



Tessent® IJTAG User's Manual

Software Version 2017.1

March 2017

Document Revision 4

© 2012-2017 Mentor Graphics Corporation
All rights reserved.

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

Note - Viewing PDF files within a web browser causes some links not to function (see [MG595892](#)).
Use HTML for full navigation.

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

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

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

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

U.S. GOVERNMENT LICENSE RIGHTS: The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: www.mentor.com/trademarks.

The registered trademark Linux[®] is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

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

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

Revision History

Revision	Changes	Status/ Date
4	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the <i>Tessent Release Notes</i> for this product are reflected in this document. Approved by Ron Press.	Released Mar 2017
3	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the <i>Tessent Release Notes</i> for this product are reflected in this document. Approved by Ron Press.	Released Dec 2016
2	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the <i>Tessent Release Notes</i> for this product are reflected in this document. Approved by Ron Press.	Released Sep 2016
1	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the <i>Tessent Release Notes</i> for this product are reflected in this document. Approved by Ron Press.	Released Jun 2016

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 Mentor Graphics Technical Publication's source. For specific topic authors, contact Mentor Graphics Technical Publication department.

Revision History: Released documents maintain a revision history of up to four revisions. For earlier revision history, refer to earlier releases of documentation which are available on <http://support.mentor.com>.

Table of Contents

Revision History

Chapter 1

Introduction to Tessent IJTAG	13
Tessent IJTAG Flow	14
ICL and PDL Limitations	16
License Usage/Requirements	17

Chapter 2

ICL and PDL Modeling	19
ICL Instrument Description	19
How to Build an ICL Netlist	20
How to Model Global Reset, Local Reset and Embedded TAPs	23
How to Define an iProc	32
How to Call an iProc	32

Chapter 3

A Typical PDL Retargeting Flow	35
The Basic PDL Retargeting Flow	36
Invoke Tessent Shell	37
Set the IJTAG Context and System Modes	38
Read ICL Files	38
Read PDL Files	39
Set the Retargeting Level	39
Define Clocks and Timing	41
Test Clock	41
Synchronous System Clock	42
Asynchronous System Clock	43
Design Rule Checks	44
Create Pattern Sets	44
Write PDL, Pattern, and Test Bench Files	47
Exit the Tool	48
Optional Elements of a PDL Retargeting Flow	49
Test Setup and Test End Procedures	49
How to Define and Use Clocks Outside ICL	50
How to Constrain Inputs	50
Report Generation	51
IJTAG Introspection	52
How to Run iCalls in Parallel	56
PDL Specialties and Exceptions	57
iMerge Conflict Reporting	57
PDL Retargeting Commands	63

Introspection and Reporting Commands	65
Chapter 4	
ICL Extraction	69
ICL Extraction Flow	71
Required Inputs for ICL Extraction	72
Optional Inputs for ICL Extraction	72
Performing ICL Extraction	72
Top-Down and Bottom-Up ICL Extraction Flows	75
Top-Down ICL Extraction Flow	76
Bottom-Up ICL Extraction Flow	77
ICL Extraction Design Rule Checks	78
Debugging DRC Violations with DFTVisualizer	79
How to Influence the ICL Extraction Process	80
How to Influence ICL Extraction through Commands	80
How to Influence ICL Extraction Through ICL Module Attributes	84
ICL Network Extraction of Parameterized Modules	87
ICL Extraction Commands	88
Chapter 5	
IJTAG Network Insertion	91
The IJTAG Network Insertion Flow	92
IJTAG Network Insertion Example	93
Modification of the IJTAG Network Insertion Flow	94
How to Edit or Modify a DftSpecification	96
DftSpecification Examples	98
Examples	98
Chapter 6	
IJTAG and ATPG in Tessent Shell	109
IJTAG ATPG Flow Overview	109
IJTAG Features of ATPG in Tessent Shell	111
EDT IP Setup for IJTAG Integration	111
How to Set Up Embedded Instruments Through Test Procedures	113
How to Set Up Embedded Instruments Through the Dofile	114
Implicit and Explicit iReset Commands	115
A Detailed IJTAG ATPG Flow	117
Chapter 7	
IJTAG Examples	121
ICL Modeling versus Verilog Modeling	122
ICL Namespaces	123
PDL Namespaces	125
How to Define Default Values in ICL	125
Attributes of the ICL Extraction Flow	127
Scan Chain Integrity Test in Tessent IJTAG	128
How to Define Auto-Return Values in ICL	129
How to Model Addressable Registers in ICL	131

Table of Contents

Chapter 8

Verification and Debug of IJTAG Instruments and Networks 135

General Guidelines for Debugging Simulation Results.....	136
Creating ICL Verification Patterns	136
Using ICL Verification Patterns	136
ICL Verification Patterns Summary	140
Displaying the Comparison Failure Counter	140
Conclusion	141

Appendix A

Getting Help 143

The Tessent Documentation System	143
Mentor Graphics Support.....	144

Third-Party Information

End-User License Agreement

List of Figures

Figure 1-1. IJTAG High-Level Architecture	13
Figure 1-2. Tessent IJTAG Flow	14
Figure 2-1. Example ICL Description	21
Figure 2-2. Association Between ResetPorts and Registers	27
Figure 2-3. Hierarchical Reset	28
Figure 3-1. PDL Retargeting Flow	36
Figure 3-2. First Pattern Set	46
Figure 3-3. First and Second Pattern Sets	46
Figure 3-4. Pattern Set Report with Two Patterns	52
Figure 3-5. iMerge Flow Graph	62
Figure 4-1. Generic ICL Extraction Flow	71
Figure 4-2. ICL Rule Violation Debug in DFTVisualizer	79
Figure 4-3. Logical Connection Example	82
Figure 5-1. IJTAG Network Insertion Flow	92
Figure 5-2. Configuration Data Window for DftSpecification	96
Figure 7-1. Gate-Level Verilog Module Example	122
Figure 7-2. Schematic View of an Indirect Addressing Scheme	132
Figure 7-3. ICL Description of an Indirect Addressing Scheme	132
Figure 8-1. Example Design	138

List of Tables

Table 1-1. Tessent IJTAG Flow	15
Table 3-1. Conflict Report Terminology	58
Table 3-2. PDL Retargeting Command Summary	63
Table 3-3. ICL Introspection and Reporting Command Summary	65
Table 4-1. Values for ICL Extraction Attribute connection_rule_option	85
Table 4-2. ICL Extraction Command Summary	88
Table 5-1. Modifications to the IJTAG Network Insertion Flow	94
Table 5-2. IJTAG Network Insertion Command Summary	97
Table 6-1. EDT Configuration Keywords and Values	112
Table 8-1. Scan Path Configurations	138

Chapter 1

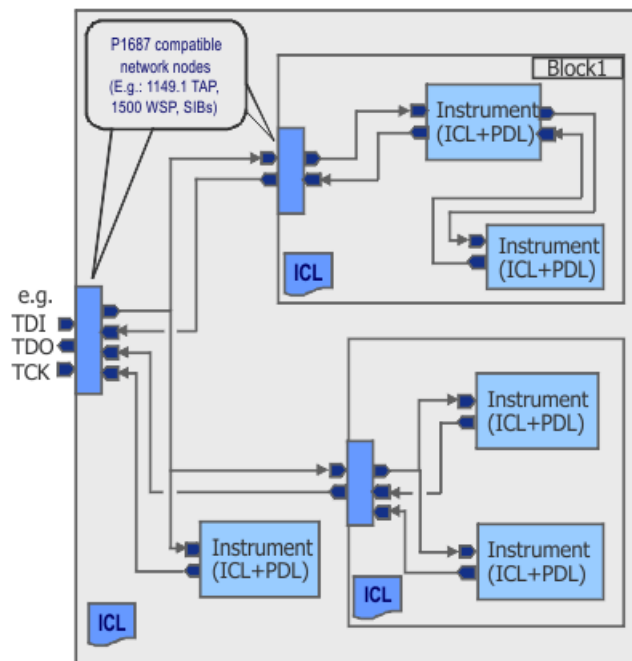
Introduction to Tessent IJTAG

Tessent IJTAG is Mentor Graphics 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 high-level IJTAG implementation.

Figure 1-1. IJTAG High-Level Architecture



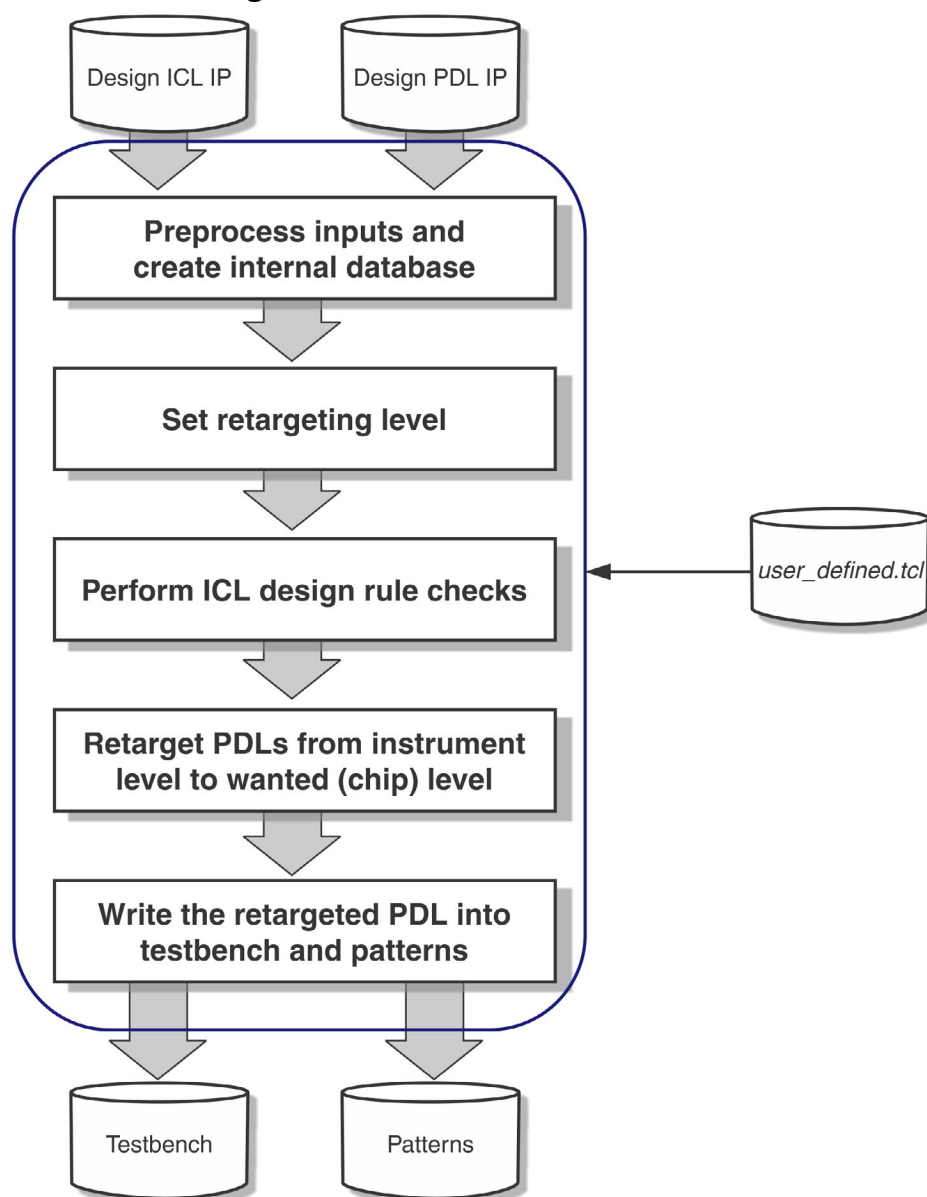
Tessent IJTAG Flow	14
ICL and PDL Limitations	16
License Usage/Requirements	17

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	<p>Tessent Shell is the platform you use to perform the Tessent IJTAG operations.</p> <p>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.</p> <p>Refer to the Tessent Shell User's Manual and Tessent Shell Reference Manual for complete information.</p>
IJTAG Flow Inputs	<p>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.</p> <p>The tool loads this information into the internal ICL database. See "Read ICL Files" and "Read PDL Files."</p>
Pattern Retargeting Level	<p>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.</p> <p>Before proceeding, you must specify the ICL hierarchy level to which the PDL commands should be retargeted. See "Set the Retargeting Level."</p>
Design Rule Checking	<p>Tessent IJTAG automatically performs ICL design rule checking on the set ICL hierarchy level, down to the instrument level.</p> <p>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".</p>
ICL Introspection	<p>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.</p> <p>For more information, see "IJTAG Introspection."</p>
PDL Command Retargeting	<p>Tessent IJTAG performs automatic PDL retargeting based on the ICL hierarchy level you specified.</p> <p>The tool can retarget the PDL commands from instrument level up to chip level. See "A Typical PDL Retargeting Flow."</p>

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

Flow Step	Description
Patterns and Test Bench	The final phase of the Tessent IJTAG flow is writing out the retargeted PDL, test bench, and patterns. For more information, see “ Write PDL, Pattern, and Test Bench Files. ”
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

Some of the ICL features described in IEEE 1687-2014 are not yet supported by Tessent IJTAG.

The unsupported ICL features are listed below:

- ICL namespaces:
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 are not supported, the ReadCallBack and WriteCallBack properties cannot be used within a DataRegister definition.
- The AccessTogether property of the alias construct.
- The DefaultLoadValue for data registers. DefaultLoadValue for ScanRegister is supported however.
- 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 quotes. If the next character is something else, the backslash is interpreted as an ordinary character of the string.

PDL features described in IEEE 1687-2014 but not yet supported by Tessent IJTAG:

- 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 iScan command is missing.
- The iState command is missing.

- The -comment and -status options of the iNote command are not supported.
- The -together option for the iApply command is missing.
- The -broadcast option of the iOverrideScanInterface command is not supported
- iOverrideScanInterface only works for scan interfaces of the top module.

License Usage/Requirements

An IJTAG license is required for stand-alone verification of non-Tessent or non-Mentor Instruments irrespective of where the Instruments are to be used. With the IJTAG license, the ICL and PDL created for the Instrument can be validated by using commands such as iWrite, iRead, iApply etc. With the use of these ICL and PDLs you can generate test benches and simulate these test benches with the Verilog model of the Instrument.

Any license other than FastScan or TessentScan (formerly called DFTAdvisor) can be used to connect non-Tessent/non-Mentor instruments to an IJTAG network as long as there is at least one Tessent Instrument in the design or at least one Tessent Instrument is inserted along with the non-Tessent/non-Mentor Instrument. Similarly, any license other than FastScan or TessentScan can be used 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. To read from a non-Tessent Instrument requires an IJTAG license (that is, to use iRead from any non-Tessent Instrument requires an IJTAG license).

Setting up or using any Tessent IP/Instrument via the IJTAG network does not require an IJTAG license. The license that creates the Tessent IP/Instrument can be used to set it up via the IJTAG network.

Chapter 2

ICL and PDL Modeling

This chapter provides some insight into ICL and PDL without rephrasing the entire IEEE 1687 document, but provides enough information so you can understand the remainder of the user guide.

Additionally, the “[IJTAG Examples](#)” chapter later in this document provides more examples. You can also download several complete example test cases from SupportNet. 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	19
How to Build an ICL Netlist	20
How to Model Global Reset, Local Reset and Embedded TAPs.....	23
How to Define an iProc	32
How to Call an iProc	32

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

allow 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 allow for 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.

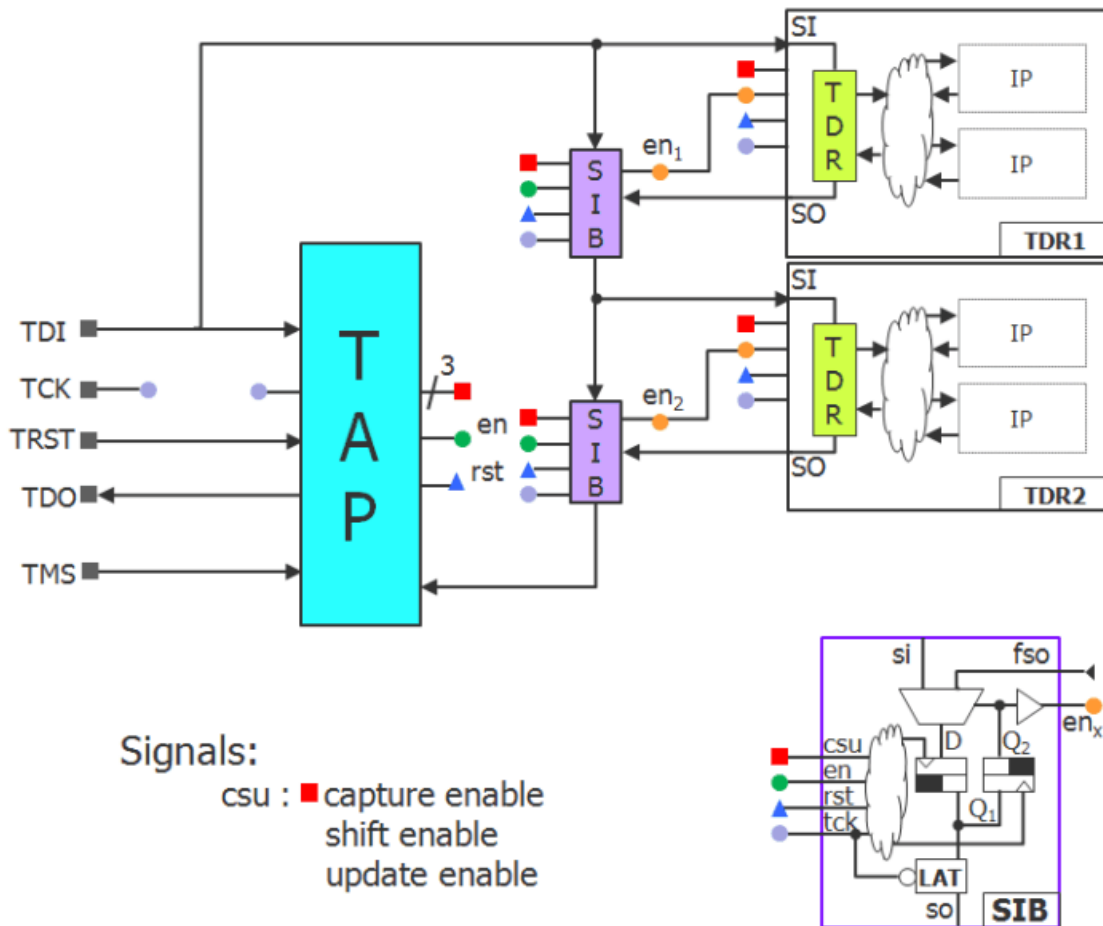
After the listing of the ports, this example shows an 8-bit scan register named *R*. The keywords in the attribute section of the scan register declaration are the ICL code between the brackets (`{ }`). These keywords provide further information about the scan register. The example also declares that 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



The depicted logic shall only serve as an example.
P1687 does not require any particular implementation as long as the IO protocol is served correctly.

The task now is to instantiate each instrument, connect them, and create a top-level ICL description, you name *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 top ICL module is shown in the following example:

```
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
    }

}
```

Note that you must define the port connections of an instrument and ensure that the connections of each input port are declared. Consequently, the connection list of each instrument instantiation only lists input ports and the ports driving them. For example, the input port *si* of the instance *MyTdr2* of the instrument *tdr2* is driven by the port *so* of instance *MySib1* as shown here:

```
Instance MyTdr2 Of tdr2 { InputPort si      =
MySib1.so ;
```

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 in order 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 “reset” state or in “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 will 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 holds, 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 at all.

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 holds, 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 at all.

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 the “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 the “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 the “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 the “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.

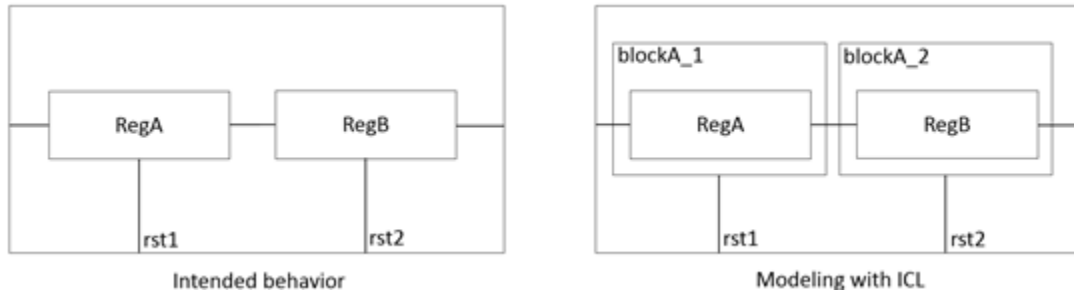
The following rules apply to the implied reset connectivity of the registers with ResetValue specifications (registers without ResetValue specification are not affected by any reset activity):

- 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 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 holds, 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 the “Global Reset” as if there was no suitable reset signal source at all.

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. This is shown in Figure 2-2 below. In order to achieve the unique association between the ScanRegister RegA and the ResetPort rst1 and the unique association between ScanRegister RegB and the ResetPort rst2, the modules blockA_1 and blockA_2 must be introduced.

Figure 2-2. Association Between ResetPorts and Registers



The Role of the DataMux

In order 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, or the isolation of TAP controllers including the possibility to park them in “reset” state or in “idle” state, it is necessary to intercept reset/trst/tms signals. 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, in a manner of speaking, a “reset mux”, “trst mux” or “tms mux” resp. All the other inputs as well as the select inputs must be ordinary data signals, that is, they must be driven by the parallel outputs of a register, by DataOutPorts or DataInPorts, by other DataMuxes or by combinational logic modelled by means of 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 way that it selects the data signal with the active reset value. The Local Reset rests until the DataMux is configured again in such way that it presents the ordinary reset signal or an inactive reset value. (Note: The possibility to implement a self-clearing reset, as described in the IEEE 1687-2014 standard, is currently not supported.) While the downstream registers are subject to the Local Reset, it is not possible to make them part of the active scan path.

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 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.

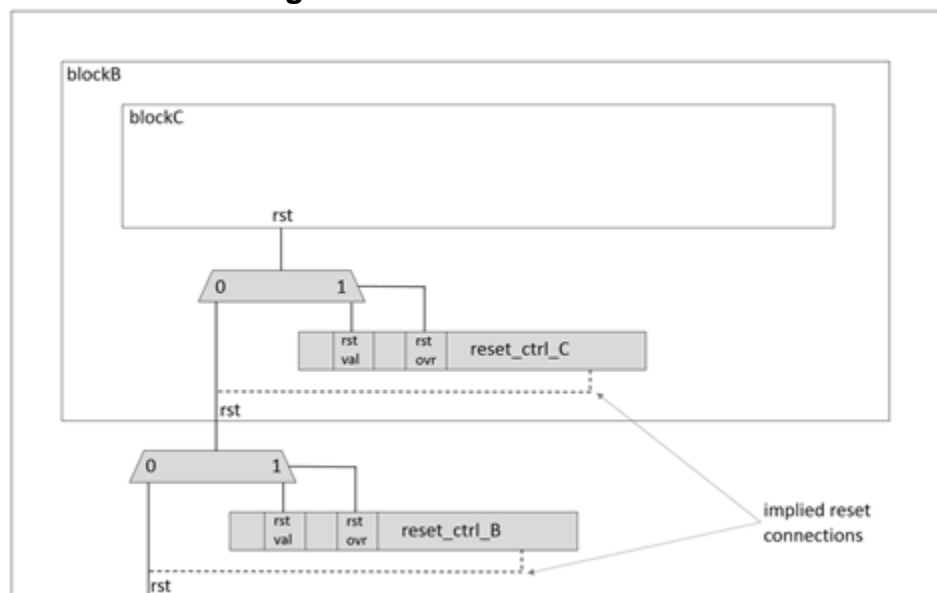
A DataMux can also be used to intercept a tms signal. If a DataMux drives the value '1' to the TMSPort of a TAP controller state machine (that is, an ICL module with a ToIRSelectPort), the TAP controller is held in reset state. If a DataMux drives the value '0' to the TMSPort of a TAP controller state machine, the TAP controller is held in 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 idle state or in reset state. In the latter case, the TAP controller will not only be isolated, its registers will also be 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 [Figure 2-3](#) demonstrates this concept.

Figure 2-3. Hierarchical Reset



The register `reset_ctrl_B` can be used to trigger a Local Reset in blockB, the register `reset_ctrl_C` can be used to trigger a Local Reset in blockC. If the `ResetValue` specifications for the register bit “`rst_ovr`” of the two registers are set to 0, then the Global Reset will automatically trigger the reset in blockB, and the reset of blockB will automatically trigger the reset of blockC. This is because the Global Reset resets the register `reset_ctrl_B`. 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.

In order 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.

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 toplevel `ResetPorts` and `TRSTPorts`. The effect of this reset is simulated in the retargeter considering all the explicit and implied reset connections, while 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 toplevel `ResetPorts` and `TMSPorts`, in order to enforce the transition of the TAP state machine/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, while 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` which 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 applied to the embedded TAP controllers without explicitly connected `TMSPorts`, too.

An ICL module, which 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 will arrive in the state “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, which is connected like this, does not react on the synchronous reset.

A Local Reset is simulated nearly 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. So the Local Reset can only happen at the end of an iApply. 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 toplevel TRSTPort is constrained to the inactive value (CT1 constraint)
- If the toplevel TRSTPort is forced to the inactive value by means of [iForcePort](#)
- If one of the TAP controller 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:

- **Unsupported features:** Self-clearing Local Reset as described in chapter 5.7 of the IEEE 1687-2014 standard is not supported, yet. The standard also describes the possibility to shift through registers while the update stage is subject to a Local Reset. This is not supported, either. Registers, which are associated with a reset signal that is constantly held active, are treated as completely inaccessible. The active scan path cannot contain such registers. Therefore the IJTAG network must be designed with care, such that an active Local Reset does not cause a deadlock.
- **Dedicated control registers for Local Reset:** In order to prevent the retargeter from accidentally triggering a Local Reset or from accidentally parking a TAP controller, the register bits controlling the Local Reset and the TAP isolation must be kept separate from the register bits that control the scan path configuration. If there were register bits controlling both, Local Reset as well as scan path configuration, the attempt to select a particular scan path could result in an undesirable Local Reset or in the disabling of a

TAP controller. The Design Rule [ICL131](#) checks that the register bits do not serve several conflicting purposes.

- **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. In order to ensure that the ordinary iRead/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, the user must separate the ordinary iRead and iWrite commands from those which are meant to trigger/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. 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 as well as the state machine of the other TAPs must be running. While the other TAPs are currently in the Update-DR state, the embedded TAP, which has just been woken up, is still in 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 reset state and the TAP controllers get out of sync.
- **Prevention of circular reset paths:** Tessent Shell does not allow circular reset logic, if this logic can be used to trigger a Local Reset. In other words, a Local Reset cannot set back the register that has triggered just this Local Reset. Circular paths in the reset logic are allowed only if the logic serves exclusively the purpose of *suppressing* the reset of

the registers. Therefore, it is possible to design the reset circuitry such that a register controls a reset mux to prevent its own subsequent reset. Once a register is configured such that it prevents its own reset, another scan load is required to load the appropriate control value into the register, such that the reset mux allows the reset signal to pass through to the register, again.

How to Define an iProc

Most of your ICL instruments will 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 anyway 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*, which is instantiated in an ICL module instance of *Core3*. The full hierarchical ICL instance path is therefore *Core3.Block1.MyTdr*. Since 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.

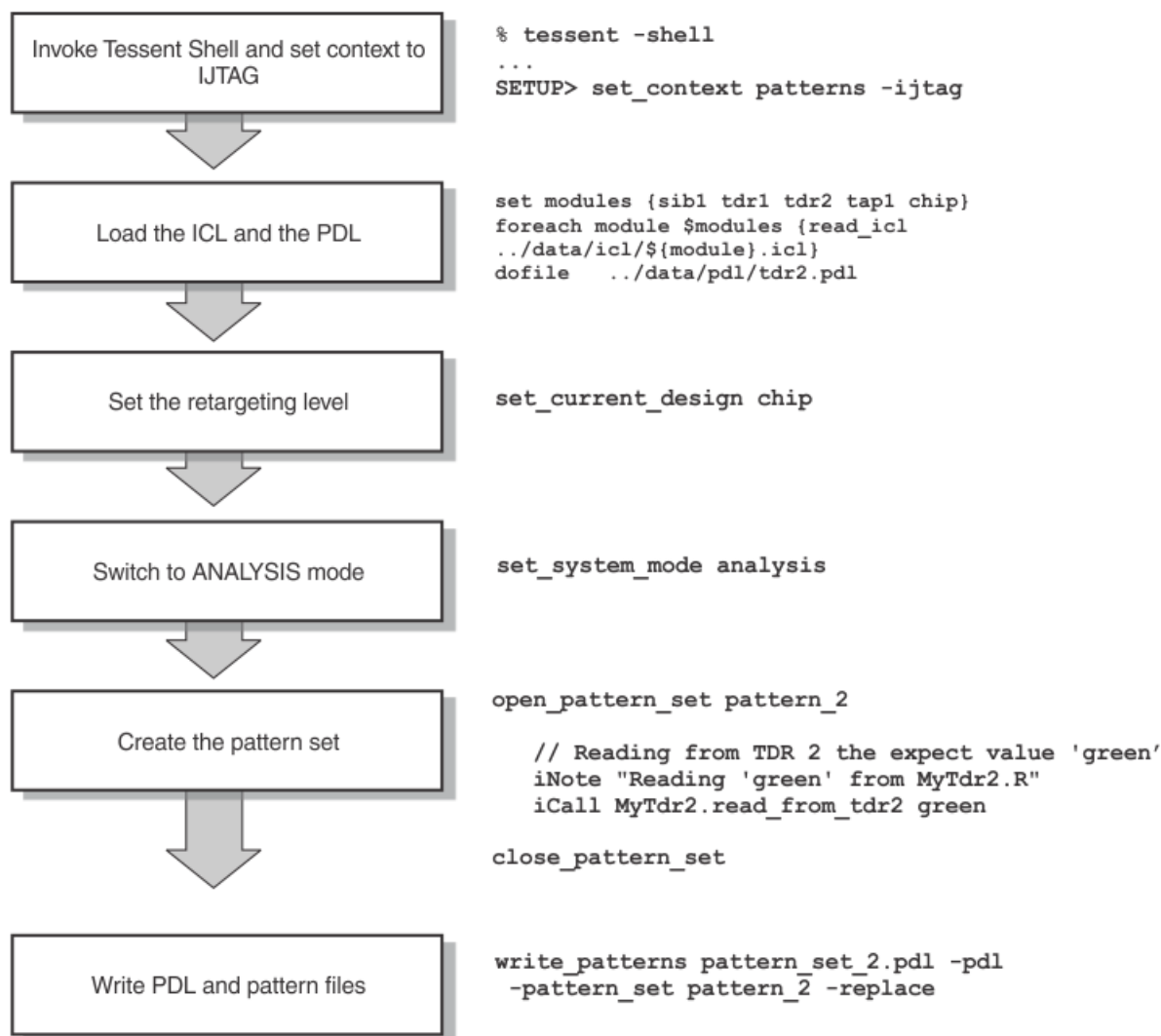
The Basic PDL Retargeting Flow	36
Invoke Tessent Shell	37
Set the IJTAG Context and System Modes	38
Read ICL Files	38
Read PDL Files	39
Set the Retargeting Level	39
Define Clocks and Timing	41
Design Rule Checks	44
Create Pattern Sets	44
Write PDL, Pattern, and Test Bench Files	47
Exit the Tool	48
Optional Elements of a PDL Retargeting Flow	49
Test Setup and Test End Procedures	49
How to Define and Use Clocks Outside ICL	50
How to Constrain Inputs	50
Report Generation	51
IJTAG Introspection	52
How to Run iCalls in Parallel	56
PDL Specialties and Exceptions	57
iMerge Conflict Reporting	57
PDL Retargeting Commands	63
Introspection and Reporting Commands	65

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 follow the flow and discuss each step separately. The sections describe only 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 which are outside of the ICL description, you also must read at least the top level Verilog description of your netlist. This allows 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 test bench. These optional steps are discussed in the section “[Optional Elements of a PDL Retargeting Flow](#)”. For these reasons, Mentor Graphics recommends reading the top netlist module as well, as shown in the section “[Set the Retargeting Level](#).”

If you do not have a top-level ICL file for your design, Tessent ITJAG 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	37
Set the IJTAG Context and System Modes.....	38
Read ICL Files	38
Read PDL Files.....	39
Set the Retargeting Level	39
Define Clocks and Timing.....	41
Test Clock.....	41
Synchronous System Clock	42
Asynchronous System Clock	43
Design Rule Checks	44
Create Pattern Sets	44
Write PDL, Pattern, and Test Bench Files	47
Exit the Tool	48

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.

You 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

The execution of 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 will automatically determine 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 will 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 since they describe actions that the tool must execute, 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 executes 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 will be registered and can be called later. Other commands are executed immediately, just like any other dofile. Therefore, the second type of PDL file is not allowed in setup mode. In fact, it is only allowed to be executed inside of an open pattern set.

You should 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 ultimately turns out to be error prone in its usage. Encapsulating all PDL reading and writing commands inside of iProcs allows the PDL retargeting later in the flow to use only iCalls to meaningful instance and iProc names.

PDL files of the first type may be called in both, setup and analysis mode, as well as 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 topmost module and if it is your intention to retarget the PDL to this level. Executing 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, will be flagged as a design rule violation at that point.

Usually, the current design is set to the top level ICL module. At this point, Mentor Graphics recommends to also load the top most Verilog module, at least as a black box. This allows 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 test bench out of the ICL flow will still be 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 will not be part of your ICL test bench. Loading the netlist into Tessent IJTAG makes all ports known to the tool. It will then automatically add all ports to the ICL test bench.

Therefore, Mentor Graphics 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 test bench computed from the retargeted PDL, you need to 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	41
Synchronous System Clock	42
Asynchronous System Clock	43

Test Clock

In the simplest PDL retargeting flow, clocks and timeplates do not need to be declared to the tool. The ICL test clock is known to the tool through the ICL port function 'TCKPort' in the top most ICL module.

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 of PI, measure PO, pulse of 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 declared by you)
- The test clock period is 100ns
- The relative timing of the ports is force of PI, pulse of 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 wish 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. Since 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
```

Note that you do not need to declare the -tck_ratio, since it defaults to 1, which means the tck period is equal to the tester period. You will need to modify the -tck_ratio option only when you have synchronous system clocks as well, as explained in the section “[Synchronous System Clock](#)”.

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, will have an off state of 0, whereas the other, sclk1, will have an off state of 1.

System clocks require the [add_clocks](#) command. You can then use options to [open_pattern_set](#) to define the timing of the test clock versus the system clocks. You have to set the tester period

to the speed of the system clock, then use the `tck` ratio to time the test clock in relationship 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
```

Since the asynchronous clocks are independent from the tester period, you can default back to use the `-tester_period` option with the default `-tck_ratio` of 1 to define the period of the test clock.

As a second example, assume that only `sclk0` is a 30ns asynchronous clock. Since `sclk1` is a synchronous clock with respect to the tester period (50ns again), the example would now look 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 exact 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 of the flow. Only synchronous clocks in relationship to the test clock period need to be considered when using the options to [open_pattern_set](#).

Consult 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, etc ... >...

close_pattern_set [options]

The name of the pattern set is mandatory and must be unique among all used pattern sets. This name will be 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, since Tessent IJTAG executes 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.

Since an iReset is executed 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 wish 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 execute 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 each single 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
iCallmytest1
close_pattern_set
report_pattern_sets
```

This produces the following report:

Figure 3-2. First Pattern Set

// name	timeplate	tester period	tck ratio	tester cycles	initial iReset	active scan interfaces	network end state	TAP start state	TAP end state	saved
// 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 will get the following report listing both pattern sets:

Figure 3-3. First and Second Pattern Sets

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

The eleven column headings in the report are described as follows:

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, it defaults to 100ns.

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. Defaults to 1.

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 means of 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

Write PDL, Pattern, and Test Bench 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 will be 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 test bench 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 can now generate SVF even if a TAP interface was not positively identified in the current design. In such 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 about all options of the [write_patterns](#) command.

Exit the Tool

Use the Tessent Shell exit command to exit the tool.

exit

You can optionally specify the -force switch, which 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	49
How to Define and Use Clocks Outside ICL	50
How to Constrain Inputs	50
Report Generation	51
IJTAG Introspection	52

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 will be considered in Tessent IJTAG as well.

Since 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, since 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 execution are regarded as input constraints. This behavior is equivalent to the Tessent ATPG tools. In order 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 prior to 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, since 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 which 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, which is needed 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.

In order to define either aspect of a non-ICL clock, you first have to load at least a 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. By means of the `add_clocks` command you 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 will be 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 will be used when processing the procedures as part of writing the patterns (other than PDL) to disk. These clocks will not be pulsed during any IJTAG operation.

How to Constrain Inputs

It is possible that the die must be statically configured outside of the ICL and PDL to be 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 constraint 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 a 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. By means of the `add_input_constraints` command you 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 `TRSTPort` with a `CT1` constraint. This allows 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 will execute 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 will detect constrained clock sources, including constrained differential or inverse clock sources.

In the following example, the tool traces `iClock` to the constrained port `ClkA` and to 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 and/or the transcript file.

With Tessent Shell, Mentor Graphics 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 as well as their extracted sources and cumulative freqMultiplier and freqDivider values. You can only specify this when there is 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 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 command. 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 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 in Tessent Shell](#)” in the *Tessent Shell User’s Manual*. There you will 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 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 executed 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 executed 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 for 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 executed 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, which is applied simultaneously with the first one, is the function of the ICL port. The `get_icl_ports` command together with these two filters compute and return 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 called 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;  
  [...]
```

How to Run iCalls in Parallel

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

The PDL command [iMerge](#) instructs the PDL retargeter to try to execute 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. In fact, the PDL retargeter might identify that some of the PDL commands cannot be executed in parallel and chooses to serialize them instead. Overall, the order of the parallel execution 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 which is allowed in between iMerge -begin and iMerge -end.

[iTake](#) can be used inside of an iProc to reserve a "resource" (i.e. a port, a register or an instance) for exclusive use with this iProc. No other of the 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, i.e. 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 means of 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	57
iMerge Conflict Reporting	57

PDL Specialties and Exceptions

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

The following situations will 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, which 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 3-1 defines the terminology used in the description section of the conflict report.

Table 3-1. 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 during 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 3-1. 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

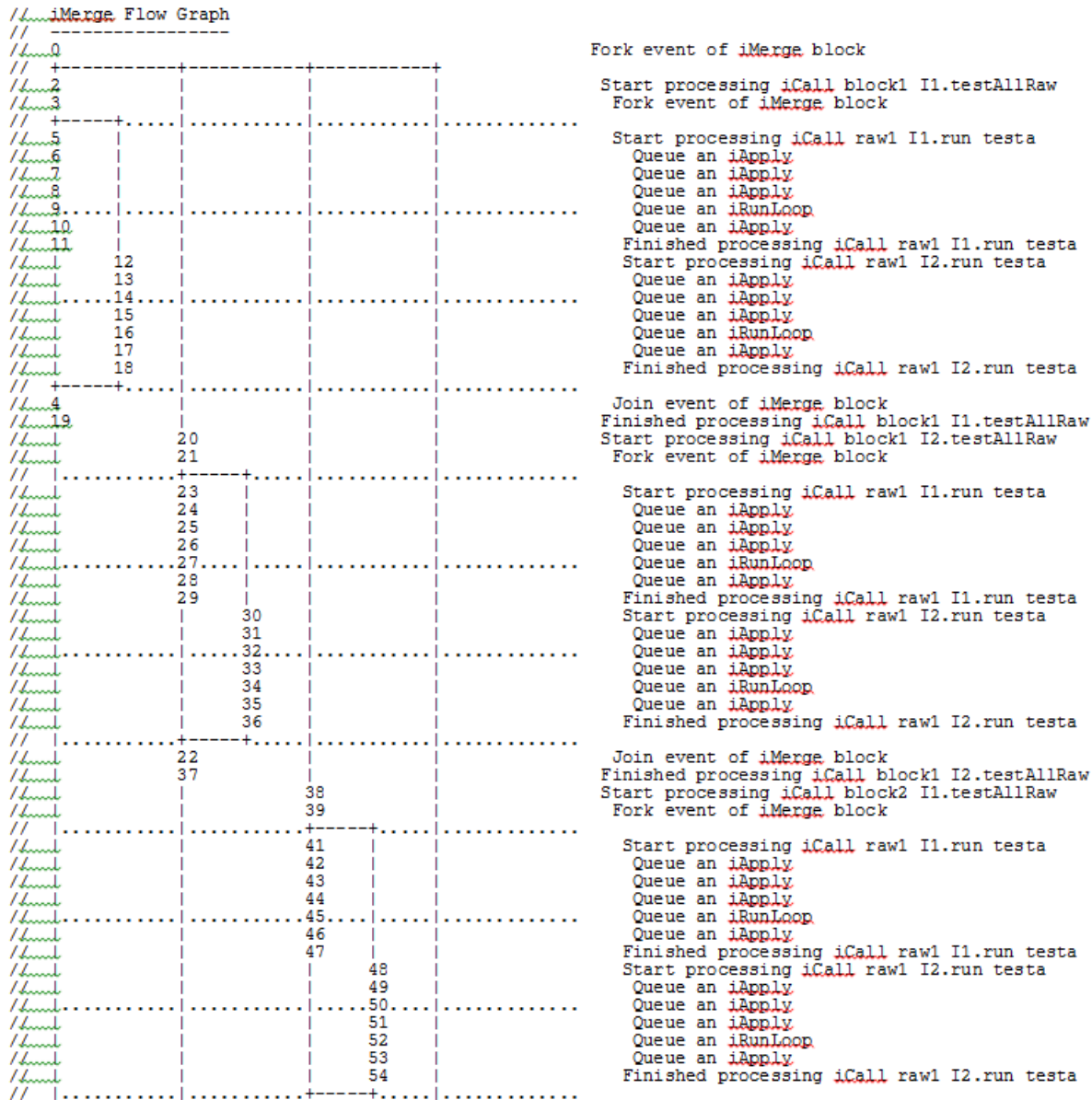
write_patterns pat1.stil -stil -repl
```

The resulting conflict report appears 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 3-5](#) 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 which 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 which in turn is also called from within iCall block2_I1.testAllRaw.

Figure 3-5. iMerge Flow Graph



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 be on the safe side.

```

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 brief summary of the PDL retargeting commands available in Tessent Shell.

Table 3-2. 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 <i>effective_icl_instance_path</i> .

Table 3-2. 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, which may be executed 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 <i>effective_icl_instance_path</i> 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 which subsequent iProc commands refer to and optionally its PDL name space.
iRead	Adds a read operation to the command queue that will be solved by the next iApply command.
iRelease	Releases a resource previously taken by means of 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 will be 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 3-2. 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 execution of this command.
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 88.

All Tessent Shell reporting commands start with “report_”, whereas all introspection commands returning information about objects start with “get_”. Besides these, there are commands starting with “delete_” for removing objects from memory of the tool.

Table 3-3. 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.
get_icl_fanins	Returns a collection of all requested objects found in the fanin of the specified pin or port objects.

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

Command	Description
get_icl_fanouts	Returns a collection of all requested objects found in the fanout of the specified ICL pin, or port objects.
get_icl_instances	Returns a collection of all ICL instances instantiated relative to the current design that match the specified <i>*name_patterns</i> list.
get_icl_modules	Returns a collection of all ICL modules that match the specified <i>name_patterns</i> 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_pins	Returns a collection of all hierarchical ICL pins instantiated relative to the current design that match the specified <i>name_patterns</i> list.
get_icl_ports	Returns a collection of all ports on a given module that match the specified <i>name_patterns</i> 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 will be created in the new ICL top module by ICL extraction after the <i>add_icl_scan_interfaces</i> 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 will be created in the new ICL top module during ICL extraction.
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 <i>icl_port_or_pin_name</i> .
get_iproc_argument_default	Returns the default value for the specified <i>arg_name</i> of the specified <i>proc_name</i> attached to the ICL module specified by the last <i>iProcsForModule</i> 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 <i>iProcsForModule</i> command.

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

Command	Description
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.
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 Tessent Shell command [extract_icl](#) 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, then 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 the DFTVisualizer 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	71
Required Inputs for ICL Extraction	72
Optional Inputs for ICL Extraction	72
Performing ICL Extraction	72
Top-Down and Bottom-Up ICL Extraction Flows	75
Top-Down ICL Extraction Flow	76
Bottom-Up ICL Extraction Flow	77
ICL Extraction Design Rule Checks	78
Debugging DRC Violations with DFTVisualizer	79
How to Influence the ICL Extraction Process	80
How to Influence ICL Extraction through Commands	80
How to Influence ICL Extraction Through ICL Module Attributes	84
ICL Network Extraction of Parameterized Modules	87

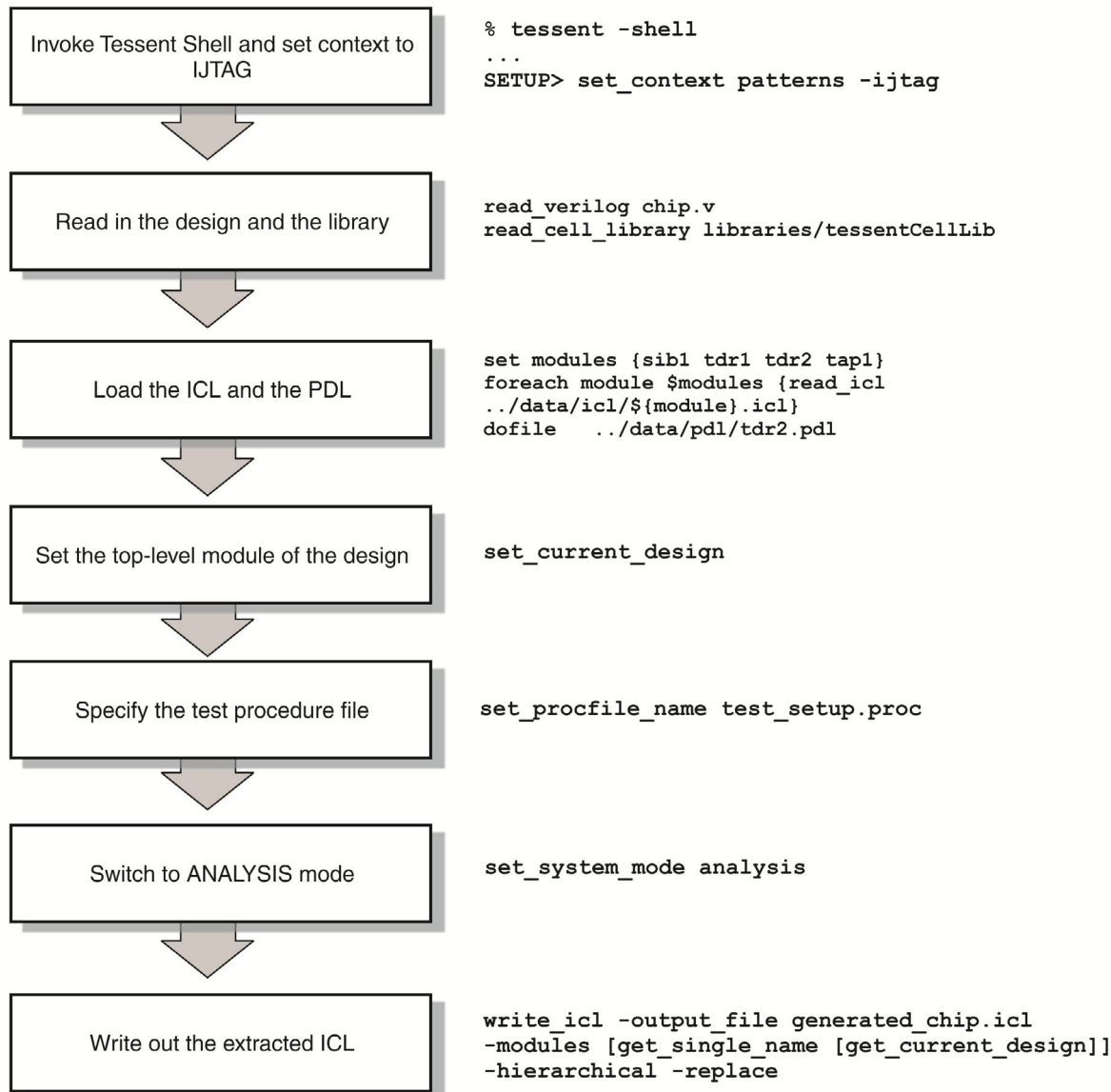
ICL Extraction Commands.....	88
-------------------------------------	-----------

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



Required Inputs for ICL Extraction	72
Optional Inputs for ICL Extraction.....	72

Required Inputs for ICL Extraction

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

The following inputs are required:

- **DesignData** — 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 which can influence the ICL extraction process.

- **Test Procedure File and/or Input Constraints** — The purpose is to set the design into a mode which 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 process will be executed. 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 you need to, 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 will match 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 afterwards will not be 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 execute 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 as well as special ICL extraction related design rule checks, which 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 and / 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 fanin 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** — The task this flow addresses is generating 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 set 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. [set_module_matching_options](#) is the command that allows 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, which will return “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 executes 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 executed 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 as well as on internal (cut) points. You can also declare logical connections between points in the design which 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. In addition, 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, i.e. the design setup during ICL extraction is a compatible subset of the design setup during PDL retargeting.

Tessent Shell enforces this automatically by issuing the [add_input_constraints](#) command for all current top-level constraints listed in the attributes. Non top-level constraints are not enforced. However a later version of Tessent IJTAG may check that the internal constraints set during ICL extraction are satisfied by the overall design setup.

Top-Down ICL Extraction Flow	76
Bottom-Up ICL Extraction Flow	77

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 Chapter 3, “[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 level 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 will be done only for a sub module which 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) which are executed at the beginning of the extraction process and after each loaded ICL module passes its own syntax and semantics check. 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 check.

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 execution of the “set_current_design” command. This includes verifying that all ports on the ICL module are present in the corresponding Verilog module. Note that the reverse is not necessary, since 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 to 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, that is, before the ICL extraction is executed in 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 -regex 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 double 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 in detail later in this chapter in the section “[How to Influence the ICL Extraction Process](#)”.

Debugging DRC Violations with DFTVisualizer 79

Debugging DRC Violations with DFTVisualizer

Some ICL extraction DRC violations can also be debugged using DFTVisualizer. This is especially helpful for the I2 connection tracing DRC.

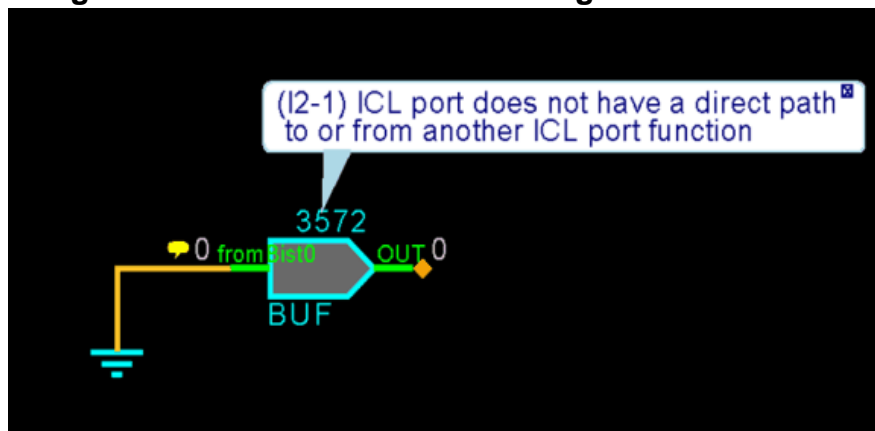
Prerequisites

DRC violations reported by Tessent IJTAG

Procedure

1. Open DFTVisualizer, using the [open_visualizer](#) command.
2. In the Design Browser window, which opens by default, select the **DRC Violations** tab. The DRC violations are listed by severity. Click the “+” to expand the listing and select the ICL extraction DRC violation you want to debug. Double clicking on the listed violation, or right clicking and selecting **Analyze DRC Violation** from the popup menu will display the violation and the associated instances. This is the same process as ATPG DRC debugging with DFTVisualizer. Alternatively, you can use the command “analyze_drc_violation.” at the tool prompt in the Console Window.
3. Once you analyze an ICL extraction DRC violation, you will notice additional call-out boxes, which will assist you in determining the cause of the violation, as illustrated in [Figure 4-2](#). In DFTVisualizer, you can use all applicable features, such as tracing the Verilog design netlist or looking at input constraint values and their propagation.

Figure 4-2. ICL Rule Violation Debug in DFTVisualizer



How to Influence the ICL Extraction Process

You can influence how ICL extraction is being executed 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 to the tool 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 will be translated into the ICL syntax of the generated ICL module.

How to Influence ICL Extraction through Commands 80

How to Influence ICL Extraction Through ICL Module Attributes 84

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 however, these name modifications follow specific patterns.

Typically, synthesis adds a prefix or postfix to the design module’s original name. For example, names are changed from “MyModule” to “MyModule_X1”, to “MyModule_X2”, etc. Using the Tessent Shell command “set_module_matching_options” you can tell the tool these module name mapping patterns specific 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 of at least one digit. Assume now, 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”, etc. Since 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 would overwrite 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". Make sure you use the [set_module_matching_options](#) command before you use [set_current_design](#), since part of setting the current design is creating the ICL to design 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, ports of different types, like DataInPort, DataOutPort, ScanInPort or ScanOutPort can be added.

If you have [add_clocks](#) -pulse_always defined on any port of your current design, it will be 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 commands [add_input_constraints](#) and [add_clocks](#) as shown in the following example:

**add_input_constraints -equivalent CLKP -invert CLKN
add_clocks 0 CLKP -period 50ns**

By default, the port with the offstate "0" will be used as the ordinary ClockPort in ICL ("representative port"), and the port with the offstate "1" will be used as the ClockPort with the "DifferentialInvOf" property ("associated port"). This default is overridden if the ports are connected to other ICL ClockPorts or internal clocks that unambiguously determine their roles in the differential group.

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 very 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:

**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 relate 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 at the designated target location, ignoring the actual design netlist structure. Similarly, during backward tracing the process continues at the designated source location when a target pin was reached. Note that the source and the target location may also be ports.

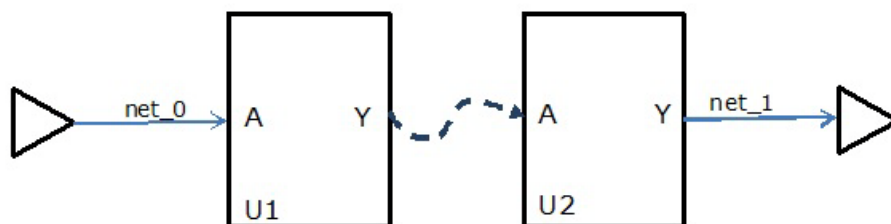
You would use logical connections to move forward with your IJTAG work for example for incompletely defined designs. Assume, for example, a case in which you have design module black boxes on the ICL extraction path. Using logical connections you can connect the ICL relevant pins of the black boxes, allowing 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 will use 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



Note that 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 will create 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 will be 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 has no influence over the other operations that are part of the ICL extraction process. In particular, any design simulations executed 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 will determine the port name and ICL port function for all module ports identified during the ICL extraction process. However, ICL extraction cannot identify which ports form a logical group, captured in an ICL ScanInterface. You have to provide this information using 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 65 and “[ICL Extraction Command Summary](#)” on page 88.

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, which will be created through ICL extraction. The created scan interface must follow all rules defined in the standard.

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. You 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 violation 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 will be 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/driving such ports can be unused for the purpose of ICL extraction tracing. In addition the “connection_rule_option” attribute 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 allows 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, since 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 will happen 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 simulation value of 0 or 1 in stable_after_setup simulation context

Table 4-1. Values for ICL Extraction Attribute `connection_rule_option` (cont.)

Attribute Value	Port Direction	Allowed Simulation Value (<code>stable_after_setup</code>)	Description
<code>allowed_tied_low</code>	input	0	The port can be tied to low or be driven by any logic with simulation value of 0 in <code>stable_after_setup</code> simulation context
<code>allowed_tied_high</code>	input	1	The port can be tied to high or be driven by any logic with simulation value of 1 in <code>stable_after_setup</code> simulation context
<code>allowed_no_destination</code>	output	0/1/Z/X	The port can be open or connected to any logic regardless of simulation values. No synthesis will happen in the output cone of this port
<code>must_connect_to_top_port</code>	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 allow a data-in port named “din” of the ICL module “block1” to be constrained to high. Note that such a constraint in the Verilog netlist would cause a tracing violation, since 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_l1 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 will allow 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(topleve_datain[17:10]), <...> );
```

ICL Network Extraction will recognize 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 will generate 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 will be 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 will become part of the generated ICL module.
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.

Table 4-2. ICL Extraction Command Summary (cont.)

Command	Description
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.
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 execution of this command.
set_design_sources	Specifies where the tool should look for the definition of undefined modules in the list of files specified by the read_icl command.
set_design_sources	Customizes the behavior of ICL extraction.

Table 4-2. ICL Extraction Command Summary (cont.)

Command	Description
set_icl_extraction_options	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_ical	Adds an iCall to the start of the test_end procedure.
set_test_setup_ical	Adds an iCall to the end of the test_setup procedure.
write_icl	Writes out ICL modules created and/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 and/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 can connect to a pre-existing TAP controller. If the IJTAG network needs to connect to pre-existing TAP controller then an ICL for the TAP controller needs to be provided.

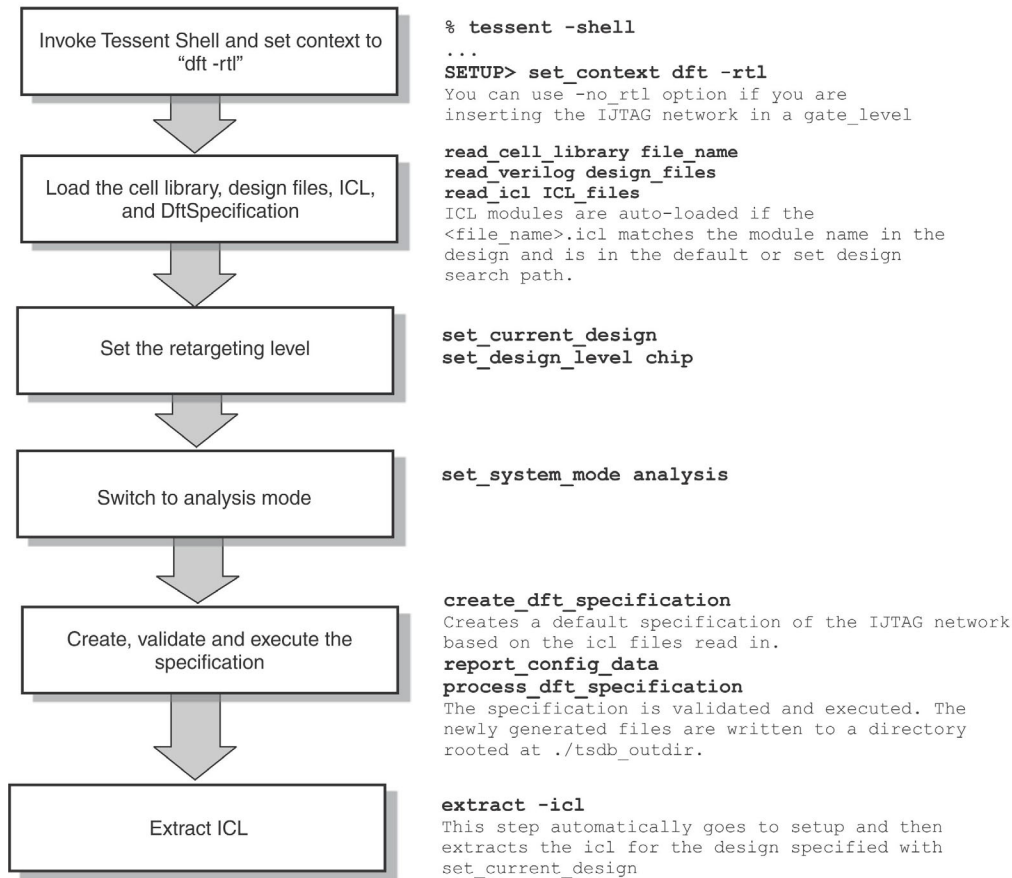
The IJTAG Network Insertion Flow	92
IJTAG Network Insertion Example	93
Modification of the IJTAG Network Insertion Flow	94
How to Edit or Modify a DftSpecification	96
DftSpecification Examples	98

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. Since 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 command `create_dft_specification` 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, as well as all modified design files. The IJTAG network itself is

automatically generated using `create_dft_specification`. 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 itself is not automatically generated since you may want to do further design editing. However, since all data resides in memory, you can perform the subsequent IJTAG Network Extraction step using the `extract_icl` command.

IJTAG Network Insertion Example.....	93
Modification of the IJTAG Network Insertion Flow	94

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 the verilog netlist which already has scan inserted and EDT IP inserted with the PLL module already present. For the PLL module, an ICL and PDL has 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 executing `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 DFTVisualizer to edit. The last step is `extract_icl` which provides the ICL for the level that was set using `set_current_design`.

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 <code>./tsdb_outdir</code> . 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 <code>"-no_insertion"</code> and <code>"-validate_only"</code> of the <code>process_dft_specification</code> command.
Transcript all design edits performed	The high level command process_dft_specification executes a series of Tessent Shell editing commands such as create_connections or create_instance . Usually, these commands are not transcribed, but they may come in handy when debugging. The command process_dft_specification provides you with the <code>-transcript_insertion_commands</code> option which adds all design editing steps performed during the execution of the DftSpecification to the transcript.

Table 5-1. Modifications to the IJTAG Network Insertion Flow (cont.)

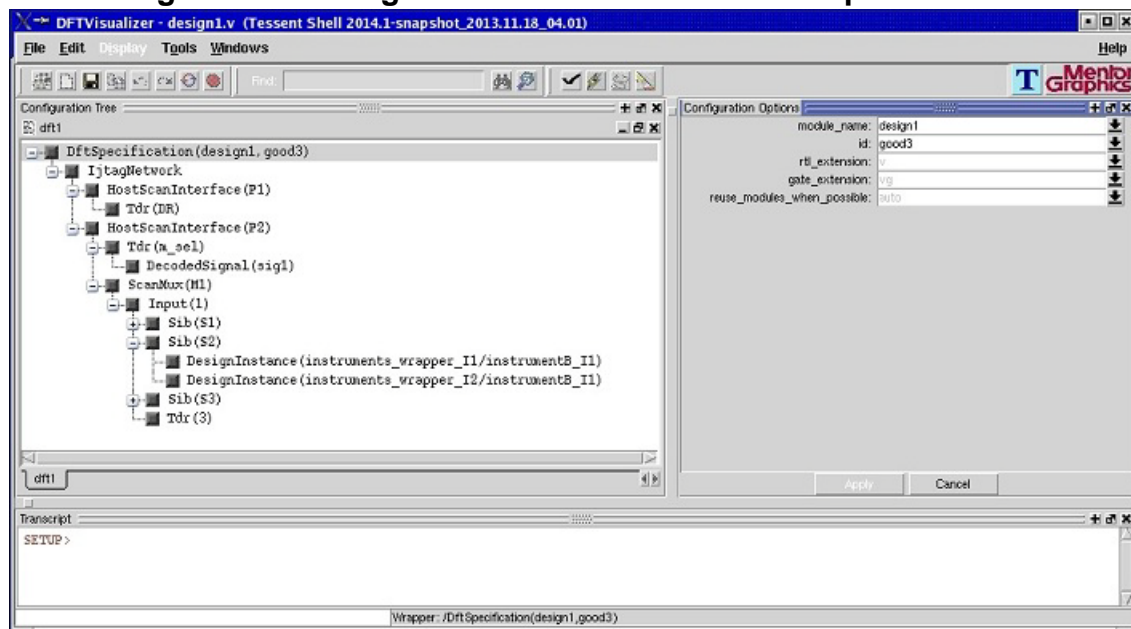
To...	Description
Execute a DftSpecification	You can have more than one DftSpecification loaded for the current design; they differ based on a user-specified identifier. For more information, see the DftSpecification wrapper description. Selecting one or the other DftSpecification is easily done through the "id" option of the process_dft_specification command.
Automatically execute 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 execute your design editing command before writing out the final, modified design file; this file then includes both the inserted IJTAG network as well as 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 executed.

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 DFTVisualizer Configuration Data window or an ASCII text editor.

The DFTVisualizer Configuration Data window provides you with a graphical interface that facilitates the creation and modification of specification elements as shown in [Figure 5-2](#). 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 5-2. Configuration Data Window for DftSpecification



The graphical interface guides you through the IJTAG Network Insertion process by allowing 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 will highlight any errors.

You can create a new DftSpecification by using the following command:

display_specification –create

The command opens DFTVisualizer 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 named 'design1':

```
display_specification good3
```

For information on using the Configuration Data window, see sections “[Configuration Data Window](#)” and “[Modifying the Contents of the Configuration Data Window](#)” in the *Tessent Shell User's Manual*. For information on DftSpecification syntax and examples, see “[Configuration-Based Specification](#)” in the *TessentShell Reference Manual*. The [DftSpecification](#) grammar is completely described in the *TessentShell 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	Starts the DFTVisualizer and displays the DftSpecification in the Configuration window. Allows subsequent editing of the displayed DftSpecification. Use the "-create" option to get a new, empty DftSpecification.
process_dft_specification	Executes 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 the purpose of 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 *TessentShell 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..... 98

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 DFTVisualizer Configuration Data window. 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 will be 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 will not create the design instance; it will only connect it as specified.

```
(SSib3) {
    (design2_I1/instrumentB_I1) {
    }
}
```

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 will be connected to the second scan-in port of the SIB; the scan-in of the instrument will be 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 will be connected to the same source from which the SIB receives these control signals. Tck will be connected in a similar manner.

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 will be 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 which will then be inserted as part of the specification processing. You use the following basic connection specification:

```
(TTdr3) {  
    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 will contain a TDR register of eight bits. The size of the TDR is automatically determined based on the connectivity requirements. In addition, 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:

```
(TIjtagNetwork3) {  
  {  
    Connection(6:0) : design2_I1/instrumentA_I1/INA[6:0];  
  }  
  {  
    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(s), 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 fanin (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(T3) {  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, which will be 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 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

```
(TTdr3) {  
  {  
    Connection(6:0) : design2_I1/instrumentA_I1/INA[6:0];  
    multiplexing : on;  
  }  
  {  
    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 will contain 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 not known 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.

```
(TTdr3) {  
  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 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 109 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.

```
(Ttdr3) {
  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 function each of the design instance pins perform on the host scan interface:

```
(iHostScanInterfacejtag) {
  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” which includes the scan and outgoing controlling port functions: the ScanInPort (from the chip-side back to the TAP), ToSelectPort, ToCaptureEnPort, ToShiftEnPort, and ToUpdateEnPort.

```
(iHostScanInterfacejtag) {  
    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 for this example, that 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 “S3” is connected to the scan input pin of “S1”. For a detailed example and schematic refer to the description of the wrapper in the *Tessent Shell ReferenceManual*.

```
(iHostScanInterfacejtag) {  
    Interface {  
        design_instance : main_tap ;  
        scan_interface: host1 ;  
    }  
    (S1) {  
    }  
    (S2) {  
    }  
    (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.


```

(iHostScanInterfacejtag) {
  Interface {
    design_instance : main_tap ;
    scan_interface : host1 ;
  }
  (S1) {
  (S2) {
    }
  (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. Note how 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 in the *Tessent Shell Reference Manual* for complete information about the ScanMux.

```

(SSib1) {
  (Tsel) {
  }
  (SM1) {
    Select : tdr(Tsel)/DataOut(0);
    Input(0) {
      (T0) {
        length : 3 ;
      }
    }
    Input(1) {
      (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 , , or connections. If the TDR does not use any of them, 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.

```
(SSib3) {  
    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.

```
(SSib3) {  
    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 ;  
  }  
}
```


Chapter 6

IJTAG and ATPG in Tessent Shell

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.

Since IJTAG is only available in Tessent Shell and is not part of the classic ATPG point tools (FastScan and TestKompress), ATPG must be used within Tessent Shell to leverage this feature. The [Tessent Shell User's Manual](#) explains the steps required to transition existing dofiles from the ATPG point tools to ATPG in Tessent Shell.

IJTAG ATPG Flow Overview	109
IJTAG Features of ATPG in Tessent Shell	111
EDT IP Setup for IJTAG Integration	111
How to Set Up Embedded Instruments Through Test Procedures	113
How to Set Up Embedded Instruments Through the Dofile	114
Implicit and Explicit iReset Commands	115
A Detailed IJTAG ATPG Flow	117

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 a number of steps that are optional, 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 which 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 which reference iProcs on any ICL instance. This allows you to enable the low-power mode of an EDT IP, for example, and 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 117.

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	111
How to Set Up Embedded Instruments Through Test Procedures.....	113
How to Set Up Embedded Instruments Through the Dofile	114
Implicit and Explicit iReset Commands	115

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.

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, since 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 will be 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
```

Note that generic semantic terms, like “`edt_configuration`” or “`edt_bypass`” are used for the parameter denoting the static configuration ports of an EDT IP. The generated PDL file will translate these semantic terms into the pin names actually 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 needs to be specified. [Table 6-1](#) below lists all the possible ports by parameter keyword and their default values.

Table 6-1. EDT Configuration Keywords and Values

Parameter Keyword	Default Value
<code>edt_configuration</code>	0 (== LC)
<code>edt_low_power_shift_en</code>	0 (== off)
<code>edt_bypass</code>	0 (== off)
<code>edt_single_bypass_chain</code>	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 need to 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). Of course, 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 in order to set the static configuration bits in the PDL. This is done automatically by the PDL regargeting engine as part of the `test_setup` simulation.

How to Set Up Embedded Instruments Through Test Procedures

In the ATPG context patterns -scan, IJTAG is allowed in test_setup and test_end procedures only. It is neither allowed in any other procedure, nor in ATPG's analysis mode. Also, only iReset and iCall commands are allowed in the test procedures.

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_tpl =
    force_pi 0 ;
    measure_po 10 ;
    pulse tck_a 40 20;
    period 100 ;
end;

procedure test_setup =
    timeplate gen_tpl ;
    // 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;
```

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 will contain patterns derived from the PDL within the test_setup and/or 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 as well as the entire ICL hierarchy and PDL command files have to 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, allows for 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 commands [set_test_setup_icall](#) and [set_test_end_icall](#) 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 to 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 will be executed 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, which 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_ical` command without any error given by the tool.

In all the examples listed above, the iCalls are executed sequentially in the order they were declared. On the other hand, IJTAG also allows the parallel execution of iCalls. The next example shows an iProc that sets up multiple OCC instruments, all in parallel. The PDL retargeting engine will try to find a solution within the given hardware constraints that allows for this parallel execution. 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 execute for all OCCs:

```
SETUP> set_test_setup_ical {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_ical 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_tpl ;
    // 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_tpl ;  
    // 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 will happen 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 which 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_tpl ;  
    // 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 will then no longer issue 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_tpl ;  
    // 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 will be inserted after the last cycle statement, but with no implicit iReset added before, since 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 needs to 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, flow 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 will be 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 will be 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 context patterns –ijtag. 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 will be 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 typically the case for SIB, TAP, and TDR modules, especially if they
were
;# inserted through the ICL Network insertion functionality of Tessent Shell,
shown
;# in Step 1. But this is typically not the case for the EDT IP ICL module generated
in
;# Step 2, since the user 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 afterwards will not be 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 that 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 executes 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.

Chapter 7

IJTAG Examples

This chapter presents selected IJTAG topics of interest.

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.
- **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.

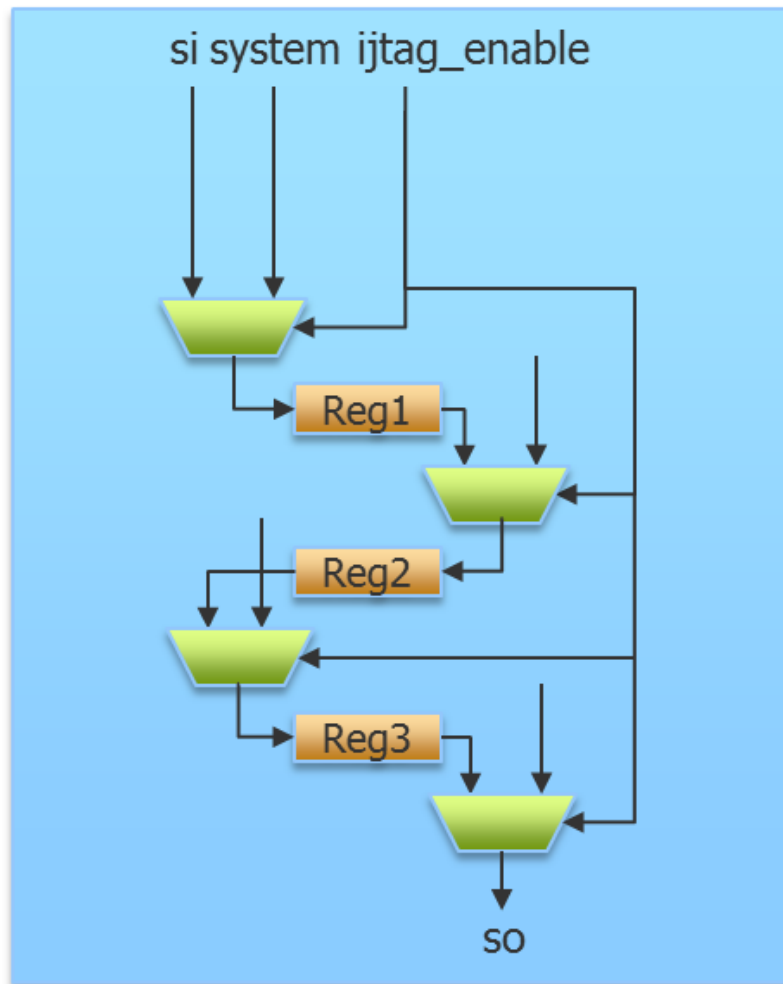
ICL Modeling versus Verilog Modeling	122
ICL Namespaces	123
PDL Namespaces	125
How to Define Default Values in ICL	125
Attributes of the ICL Extraction Flow	127
Scan Chain Integrity Test in Tessent IJTAG	128
How to Define Auto-Return Values in ICL	129
How to Model Addressable Registers in ICL	131

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 multiplexor 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

Consider the following problem: Two suppliers deliver ICL and PDL of instruments, but happen to choose the same name for the instrument. You cannot instantiate both at the same time. Also binding of PDL iProcs to the modules is no longer unambiguous. IJTAG resolves this problem with namespaces for ICL and PDL.

Note



ICL Namespaces are not currently supported; only PDL Namespaces are. This section is provided for completeness purposes only.

ICL file from supplier A:

```
NameSpace A ;  
Module MBIST { ... }
```

Now you can bind the PDL that comes with the module 'MBIST' from supplier A to this module by referencing the namespace.

PDL file from supplier A:

```
iProcTargetModule A::MBIST  
iProc init {} { ... }
```

In a similar way, supplier B provides ICL and PDL as follows

ICL file from supplier B:

```
NameSpace B ;  
Module MBIST { ... }
```

PDL file from supplier B:

```
iProcTargetModule B::MBIST  
iProc init {} { ... }
```

You now have two ICL modules with their respective PDLs, but separated by individual namespaces. The ICL instantiations of these two instruments may look like this:

```
Module TOP {  
    ...  
    Instance I1 of A::MBIST { ... }  
    Instance I2 of B::MBIST { ... }  
    ...  
}
```

Calling the iProcs made available through I1 and I2 is no different than calling iProcs for any other instrument:

```
iCall I1.init  
iCall I2.init
```

The PDL retargeter resolves the iCall of an iProc named 'init' backwards identifying for example I2 being an instance of MBIST in the ICL namespace 'B'. In this namespace, bound to this ICL module, the PDL retargeter found the iProc 'init' and executed the PDL commands in it.

PDL Namespaces

The problem a PDL namespace resolves is related, but distinct from the ICL namespace problem.

Assume you own an ICL module named M. Again, you have two suppliers, who instantiate their instruments in your ICL module, but this time they bind the provided PDL to 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. Obviously ICL namespaces do not resolve this problem. The IJTAG standard provides therefore a PDL namespace to further separate PDL for the same ICL module.

Your ICL module file:

```
NameSpace MyNS ;  
Module M { ... }
```

Your PDL file(s), defining the iProcs, which may be modified by the respective supplier:

```
iProcTargetModule MyNS::M -iProcNameSpace R  
iProc init {} { ... }  
  
iProcTargetModule MyNS::M -iProcNameSpace S  
iProc init {} { ... }
```

Your ICL file instantiating I1 of module M:

```
Module TOP {  
    ...  
    Instance I1 of MyNS::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
```

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 keywords “ResetValue” and “DefaultLoadValue”. Both define a scan load value that the PDL retargeter must abide. When an iReset is issued, the 4-bit scan register Reg_1 in this example will assume the value '1001' for its register bits. Note that 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, which 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 which 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, in this case named “chip”, use the following command:

```
report_attributes [ get_icl_modules chip ]
```

In case the ICL module “chip” was created through ICL extraction there will be a number of 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 a 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 will 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 allows 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. Please note the option '-end_in_pause' of the **iApply** command. In this example, you use the same chain test value that Mentor Graphics 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 will still be executed as expected, however after the bit changes value due to a write operation, the PDL retargeter will return 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 will change this bit:

User PDL file:

```
iWrite TDR_1 0b111001100  
iApply
```

It is up to the PDL retargeter to first execute the 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 will compute 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 7-3](#) 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 7-3](#) demonstrates using the iApplyEndState ICL feature. Notice that the bit TDR1[63] in the ICL code in [Figure 7-3](#) 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 allows 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 perfectly modeled with this syntax. More complex addressing schemes need to be modeled explicitly with DataMux construct on the read path and DataRegister with WriteDataSource and WriteEnSouce properties on the write path.

[Figure 7-2](#) 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 exact 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 PCL 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 7-3](#).

Figure 7-2. Schematic View of an Indirect Addressing Scheme

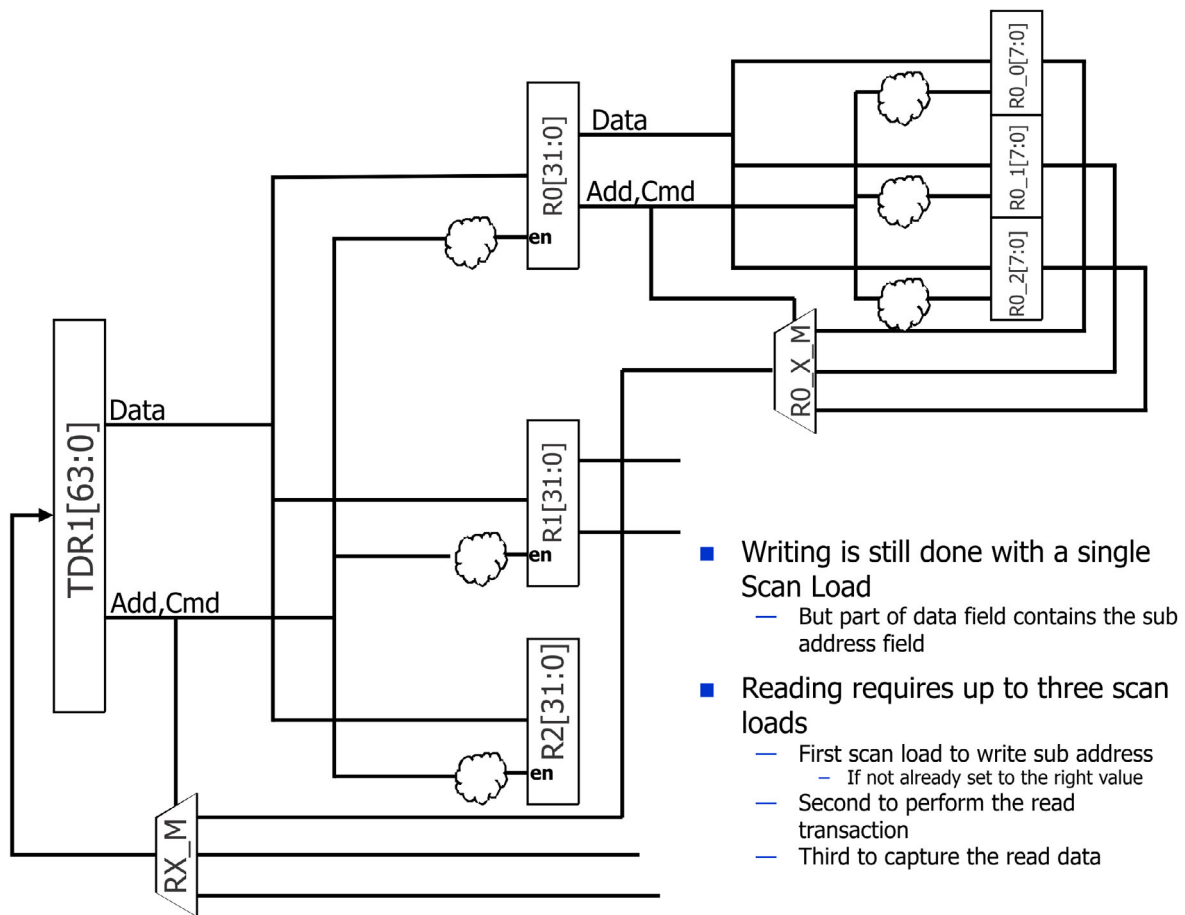


Figure 7-3. ICL Description of an Indirect Addressing Scheme

```

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; }
}

```

Chapter 8

Verification and Debug of IJTAG Instruments and Networks

IEEE 1687-2014 (IJTAG) allows describing various instruments and network components through ICL (Instrument Connectivity Language) 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 isn't obvious to determine. Additionally, when simulations fail it is increasingly difficult to figure why.

This chapter provides guidelines and pointers to verify and debug such IJTAG networks and instruments.

General Guidelines for Debugging Simulation Results	136
Creating ICL Verification Patterns	136
Using ICL Verification Patterns.....	136
ICL Verification Patterns Summary	140
Displaying the Comparison Failure Counter	140
Conclusion	141

General Guidelines for Debugging Simulation Results

This section describes guidelines that will help in debugging 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).

The verification testbenches generated with **create_icl_verification_patterns** can be simulated with standard EDA simulators (for example, ModelSim or QuestaSim).

Using ICL Verification Patterns

The test patterns generated from the **create_icl_verification_patterns** command are actually 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 actually 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/out to/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 case, the related scan muxes are being flagged as "to be processed" in step 2 (below).

2. For each scan mux input that was not tested in step 1, the scan mux condition will be 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/out the related scan register.

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 which activates all necessary conditions to reach this scan register.

The last two steps above will be skipped if everything is tested during step 1.

A scan register could be part of more than one scan path. In that case it will be 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 patterns used to test the scan registers and their connectivity are as follows:

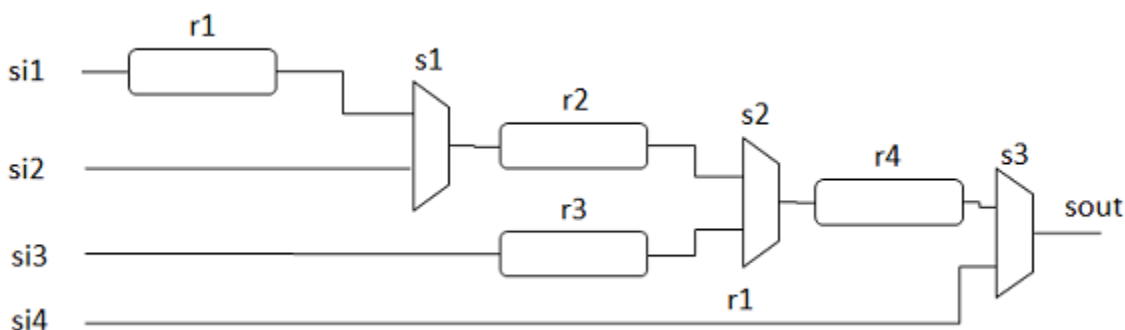
```
1.1000...000 (one '1')
2.0000...000 (only '0')
3.0111...111 (one '0')
4.1111...111 (only '1')
```

The above patterns ensure 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



The above example will result 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 will be tested in this scan path configuration. Remember that tracing starts at a scan out and keeps tracing backwards until a scan in is encountered.

The algorithm takes into account a list of instances to exclude from the test. This list is generated from the **-instances/-modules** and **-exclude_instances/-exclude_modules** switches optionally specified by the user. This list is empty by default, that is, all instances will be 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”.

DataOut pins that cannot be observed are ignored as well as DataIn pins which cannot be activated, for example, due to constraints or missing connectivity.

Note that 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 case that the synthesis modifies the design hierarchy (for example, because of “generate” loops). In order to obtain data pin verification patterns which 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 to 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 don’t know where to look.

To help debugging those mismatches, the verification TB contains an internal variable named **`_compare_fail_count`**. This variable is expected to be 0 initially and will increment whenever any comparison mismatch is observed.

It is thus a good idea to display this variable when looking at simulation waveforms. Since it only increments when a miscompare is recorded, first focus around the simulation time at which it becomes equal to 1. Much before that time, everything is likely OK — and any additional failure afterwards may have the very same cause.

Once you zoom in that very first failure, look at 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 to look at 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 allows 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 looking for 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.

There are several ways to get help when setting up and using Tessent software tools. Depending on your need, help is available from documentation, online command help, and Mentor Graphics Support.

The Tessent Documentation System	143
Mentor Graphics Support.....	144

The Tessent Documentation System

At the center of the documentation system is the InfoHub that supports both PDF and HTML content. From the InfoHub, you can access all locally installed product documentation, system administration documentation, videos, and tutorials. For users who want to use PDF, you have a PDF bookcase file that provides access to all the installed PDF files. Both the InfoHub and the PDF bookcase also provides direct access to SupportNet for software downloads and Knowledge Base articles.

For information on defining default HTML browsers, setting up browser options, and setting the default PDF viewer, refer to the “[Documentation Options](#)” in the *Mentor Graphics Documentation System* manual.

You can access the documentation in the following ways:

- **Shell Command** — On Linux platforms, enter **mgcdocs** at the shell prompt or invoke a Tessent tool with the **-manual** invocation switch.
- **File System** — Access the Tessent InfoHub or PDF bookcase directly from your file system, without invoking a Tessent tool. For example:

HTML:

```
firefox $MGC_DFT/docs/infohubs/index.html
```

PDF

```
acroread $MGC_DFT/docs/pdfdocs/_bk_tessent.pdf
```

- **Application Online Help** — ou can get contextual online help within most Tessent tools by using the “**help -manual**” tool command. For example:

```
> help dofile -manual
```

This command opens the appropriate reference manual at the “dofile” command description.

- **SupportNet** — SupportNet provides the entire Tessent documentation suite and Knowledge Base articles. You access SupportNet at the following URL:

<http://supportnet.mentor.com>

Mentor Graphics Support

Mentor Graphics software support includes software enhancements, access to comprehensive online services with SupportNet, and the optional On-Site Mentoring service.

For details, refer to this page:

<http://supportnet.mentor.com/about>

If you have questions about a software release, you can log in to SupportNet and search thousands of technical solutions, view documentation, or open a Service Request online:

<http://supportnet.mentor.com>

If your site is under current support and you do not have a SupportNet login, you can register for SupportNet by filling out a short form here:

<http://supportnet.mentor.com/user/register.cfm>

All customer support contact information is available here:

<http://supportnet.mentor.com/contacts/supportcenters/index.cfm>

Third-Party Information

For information about third-party software included with this release of Tessent products, refer to the “[Third-Party Software for Tessent Products](#).”



End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/eula

IMPORTANT INFORMATION

USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Software received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. ORDERS, FEES AND PAYMENT.

- 1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (each an "Order"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not those documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order or presented in any electronic portal or automated order management system, whether or not required to be electronically accepted, will not be effective unless agreed in writing and physically signed by an authorized representative of Customer and Mentor Graphics.
- 1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice. Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.
- 1.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation, setup files and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Except for Software that is embeddable ("Embedded Software"), which is licensed pursuant to separate embedded software terms or an embedded software supplement, Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 4.2); (b) for Customer's internal business purposes; (c) for the term of the license; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if Customer provides any feedback or requests any change or enhancement to Products, whether in the course of receiving support or consulting services, evaluating Products, performing beta testing or otherwise, any inventions, product improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.

3. BETA CODE.

- 3.1. Portions or all of certain Software may contain code for experimental testing and evaluation (which may be either alpha or beta, collectively "Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. Mentor Graphics may choose, at its sole discretion, not to release Beta Code commercially in any form.
- 3.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.
- 3.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 3.3 shall survive termination of this Agreement.

4. RESTRICTIONS ON USE.

- 4.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Except for Embedded Software that has been embedded in executable code form in Customer's product(s), Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer becomes aware of such unauthorized disclosure or use. Customer acknowledges that Software provided hereunder may contain source code which is proprietary and its confidentiality is of the highest importance and value to Mentor Graphics. Customer acknowledges that Mentor Graphics may be seriously harmed if such source code is disclosed in violation of this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, disassemble, reverse-compile, or reverse-engineer any Product, or in any way derive any source code from Software that is not provided to Customer in source code form. Log files, data files, rule files and script files generated by or for the Software (collectively "Files"), including without limitation files containing Standard Verification Rule Format ("SVRF") and Tcl Verification Format ("TVF") which are Mentor Graphics' trade secret and proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Products or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.
 - 4.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use, or as permitted for Embedded Software under separate embedded software terms or an embedded software supplement. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer's employees or on-site contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.
 - 4.3. Customer agrees that it will not subject any Product to any open source software ("OSS") license that conflicts with this Agreement or that does not otherwise apply to such Product.
 - 4.4. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense, or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.
 - 4.5. The provisions of this Section 4 shall survive the termination of this Agreement.
5. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer with updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics' then current End-User Support Terms located at <http://supportnet.mentor.com/supportterms>.
6. **OPEN SOURCE SOFTWARE.** Products may contain OSS or code distributed under a proprietary third party license agreement, to which additional rights or obligations ("Third Party Terms") may apply. Please see the applicable Product documentation (including license files, header files, read-me files or source code) for details. In the event of conflict between the terms of this Agreement

(including any addenda) and the Third Party Terms, the Third Party Terms will control solely with respect to the OSS or third party code. The provisions of this Section 6 shall survive the termination of this Agreement.

7. LIMITED WARRANTY.

- 7.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification, improper installation or Customer is not in compliance with this Agreement. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."
- 7.2. THE WARRANTIES SET FORTH IN THIS SECTION 7 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

8. LIMITATION OF LIABILITY. TO THE EXTENT PERMITTED UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

9. THIRD PARTY CLAIMS.

- 9.1. Customer acknowledges that Mentor Graphics has no control over the testing of Customer's products, or the specific applications and use of Products. Mentor Graphics and its licensors shall not be liable for any claim or demand made against Customer by any third party, except to the extent such claim is covered under Section 10.
- 9.2. In the event that a third party makes a claim against Mentor Graphics arising out of the use of Customer's products, Mentor Graphics will give Customer prompt notice of such claim. At Customer's option and expense, Customer may take sole control of the defense and any settlement of such claim. Customer WILL reimburse and hold harmless Mentor Graphics for any LIABILITY, damages, settlement amounts, costs and expenses, including reasonable attorney's fees, incurred by or awarded against Mentor Graphics or its licensors in connection with such claims.
- 9.3. The provisions of this Section 9 shall survive any expiration or termination of this Agreement.

10. INFRINGEMENT.

- 10.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to such action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.
- 10.2. If a claim is made under Subsection 10.1 Mentor Graphics may, at its option and expense: (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.
- 10.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; (h) OSS, except to the extent that the infringement is directly caused by Mentor Graphics' modifications to such OSS; or (i) infringement by Customer that is deemed willful. In the case of (i), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.
- 10.4. THIS SECTION 10 IS SUBJECT TO SECTION 8 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, FOR DEFENSE,

SETTLEMENT AND DAMAGES, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

11. TERMINATION AND EFFECT OF TERMINATION.

- 11.1. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.
- 11.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination of this Agreement and/or any license granted under this Agreement, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.
12. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and European Union ("E.U.") and United States ("U.S.") government agencies, which prohibit export, re-export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export or re-export Products in any manner without first obtaining all necessary approval from appropriate local, E.U. and U.S. government agencies. If Customer wishes to disclose any information to Mentor Graphics that is subject to any E.U., U.S. or other applicable export restrictions, including without limitation the U.S. International Traffic in Arms Regulations (ITAR) or special controls under the Export Administration Regulations (EAR), Customer will notify Mentor Graphics personnel, in advance of each instance of disclosure, that such information is subject to such export restrictions.
13. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. The parties agree that all Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to U.S. FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. government or a U.S. government subcontractor is subject solely to the terms and conditions set forth in this Agreement, which shall supersede any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.
14. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.
15. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 15 shall survive the termination of this Agreement.
16. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the U.S. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, U.S., if Customer is located in North or South America, and the laws of Ireland if Customer is located outside of North or South America or Japan, and the laws of Japan if Customer is located in Japan. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply, or the Tokyo District Court when the laws of Japan apply. Notwithstanding the foregoing, all disputes in Asia (excluding Japan) arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
17. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
18. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements. Any translation of this Agreement is provided to comply with local legal requirements only. In the event of a dispute between the English and any non-English versions, the English version of this Agreement shall govern to the extent not prohibited by local law in the applicable jurisdiction. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.