
      scan_interface  : host1 ;
    }
    Sib(S1) {
      Sib(S2) {
      }
    }
    Sib(S3) {
    }
  }
}
```

Usage of a ScanMux

The following example shows how to use a ScanMux. A ScanMux has two scan input ports (input 0 and input 1), one scan output port, and one 1-bit wide select port. If you need a larger ScanMux, you must concatenate multiple ScanMuxes.

You should try to control the mux select line using a TDR bit at the scan output side of the ScanMux; this enables you to change the select line value by scan shifting through the mux and the TDR. If this is not possible, you should try to control the ScanMux from a TDR or the TAP higher up in the hierarchy. Avoid trying to control the mux select line using an ICL object (like a TDR) that is in only one of the two scan input paths because this can lock the mux into only one configuration.

DftSpecification

The following example shows the usage of a ScanMux in which the mux input 0 is connected to a 3-bit TDR, input 1 is connected to a 5-bit TDR, and there is a single-bit TDR after the mux, further towards the scan out, which means it is defined first in the DftSpecification. This TDR is connected to the mux select line. The connection is done by referencing the ID of the TDR and the generic DataOut(0) token and referencing the data output port connected to the TDR register bit 0 (which, in this case, is the only bit of the TDR). Refer to the *Tessent Shell Reference Manual* for complete information about the ScanMux.

```
Sib(S1) {
  Tdr(Tsel) {
  }
  ScanMux(SM1) {
    Select : tdr(Tsel)/DataOut(0);
    Input(0) {
      Tdr(T0) {
        length : 3 ;
      }
    }
    Input(1) {
      Tdr(T1) {
        length : 5 ;
      }
    }
  }
}
```

Move of a SIB, TDR, or ScanMux Deeper Into the Design Hierarchy

The following example shows how to change the default placement of an element. By default, all inserted SIBs and ScanMuxes are inserted into the top-level design module. The default location for a TDR depends on whether or not the TDR uses DataInPorts, DataOutPorts, or DecodedSignals connections. If the TDR does not use any of these, the TDR is also placed in the top-level design module; otherwise, it is automatically placed in the common ancestor of the connections.

DftSpecification

You use the "parent_instance" parameter to specify where in the design hierarchy the ICL object should be inserted. The SIB, TDR, and ScanMux elements all have a "parent_instance" parameter. You can force the TDR to be placed in the top-level design module by specifying a period (.) as the parent_instance. This example specifies that the SIB should be inserted at the design instance path "design2_I1/core1." The instance path must exist, but any missing ports are created as needed to connect the object to the rest of the IJTAG network.

```
Sib(S3) {
  parent_instance : design2_I1/core1;
  (design2_I1/core1/instrumentB_I1) {
  }
}
```

Change of the Instance Name of a SIB, TDR, or ScanMux

In the following example, you specify the name of an IJTAG network object. By default, all inserted IJTAG network objects such as SIB, TDR, and ScanMux have a predetermined name that is composed of the DftSpecification id, the id of the object (like "S3" below), and the type of the object (like SIB).

You can use the "leaf_instance_name" parameter to change this default naming convention. However, you are completely responsible for ensuring that this name is a legal design instance name. The tool validates the given name before insertion. The tool also uniquifies the name if

needed, based on the default or specified uniquification rules; you can change the uniquification rules using the command.

DftSpecification

You name the SIB "sib_S3". Of course, the "leaf_instance_name" can be combined with the "parent_instance" parameter.

```
Sib(S3) {
  leaf_instance_name : sib_S3;
  (design2_I1/instrumentB_I1) {
  }
}
```

Change of the Design and ICL Port Names of a SIB, TDR, or ScanMux

In the following example, you specify the name for ports of an IJTAG network object. The SIB, TDR, and ScanMux modules have default names for all ports. You can use the Interface wrapper, to change the default names for one, several, or all ports. The mechanism is the same for the SIB, TDR, and ScanMux, although the names and semantics of the ports differ. The example below shows this mechanism for the ScanMux only.

DftSpecification

You change the name of the ScanMux input ports from the default "mux_in0" and "mux_in1", respectively, to "mux_input0" and "mux_input1"; the names of all other ports of the ScanMux are not changes and still have their default names:

```
ScanMux(SM1) {
  Interface {
    input0 : mux_input0 ;
    input1 : mux_input1 ;
  }
}
```

The purpose of the IJTAG functionality within ATPG is to significantly simplify the test_setup procedure as well as the optional test_end procedure by using IJTAG to configure EDT IPs and any other embedded IJTAG instruments needed for scan ATPG.

Because IJTAG is available only in Tessent Shell and is not part of the classic ATPG point tools (FastScan and TestKompress), you must use ATPG within Tessent Shell to take advantage of this feature. Refer to "Tessent Shell Introduction" in the *Tessent Shell User's Guide* for more information.

IJTAG ATPG Flow Overview

This section outlines the steps of the IJTAG ATPG flow, especially for enabling IJTAG to set up EDT IPs in the design.

The flow has some optional steps, depending on what files are already available. For example, if a complete ICL description is already available, Step 1, ICL Network Insertion as well as Step 4, ICL Extraction are not necessary. Similarly, if no embedded compression is used, no EDT IP needs to be inserted and Step 2 of the flow can consequently be skipped.

1. Use the "IJTAG Network Insertion" feature to add the hardware that controls the static signals of EDT and any other instruments to be driven through IJTAG. This can be done on the RTL or synthesized netlist.

2. Have the tool generate ICL and PDL for the EDT IP when the EDT IP is generated.

3. Provide ICL models for any other modules involved in the network (if any), such as the TAP and TDRs.

4. Perform ICL extraction so the connectivity of the ICL network is extracted from the design.

5. Run ATPG. The test_setup (or test_end) procedure may include iCalls that reference iProcs on any ICL instance. This enables the low-power mode of an EDT IP, for example, and you can have the tool generate the sequence needed to do that in test_setup.

For a detailed description of the IJTAG ATPG flow including details of all the Tessent Shell commands used in the flow, see "A Detailed IJTAG ATPG Flow" on page 160.

IJTAG Features of ATPG in Tessent Shell

This section introduces the ICL module and PDL of the EDT IP. It then explains how this and the ICL/PDL of other instruments can be used as part of test_setup and test_end for ATPG in Tessent Shell.

The following topics are described in this section:

EDT IP Setup for IJTAG Integration

Tessent Shell commands are used to generate the IJTAG files needed to integrate EDT IP with IJTAG.

As part of the EDT IP creation, the Tessent Shell command write_edt_files is used to generate several files needed for subsequent flow steps, like synthesis or test pattern generation. By default, the write_edt_files command causes the tool to generate dofiles that use IJTAG to describe the static configuration inputs of the EDT IP. These static configuration inputs select certain features of the EDT IP: edt bypass, single chain bypass, low power, and edt configuration. Please see set_edt_pins in the *Tessent Shell Reference Manual* to learn more about the purpose and usage of these static configuration pins.

> ⎯⎯ **Note** ⎯⎯
> When no static control ports are present in the EDT logic, the ICL description of the EDT IP is not required for pattern generation. However, the ICL is still generated referencing only the edt_clock port to support other Tessent flow functionality such as SDC generation, that is tied to the presence of the ICL description of DFT instruments.

The write_edt_files command generates an ICL file similar to the following:

```
Module CA_edt {
    DataInPort CA_CONFIGURATION    { RefEnum ConfigTable; }
    DataInPort CA_LOW_POWER        { RefEnum OnOffTable; }
    DataInPort CA_BYPASS           { RefEnum OnOffTable; }

    Enum ConfigTable {
        LC = 1'b0;
        HC = 1'b1;
    }

    Enum OnOffTable {
        off = 1'b0;
        on  = 1'b1;
    }

    Attribute tessent_instrument_type = "mentor::edt";
    Attribute tessent_signature = "7b07783c53f9534b437c62964b2aad63";
}
```

Your ICL file may vary in the descriptions of the actual data ports because only those static configuration inputs that you have defined for the particular EDT IP are used. The name of the ICL data port is identical to the name of the corresponding design port. Similarly, the name of the ICL module is identical to the name of your EDT IP in the design.

In addition to the ICL file, the write_edt_files command also generates a matching PDL file linked to the generated ICL module. It features a single iProc named "setup" that is iCall'ed for the respective EDT logic instance. The setup iProc takes parameter-value pairs for the static configuration inputs. An example iCall for an EDT IP instance named CA_edt_instance might look as follows:

```
iCall CA_edt_instance.setup edt_configuration LC \
                        edt_bypass on edt_single_bypass_chain off
```

Generic semantic terms, such as "edt_configuration" or "edt_bypass" are used for the parameter denoting the static configuration ports of an EDT IP. The generated PDL file translates these semantic terms into the pin names used for your EDT IP instance. There is no need to provide those to the PDL file. Further, there is no need to list every other option possible for the EDT IP. Only the parameter-value pair that is changed from its default value must be specified. Table 6-1 lists all the possible ports by parameter keyword and their default values.

Table 6-1. EDT Configuration Keywords and Values

Parameter Keyword	Default Value
edt_configuration	0 (== LC)
edt_low_power_shift_en	0 (== off)
edt_bypass	0 (== off)
edt_single_bypass_chain	0 (== off)

When the iCall to the generated setup iProc is placed in the test_setup procedure using the desired parameter-value pairs, it statically configures the EDT IP automatically as part of test_setup. On the design side, these ICL data ports must be added to the ICL network. For example, by connecting them to the parallel data output of a Test Data Register, which is in turn part of the ICL scan network. (See Chapter 5, IJTAG Network Insertion to learn how to create such a network). They can also be connected directly to ports. The PDL retargeting engine reads the PDL that is called by the test_setup procedure and determines what needs to be shifted into the top-level design to set the static configuration bits in the PDL. This is done automatically by the PDL retargeting engine as part of the test_setup simulation.

How to Set Up Embedded Instruments Through Test Procedures

In the patterns -scan ATPG context, IJTAG is only allowed in test_setup and test_end procedures. It is not allowed in any other procedure or in ATPG's analysis mode. Also, only iReset and iCall commands are allowed in the test procedures.

> **Note**
> For complete information on test procedure file creation, syntax, and structure, refer to "Test Procedure File" in the *Tessent Shell User's Manual*.

The iProc called by the iCall command in the test procedure may use all supported PDL commands. The following example illustrates the usage:

```
set time scale 1.000000 ns ;
   timeplate gen_tp1 =
      force_pi 0 ;
      measure_po 10 ;
      pulse tck_a 40 20;
      period 100 ;
end;

procedure test_setup =
   timeplate gen_tp1 ;
   // cycle 1 starts at time 0
   cycle =
      force test_mode_EDT 1 ;
      force test_mode_MBIST 0 ;
   end;

   iCall OCC_Inst1.setup
   iCall coreA.blockA.edtInst1.setup edt_bypass ON ;
   iCall coreA.blockB.edtInst1.setup edt_bypass OFF ;

end;
```

When a pulse statement exists in the timeplate that is being used at the time of the iCall for the TCK clock, then the tck ratio is 1. If a pulse statement does not exist, the tck ratio defaults to 4.

While the tool is processing the test_setup procedure during the transition to the analysis system mode, if it encounters an iCall statement, it calls the PDL retargeting engine to retarget the called iProcs to the current top level. The computed, internal sequence then replaces the iCall in the internal representation for the test procedure. This processed test procedure, which only includes events with respect to the port of the current design, is what is passed on to DRC. So DRC indirectly verifies the resulting sequence. For DRC, there is no difference between test_setup (or test_end) patterns defined through "force" and "pulse" statements or those defined through PDL. The latter is just much more convenient, especially when there are many embedded instruments to set up through a TAP controller. The iCalls in the test procedures must invoke loaded iProcs, which in turn may use any legal PDL command.

Note that IJTAG is not part of the actual ATPG pattern creation. It only provides the means to specify the test_setup and test_end procedures, or part of them. Consequently, the ATPG patterns written to disk contain patterns derived from the PDL within the test_setup and test_end sections. Again, there is no difference from the traditional way of defining test_setup or test_end patterns.

For IJTAG to work within ATPG's test_setup and test_end, the Verilog netlist and the entire ICL hierarchy and PDL command files must be loaded into Tessent Shell. This includes the top-level ICL file. If there is no top-level ICL, Tessent Shell can generate one using the "IJTAG Network Insertion" functionality.

How to Set Up Embedded Instruments Through the Dofile

Tessent Shell provides a convenient way of adding IJTAG iCalls to test_setup and test_end procedures from within the dofile, where all design introspection and Tcl commands are available to specify the needed IJTAG commands.

The test procedure example described above in the "How to Set Up Embedded Instruments Through Test Procedures" section shows the explicit usage of the iCall PDL command inside a test procedure. But it may not be convenient to embed the iCall commands within the test procedure file especially if they are generated from the tool based on introspection. The tool, therefore, enables a much more convenient way of adding IJTAG iCalls to test_setup and test_end procedures from within the dofile, where all design introspection and Tcl commands are available to specify the needed IJTAG commands.

The set_test_setup_icall and set_test_end_icall commands are available in the setup mode of the "patterns -scan" context to declare one or more iCalls to be added to the end of test_setup or the beginning of test_end, respectively, without the need to edit the procedure file itself.

Command lines in the dofile matching the test_setup procedure example above would look like this:

```
SETUP> set_test_setup_icall {OCC_Inst1.setup} -append
```

SETUP> set_test_setup_icall {coreA.blockA.edtInst1.setup edt_bypass ON} -append

SETUP> set_test_setup_icall {coreA.blockB.edtInst1.setup edt_bypass OFF} -append

As before, these three iCalls declared to the ATPG tool through the set_test_setup_icall command become part of the test_setup procedure and are run when the test_setup procedure is processed.

The next example demonstrates the convenience provided by these dofile commands. It sets all EDT instances anywhere in the design to the edt_bypass off mode of operation. For this, it first introspects the design to find all EDT ICL modules that were named MyEDT in this example, then calls the set_test_setup_icall command for each instance by looping through a (string) list of instance path names.

SETUP> foreach edt_inst [get_name_list [get_icl_instances -of_module MyEDT]] {

set_test_setup_icall [list $edt_inst.setup edt_bypass off] -append

}

Observe that the "-append" option can also be used for the very first set_test_setup_icall command without any error given by the tool.

In all of the examples listed above, the iCalls are run sequentially in the order they were declared. IJTAG also enables the parallel running of iCalls. The next example shows an iProc that sets up multiple OCC instruments, all in parallel. The PDL retargeting engine tries to find a solution within the given hardware constraints that enables this parallel run. If it cannot find such a solution, it serializes the parts that cannot be parallelized. This iProc example below assumes that the top-level design/ICL module is named "top":

```
iProcsForModule top
iProc parallel_OCC_setup {
    iMerge -begin
        iCall coreA.OCC_Inst.setup
        iCall coreB.OCC_Inst.setup
    iMerge -end
}
```

Then, in the dofile, there is only one iCall in test_setup to run for all OCCs:

SETUP> set_test_setup_icall {parallel_OCC_setup} -append

There are two ways of defining the iProc above, either in a separate file, which can optionally be generated from the dofile and then sourced, or in the dofile directly, since the iProcsForModule as well as the iProc keywords are all registered dofile commands.

Implicit and Explicit iReset Commands

Whether using explicit iCalls in the test_setup procedure or through the set_test_setup_icall command, in both scenarios the tool needs to know the initial state of the ICL network. To this end, it issues an implicit iReset command before it commences with the PDL retargeting.

Assume again that the following test_setup procedure is given to the tool:

```
procedure test_setup =
   timeplate gen_tp1 ;
   // cycle 1 starts at time 0
   cycle =
      force test_mode_EDT 1 ;
      force test_mode_MBIST 0 ;
   end;

   iCall OCC_Inst1.setup ;
   iCall coreA.blockA.edtInst1.setup edt_bypass ON ;
   iCall coreA.blockB.edtInst1.setup edt_bypass OFF ;
end;
```

To explain this implicit iReset, the test_setup procedure below shows what the tool is actually evaluating:

```
procedure test_setup =
   timeplate gen_tp1 ;
   // cycle 1 starts at time 0
   cycle =
      force test_mode_EDT 1 ;
      force test_mode_MBIST 0 ;
   end;

   iReset ;
   iCall OCC_Inst1.setup ;
   iCall coreA.blockA.edtInst1.setup edt_bypass ON ;
   iCall coreA.blockB.edtInst1.setup edt_bypass OFF ;

end;
```

Notice the tool inserted an "iReset" command, just before the very first iCall. This implicit iReset happens if the set_test_setup_icall command was used instead of the explicit iCalls in the test_setup procedure.

While this iReset is needed to establish the initial state of the ICL Network, it could destroy your design setup state reached through the cycles of force and pulse statements, especially if there is a TAP controller that would be reset through the iReset command. To prevent this

implicit iReset, you place an explicit iReset command at a convenient location in the test_setup procedure, for example right at the beginning of test_setup:

```
procedure test_setup =
   timeplate gen_tp1 ;
   // cycle 1 starts at time 0

   iReset ;

   cycle =
      force test_mode_EDT 1 ;
      force test_mode_MBIST 0 ;
   end;

   iCall OCC_Inst1.setup ;
   iCall coreA.blockA.edtInst1.setup edt_bypass ON ;
   iCall coreA.blockB.edtInst1.setup edt_bypass OFF ;
end;
```

The tool then no longer issues the implicit iReset. However, you must make sure that after all cycles are applied, the state of the ICL network components is the reset state. This must hold true in particular for the state of a TAP controller you might have operated in the cycle statements.

You can also combine the explicit iReset with the set_test_setup_icall command to achieve the same result as the test_setup procedure above:

```
procedure test_setup =
   timeplate gen_tp1 ;
   // cycle 1 starts at time 0

   iReset ;

   cycle =
      force test_mode_EDT 1 ;
      force test_mode_MBIST 0 ;
   end;
end;
```

With the dofile:

SETUP> set_test_setup_icall {OCC_Inst1.setup} -append

SETUP> set_test_setup_icall {coreA.blockA.edtInst1.setup edt_bypass ON} -append

SETUP> set_test_setup_icall {coreA.blockB.edtInst1.setup edt_bypass OFF} -append

As before, the iCalls are inserted after the last cycle statement, but with no implicit iReset added before, because there is already an explicit iReset in the test_setup procedure.

> _____ **Note** _____
>
> In the 2014_1 release and all subsequent releases, PDL commands (iReset, iCall, and iMerge) are no longer allowed in procfiles in the patterns -ijtag context.

A Detailed IJTAG ATPG Flow

This section illustrates the entire IJTAG ATPG flow, including all required commands, for using IJTAG to set up an EDT IP as well as an OCC instrument.

The flow has the following entry points:

1. (Optional) ICL Network Insertion.

2. (Optional) EDT IP insertion.

3. (Optional) Generation of the top-level ICL description.

4. ATPG.

The information shown in the following example can be similarly extended to any other embedded instrument that must be set up prior to scan. The flow starts with the insertion of the ICL Network, then continues to the insertion of the EDT IP, the top-level ICL generation through extraction from the design, and concludes with ATPG.

For example, if no embedded compression is used, Step 2 can be skipped. Similarly, if the design already comes with a complete ICL model including the top-level ICL module and the ICL network connecting all relevant instruments, the flow simplifies to only Step 4.

Flow Step 1 inserts an ICL Network. In particular, the DFT specification defines a TDR to which Step 2b connects the EDT IP's edt_bypass port. (For simplicity of the example, only the edt_bypass signal of the EDT IP is shown.) In this example, the TDR has the instance name "MyTDR" and a DataOutPort named "td". This DataOutPort is connected to the edt_bypass port of the EDT IP in Step 2b of the flow.

1. (Optional) ICL Network Insertion.

   See Chapter 5, "IJTAG Network Insertion" for details. If there is already a complete ICL network, proceed to Step 2.

   > **set_context dft**
   >
   > **read_cell_library <library files>**
   >
   > **read_verilog <design files>**
   >
   > **read_icl <ICL files for all instruments>**
   >
   > **;# Note that the ICL modules are auto-loaded if the <filename>.icl matches**
   >
   > **;# the module name in the design and is in the default or set design search path.**

> **set_current_design**
>
> **set_system_mode analysis**
>
> **read_config_data <the specification file of the ICL Network to be inserted>**
>
> **process_dft_specification**

2. (Optional) EDT IP insertion.

   If you entered the flow with Step 1, you can proceed to EDT IP insertion without leaving Tessent Shell. If you exited Tessent Shell or this is your first flow step, just read the cell library and the design and ICL files again. If you do not use EDT IP, proceed to Step 3.

   a. Follow through with the EDT IP insertion flow as usual.

   b. While in setup mode, instruct the tool to connect the Bypass port of the EDT instance to the td output port of the IJTAG TDR:

   > **set_edt_pins bypass - MyTdr/td**

   c. When in analysis mode, write out the EDT IP inserted files. By default, the tool performs IJTAG mapping and writes out the ICL and PDL file.

   > **write_edt_files <all other option>**

3. (Optional) Generation of the top-level ICL description.

   After synthesis of the EDT IP, generate the top-level ICL file. This functionality is only available in the patterns -ijtag context. If you already have a complete top-level ICL file from somewhere else, proceed to Step 4.

   > **set_context patterns -ijtag**
   >
   > **read_cell_library < library files>**
   >
   > **read_verilog < design files>**
   >
   > **read_icl <your ICL files (EDT module, TAP module, TDR module, …)>**
   >
   > **;# Note that the ICL modules are auto-loaded if the <filename>.icl matches**
   >
   > **;# the module name in the design and is in the default or set design search path.**
   >
   > **;# This is the case for SIB, TAP, and TDR modules if they were inserted**
   >
   > **;# using the ICL Network insertion functionality of the Tessent Shell, shown in**
   >
   > **;# Step 1. This is not the case for the EDT IP ICL module generated in Step 2**
   >
   > **;# because the provided EDT filename is usually different from the EDT IP**
   >
   > **;# module name.**
   >
   > **set_module_matching_options <as needed only>**
   >
   > **;# to bridge naming conventions between the design and ICL**
   >
   > **set_current_design**
   >
   > **;# Make sure that you have loaded the design as well as all ICL files before you**

;# issue this command. You also must have issued the module matching

;# command beforehand. The reason for this is that set_current_design

;# processes all design information and makes the link between the ICL and

;# design modules. Anything loaded afterward is not part of the design

;# going into analysis mode.

set_system_mode analysis

write_icl -output_file [get_single_name [get_current_design]].icl -replace

;# Assume the top-level module is named "top".

4. ATPG.

set_system_mode setup

;# Only needed if you come from a previous step in the flow.

set_context patterns -scan

set_test_setup_icall { OCC_Inst1.setup } -append

;# Adds the iCall to a PDL iProc to test_setup that implements the setup of an

;# instrument before ATPG. In this example the setup iProc comes with the OCC.

dofile ./MyEDT/top_setup.pdl

;# This is the dofile that was generated by the earlier EDT IP insertion step.

;# The actual filename depends on the filename chosen in Step 2.c.

;# As a key IJTAG setup component it runs the "set_test_setup_icall"

;# command explained earlier. It is important to understand the generated dofile.

set_system_mode analysis

create_patterns

;# Do not forget to save your patterns and the flat model.

This chapter presents selected IJTAG topics of interest.

Note
For an example to create a fully compliant IEEE 1500 Wrapper Serial Port (WSP) using Tessent Shell's IJTAG features, see "How to Create a WSP Controller in Tessent Shell" in the *Tessent Shell User's Manual*.

The topics covered include the following:

- **ICL Modeling versus Verilog Modeling** — The first example demonstrates that there is no need to model the Verilog description of a module 1:1 in ICL. It is sufficient to model the IO-behavior of an instrument while the die is in IJTAG mode of operation.

- **ICL and PDL Namespaces** — The ICL namespaces are not currently supported, but the concept is discussed for completeness.

- **Skipping the Run-Test/Idle State** — A discussion of the option to skip the run-test/idle state that the TAP controller can pass through during the scan operation.

- **Default Values in ICL** — Different ways to define default values in ICL are described in this section along with examples.

- **Attributes of the ICL Extraction Flow** — ICL attributes follow the same use model as attributes elsewhere in Tessent Shell. The example in this section shows the role attributes play in the ICL extraction flow.

- **Scan Chain Integrity Test in Tessent IJTAG** — An example in this section shows how to create an ICL scan chain integrity test.

- **How to Define Auto-Return Values and Addressable Registers in ICL** — The example in the "How to Define Auto-Return Values in ICL" section describes a particular ICL construct that instructs the PDL retargeter to automatically restore defined bits on a register to a prescribed value by the end of the iApply time frame. The example is designed to automatically turn off a bit in a scan register encoding a read enable, so that subsequent read operations may proceed correctly. This automation is enabled in the PDL retargeter without your intervention.

ICL Modeling versus Verilog Modeling

This section describes how to create the ICL model of a Verilog module.

Figure 7-1 shows a gate-level description of a Verilog module.

Figure 7-1. Gate-Level Verilog Module Example
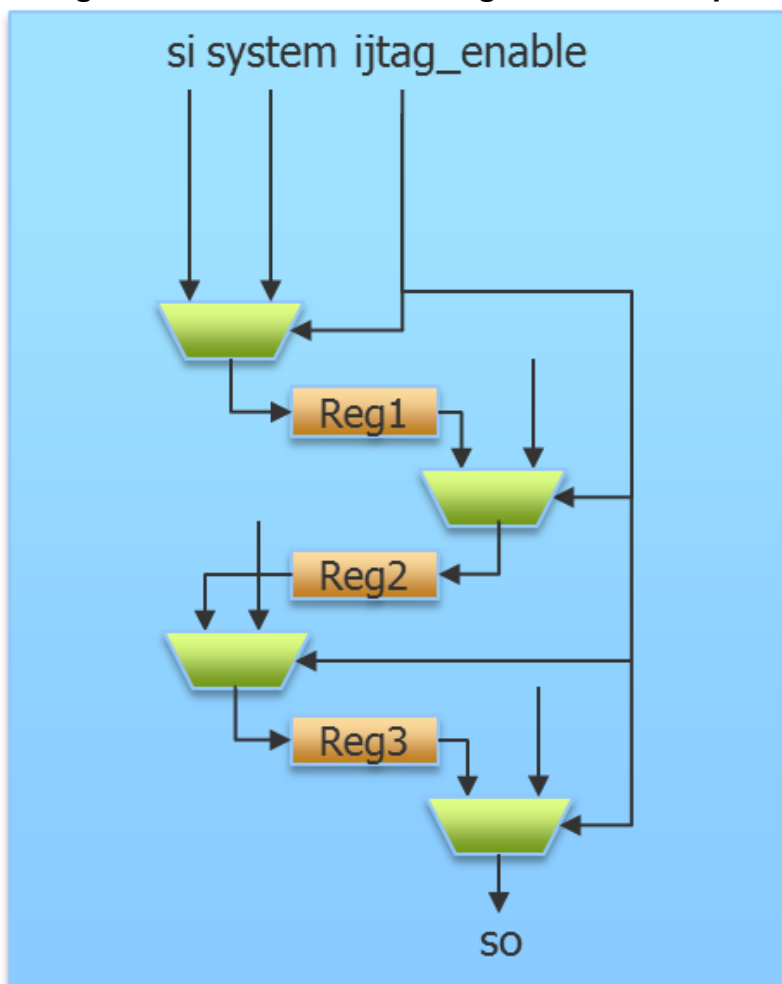
For simplicity, the example does not show any clock or enable signals in the Verilog or the ICL.

Assume that the input port 'ijtag_enable' is active and selects the left-most input of each multiplexer while the die is in the ijtag mode of operation. Under the assumption of the ijtag_enable value being constant, you can model the Verilog module in ICL as follows:

```
Module M1 {
  ScanInPort  si ;
  ScanOutPort so { Source Reg3[0] ; }

  ScanRegister Reg1[2:0] {
    ScanInSource si ;
  }

  ScanRegister Reg2[2:0] {
    ScanInSource Reg1[0];
  }

  ScanRegister Reg3[2:0] {
    ScanInSource Reg2[0] ;
  }
}
```

This is a straightforward translation of the Verilog module's scan register chain. Just to show that this is not the only possible translation, consider this following ICL module:

```
Module M2 {
  ScanInPort  si ;
  ScanOutPort so { Source Reg[0] ; }

  ScanRegister Reg[8:0] {
    ScanInSource si ;
  }

  Alias Reg1[2:0] = Reg[8:6] ;
  Alias Reg2[2:0] = Reg[5:3] ;
  Alias Reg3[2:0] = Reg[2:0] ;
}
```

This is an equivalent description of the IO behavior of the instrument. For example, both ICL modules M1 and M2 allow addressing three 3-bit registers named Reg1, Reg2, and Reg3, respectively from PDL.

ICL Namespaces

Note: ICL Namespaces are not currently supported; only PDL Namespaces are. This section is provided for completeness purposes only.

PDL Namespaces

Assume you own an ICL module named M. You have two suppliers, who instantiate their instruments in your ICL module. At the supplier's instrument top-level and downwards, there are no conflicts of either ICL or PDL objects. However, the PDL these suppliers provide for your module M may conflict. Both may provide a PDL named "init" bound to M. The IJTAG standard provides therefore a PDL namespace to further separate PDL for the same ICL module.

Your ICL module file:

```
Module M { … }
```

Your PDL file(s), defining the iProcs that may be modified by the respective supplier:

```
iProcTargetModule M -iProcNameSpace R
iProc init {} { … }

iProcTargetModule M -iProcNameSpace S
iProc init {} { … }
```

Your ICL file instantiating I1 of module M:

```
Module TOP {
  …
   Instance I1 of M
  …
}
```

You can now iCall both iProcs named 'init' of instance I1 of module M as follows

```
iCall I1.R::init
iCall I1.S::init
```

Skipping the Run-Test/Idle State

Run-Test/Idle (RTI) is a wait state between scan loads. The state machine allows skipping the RTI state when there are successive scan loads to run.

Run-Test/Idle is a state of the IEEE 1149.1 TAP finite state machine (FSM) as shown in Figure 7-2. The RTI state allows the TAP controller to wait on an action. A basic scan load has the following sequence of states: RTI, Select-DR, Capture-DR, Shift-DR, Exit1-DR, Update-DR, and RTI.

Figure 7-2. Partial IEEE 1149.1 TAP Controller State Diagram
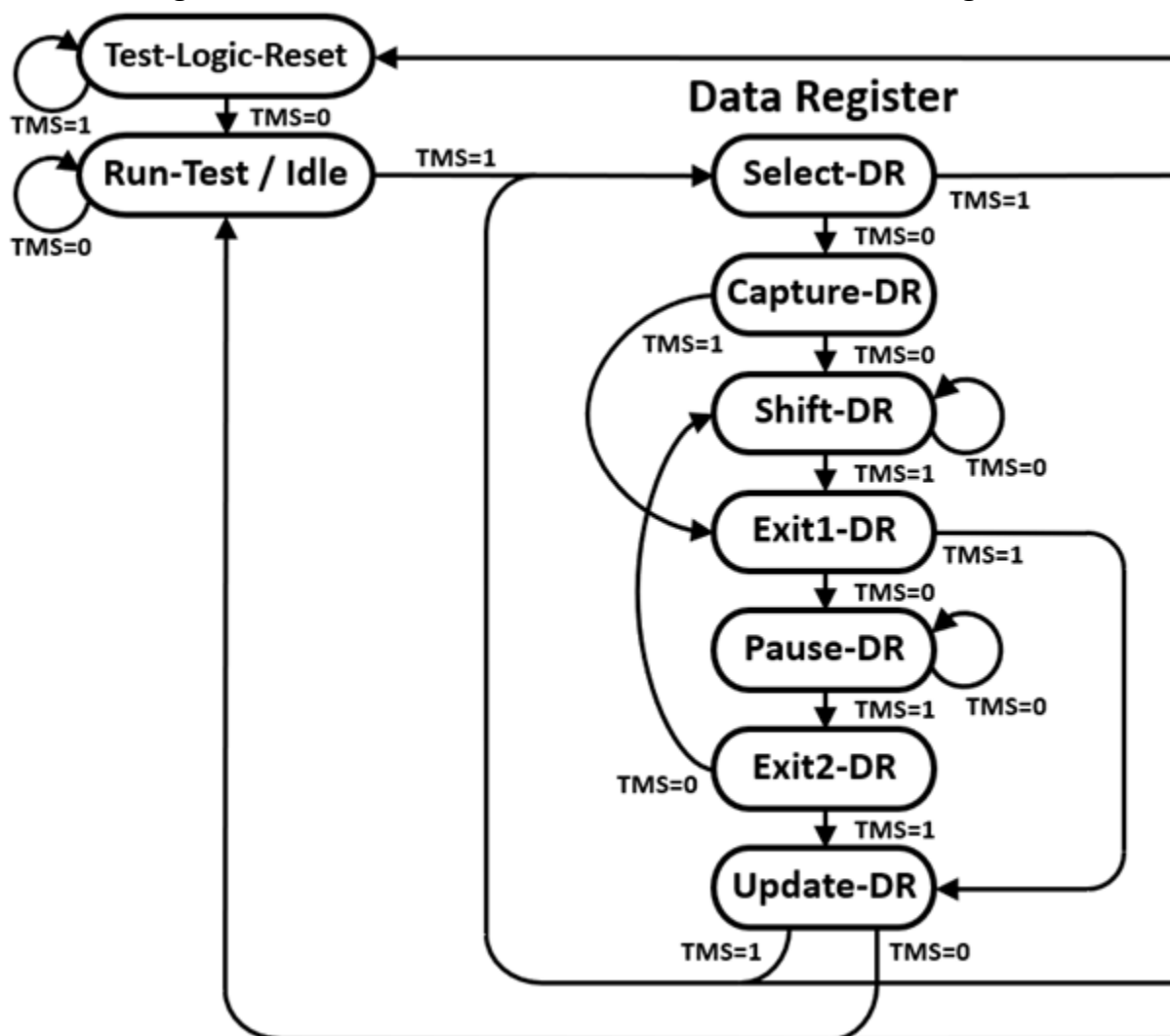
Because there is a direct transition from Update-DR to Select-DR with a TMS=1 transition, the RTI state may be skipped between two or more consecutive scan loads. A basic sequence of states for two consecutive scan loads is: RTI, Select-DR, Capture-DR, Shift-DR, Exit1-DR, Update-DR, Select-DR, Capture-DR, Shift-DR, Exit1-DR, Update-DR, and RTI. The second Select-DR begins the second scan load after skipping the RTI state. RTI can be skipped between any combination of DR and IR scan loads.

During IJTAG retargeting, the retargeting engine does not skip the RTI state if any of the following apply:

- In-system test is performed.

- Two-Pin Serial Port is present.

- An action follows the scan load that is neither an iNote nor a scan load, such as iRunLoop or iReset.

- A TAP FSM is taken out of the idle state and put onto the active scan path due to the scan load.

- A TAP FSM is put into the idle state and removed from the active scan path due to the scan load.

- The next scan load targets a different scan interface than the current scan load.

- It is the last scan load of an iCall in test_setup or test_end (even if there is a scan load following directly afterward in the next iCall).

- The target scan interface is a non-TAP client and TCK has an off-state of 1.

- The scan load changes the value of a ScanRegister bit with the tessent_force_rti_on_value_change attribute set to 1.

- When a Local Reset is triggered in the IJTAG network, the next scan load that shifts a value into a ScanRegister bit with the tessent_force_rti_on_value_change attribute set to 1 results in that scan load ending in the Run-Test/Idle state. This includes registers not affected by the Local Reset. See "How to Model Global Reset, Local Reset, and Embedded TAPs" on page 23 for details. Subsequent scan loads shifting the same value into the ScanRegister bits may skip the Run-Test/Idle state.

> **Note**
>
> The definition of the tessent_force_rti_on_value_change attribute changes when you combine it with the write_latency attribute for a ScanRegister. Refer to "How to Add Extra Delays After Writing to a ScanRegister" on page 179 for more information.

How to Define Default Values in ICL

This section describes how to define default values in ICL.

Consider the following part of an ICL module definition:

```
ScanRegister Reg_1[3:0] {
    ScanInSource      si ;
    ResetValue        4'b1001 ;
    DefaultLoadValue  4'b1001 ;
}

ScanRegister Reg_2[3:0] {
    ScanInSource      si ;
    DefaultLoadValue  4'b1111 ;
}
```

In the properties section of the scan register declaration, there are the "ResetValue" and "DefaultLoadValue" keywords. Both define a scan load value that the PDL retargeter must abide by. When an iReset is issued, the 4-bit scan register, Reg_1 in this example, assumes the

value 4'b1001 for its register bits. The reset signal does not need to be ICL-routed. It is implicitly assumed.

Better ICL coding style uses enumeration tables to abstract from data values. The scan register example above would resemble the following:

```
ScanRegister Reg_1[3:0] {
    ScanInSource      si ;
    ResetValue        resetvalue ;
    DefaultLoadValue  defaultvalue ;
    RefEnum           scanRegValTable ;
}

Enum scanRegValTable {
    resetvalue    4'b1001 ;
    defaultvalue  4'b1001 ;
    green         4'b1101 ;
    blue          4'b1110 ;
}
```

The string-value pairs defined by the enumeration table are only a shorthand. You can always use numbers for reading and writing as usual.

Assume the following user PDL for the ICL example above:

```
iWrite Reg_1 green
iApply

iWrite Reg_1 0b0011
iApply
```

Another interesting behavior of the PDL retargeter is due to the 'DefaultLoadValue' ICL keyword in the ICL example above. Assume the following user PDL for the ICL example:

```
iReset

iWrite Reg_1[0] 0b0
iApply

iWrite Reg_2[0] 0b0
iApply
```

The iWrite command specifies only scan register bit 0. However, the PDL retargeter must shift in something for the other bits. The default load value defines this.

This is the retargeted PDL:

```
iReset

iWrite Reg_1 0b1000
iApply

iWrite Reg_2 0b1110
iApply
```

If no default load values were defined and the left-most three bits were never written before, neither explicitly through an iWrite command nor implicitly through an iReset, the IJTAG default rule is then to write the value 0. This is particularly important for data registers that do not necessarily have a reset value defined.

Attributes of the ICL Extraction Flow

This section explains the usage of some attributes for IJTAG.

Overall, there is no difference in the use model compared to the rest of Tessent Shell. IJTAG only adds a few built-in attributes, some of which are shown in the following example that is derived from the ICL extraction flow.

In Tessent Shell, there are several ways to gain access to an attribute or attribute value. This example uses a reporting function to get a list of all attributes of an entity.

report_attributes

Here, it is important to use the correct introspection commands to get access to the ICL entities and not the (Verilog) design entities. Hence, to report all attributes for the top-level ICL module named "chip", use the following command:

report_attributes [get_icl_modules chip]

In case the ICL module "chip" was created through ICL extraction, there are several built-in attributes listed that are of interest. Below is a partial report:

```
ANALYSIS> report_attributes [ get_icl_modules chip ]
Attribute Definition Report

Name                                 Value                                         Inheritance
------------------------------------ --------------------------------------------- ----------
forced_high_input_port_list          {A[1]} {A[0]}                                 -
forced_high_internal_input_port_list {a_inst3/A[1]} {a_inst2/A[2]} {a_inst2/A[0]}  -
forced_low_input_port_list           {\B[0] } {\B[1] } {A[2]} {pmu_se}             -
forced_low_internal_input_port_list  {b_inst2/A}                                   -
icl_extraction_date                  Wed Aug 15 00:21:13 2012                      -
is_created                           true                                          -
```

This list of attributes shows that the module 'chip' was created through ICL extraction (is_created == true), and when this happened (icl_extraction_date). The next line shows how to get access to the value of an attribute. It returns a Tcl list.

> **get_attribute_value_list [get_icl_modules chip] -name is_created**

If you do not know the top-level name or if you want to have a more generic script, you can introspect the name as well, as follows. Please note again the use of the correct icl introspection commands and options. Otherwise, you get the design introspection versions.

Examples are as follows:

> **report_attributes [get_current_design -icl]**

> **get_attribute_value_list [get_current_design -icl] -name is_created**

Scan Chain Integrity Test in Tessent IJTAG

Tessent IJTAG provides an extension to IJTAG that enables the creation of scan chain integrity tests.

The current release of this functionality provides only the building blocks. Using these building blocks, a chain test for a provided scan register can easily be constructed. Currently no automation is provided in Tessent IJTAG to compute chain integrity tests for all ICL scan chains with a single command.

An ICL scan chain integrity test is defined in two steps: an iWrite to the register, followed by an iRead from the register.

_____ **Note** _____

Use the optional '-end_in_pause' switch of the iApply command to create IJTAG scan chain integrity tests.

In this example, you use the same chain test value that Tessent ATPG tool uses. You can also use a running 1 or running 0, which is useful to validate a register length and access, or any other value you determine is meaningful.

```
set chaintest [ string range [ string repeat "0011" \
:  [ expr $reg_length / 4 +1 ] ] 0 [expr $reg_length -1] ]
:
:iWrite MyTdr1.R 0b${chaintest}
iApply -end_in_pause

iRead  MyTdr1.R 0b${chaintest}
iApply
```

How to Define Auto-Return Values in ICL

This section presents an ICL feature that makes the PDL retargeter automatically keep a certain value in a register bit.

Writing to this bit is still performed as expected, however after the bit changes value due to a write operation, the PDL retargeter returns the bit to this specified value at the next opportunity. It might require that the PDL retargeter issue another scan load operation to fulfill this requirement.

Assume you have an application where the enable bits in a TDR must be kept in their off state at all times, except for the short moments when they are needed. You could require that the author of the module's PDL remember to turn off these bits, but this would be cumbersome and error-prone. IJTAG provides an automated mechanism in the form of the "iApplyEndState" in ICL.

```
Module M {

    ScanInPort sin;
    ScanOutPort sout { Source TDR_2[0] ; }
    ...

    ScanRegister TDR_1[8:0] {
        ScanInSource sin;
        ResetValue 9'b0;
    }

    Alias myDataWriteEnable = TDR_1[8] {
        iApplyEndState 1'b0;
    }

    ScanRegister TDR_2[8:0] {
        ScanInSource TDR1[0];
        ResetValue 9'b0;
    }

}
```

In this ICL module example, the scan register bit TDR_1[8] must be kept at 0. Writing to the scan register as follows changes this bit:

User PDL file:

```
iWrite TDR_1 0b111001100
iApply
```

It is up to the PDL retargeter to first run per your intention, and then return bit TDR_1[8] to the 0 value, (as specified in the iApplyEndState) at the earliest possible opportunity. This opportunity is usually given with the next iApply statement.

To continue this example, assume that you read from TDR_2 after the above iWrite to TDR_1. In this example the PDL retargeter computes the following PDL:

Retargeted PDL:

```
iWrite TDR_1 0b111001100
iApply

iWrite TDR_1 0b011001100
iRead  TDR_2 0x1ff
iApply
```

A second possible opportunity for the tool to restore the iApplyEndState value, if there is no subsequent iApply command, is through options to the close_pattern_set command. If you use either the option "close_pattern_set -network_end_state initial" or "close_pattern_set -network_end_state reset", the tool has the opportunity through one or several of the automatically computed iApply blocks statements to not only bring the ICL network into the requested state but also put the iApplyEndState value back in place.

The ICL code example in Figure 7-4 shown in the "How to Model Addressable Registers in ICL" section is extended to demonstrate a practical example.

How to Model Addressable Registers in ICL

This section describes some of the ways you can model addressable registers in ICL.

The ICL code example in Figure 7-4 demonstrates using the iApplyEndState ICL feature. Notice that the bit TDR1[63] in the ICL code in Figure 7-4 encodes the 'read enable' instruction. This bit has to assume the value 1 when values from data registers should be read (Enum ReadWriteCmd). However, it must be kept 0 at all other times. The iApplyEndState ICL feature ensures that the PDL retargeter automatically 'turns off' the read enable bit, once it is no longer needed.

ICL enables modeling a register addressing scheme controlled from IJTAG ScanRegisters. The address, data, read enable, and write enable values are automatically calculated by the PDL retargeter.

ICL supports the direct modeling of addressable registers using the AddressPort, ReadEnPort, and WriteEnPort port functions, and the AddressValue property within the DataRegister and Instance construct. Many standard addressing schemes are correctly modeled with this syntax. More complex addressing schemes must be modeled explicitly with a DataMux construct on the read path and a DataRegister with WriteDataSource and WriteEnSouce properties on the write path.

Figure 7-3 is a schematic view of an example with an indirect addressing scheme. To read the DataOut port of an instrument, the bank register must first be written to select the proper

instrument within the bank. Then, a read transaction is performed on the given bank followed by a last scan load to capture the result of the read. The read path is modeled with cascaded DataMux where the first level is selected by the bank register and the second level is selected by the ReadEnable signal and the bank address.

When doing the final scan load to capture the read value, the solver would normally scan in the same values into the TDR, as it did during the second scan load. However, if this is done, the third scan load would initiate a second read transaction, which is not desired. The PDL retargeter is told to leave the ReadEnable signal to its off value on the last scan load using the iApplyEndState property within the Alias construct, as shown in Figure 7-4.

Figure 7-3. Schematic View of an Indirect Addressing Scheme

- Writing is still done with a single Scan Load
  — But part of data field contains the sub address field
- Reading requires up to three scan loads
  — First scan load to write sub address
    – If not already set to the right value
  — Second to perform the read transaction
  — Third to capture the read data
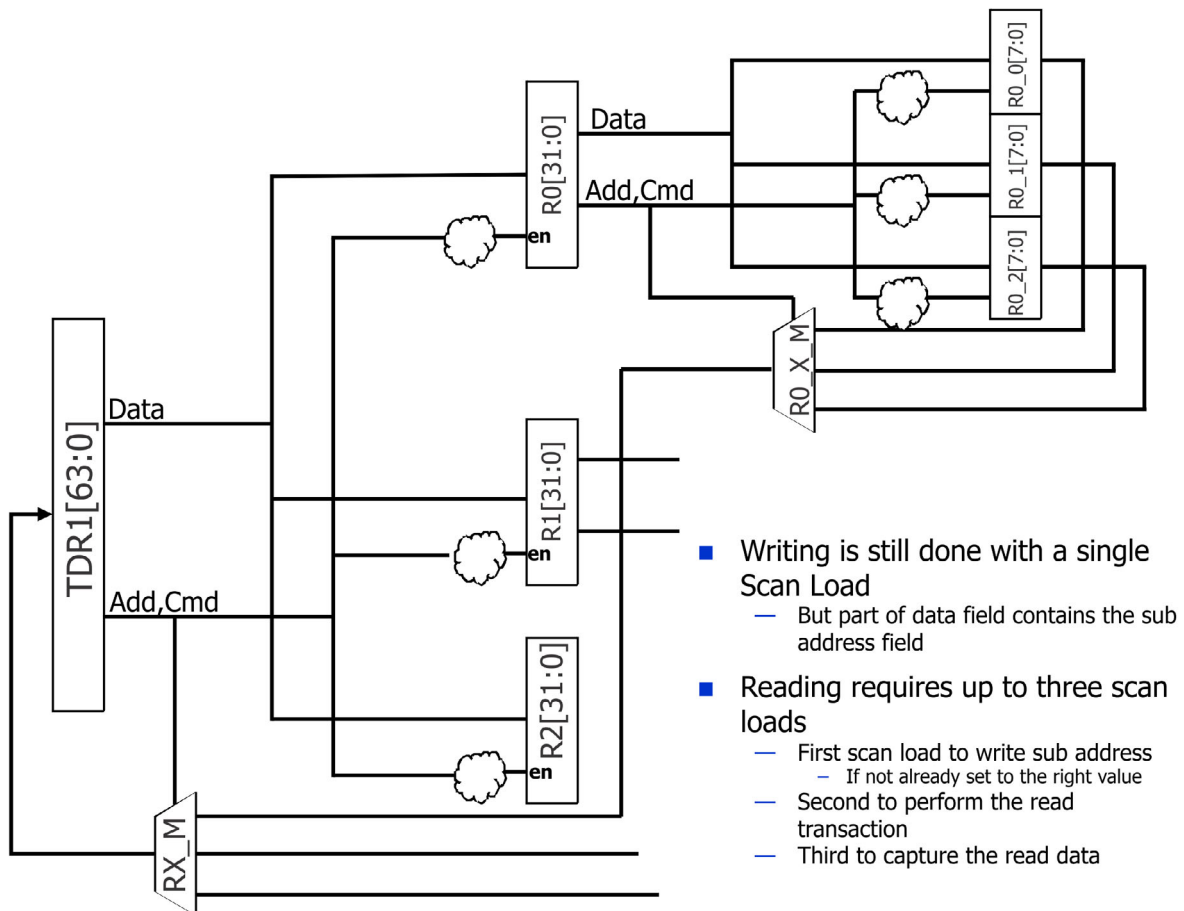
Figure 7-4. ICL Description of an Indirect Addressing Scheme

```
Module block1 {
  ScanInPort si1;
  ScanOutPort so1 { Source TDR1[0]; }
  SelectPort en1;
  ShiftEnPort se;
  CaptureEnPort ce;
  UpdateEnPort ue;
  TCKPort tck;

  ScanRegister TDR1[63:0] {
    ResetValue 64'b0;
    ScanInSource si1;
    CaptureSource 32'b0,RX_M;
  }

  Alias RE = TDR1[63] { iApplyEndState 1'b0; }
  Alias cmd[1:0] = TDR1[63:62] { RefEnum ReadWriteCmd; }
  Alias R0_cmd[1:0] = R0[31:30] { RefEnum ReadWriteCmd; }
  Alias R1_cmd[1:0] = R0[31:30] { RefEnum ReadWriteCmd; }
  Alias R2_cmd[1:0] = R0[31:30] { RefEnum ReadWriteCmd; }

  DataRegister R0[31:0] { WriteEnSource R0_W; WriteDataSource TDR1[31:0];}
  DataRegister R1[31:0] { WriteEnSource R1_W; WriteDataSource TDR1[31:0];}
  DataRegister R2[31:0] { WriteEnSource R2_W; WriteDataSource TDR1[31:0];}

  DataRegister R0_0[7:0] { WriteEnSource R0_0_W; WriteDataSource R0[7:0];}
  DataRegister R0_1[7:0] { WriteEnSource R0_1_W; WriteDataSource R0[7:0];}
  DataRegister R0_2[7:0] { WriteEnSource R0_2_W; WriteDataSource R0[7:0];}
  DataRegister R1_0[7:0] { WriteEnSource R1_0_W; WriteDataSource R1[7:0];}
  DataRegister R1_1[7:0] { WriteEnSource R1_1_W; WriteDataSource R1[7:0];}
  DataRegister R1_2[7:0] { WriteEnSource R1_2_W; WriteDataSource R1[7:0];}
  DataRegister R2_0[7:0] { WriteEnSource R2_0_W; WriteDataSource R2[7:0];}
  DataRegister R2_1[7:0] { WriteEnSource R2_1_W; WriteDataSource R2[7:0];}
  DataRegister R2_2[7:0] { WriteEnSource R2_2_W; WriteDataSource R2[7:0];}

  LogicSignal R0_W { cmd,TDR1[61:32] == write,30'd0; }
  LogicSignal R1_W { cmd,TDR1[61:32] == write,30'd1; }
  LogicSignal R2_W { cmd,TDR1[61:32] == write,30'd2; }
  DataMux RX_M[31:0] SelectedBy cmd,TDR1[61:32] {
    2'b10, 30'd0 : 24'b0,R0_X_M[7:0];
    2'b10, 30'd1 : 24'b0,R1_X_M[7:0];
    2'b10, 30'd2 : 24'b0,R2_X_M[7:0]; }

  LogicSignal R0_0_W { R0_cmd,R0[29:24] == write, 6'd0; }
  LogicSignal R0_1_W { R0_cmd,R0[29:24] == write, 6'd1; }
  LogicSignal R0_2_W { R0_cmd,R0[29:24] == write, 6'd2; }
  DataMux R0_X_M[7:0] SelectedBy R0_cmd,R0[29:24] {
    2'b10, 6'd0 : R0_0[7:0];
    2'b10, 6'd1 : R0_1[7:0];
    2'b10, 6'd2 : R0_2[7:0]; }

  LogicSignal R1_0_W { R1_cmd,R1[29:24] == write, 6'd0; }
  LogicSignal R1_1_W { R1_cmd,R1[29:24] == write, 6'd1; }
  LogicSignal R1_2_W { R1_cmd,R1[29:24] == write, 6'd2; }
  DataMux R1_X_M[7:0] SelectedBy R1_cmd,R0[29:24] {
```

```
      2'b10, 6'd0 : R1_0[7:0];
      2'b10, 6'd1 : R1_1[7:0];
      2'b10, 6'd2 : R1_2[7:0]; }

   LogicSignal R2_0_W { R2_cmd,R2[29:24] == write, 6'd0; }
   LogicSignal R2_1_W { R2_cmd,R2[29:24] == write, 6'd1; }
   LogicSignal R2_2_W { R2_cmd,R2[29:24] == write, 6'd2; }
   DataMux R2_X_M[7:0] SelectedBy R2_cmd,R0[29:24] {
      2'b10, 6'd0 : R2_0[7:0];
      2'b10, 6'd1 : R2_1[7:0];
      2'b10, 6'd2 : R2_2[7:0]; }

   Enum ReadWriteCmd { read = 2'b10; write = 2'b01; }
}
```

How to Model a ScanMux Selection Preference

During IJTAG pattern retargeting, the IJTAG engine always attempts to utilize a scan path with the optimal length when performing the operations of an iApply.

In many cases, this automatically implies the assignment of the select signal of the ScanMuxes along the scan path. Nonetheless, there might be multiple ScanMux selections that have the same overall scan path length. Even though these selections seem equivalent to the IJTAG engine, they could have special meanings in the design. Therefore, the IJTAG engine chooses a selection in a deterministic manner, using the following rules:

1. If selections have the same direct source, the IJTAG engine prefers them in the order specified in the ICL file.

2. If selections have the same scan path length to a common source, the IJTAG engine prefers them in the order specified in the ICL file. The computation of the scan path length is static and performed during ICL elaboration. Therefore, it cannot take dynamic changes in scan path length (for example, due to ScanMuxes) into account. As a result, the scan path length computation only considers direct paths to a common source without any scan path configuration elements. For more details, refer to Computation of Scan Path Length to a Common Source.

3. The top selection of a ScanMux supersedes all other selection rules. In other words, the first mux_select_value line in the list of selections for the mux_select_signals of a ScanMux supersedes all other selection rules. When set to on, the -prefer_top_mux_selection switch of the set_ijtag_retargeting_options command directs the IJTAG engine to always use top selection. By default, the top selection rule is off.

Computation of Scan Path Length to a Common Source

The tool uses a tracing procedure to determine the length of the scan path to a common source point. This scan path analysis is static and occurs during the ICL elaboration. Therefore, the tool cannot take the current network configuration into account. As a result, it cannot trace through complex ScanMuxes.

The exception to this rule is simple ScanMuxes of segment insertion bits (SIBs). When the tool encounters a SIB during the trace, the algorithm assumes that the SIB is not active and the trace continues at the input of the SIB.

Examples

Example 1. Preferred Solution in a TAP DR-MUX:

```
ScanMux DRMux SelectedBy instruction[2:0] {
      3'b000 : bypass; // BYPASS instruction
      3'b100 : bypass; // HIGHZ  instruction
      3'bx01 : fromTdr1;
   }
```

In this example, the IJTAG engine prefers the instruction "000" to "100", even though they both result in the same selection of the ScanMux because "000" is specified before "100". This ensures that the IJTAG engine does not accidentally activate the "HIGHZ" mode of the TAP. The preference only becomes relevant if the IJTAG retargeting engine has no implicit or explicit requirements for the value of the instruction.

Example 2. Preferred Solution With Different ScanRegisters:

```
ScanRegister r1[1:0] { ScanInSource tdi; }
ScanRegister r2[1:0] { ScanInSource tdi; }
ScanRegister r3[1:0] { ScanInSource tdi; }
ScanRegister r4[2:0] { ScanInSource tdi; }
ScanMux m1 SelectedBy select[1:0] {
                2'b10: r4[0];
                2'b11: r3[0];
                2'b00: r2[0];
                2'b01: r1[0];
}
```

In this example, the selection of "11" is preferred to "00" and "01". The "10" selection is not preferred because it would result in a longer overall scan path, as the r4 ScanRegister has a length of 3, whereas the other possible selections all have a length of 2.

Example 3. Preferred Solution for Scan Paths to a Common Source:

Figure 7-5. Tracing Scan Paths to a Common Source
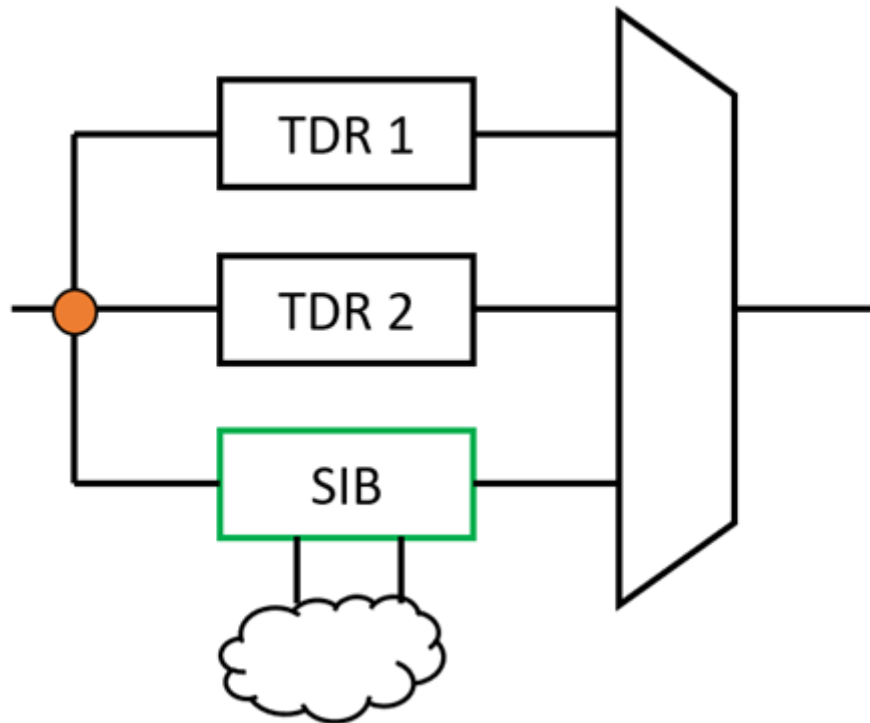
In this example, the ScanMux on the right has three different possible selections including a SIB drawn in green. The tool traces all three to the same common source and prioritizes selections based on the order defined in the ICL file.

Example 4. Preferred Solution when a ScanMux Drives Another ScanMux:

Figure 7-6. ScanMux Driving Another ScanMux
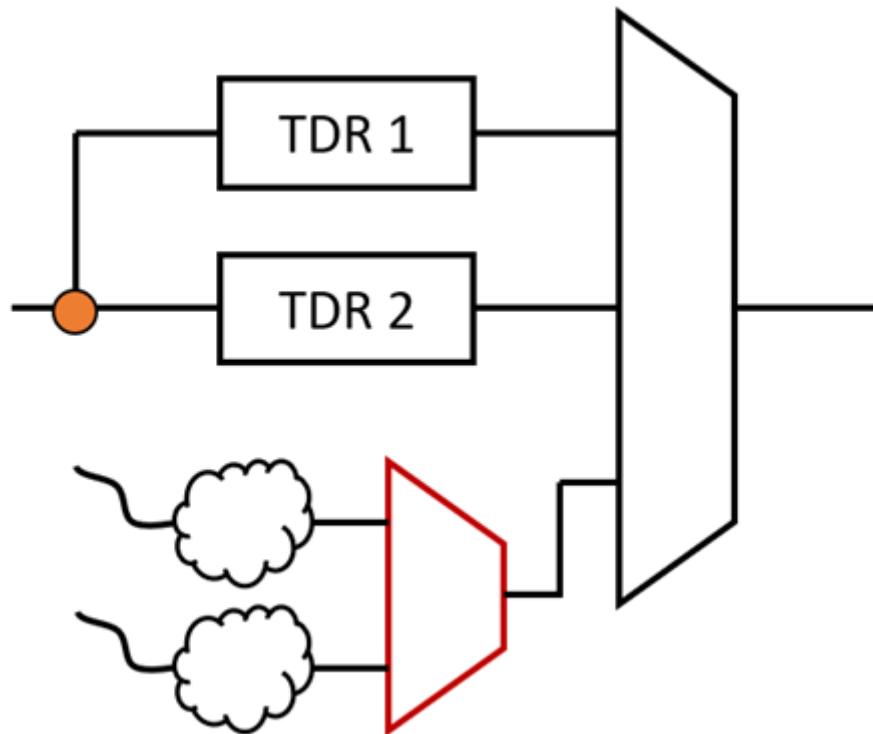
In this example, the ScanMux on the right has three different possible selections. The third selection is driven by another ScanMux that is not within a SIB. The tool cannot statically compute the scan path length for the red ScanMux, and as a result, there is no prioritization for the third selection.

How to Add Extra Delays After Writing to a ScanRegister

If you define an ICL ScanRegister that has the write_latency attribute, any scan load that shifts through that ScanRegister (and triggers the update functionality) is automatically extended with the specified number of IDLE TCK cycles in the Run-Test/Idle (RTI) state. This occurs before the next IJTAG operation starts.

You can also define the tessent_force_rti_on_value_change attribute on a ScanRegister object. This attribute instructs the Tessent Shell tool to force any scan load that changes the specified scan register bits to automatically end in the RTI state.

If you define both attributes on a ScanRegister, the write_latency attribute extends the scan load timing only if the scan load changes the register value. In this case, shifting a value into the ScanRegister that matches the current value does not add extra IDLE TCK cycles.

When used together, the write_latency attribute takes precedence over the tessent_force_rti_on_value_change attribute. The tool adds the extra cycles when any register bit changes its value, not just the bits for which the value of the tessent_force_rti_on_value_change attribute is 1. Therefore, setting a single bit to 1 for the value of the tessent_force_rti_on_value_change attribute is sufficient to trigger this behavior. Also, setting the mask to 1 for every bit of the ScanRegister is not required when both attributes are specified.

_____ **Note** _____

The width of the tessent_force_rti_on_value_change attribute must match the width of the ScanRegister; otherwise, the tool generates an ICL52 DRC violation during ICL file parsing.

The following table summarizes the behavior of the attributes and their combination.

Table 7-1. Behavior of the tessent_force_rti_on_value_change and write_latency Attributes

Specified ScanRegister Attribute	Behavior	Example

tessent_force_rti_on_value _change	Any scan load that changes the value of the specified ScanRegister bits ends in the RTI state.	```ScanRegister reg[2:0] {```
		 ``` Attribute \```
		 ```tessent_force_rti_on_value_change \```
		 ``` = 3'b110;```
		 ``` ScanInSource si;```
		 ```}```
		
		 Any scan load that modifies the value of reg[2] or reg[1] ends in the RTI state.

write_latency	Any scan load pulsing the update signal on the ScanRegister gets *<n>* IDLE TCK cycles after the scan load ends.	```ScanRegister reg[2:0] {```
		 ``` Attribute write_latency = "tck 5";```
		 ``` ScanInSource si;```
		 ```}```
		
		 Applies five IDLE TCK cycles after the scan load ends.

tessent_force_rti_on_value _change
 	
 and
 	
 write_latency
	Any scan load that changes the value of any bit of the ScanRegister gets *<n>* IDLE TCK cycles after the scan load ends.	```ScanRegister reg[2:0] {```
		 ``` Attribute \```
		 ```tessent_force_rti_on_value_change \```
		 ``` = 3'b110;```
		 ``` Attribute write_latency = "tck 5";```
		 ``` ScanInSource si;```
		 ```}```
		
		 Note: In this example, setting the value of the tessent_force_rti_on_value_change attribute to 3'b001 would have the same effect as the specified value.

If you specify multiple ScanRegisters with the write_latency attribute on the active scan path, the tool adds the maximum number of IDLE TCK cycles defined in any of the attributes. For example, consider a scan path with two ScanRegisters on it, where the first register has the write_latency attribute defined as "tck 7", and the second register has the attribute defined as "tck 3". A scan load shifting through these registers adds seven IDLE TCK cycles after the end of the scan load (maximum of 3 and 7).

Accessing the Attributes

The write_latency and tessent_force_rti_on_value_change attributes for the icl_scan_register_in_module object type are writable only while in setup mode. The tool reports an error if you try to set the value of these attributes (using the set_attribute_value command) while in analysis mode or during system mode transition.

These attributes are read-only for the icl_scan_register object type.

Examples

The following example is an ICL definition of a module with ScanRegisters that have at least one of the write_latency and tessent_force_rti_on_value_change attributes defined.

```
Module SimpleModule {
  ScanInPort si1;
  ScanInPort si2;
  ScanOutPort so1 { Source reg3[0]; }
  ScanOutPort so2 { Source reg1[0]; }
  SelectPort sel1;
  SelectPort sel2;
  ShiftEnPort se;
  CaptureEnPort ce;
  UpdateEnPort ue1;
  UpdateEnPort ue2;
  TCKPort tck;
  ResetPort rst { ActivePolarity 0; }
  ScanInterface intf1 { Port si1; Port so1; Port sel1; Port ue1; }
  ScanInterface intf2 { Port si2; Port so2; Port sel2; Port ue2; }

  ScanRegister reg1[4:0] {
    ScanInSource si2;
    ResetValue 5'b00000;
    Attribute write_latency = "tck 5";
  }

  ScanRegister reg2[3:0] {
    ScanInSource si1;
    ResetValue 4'b0000;
    Attribute tessent_force_rti_on_value_change = 4'b0101;
  }

  ScanRegister reg3[5:0] {
    ScanInSource reg2[0];
    ResetValue 6'b000000;
    Attribute write_latency = "tck 3";
    Attribute tessent_force_rti_on_value_change = 6'b000001;
  }
}
```

This module has two ScanInterfaces with two separate SelectPorts. This is necessary to demonstrate the behavior of the different attribute combinations. If all ScanRegisters are accessible through the same ScanInterface, they are driven by the same UpdateEnPort. Therefore, any scan load would have five IDLE TCK cycles of wait time appended, because ScanRegister reg1 has only the write_latency attribute defined.

The following pattern set performs writes on the ScanRegisters. The comments describe whether the tool adds extra IDLE TCK cycles after each scan load.

```
open_pattern_set test
  # Scan loads for this iApply have 5 TCK cycles added after the
  # update event because reg1 has the write_latency attribute defined.
  iWrite reg1[4:2] 0b111
  iApply

  # Scan loads for this iApply have 5 TCK cycles added after the
  # update event because reg1 has the write_latency attribute defined.
  # This happens even if reg1 does not change its value.
  iWrite reg1[4:2] 0b111
  iApply

  # Scan loads for this iApply end in the RTI state (= 1 extra TCK
  # cycle added after the update event) because reg2 has the
  # tessent_force_rti_on_value_change attribute specified.
  iWrite reg2[0] 1
  iApply

  # Scan loads for this iApply do not end in the RTI state because reg2
  # does not change its value.
  iWrite reg2[0] 1
  iApply

  # Scan loads for this iApply have 3 TCK cycles added after the
  # update event because reg3 changes its value.
  iWrite reg3[2] 1
  iApply

  # Scan loads for this iApply do not have any extra TCK cycles added
  # after the update event because reg3 does not change its value.
  iWrite reg3[2] 1
  iApply
close_pattern_set
```

Chapter 8
Verification and Debug of IJTAG Instruments and Networks

IEEE 1687-2014 (IJTAG) enables you to describe various instruments and network components through ICL files; these ICL files are read and processed by Tessent IJTAG. A higher-level ICL file representing the current design can then be exported (ICL extraction) and lower-level PDL test procedures can be regenerated (PDL retargeting) to a higher-level module.

Inserting new IJTAG instruments in a design and connecting them together modifies the overall access network. For instance, an instrument may be connected to an IEEE 1500 Wrapper Test Access Port (WTAP) and this WTAP may in turn be interfaced to an IEEE 1149.1 TAP. To access the instrument, one thus has to go through the TAP and WTAP to reach it. Depending on how the connections to the TAP and WTAP are made, accessing the target instrument may require implementation-specific instruction opcodes and loading data registers with appropriate data.

Assuming a design has a syntactically-valid ICL description, how do you validate its contents? Do all described test data registers (TDRs) have the expected length and are connected exactly as indicated?

An obvious method is to take an existing instrument-level test, retarget it to the current top-level and then simply simulate it – exactly like a functional test. However, the coverage of such a functional test is usually relatively low, and knowing exactly what gets tested is not obvious to determine. Additionally, when simulations fail, it is increasingly difficult to figure out why.

This chapter provides guidelines and pointers to verify and debug such IJTAG networks and instruments.

Tip
When using the Two-Pin Serial Port for IJTAG, refer to the topic "How to Avoid Simulation Issues When Using the Two-Pin Serial Port Controller" in the *Tessent Shell User's Manual* .

General Guidelines for Debugging Simulation Results

This section describes guidelines to help you debug simulation results.

- Create ICL verification testbenches

- Display the comparison failure counter in waveforms

Creating ICL Verification Patterns

Tessent IJTAG provides an automated method to structurally validate an ICL file: verification patterns are generated based on the current ICL model of the design. Those patterns can be exported as Verilog testbenches and then be simulated against the actual HDL view. This ensures the ICL-based Tessent IJTAG view matches the actual test access infrastructure of the current design.

Once this test access infrastructure has been validated (either via simulation or by testing an actual silicon device), one can confidently assume the target instrument is accessible; settings can be correctly applied to the instrument and its status can be monitored. If functional tests are subsequently run and fail, the debug should therefore focus on the actual instrument behavior – not on whether it is correctly accessed through the IJTAG network.

To generate those ICL verification patterns, use the create_icl_verification_patterns command. For example:

```
[...]
extract_icl
set_system_mode analysis
open_pattern_set pset
create_icl_verification_patterns
close_pattern_set
report_pattern_set
write_patterns patterns/check_network.v -verilog -replace[...]
```

In the above dofile excerpt, a pattern set named "pset" is opened. The verification patterns are automatically generated by Tessent IJTAG, based on the current design's extracted ICL. Those patterns are saved as Verilog testbenches (TBs) or ATE patterns for simulation with digital EDA simulators (or for testing an actual silicon device on an ATE).

You can simulate verification testbenches generated by the create_icl_verification_patterns command with Siemens EDA Questa™ SIM or other standard EDA simulators.

___ **Tip** _____

🛈 When using the Two-Pin Serial Port for IJTAG, refer to the topic "How to Avoid Simulation Issues When Using the Two-Pin Serial Port Controller" in the *Tessent Shell User's Manual* .

_____

Using ICL Verification Patterns

The test patterns generated from the create_icl_verification_patterns command are divided into two categories.

- **Scan register integrity test** — Verifies that the scan-in to scan-out path of every scan register works correctly. It also checks whether every scan register can be accessed and has the correct length. For every scan multiplexer in place, each input must be exercised at least once.

- **Data input & output certification** — Ensures the parallel IOs of an ICL module capture and drive the intended values, using simulation-based "force" and "observe" commands. Shifted-in values should be updated on the parallel output and captured parallel inputs should be shifted-out appropriately.

Scan Register Integrity Test

The scan register integrity test is performed after the tools first create a scan path table. This table is constructed by tracing backward from each scan output described in the current ICL module. Whenever a scan mux is reached, a path that was not used before is selected. When reaching a scan input, tracing stops and the traced scan path is stored in the scan path table. This tracing process is then repeatedly performed, every iteration choosing the next path from the last reached scan mux. Once all scan paths have been traced, the actual pattern generation process is initiated.

The pattern generation comprises three distinct steps:

1. For every possible scan path, issue iWrite commands to set the intended scan mux select conditions, then issue iWrite and iRead commands to shift the test pattern in or out to or from the scan registers. If successful, the tested scan mux inputs and the related scan registers are flagged as tested.

   The activation of the scan path could fail due to mutually exclusive scan mux conditions. In such a case, the related scan muxes are being flagged as "to be processed" in step 2.

2. For each scan mux input that was not tested in step 1, the scan mux condition is activated; the pattern for the scan register connected to the scan mux at the input or output side then gets exercised. If successful, the related scan mux input and scan register is flagged as tested.

   In this step, only one scan mux is activated and the solver has the freedom to activate everything else that may be required to scan in or out the related scan register.

   ScanMux inputs can be excluded from the test by using the attribute exclude_codes_from_icl_verify. Use this attribute in the ScanMux declaration and

specify the values that must not be applied to the select inputs of the multiplexer. If there are multiple values that must be excluded, enclose each value in braces, as follows:

```
ScanMux mux SelectedBy tdr[2:0] {
    Attribute exclude_codes_from_icl_verify = "{3'b100} {3'b110}";
    3'b000: registerA[0];
    3'b100: registerB[0];
    3'b101: registerC[0];
    3'b110: registerD[0];
}
```

The specification of the attribute shown in the previous example instructs the verification pattern generator not to configure the mux in such a way that the registerB or registerD are used as input selectors.

3. For each scan register not flagged as tested, iWrite and iRead commands apply the pattern directly and expect the solver to find a solution that activates all necessary conditions to reach this scan register.

The last two steps above are skipped if everything is tested during step 1.

A scan register could be part of more than one scan path. In that case, it is tested multiple times.

No patterns are created or generated for paths not containing any scan register (also known as transparent paths) or only made of scan registers that were explicitly excluded.

An error is generated if scan registers are left untested. This error indicates the scan registers could not be activated:

```
//  Error: Failed to access following scan registers:
//          i1.r1[10:0]
//          i1.r2[10:0]
```

The pattern used to test the scan registers and their connectivity is the following:

```
1011001111000011111111100000000…
```

The above pattern ensures the concatenated scan register length is as expected.

Example

Assume an ICL file describes the following design consisting of four scan registers (r1-r4) and scan muxes (s1-s3):

Figure 8-1. Example Design
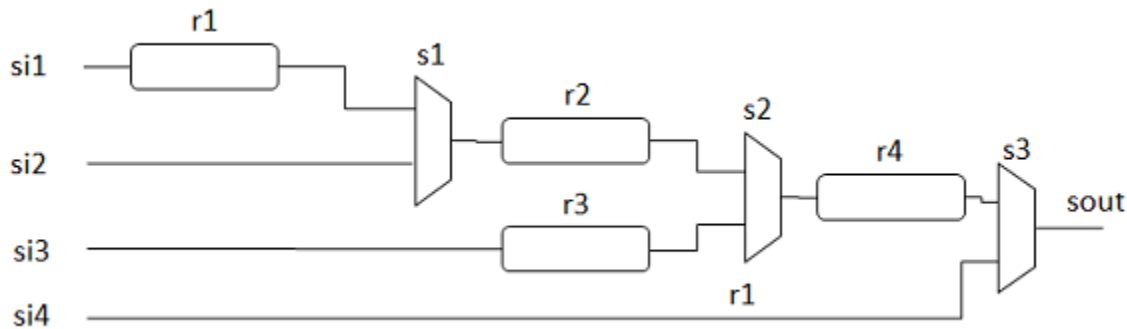
This example results in four scan path configurations being added to the table in the following order:

Table 8-1. Scan Path Configurations

s1	s2	s3	r1	r2	r3	r4
0	0	0	T	T	-	T
1	0	0	-	T	-	T
1	1	0	-	-	T	T
1	1	1	-	-	-	-

A 'T' means the scan register is tested in this scan path configuration. Tracing starts at a scan out and keeps tracing backward until a scanin is encountered.

The algorithm takes into account a list of instances to exclude from the test. This list is generated from the -instances or -modules and -exclude_instances or -exclude_modules switches optionally specified by you. This list is empty by default, that is, all instances are considered for test unless specified otherwise.

Data Input and Output Certification

The tests applied in the previous section validate serial scan paths. However, for a complete test, parallel IO capture and update should also be verified. Parallel data captured at inputs can be shifted out for examination while known shifted-in data can be applied (updated) at outputs.

For such tests to be performed on a given ICL module instance, its tessent_design_instance attribute must be defined - because the tools then know the related design instance and can address these pins directly in the design (via Verilog simulation).

The verification process forces and measures internal DataIn and DataOut pin in conjunction with iWrite and iRead commands. Within a given design, hierarchical parallel DataIn pins are forced to a specific value before being captured and shifted out with an iRead. Similarly, iWrite commands are applied to shift in a known value and then update it to DataOut pins where this value can be compared.

The following steps are performed to certify DataIn pins on every instance with the tessent_design_instance attribute:

1. iWrite a "0" to DataIn pins.

2. iApply.

3. Create measure/compare actions and add them to the solution.

4. Repeat the above with "1".

The following steps are performed to certify the DataOut pins on every instance with the tessent_design_instance attribute:

1. Create force actions to force a "0" to all DataOut pins.

2. iRead the "0" from the DataOut pins.

3. iApply.

4. Release the force actions.

5. Repeat the above with "1".

_____ **Note** _____

ICL Extraction sets the tessent_design_instance attribute to the pre-synthesis instance names whenever those names are available. During the creation of the data pin verification patterns, the actual design instance names are ignored. The verification pattern generator only uses the tessent_design_instance attribute to obtain the references to the design pins. Consequently, the RTL design files provided to ICL extraction must be used during the simulation of the verification patterns, at least in the case where synthesis modifies the design hierarchy (for example, because of "generate" loops). To obtain data pin verification patterns that can be simulated using the synthesized netlist with instance names that have been modified by synthesis, you either have to modify the tessent_design_instance attributes accordingly or run ICL Extraction based on the synthesized netlist from the beginning (without providing rtl files to the tool).

ICL Verification Patterns Summary

The create_icl_verification_patterns command automatically generates patterns to validate the IJTAG network & instrument connectivity of the current ICL module. Those test patterns ensure that the instrument can be properly accessed from the outside world.

If simulations (or even ATE tests) indicate mismatches during functional tests but not during ICL network and instrument verification, then it means that debug should focus on the targeted instrument itself, not on the test access network. This simplifies the overall debug process, as it narrows the failure's scope and enables a quicker issue resolution.

Displaying the Comparison Failure Counter

When simulation mismatches are reported, the design may be debugged by analyzing waveforms. This is typically done by displaying values over time for ports and signals of interest. However, debugging complete simulation waveforms could be overwhelming if you do not know where to check.

To help debug those mismatches, the verification testbench contains an internal variable named _compare_fail_count. This variable is expected to be 0 initially and increments whenever a comparison mismatch is observed.

It may be useful to display this variable when examining simulation waveforms. Because it only increments when a miscompare is recorded, first focus around the simulation time at which it becomes equal to 1. Before that time, everything is likely to be good, and any additional failure afterward may have the same cause.

Once you zoom in on that very first failure, examine the few preceding clock cycles and investigate what went wrong. Keep in mind that a failing comparison occurring on a serial scan-out port (such as TDO) is often caused by an erroneous captured value. Although the error may only be reported once that specific bit is shifted out, you need observe the time the bit was captured to diagnose further.

Conclusion

Using an ICL description at a given design level (for example: for an entire chip), automatically-generated test patterns can be applied to ensure the integrity of the test access network. This enables debug to focus on the target instrument itself, rather than having to figure whether the instrument is properly accessed or not. The command to generate such patterns is create_icl_verification_patterns.

Debugging simulation waveforms is a tedious process; fortunately one can zoom in closer to the failure location by examining the testbench internal variable named _compare_fail_count. This variable initially starts at 0 and increments +1 whenever mismatches are recorded. Signal waveforms can then be analyzed near the point in time where the variable increment first occurred.

Chapter 9
IJTAG Network Performance

The IJTAG network requires the distribution of control signals (capture enable, shift enable, and update enable) over the entire design. Typically, these signals are sourced by a single IEEE 1149.1 Test Access Port (TAP) controller. The signals are launched and captured on opposite edges of the network clock, TCK. The network is guaranteed to work even if no special care is taken to propagate TCK and the network control signals. However, the maximum frequency of operation might be too low for large designs or applications requiring a high network throughput (for example, memory BIST diagnosis). This chapter describes a method for optimizing the performance of the IJTAG network in those cases. The method takes into account that most designs are designed hierarchically.

Note that the maximum frequency of network operation is also determined by the presence and configuration of memory BIST testing, which is usually performed at the fastest functional rate applicable to each memory BIST controller under test. To ensure correct timing and operation of IJTAG accesses to the internal setup chain of the memory BIST controller, memory BIST controllers implemented prior to v2020.4 require a minimum functional clock to TCK frequency ratio of 4. Controllers implemented with versions v2020.4 and later incorporate Enhanced MBIST Access (EMA), where TCK is sourced to the memory BIST controller during the shift operation of the internal chains, rather than the functional clock, eliminating the need for the 4:1 frequency ratio.

A condition that requires a minimum frequency ratio of 2:1 of the *slowest* memory BIST controller functional clock to TCK, is the case of an IJTAG access to the internal setup chain of a memory BIST controller with the following properties:

- EMA is present in the controller

- The DftSpecification AdvancedOptions/use_multicycle_paths property is enabled by either:

  o DefaultsSpecification ControllerOptions/use_multicycle_paths set to "on" OR

  o DefaultsSpecification ControllerOptions/use_multicycle_paths set to "auto" AND DefaultsSpecification ControllerOptions/use_multicycle_paths_period_threshold set to "none" or to a period value larger than the period of the slowest functional clock at the memory BIST controller.

_____ **Note** _____

It is your responsibility to ensure you meet this requirement during synthesis and layout. There is no rulecheck to ensure this requirement is satisfied when you run process_patterns_specification. The TCK period in the patterns must be consistent with the one used during timing closure.

IJTAG Network Performance Optimization

This section describes how you can optimize IJTAG network performance. The methods described are based on the concept of source synchronous timing, where the clock launching the signals is propagated together with them and used to capture the signals at the destination. The smaller the difference in the propagation time of the clock and signals, the better the performance.

Figure 9-1. Example Chip With Embedded Blocks and IJTAG Network
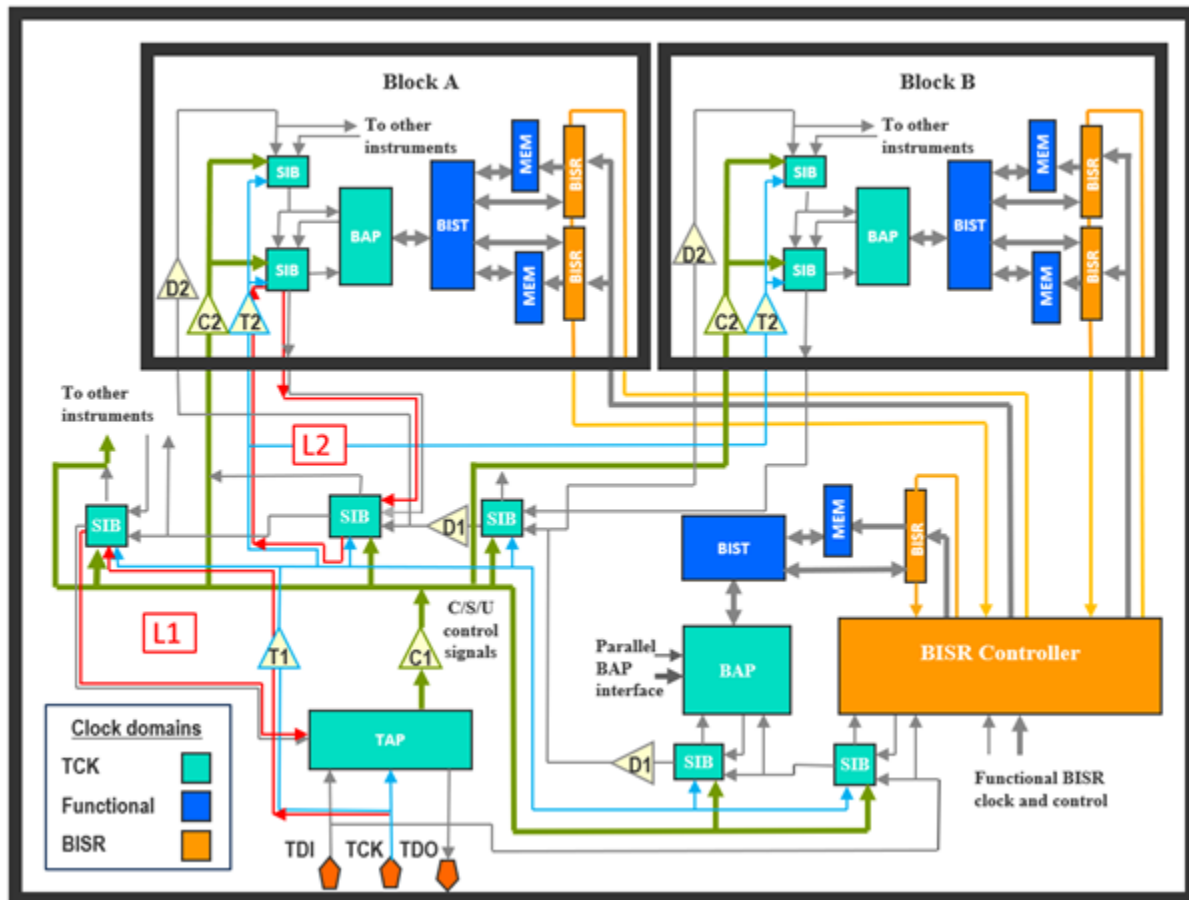
Figure 9-1 shows a design with embedded blocks. Blocks are physical regions that are laid out and signed off separately. The components of the IJTAG network are shown in teal and are all clocked by TCK. The network is primarily used to control memory BIST and repair controllers in this example, but any other controller type could be connected to the network.

TCK, the control signals of interest (capture enable, shift enable, update enable, and select), and scan data paths may be heavily loaded due to a high fanout or long wires and therefore need buffering, which introduces a delay. TCK buffers and their associated delay are represented by triangles labeled T1 and T2, whereas the triangles labeled C1 and C2 represent buffers and delay for control signals. The triangles labeled D1 and D2 represent the data path delays due to high-load fanout or buffering.

High-load fanouts can cause data path delays, which may be cumulative across the hierarchy. An example is a physical block instantiated within a second physical block which is, in turn, under a third physical block, if the same line feeds all three levels of hierarchy. Another example is a serial scan path delay, where a particular SIB has a high-load fanout that reaches

many instruments. For these two examples and other data path delay scenarios, Equation 9-1 must be satisfied:

$$| \sum D_{TCK} - \sum D_{data} | < 0.5\ TCK_{period} \qquad (9\text{-}1)$$

However, Equation 9-1 may not be satisfied for physical blocks and instruments deep in the IJTAG hierarchy and may be valid only for instruments near the top of the design.

Control paths driven by the TAP are distributed over the entire chip, resulting in cumulative delays across the hierarchy in addition to the data path delay. In that scenario, Equation 9-2 must be satisfied:

$$| \sum D_{TCK} - \sum D_{ctrl} | < 0.5\ TCK_{period} \qquad (9\text{-}2)$$

However, Equation 9-2 may not be valid for physical blocks and instruments placed away from the TAP.

The objective is to keep the total propagation delays of control signals, data paths, and TCK within ½ cycle of the target network frequency over the entire design.

Due to the timing limitations within IJTAG networks and loading, the instruments or physical blocks placed deep in the hierarchy may have problems while operating at higher TCK frequencies. Two important timing paths are highlighted in red and labeled L1 and L2 in Figure 9-1. These paths, also called timing loops, connect different levels of the TCK tree and are the source of these problems. These loops are common in IJTAG network topologies, and it is not possible to avoid them entirely. In timing loop L1, the previous (n-1) TCK pulse clocked the TAP, but the SIB that shifts data to the TAP is clocked by the current (n) TCK pulse. In timing loop L2, the previous (n-1) TCK pulse clocks the SIB that hosts the physical Block A, while the current (n) TCK pulse clocks the data coming from the child block.

The TCK signal delay must not be greater than half of the TCK period for each timing loop. Equation 9-3 determines the highest possible frequency in the entire IJTAG network:

$$D_{TCK} < 0.5\ TCK_{period} \qquad (9\text{-}3)$$

Scan Insertion Bits (SIBs) provide a natural pipeline mechanism for the serial output of instruments and embedded blocks all the way to the TDO output. The strobe on TDO is aligned with the rising edge of TCK by default in the manufacturing patterns to emulate the conditions encountered once the circuit is embedded in the system. Delaying the strobe does not provide any benefit when the TAP clock is an early version of TCK as recommended. This is because of the presence of a latch in the TAP that closes on the rising edge of TCK so that the loop timing is still ½ TCK cycle.

In a hierarchical design, it is recommended to specify a TCK clock frequency at the block level that is higher than the final one at the chip top level. This enables the tool to distribute the margin between the blocks and the top level. For example, suppose that the target frequency at the chip level is 50MHz, blocks could use 100MHz as the target when signed off. Small blocks

could use even higher frequencies as they are more likely to become part of larger blocks before being instantiated in the chip top level.

FastIJTAG Solutions

FastIJTAG extends Tessent IJTAG by providing solutions for timing closure problems. The solutions can help you create an IJTAG network that operates at the desired frequency in your chip.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

DFT Specification Implementations

You can implement any of the following IJTAG network performance optimization solutions to resolve timing closure problems when you specify the DFT Specification requirements for your design.

- Scan input pipelining adds a one-bit TDR in series with the scan path inside heavily loaded SIBs to provide an additional output port.

- Placement-aware IJTAG stitching uses physical layout information to create an IJTAG chain with the shortest path. Refer to "Placement-Aware IJTAG Stitching" and the "TDI Scan Input Pipelining" example.

- SIB output retiming stages create a reliable IJTAG network, but some SIBs may require an adjustment when not optimal in a timing loop.

Scan Input Pipelining

Scan input pipelining provides an additional output port by adding a one-bit TDR in series with the scan path inside heavily loaded SIBs.

Suppose your design uses the nesting of physical blocks to form a multilevel design hierarchy. In that case, it may contain a long data path with many fanouts from the TDI port to the lowest physical block in the hierarchy, or even from a SIB to deep hierarchies. Figure 9-2 provides an illustration.

Figure 9-2. Multilevel Design Hierarchy With Long Data Path
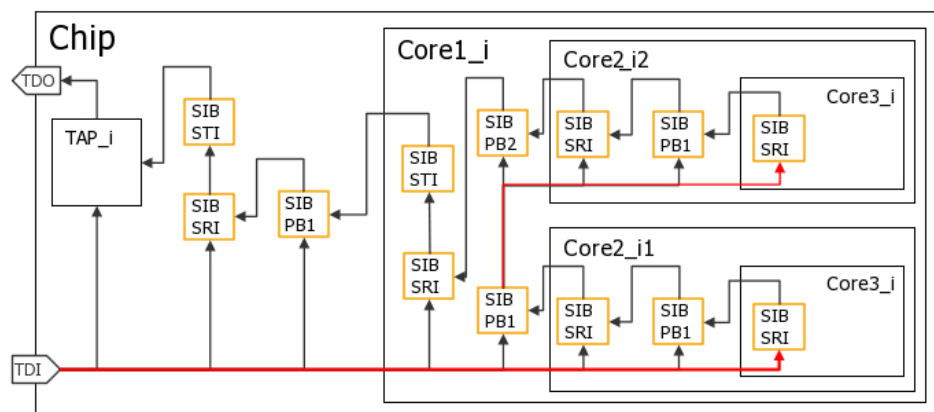
To avoid these scenarios, add a pipeline stage consisting of a one-bit TDR connected in series with the scan path inside the SIB. You add the pipeline stage by specifying the "to_scan_in_feedthrough : pipeline" property and value in the HostScanInterface/Sib(*id*) wrapper of the DftSpecification, or by specifying "to_scan_in_feedthrough_on_pb_sibs :

pipeline" in the IjtagNetwork wrapper of the DefaultsSpecification, as shown in the "TDI Scan Input Pipelining" example.

The pipeline location is not disruptive and preserves the original network topology. The pipelining of the scan input path resynchronizes the data line with the TCK clock provided to the SIB. Equation 9-4 must be valid for each timing loop for each hierarchy level separately:

$$| \ D_{TCK} - D_{data} \ | < 0.5 \ TCK_{period} \tag{9-4}$$

Figure 9-3. Multilevel Design Hierarchy Using Scan Input Pipelining
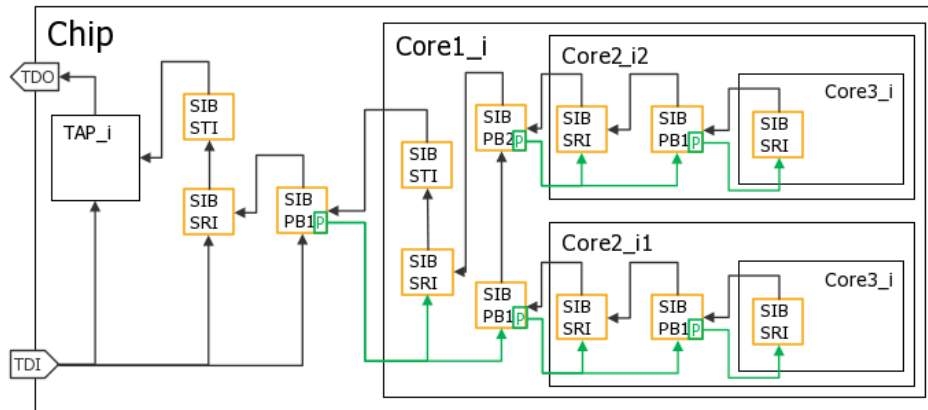
Figure 9-3 shows the use of the pipeline stage (P) for SIBs with heavily loaded fanouts, such as the SIB that hosts the physical block. This pipeline stage changes the layout footprint of affected SIBs with the addition of the output port "to_ijtag_si" on the SIB module boundaries. Figure 9-4 marks the added pipeline stage with a red circle in the circuit.

Figure 9-4. Tessent SIB With Pipeline Stage
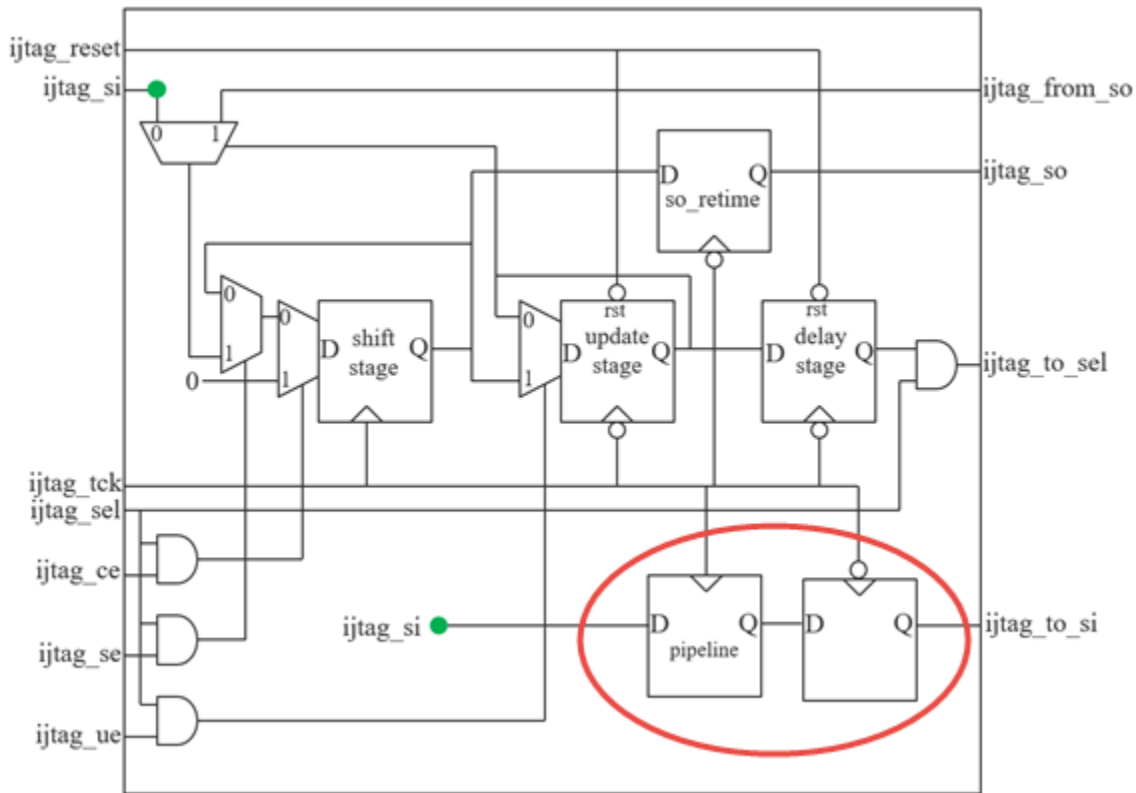
SIB Output Retiming Stage

Tessent Shell's scan path retiming creates a reliable network, but it is not optimal for timing loops. You can set the so_retiming property in the Sib wrapper of the DftSpecification to remove the retiming element from the scan-out port to help resolve loop timing issues.

The circuit of Figure 9-5 shows an example timing loop where a parent SIB hosts a SIB within a child block one hierarchy level below, similar to timing loop L2 shown in Figure 9-1. The "so_retime" flop inside the SIB (refer to Figure 9-7) may be problematic if it is the last SIB of the chain driving the "ijtag_so" output of an entire child block.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

**Figure 9-5. Timing Loop of Parent SIB Hosting a Child SIB**
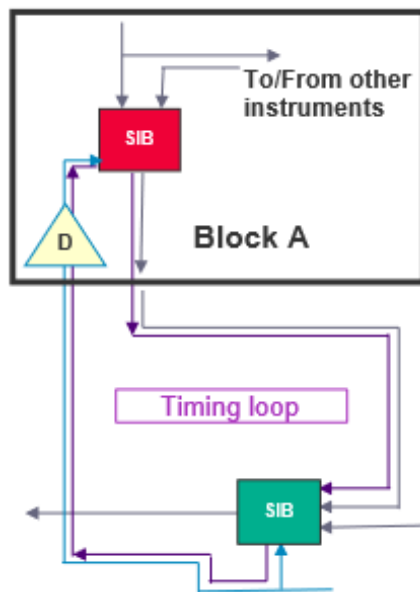
The circuit above guarantees that the TCK of the child occurs later than the parent TCK due to buffer delay. The timing diagram on the left of Figure 9-6, where the "so_retime" flop is present (so_retiming : on), shows there is less than a half of a TCK cycle for data setup time at the host SIB, since the child SIB data launches from the falling edge of its delayed TCK.

Figure 9-6. so_retiming and Data Setup at Host SIB
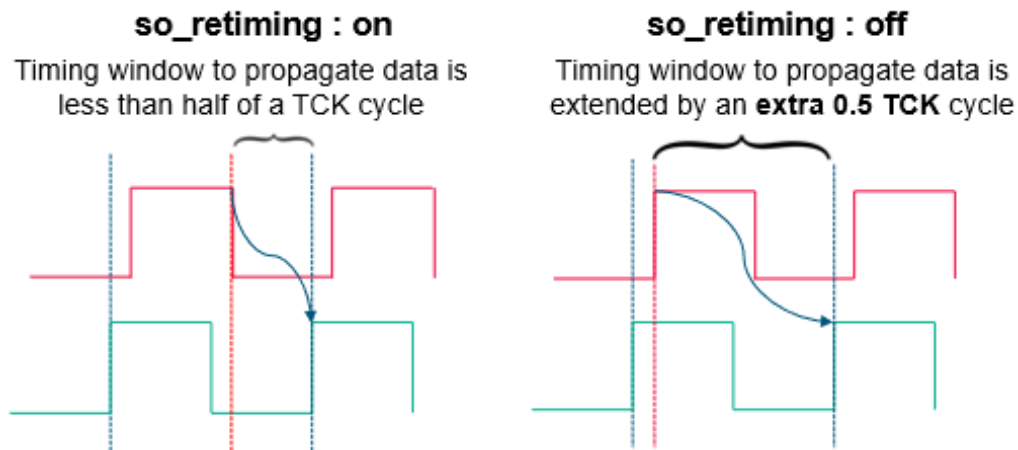
If you have loop timing closure issues, you can remove the "so_retime" flop from the child SIB to extend the time window by an extra 0.5 TCK period, as shown in the right timing diagram above. For this configuration, the child SIB data launches from the rising edge of its delayed TCK.

**Figure 9-7. Tessent SIB Circuit Diagram (Retiming Stage Circled)**
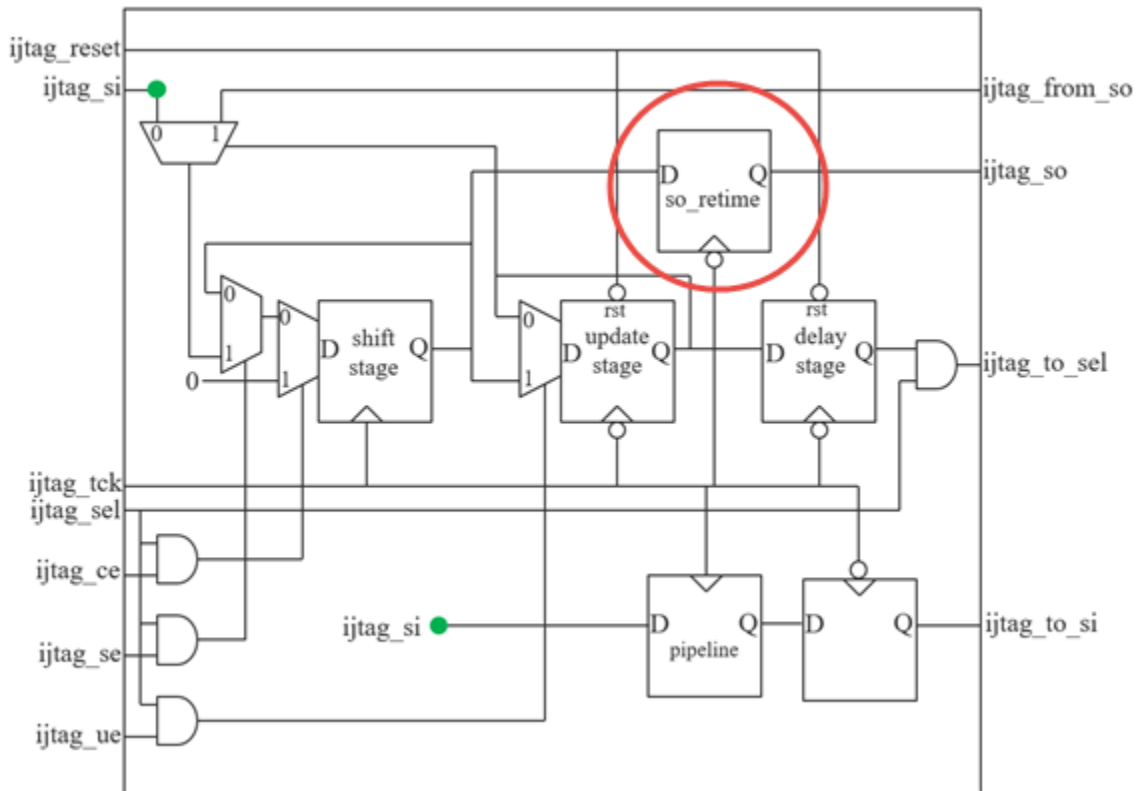
Automatically determining the last SIB in the scan chain of a block is impossible for Tessent Shell because of the possibility of future insertion passes. Therefore, you must choose the required SIBs and manually specify their so_retiming properties to remove their scan-out retiming elements. You specify "so_retiming : off" in the HostScanInterface/Sib wrapper of the DftSpecification to remove the scan-out retiming element on the final Sib.

Clock Tree Balancing

The first thing to verify in your implementation is to make sure that you connect the TAP to an early version of TCK during clock tree synthesis (CTS).

This may not happen by default. Figure 9-8 shows what you might need to do to guide the layout tool during the TCK clock tree synthesis step. You specify tessent_persistent_cell_tck_cts_stop_buf as a stop point to the clock tree synthesis command, which keeps the TAP on an early version of TCK. The layout placement of this buffer is immediately after the TCK input. You do not need to modify the timing constraints generated by the Tessent tools. The CTS buffer is always added inside the Tessent TAP controller.

Figure 9-8. TCK CLK Tree With Stop Points
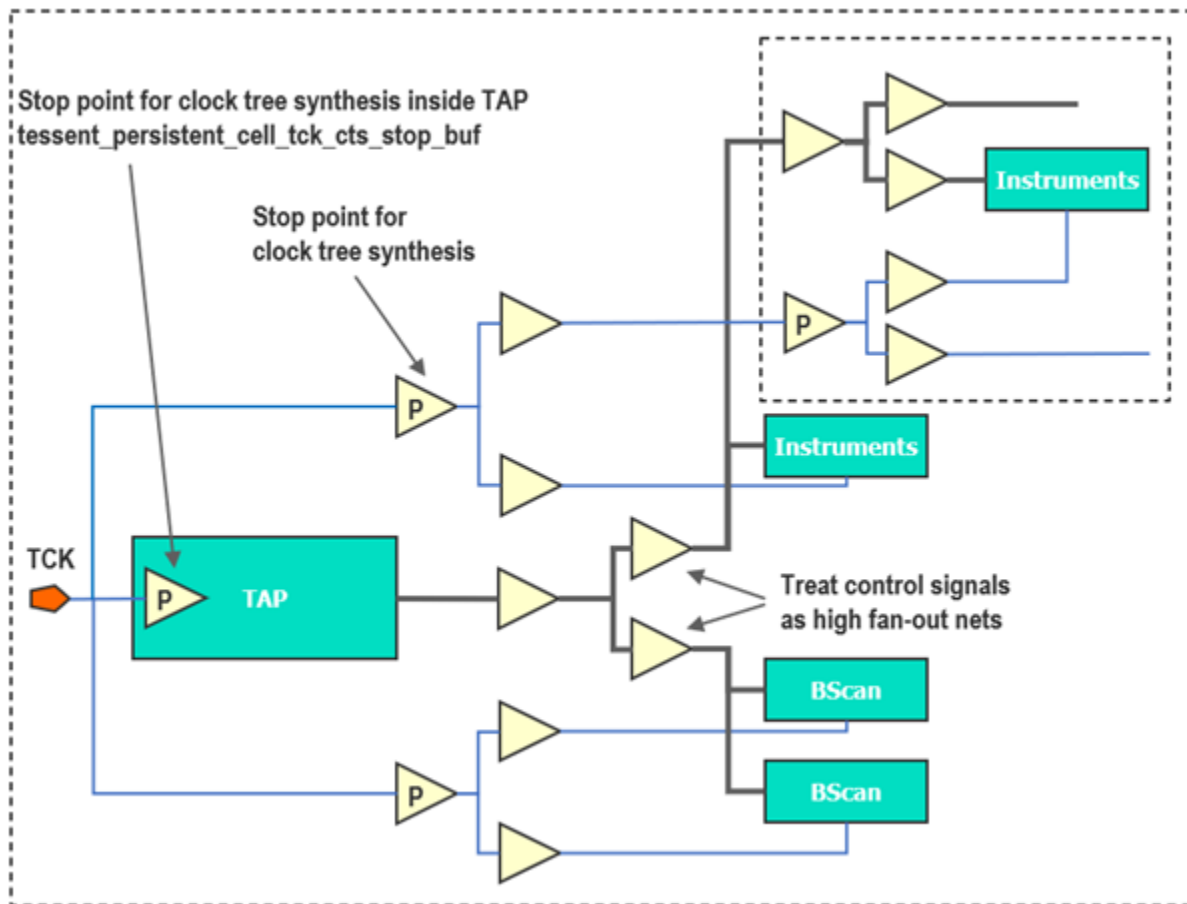
In a hierarchical design, the TCK clock input of embedded blocks must also be designated as a stop point during clock tree synthesis. This is applicable whether the blocks are inserted at the chip top level or within another block.

Another implementation aspect is that control signals might require the use of specialized layout tool commands to treat high fanout nets. The objective is to achieve roughly similar propagation delays compared to TCK to satisfy equation 9-2.

The serial input to the network, TDI, has not been discussed because it is naturally sourced by an early version of TCK from a chip pin and is less loaded than the other control signals. It must however satisfy equation 9-2 and might have to be treated like control signals.

Software Clock Stretching

Control signal delays are cumulative across the design hierarchy. Timing delays affect IJTAG instruments, especially when placed far away from their TAP controller. Timing delays also affect TMS timing, for example in designs that require routing TMS to multiple embedded TAP controllers. Selective TCK stretching solves signal delay problems for heavily loaded control signals in large IJTAG networks.

This type of clock stretching primarily affects the non-shift states of the TAP controller Finite State Machine (FSM), which minimizes the impact on test time. Selective TCK stretching may be accomplished in three ways, depending on two conditions:

- TCK clock ratio

- ATE capability to apply different test clock periods

Selective TCK Stretching

Selective TCK stretching increases the setup and hold timing margins of IJTAG control signals and the TMS TAP controller input by one or more TCK periods.

Use the set_ijtag_retargeting_options command and specify the number of extra setup and hold cycles using the "-extra_control_setup_hold_cycles" and "-extra_tms_setup_hold_cycles" switches.

Figure 9-9 shows the typical TCK waveform. All setup and hold times are equal to half of the TCK cycle. The FSM inside the TAP changes state on the positive edge, but the TAP outputs (CAPTURE_EN, SHIFT_EN) and IJTAG instrument Select update on the negative edge in the same cycle. However, UPDATE_EN updates on the same positive edge as the FSM state. Signals propagate across the hierarchy to instruments that capture the controls on the positive edge in the next cycle, except UPDATE, which captures on the negative edge. The IJTAG Select signal qualifies (logical AND) each TAP output signal for an instrument and is captured on the same edge as the control signal. Therefore, the setup and hold times for Select are more relaxed than for the TAP output signals individually.

Figure 9-9. Normal TCK (Setup/Hold 0.5 x TCK Period)
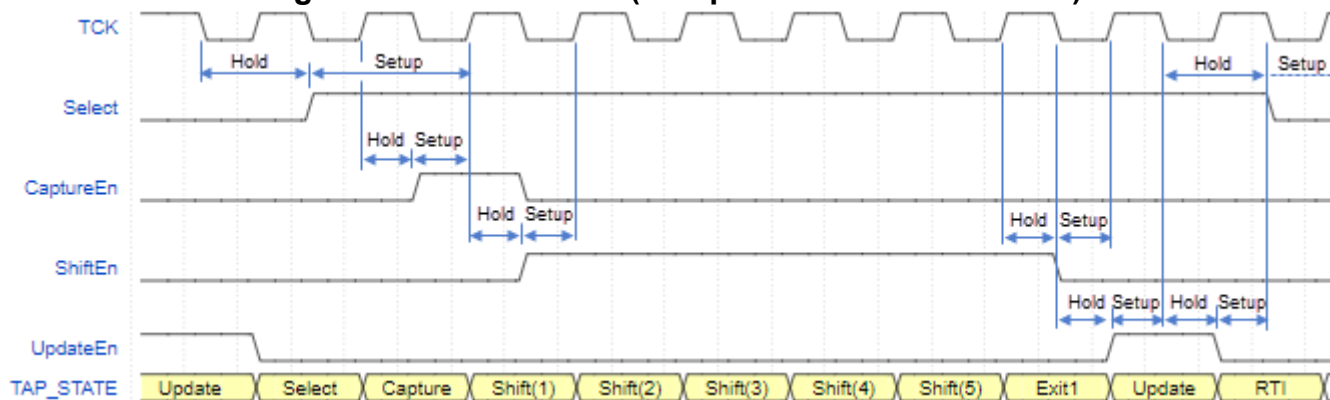
When you specify the "-extra_control_setup_hold_cycles" switch, the selective TCK stretching adds extra setup and hold time for the IJTAG control signals, and in the case shown in Figure 9-10 and Figure 9-11, these expand from 0.5TCK to 1.5TCK. The previous timing equation (Equation 9-4) modifies with "+ N × TCK_{period}", where you determine the number of additional cycles N, to become Equation 9-5:

$$| \textstyle\sum D_{TCK} - \sum D_{ctrl} | < 0.5 \; TCK_{period} + N \times TCK_{period} \tag{9-5}$$

___ **Note** ___

The maximum number of extra control or TMS setup and hold cycles is 256.

Selective TCK stretching for IJTAG control signals occur for the CAPTURE, UPDATE, and Select pulses, and the first SHIFT pulse in each iApply frame. The rest of the SHIFT phase runs at the full TCK frequency. Selective TCK stretching makes each frame longer, but it does not affect the SHIFT, which is the longest part of the frame, except in the first pulse. It is crucial for IJTAG performance that selective TCK stretching does not cause the SHIFT phase to run slower.

For a TCK ratio of 2 with a fifty percent duty cycle, the TCK primary input drives its value in the middle of the tester cycle. Therefore, two tester cycles model one TCK pulse, as shown in Figure 9-10 and Figure 9-11. A fifty percent duty cycle for both stretched and non-stretched versions of the TCK signal generates correct constraints for the SDC file.

Figure 9-10. Selective TCK Stretching (-extra_control_setup_hold_cycles 1)
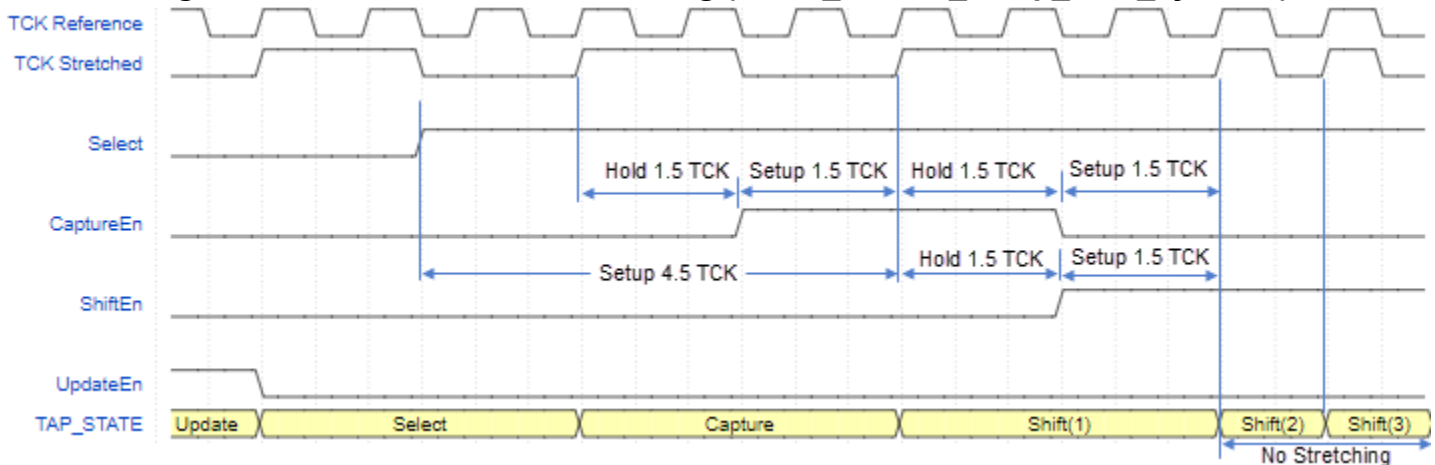
Figure 9-11. Selective TCK Stretching (-extra_control_setup_hold_cycles 1), Continued

---
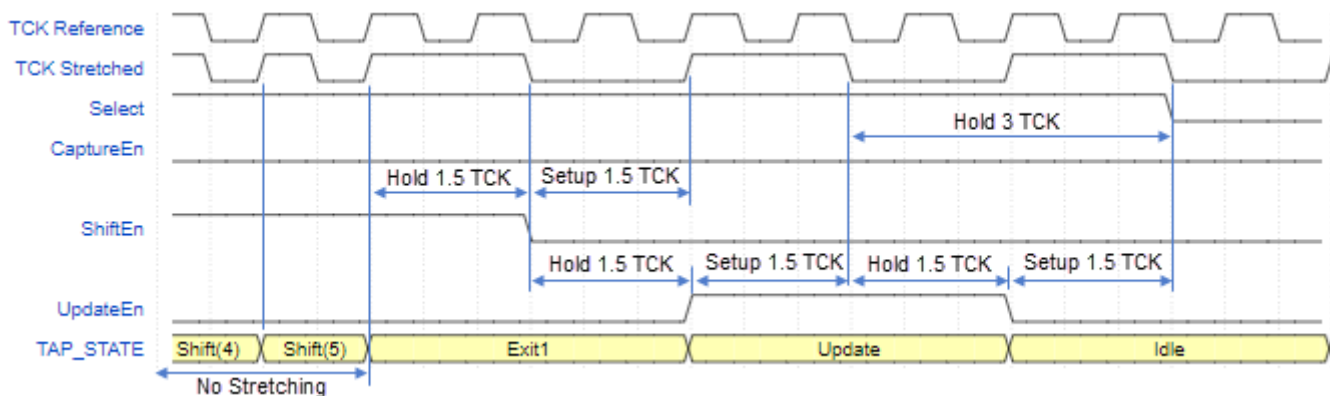
Note

When you use a custom timeplate for IJTAG operations, the timing of the IJTAG control signals can limit the extra setup and hold margins the tool adds with selective TCK stretching. Refer to the "Custom Timeplate Event Timing and TCK Stretching" section in "FastIJTAG Limitations" for more information.

The TAP controller samples TMS on the positive edge of TCK and setup and hold times are equal to 0.25 TCK and 0.75 TCK respectively. The setup and hold margins are not balanced because the Tessent generic timeplate forces the TMS signal at time "0" during a force_pi event, which forces the signal in the middle of the tester clock "off" state. When you specify the "-extra_tms_setup_hold_cycles" switch, the selective TCK stretching adds extra setup and hold time by extending the TCK cycle by a "N x TCK$_{period}$" for each, where "N" is the number of additional cycles specified. Figure 9-12 illustrates the hold time expansion from 0.75 TCK to 1.75 TCK and setup time from 0.25 TCK to 1.25 TCK when you specify a value of "1". Unlike the "-extra_control_setup_hold_cycles" switch, using the "-extra_tms_setup_hold_cycles" switch does not require the tool to generate extra timeplates, as described in the following sections. The tool automatically generates timeplates in this case.

Figure 9-12. TMS Selective TCK Stretching (1.25 TCK Setup, 1.75 TCK Hold)
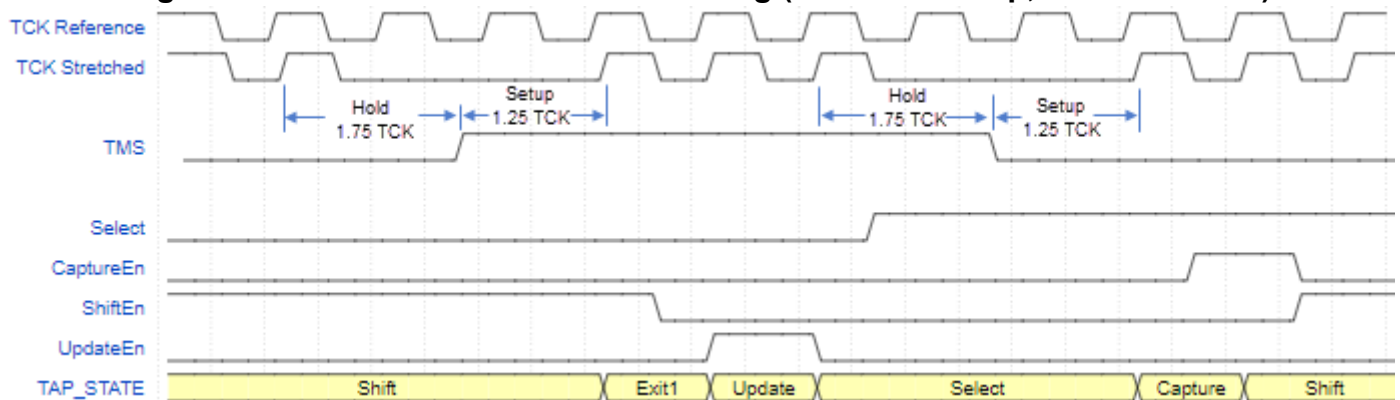
# TCK Ratio and Single Period Tester

If the tester supports only one single clock period for the pattern set and the TCK ratio equals one, the TCK waveform is split into multiple time frames. Each time frame corresponds to a single tester cycle and an associated timeplate defining TCK as a port with forced timing.

Figure 9-13 illustrates selective TCK stretching by using two additional timeplates with the same tester period. The figure shows that two additional non-return-to-zero (NRZ) timeplates are required to model the TCK waveform for three tester cycles. The first additional timeplate has a force time at 25% of the period, the second at 75%. At most, three different timeplates are required to form both the original and stretched TCK waveforms, regardless of the shape of the original TCK waveform and the number of extra pulses for stretching.

Note

The edge timings listed on top of the figure are relative. The "0.25" means that the edge occurs at 25% of the tester period. The "1" and "0" represent the value to pulse or force on TCK in the cycle. The RZ and NRZ determine the value that TCK returns to by the end of the cycle.

Figure 9-13. Selective TCK Stretching Using Two Additional Timeplates
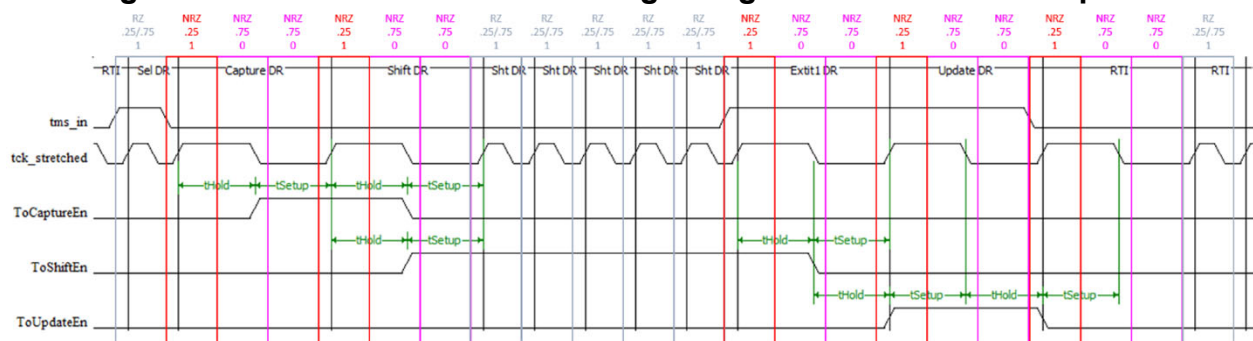
TCK Ratio Greater Than One

When a TCK ratio is greater than one, the TCK signal is treated as a data input line instead of a clock and influences the format of the timeplate for pattern generation.

For example, Figure 9-10 and Figure 9-11 show waveforms for a TCK ratio equal to two. Here, the TCK primary input is forced to its value in the middle of the tester period, so two tester cycles are required to model one pulse of TCK. Figure 9-14 shows the corresponding timeplate with a TCK ratio of "2".

Figure 9-14. Timeplate With TCK Ratio of 2

```
timeplate tessent_ijtag =
  force_pi 0;
  measure_po 2.4;
  force TCK 5;
  pulse_clock 2.5 5;
  period 10;
end;
```

Multiple Period Tester

Some testers support different clock periods per timeplate for the same pattern set. Those testers can form a signal with selective TCK stretching by using a single return-to-zero or return-to-one waveform placed in a single timeplate with a larger period. A TCK ratio of one is required in this case.

> **Note**
> Refer to the "set_tester_options" command in the *Tessent Shell Reference Manual* for more information.

Figure 9-15 shows a tester-based waveform with selective TCK stretching that uses one additional timeplate (shown in red) for a pattern. The timeplates use different tester periods.

Figure 9-15. Tester-Based Selective TCK Stretching
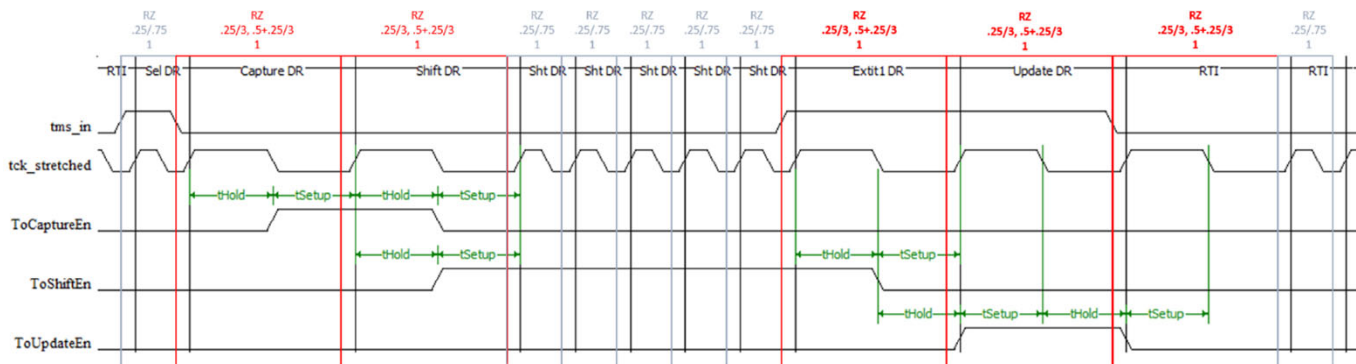
Custom TCK Timeplate and Duty Cycle

For custom TCK timeplates where the TCK clock duty cycle is not 50%, stretching of the waveform is proportional.

Figure 9-16 illustrates proportional, return-to-zero (RZ) waveforms of a custom timeplate. The timeplate has one extra setup and hold cycle. Table 9-1 lists the time of events of the custom timeplate. The period of the non-stretched timeplate is 100 ns.

The one extra cycle for both setup and hold time make the non-stretched time frame three times longer. The original hold time is extended three times, the same as the setup time.

The original, non-stretched leading edge of TCK occurs at 45 ns, and the trailing edge occurs at 70 ns. The hold time is 25 ns (70 - 45 = 25).

After selective TCK stretching, the leading edge is at the same point, but the trailing edge moves to 120 ns. The stretched hold time is 75 ns (120 - 45 = 75) and is three times longer than the original hold time.

The stretching of the timeplate is correct. However, the stretched hold time is shorter than 100 ns, which is the original period. The setup time is longer than the two original periods (225 ns). This scenario may result in incorrect SDC constraints generated for Tessent IJTAG instruments and is a known limitation.

_____ **Note** _____

Refer to "Impact on Timing" on page 211 for more details.
_____

Figure 9-16. Custom RZ Timeplate Waveforms
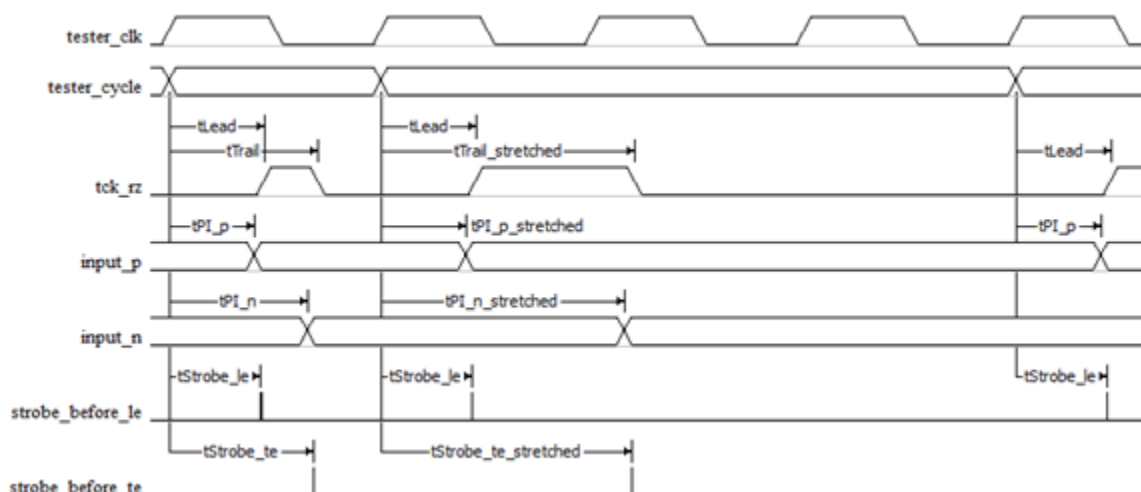
Table 9-1. Custom RZ Timeplate Time of Events

| Event	Original Time	Stretched Time
Leading edge	45 ns	45 ns
Trailing edge	70 ns	120 ns
Force PI before the leading edge	40 ns	40 ns
Force PI before the trailing edge	65 ns	115 ns
Strobe PO before the leading edge	43 ns	43 ns
Strobe PO before the trailing edge	68 ns	118 ns

Non-TCK Clocks and Selective TCK Stretching

Selective TCK stretching has no impact on non-TCK clocks defined in the generated timeplates.

When you specify "set_tester_options -timeplate_constraints same_period", the generated timeplates for selective TCK stretching distribute the TCK cycle over multiple tester cycles and the non-TCK clock pulses within each tester cycle.

When you specify "set_tester_options -timeplate_constraints none", the tool converts non-TCK clocks defined with "add_clocks -pulse_always" into multiple-pulse timing within each tester cycle in the timeplates generated for selective TCK stretching. The conversion to multiple-pulse timing also occurs for ICL ClockPorts defined with "add_icl_ports -function_modifier sync_tester_clock". Refer to "Multiple-Pulse Clocks" in the *Tessent Shell User's Manual* for additional information.

Therefore, selective stretching of TCK does not affect non-TCK clocks and the waveforms generate as shown, independent of tester timeplate constraints.

Figure 9-17. Non-TCK Clock During TCK Stretching
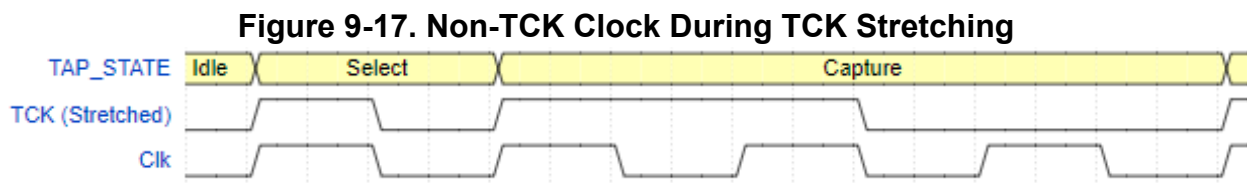
Impact on Timing

Selective TCK stretching adds programmable multicycle paths (MCPs) to the IJTAG SDC generated by Tessent Shell. The MCPs relax the setup and hold times of the TMS TAP input, and the IJTAG control signals capture_enable, shift_enable, update_enable, and select.

Timing Flow

Tessent Shell generates an SDC file with a Tcl procedure that uses the same name as the Tessent Shell command set_ijtag_retargeting_options. This procedure takes any switch and value argument pairs used by Tessent Shell, but it only processes the arguments of the "-extra_control_setup_hold_cycles" and "-extra_tms_setup_hold_cycles" switches. Then, the Tcl procedure sets the global SDC variables tessent_extra_control_setup_hold_cycles and tessent_extra_tms_setup_hold_cycles to the required values.

Next, you must create and source a side file into the timing tool and Tessent Shell to keep the design models consistent between the tools.

___ **Note** ___

Refer to "Selective TCK Stretching" for an example.

Custom Timeplates Considerations

The tool extends setup and hold times proportionally for custom timeplates with TCK duty cycles other than 50%. However, the resulting setup and hold times can be shorter than those of the original timeplate, which can cause incorrect generation of SDC constraints for Tessent IJTAG instruments.

To ensure the correct SDC, you must create the TCK clock in the SDC to represent the non-50% duty cycle. Tessent Shell creates SDC assuming the TCK clock has a 50% duty cycle. You must be aware that the timing margin of extra setup and hold cycles are proportionally distributed depending on your duty cycle.

For example, suppose you specified 2 extra setup and hold cycles. The timing margin of these extra 4 cycles (2 setup and 2 hold) are proportionally distributed based on the duty cycle.

Third-Party TAP Considerations

If you have your own, or a third-party TAP, Tessent Shell does not automatically generate the programmable MCPs in the IJTAG SDC file, and you should add an equivalent of the following pseudo-constraints to your SDC:

```
# Select from the TAP is toggled in Update-IR state
# and won't be used until three TCK cycles later, in Capture-DR
set_multicycle_path
  -setup [expr 3 + (3 * $tessent_extra_control_setup_hold_cycles)] \
    <from ijtag select source>
set_false_path -hold <from ijtag select source>

if {$tessent_extra_control_setup_hold_cycles > 0} {
  # Relaxing DR capture, shift and update based on
  # selective TCK stretching
  set_multicycle_path
    -setup [expr 1 + $tessent_extra_control_setup_hold_cycles] \
      <from capture_en, shift_en, update_en sources>
  set_multicycle_path
    -hold [expr 2 * $tessent_extra_control_setup_hold_cycles] \
      <from capture_en, shit_en, update_en sources>
}
```

Impact on SSN Patterns

SSN utilizes global TCK stretching. Selective TCK stretching can help close timing requirements. The impact of selective TCK stretching for SSN patterns is:

- The tool performs selective TCK stretching during the write_patterns command if there are IJTAG operations to write as part of the pattern set. This effectively creates TCK as a ratio of the SSN bus clock to keep the SSN bus as fast as possible.

  > **Note**
  > Refer to the "set_tester_options -timeplate_constraints same_period" command in the *Tessent Shell Reference Manual* for a description of global TCK stretching for SSN.

- During the test_setup, test_end, ssn_setup, and ssn_end, the tool further stretches TCK and applies the extra setup and hold cycles that were specified by the set_ijtag_retargeting_options command.

Additional Flow Information

This section provides additional information to consider when developing a flow.

Procedure Files

The write_procfile command writes procedure files and does not apply selective TCK stretching.

Writing procedures as part of a Tessent Core Description (TCD) flow, or writing TSDB data, does not save selective TCK stretching information.

Pattern Formats

The write_patterns command performs selective TCK stretching only for test benches and tester pattern formats. Pattern formats that do not support the application of selective TCK stretching are:

- ASCII

- Binary

- PatDB

- PDL

- SVF

Writing Patterns

When using the write_patterns command with selective TCK stretching enabled, full ICL and PDL information must be present in the following contexts if SSN is not present:

- patterns -scan

- patterns -scan_retargeting

- patterns -scan_diagnosis

The flow ensures that the required information is available in other contexts.

Backward Compatibility

Scan input pipelining and selective TCK stretching are backward compatible and have no impact on the existing flow.

- Placement-aware IJTAG stitching is enabled by default when a DEF file is loaded. For alphabetical stitching, do not load a DEF file.

- The previous behavior of the SIBs does not change because the default value of the *to_scan_in_feedthrough* property is "none".

- The previous behavior of the SIBs does not change because the default value of so_retiming is "on".

- The default number of extra pulses in selective TCK stretching is "0". You must define a value of "1" or higher to use this feature.

FastIJTAG Limitations

This section provides a collection of known limitations.

Temporary

MissionMode

The MissionMode controller (InSystemTest) is currently not compatible with selective TCK stretching.

SSN Streaming through IJTAG

SSN Streaming through IJTAG is currently incompatible with selective TCK stretching for the external capture mode (set_current_mode -type { unwrapped | external }). This limitation does not affect internal capture mode.

SVF Pattern Format

Tessent Shell can create SVF but conditionally applies selective TCK stretching as follows:

When generating patterns through a TAP interface (typically chip level), Tessent Shell writes SVF patterns using the protocol-aware SDR and SIR commands. In this case, the pattern file does not express the TCK stretching. This is not a limitation. It is a feature of the SVF format not to express the TCK cycles. Tessent Shell can read this format back in. It can apply selective stretching to generate another pattern format.

When generating patterns through an IJTAG interface (typically block level), Tessent Shell writes SVF patterns using the explicit PIO commands. In this case, the pattern file expresses the selective TCK stretching. However, Tessent Shell does not currently apply stretching in the PIO SVF. Tessent Shell can read this format back in. When generating another pattern format, it applies selective stretching as expressed in the PIO pattern regardless of the value of the extra_control_setup_hold_cycles option.

Permanent

Pattern Generation with an MBIST Asynchronous Interface

Prior to the 2020.4 release, the MBIST IP implemented an asynchronous interface (AI) to access its internal chains. When using legacy MBIST hardware with AI, the BIST_CLK frequency to TCK frequency ratio must be at least 8 to generate MBIST patterns. Otherwise, Tessent displays an error message. To decrease the TCK frequency, you can either increase the value of the tester_period property or increase the value of the tck_ratio property.

The STA scripts to validate the MBIST AI timing have not been updated to reflect selective TCK stretching. Specifically, the reporting of the BIST_SHIFT timing is pessimistic.

Starting with the 2020.4 release, AI was replaced by the Enhanced MBIST Access (EMA) mechanism. A controller equipped with EMA is not affected by this limitation.

TCK Ratio and Custom Timeplates

Default timing can use any even TCK ratio, but custom timeplates require the TCK ratio to be one or a power of 2. If the TCK ratio set by the set_ijtag_retargeting_options command is greater than one, the tool adjusts TCK automatically to the next nearest power of two. However, the "open_pattern_set -tck_ratio" command gives an error when using it to set the TCK ratio to an even value that is not a power of 2. If the custom timeplate defines TCK as a non-return-to-zero waveform, the minimum TCK ratio of 2 is required.

Custom Timeplate Event Timing and TCK Stretching

When you use a custom timeplate for IJTAG operations, the timing of the IJTAG control signals can limit the extra setup and hold margins the tool adds with selective TCK stretching. The following table outlines how you define the timeplate force time for various control signals to generate proper setup and hold margins. Refer to the timeplate descriptions and associated waveforms in the open_pattern_set command description to supplement the table information.

Table 9-2. Custom Timeplate Specifications for Full TCK Stretching

TCK Ratio	TCK Off State	Timeplate Event	Event Condition to Achieve Full TCK Stretching
1	0	Capture/Shift Enable force time	Must be after the TCK rising edge
	1	Update Enable force time	Must be after the TCK falling edge
2 or more	0 or 1	Capture/Shift/Update Enable force time	Must be after the TCK force time

Selective TCK stretching is still occurs if the custom timeplate does not meet the outlined conditions; however, the margins on the setup and hold times may not be optimal.

FastIJTAG Examples

The examples show how to use FastIJTAG solutions with the Tessent flow.

- **Scan Input Pipelining —** How to set up a scan input pipeline using SIBs.

- **Selective TCK Stretching —** How to perform selective clock stretching for TCK.

TDI Scan Input Pipelining

This example describes how to set up a scan input pipeline using SIBs in an IJTAG network with the Tessent flow.

The example uses the DefaultsSpecification property to instruct the DftSpecification creation to add scan-in pipelining on SIBs that the tool adds in front of child physical blocks.

Example 9-1. Using Tessent Shell to add a Pipeline

```
# Load placement information for placement-aware IJTAG stitching.
SETUP> read_def my_chip.def
# Instruct Tessent Shell to add a pipeline to all physical blocks SIBs
SETUP> set_defaults_value \
        DftSpecification/IjtagNetwork/to_scan_in_feedthrough_on_pb_sibs \
        pipeline
# Create the DftSpecification
SETUP> check_design_rules
ANALYSIS> set dft_spec [create_dft_specification]
# Report out the DftSpecification
ANALYSIS> report_config_data $dft_spec
DftSpecification(parent,rtl) {
  IjtagNetwork {
    HostScanInterface(tap) {
      Interface {
        tck : tck;
      }
      Tap(main) {
        HostIjtag(1) {
          Sib(sri) {
            Attributes {
              tessent_dft_function : scan_resource_instrument_host;
            }
            Sib(pb1) {
              to_scan_in_feedthrough : pipeline;
              DesignInstance(child) {
                scan_interface : ijtag;
              }
            }
          }
        }
      }
    }
  }
}
```

Selective TCK Stretching

This example describes how to perform selective TCK stretching for an IJTAG network with the Tessent flow. Selective TCK stretching provides flexibility to relax timing constraints on critical IJTAG control signals.

First, you must determine any selective TCK stretching requirements using your synthesis or timing closure tool. You may need to establish the stretching requirements for the IJTAG control ports of a physical block or the control signal registers inside the TAP controller. If you have problems closing timing on these paths, use your timing tool to determine the stretching requirements, such as how many extra setup or hold cycles to use to meet the slack requirements.

Second, using a side file, hand off the information to pattern generation to ensure consistency with the timing tool.

Suppose for an example design, the TCK frequency is 100 MHz (10 ns period) and timing analysis tools can resolve slack issues using one extra 10 ns cycle. You specify the number of additional pulses and the desired TCK period.

Step One: Establish Stretching Requirements

Step one occurs during synthesis and static timing analysis of your design outside the Tessent environment.

For this example design named "chip1", you need one extra cycle to meet the timing needs for a 10 ns TCK period. Create a side file, such as in the following example, and source it in your timing script. Add the source statement between the calls to tessent_set_default_variables and tessent_set_non_modal, as shown in the following example.

Verify that the new multicycle paths resolve the timing violations. If they do, you are ready to use the side file for pattern generation in step two.

> **Note**
> 📄 Refer to "Timing Constraints (SDC)" and "Example Scripts Using Tessent Tool-Generated SDC" in the *Tessent Shell User's Manual* for more information.

Example Side File

The following code example is a side file to add one extra cycle for a 10 ns TCK period to meet the slack requirements necessary to close timing. Hand the side file off to pattern generation and source it in Tessent Shell to ensure timing consistency.

Example 9-2. Side File for Pattern Generation

```
# Side file enable_tessent_fastIJTAG.tcl
# Replace with your own values.

set tck_period 10
set_ijtag_retargeting_options -extra_control_setup_hold_cycles 1 \
                              -extra_tms_setup_hold_cycles 1 \
                              -tck_period "${tck_period}ns"
```

Example Timing Script

The following code example is part of a timing script that sources the side file to apply the correct SDC timing constraints consistently.

Example 9-3. Partial Script Sourcing the Side File

```
...
source <tsdb_outdir>/chip1.sdc
tessent_set_default_variables
# source the side file to set the multicycle paths
source ../data/enable_tessent_fastIJTAG.tcl
tessent_set_non_modal
update_timing
...
```

Step Two: Generate Patterns with Stretching

Step two occurs within the Tessent environment.

After creating the side file with the required extra pulses and TCK clock period, you must source it in Tessent Shell before pattern generation to enable the FastIJTAG feature.

Reporting Pattern Processing Details

The following example shows how to source the side file to enable FastIJTAG with the correct SDC values to maintain consistency. It also shows how to report the pattern processing details of the process_patterns_specification command by using a callback.

_____**Note**_____
Currently, process_patterns_specification only supports TCK ratios that are a power of 2.

Example 9-4. Sourcing a Side File and Using a Callback

```
# After loading and elaborating the design in the "patterns -ijtag"
# context, source the side file to enable the FastIJTAG feature during
# pattern generation.
SETUP> source ../data/enable_tessent_fastIJTAG.tcl

# Print pattern details like tester_period and TCK ratio
proc report_pattern_sets_post {args} {
  report_pattern_sets
}
register_callback process_patterns_specification.pre_write report_pattern_sets_post

# Check lower physical blocks with chip level testbench
SETUP> set_defaults_value \
PatternsSpecification/SignOffOptions/simulate_instruments_in_lower_physical_instances on

# Create default patterns specification
SETUP> create_patterns_specification
SETUP> process_patterns_specification
```

There are several ways to get help when setting up and using Tessent software tools. Depending on your need, help is available from documentation, online command help, and Siemens EDA Support.

The Tessent Documentation System

From the Tessent 2024.1 release, Siemens provides an improved method to access your Siemens Digital Industries Software product documentation. The default option serves your product documentation from Support Center, giving you immediate access to the latest release-specific documentation, and an enhanced search of HTML and PDF files. This new method also eliminates the requirement to install product documentation as part of the local software installation.

For easy access, you can now view Support Center documentation using a documentation proxy on your network. This documentation proxy removes the need for your users to have a Support Center account or to log into Support Center to view documentation.

We understand that some customers use our products on restricted networks without internet access. You have the option to download the documentation and set up a Siemens Documentation Server to view the documentation package on your local network.

See "Configuring Documentation Access" in the *Managing Tessent Software* manual for instructions to set up the documentation proxy or the documentation server.

Global Customer Support and Success

A support contract with Siemens EDA is a valuable investment in your organization's success. With a support contract, you have 24/7 access to the comprehensive and personalized Support Center portal.

Support Center features an extensive knowledge base to quickly troubleshoot issues by product and version. You can also download the latest releases, access the most up-to-date documentation, and submit a support case through a streamlined process.

`https://support.sw.siemens.com`

If your site is under a current support contract, but you do not have a Support Center login, register here:

https://support.sw.siemens.com/register

Third-Party Information

Details on open source and third-party software that may be included with this product are available in the *<your_software_installation_location>/legal* directory.