IEEE STANDARDS ASSOCIATION

IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device

IEEE Computer Society

Sponsored by the
Test Technology Standards Committee

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 1687™-2014

IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device

Sponsor

Test Technology Standards Committee
of the
IEEE Computer Society

Approved 3 November 2014

IEEE-SA Standards Board

Abstract: A methodology for accessing instrumentation embedded within a semiconductor device, without defining the instruments or their features themselves, via the IEEE 1149.1™ test access port (TAP) and/or other signals, is described in this standard. The elements of the methodology include a hardware architecture for the on-chip network connecting the instruments to the chip pins, a hardware description language to describe this network, and a software language and protocol for communicating with the instruments via this network.

Keywords: access network, built-in self-test (BIST), boundary scan, debug, design for testability (DFT), embedded instruments, IEEE 1149.1™, IEEE 1687™, Instrument Connectivity Language (ICL), internal JTAG (IJTAG), Joint Test Action Group (JTAG), on-chip instrumentation, Procedural Description Language (PDL), test, Tool Command Language (Tcl)

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

PDF: ISBN 978-0-7381-9416-5 STD20033
Print: ISBN 978-0-7381-9417-2 STDPD20033

Important Notices and Disclaimers Concerning IEEE Standards Documents

IEEE documents are made available for use subject to important notices and legal disclaimers. These notices and disclaimers, or a reference to this page, appear in all standards and may be found under the heading "Important Notice" or "Important Notices and Disclaimers Concerning IEEE Standards Documents."

Notice and Disclaimer of Liability Concerning the Use of IEEE Standards Documents

IEEE Standards documents (standards, recommended practices, and guides), both full-use and trial-use, are developed within IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association ("IEEE-SA") Standards Board. IEEE ("the Institute") develops its standards through a consensus development process, approved by the American National Standards Institute ("ANSI"), which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims all warranties (express, implied and statutory) not included in this or any other document relating to the standard, including, but not limited to, the warranties of: merchantability; fitness for a particular purpose; non-infringement; and quality, accuracy, effectiveness, currency, or completeness of material. In addition, IEEE disclaims any and all conditions relating to: results; and workmanlike effort. IEEE standards documents are supplied "AS IS" and "WITH ALL FAULTS."

Use of an IEEE standard is wholly voluntary. The existence of an IEEE standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

IN NO EVENT SHALL IEEE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO: PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE PUBLICATION, USE OF, OR RELIANCE UPON ANY STANDARD, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND REGARDLESS OF WHETHER SUCH DAMAGE WAS FORESEEABLE.

Translations

The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE should be considered the approved IEEE standard.

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, or be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position of IEEE.

Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to comments or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in revisions to an IEEE standard is welcome to join the relevant IEEE working group.

Comments on standards should be submitted to the following address:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854 USA

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

IEEE draft and approved standards are copyrighted by IEEE under U.S. and international copyright laws. They are made available by IEEE and are adopted for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making these documents available for use and adoption by public authorities and private users, IEEE does not waive any rights in copyright to the documents.

Photocopies

Subject to payment of the appropriate fee, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Updating of IEEE Standards documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect.

Every IEEE standard is subjected to review at least every ten years. When a document is more than ten years old and has not undergone a revision process, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE standard.

In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE-SA Website at http://ieeexplore.ieee.org/xpl/standards.jsp or contact IEEE at the address listed previously. For more information about the IEEE-SA or IEEE's standards development process, visit the IEEE-SA Website at http://standards.ieee.org.

Errata

Errata, if any, for all IEEE standards can be accessed on the IEEE-SA Website at the following URL: http://standards.ieee.org/findstds/errata/index.html. Users are encouraged to check this URL for errata periodically.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE-SA Website at http://standards.ieee.org/about/sasb/patcom/patents.html. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Participants

At the time this IEEE standard was completed, the IEEE 1687 Working Group had the following membership:

Kenneth Posse, *Chair*
Alfred Crouch, *Vice Chair*
Jeff Rearick, *Editor*
Michael Laisne, *Webmaster*

Paul Abelovski
Alan Bair
Bill Bruce
Krishna Chakravadhanula
C. J. Clark
Mike Coldewey
Jean-Francois Cote
Bruce Cowan
Adam Cron
Romi Datta
Stylianos Diamantidis
Jason Doege
Richard Dugan
Theodore Eaton
Heiko Ehrenberg
William Eklow
Brian Foutz
Pradipta Ghosh
Kevin Gorman
Suresh Goyal
J. J. Grealish
Scott Hartranft
Hong-Shin Jun
Rohit Kapur
Martin Keim
Bruno Latulippe
Andrew Levy
Guoqing Li
Ed Malloy
Harrison Miles, Jr.
Skip Meyers
Jay Nejedlo
Thai-Minh Nguyen
Hari Nookala
Rick Nygaard
Victor Orona
John Parham
Srinivas Patil
Michele Portolan
John Potter
Jeff Remmers
Paul Reuter
Mike Ricchetti
Ben Rice
Thomas Rinderknecht
Franciso Russi
John Seibold
Craig Stephan
Anthony Suto
Steven Terry
Brian Turmelle
William Tuthill
Brad Van Treuren
Hans Martin von Staudt
Hugh Wallace
Brian Wang
Mike Wiznerowicz
Christian Zoellin
Songlin Zuo

The following members of the individual balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Saman Adham
Gobinathan Athimolom
John Braden
Bill Brown
Gunnar Carlsson
Krishna Chakravadhanula
Keith Chow
C. J. Clark
Jean-Francois Cote
Adam Cron
Alfred Crouch
Russell Davis
Frans G. De Jong
Jason Doege
Sourav Dutta
Theodore Eaton
Heiko Ehrenberg
Peter Eijnden
William Eklow
Joshua Ferry
Kevin Gorman
Chris Gorringe
J. J. Grealish
Randall Groves
Peter Harrod
Kazumi Hatayama
Werner Hoelzl
Neil Glenn Jacobson
Hong-Shin Jun
Rohit Kapur
Martin Keim
Anzou Ken-Ichi
Michael Laisne
James Langlois
Roland R. Latvala
Philippe Lebourg
Adam Ley
Teresa Lopes
Greg Luri
Colin Maunder
Ian Mcintosh
Reinhard Meier
Harrison Miles, Jr.
Jeffrey Moore
Richard Morren
Zainalabedin Navabi
Ion Neag
Jim O'Reilly
Kim Petersen
Ulrich Pohl
Irith Pomeranz
Michele Portolan
Kenneth Posse
John Potter
Jeff Rearick
Paul Reuter
Mike Ricchetti
Gordon Robinson
Andrzej Rucinski
Francisco Russi
Bartien Sayogo
John Seibold
Kapil Sood
Roger Sowada
Thomas Starai
Craig Stephan
Michael Stora
Walter Struppler
Anthony Suto
Efren Taboada
David Thompson
Brian Turmelle
William Tuthill
Louis Ungar
Srinivasa Vemuru
Tom Waayers
Hugh Wallace
Douglas Way
Oren Yuen
Janusz Zalewski
Daidi Zhong
Christian Zoellin
Songlin Zuo

When the IEEE-SA Standards Board approved this standard on 3 November 2014, it had the following membership:

John Kulick, *Chair*
Jon Walter Rosdahl, *Vice-chair*
Richard H. Hulett, *Past Chair*
Konstantinos Karachalios, *Secretary*

Peter Balma
Farooq Bari
Ted Burse
Clint Chaplain
Stephen Dukes
Jean-Phillippe Faure
Gary Hoffman
Michael Janezic
Jeffrey Katz
Joseph L. Koepfinger*
David J. Law
Hung Ling
Oleg Logvinov
T. W. Olsen
Glenn Parsons
Ron Peterson
Adrian Stephens
Peter Sutherland
Yatin Trivedi
Phil Winston
Don Wright
Yu Yuan

*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Richard DeBlasio, *DOE Representative*
Michael Janezic, *NIST Representative*

Michelle Turner
IEEE-SA Content Publishing

Kathryn Bennett
IEEE-SA Technical Community Programs

Introduction

This introduction is not part of IEEE Std 1687™-2014, IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device.

The development of this standard stemmed from two independent (and unaware of each other) efforts at the 2004 International Test Conference to address issues surrounding the use of the IEEE 1149.1 test access port (TAP) for purposes well beyond boundary scan testing. These efforts were merged, and the Internal JTAG (IJTAG) Working Group was born. The scope of the effort was refined in the following months as more members joined the group to focus on the access to design for testability (DFT) features (more generally called *instruments*) inside devices. The development of the ideas comprising this standard can be traced by presentations at a series of conferences, workshops, and symposia, including ITC'05, BAST'06, ITSW'06, VTS'06, ETS'06, ITC'06, VTS'07, ETS'07, ITC'07, ETS'08, ITC'08, and IOLTS'09, ITC'11, ETS'12, ITC'12, and ITC'13.

Frequently asked questions

The IEEE 1687 web site will include an FAQ.[a]

[a] See http://grouper.ieee.org/groups/1687/.

Contents

IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device

1. Overview

1.1 Scope

This standard develops a methodology for access to embedded instrumentation, without defining the instruments or their features themselves, via the IEEE 1149.1™ test access port (TAP) and additional signals that may be required. The elements of the methodology include a description language for the characteristics of the features and for communication with the features, and requirements for interfacing to the features.

1.2 Purpose

IEEE Std 1149.1 specifies circuits to be embedded within a semiconductor device to support board test; namely, the TAP, TAP Controller, and a number of internal registers.[1] In practice the TAP and TAP Controller are being used for other functions well beyond boundary scan in an ad-hoc manner across the industry to access a wide variety of embedded instruments. The purpose of the IEEE 1687 initiative is to provide an extension to IEEE Std 1149.1 specifically aimed at using the TAP to manage the configuration, operation, and collection of data from this embedded instrumentation circuitry.

[1] Information on references can be found in Clause 2.

1.3 Background

There exists the widespread use of embedded instrumentation [such as built-in self-test (BIST) engines, complex I/O characterization and calibration, embedded timing instrumentation, etc.), each of which is accessed and managed by a variety of external controllers using a variety of mechanisms and protocols. Therefore, a need exists for standardization of these methods in order to facilitate an efficient and orderly process for the preparation of tests that access and control these embedded instruments. Given the widespread use of IEEE 1149.1 TAP and boundary-scan architecture as a gateway to many internal test, debug, and programming features, it is logical to build upon the foundation of the boundary scan standards (IEEE Std 1149.1-1990 [B2], IEEE Std 1149.1-2001, IEEE Std 1149.1-2013 , IEEE Std 1149.4™-2010 [B3], IEEE Std 1149.6™-2010 [B4], IEEE Std 1532™-2002 [B5], and IEEE Std 1500™-2005) to support TAP-based access to internal device test features in a standardized manner.[2] IEEE Std 1149.1-2013 extended previous versions of that standard to include procedural access to reconfigurable test data registers (TDRs) accessible via the TAP. IEEE Std 1687 describes an instrument-centric approach to this problem that allows retargeting from instrument ports through a wide variety of on-chip network configurations (of which IEEE Std 1500 is one example) to a device interface (of which the TAP is one example). Like IEEE Std 1500, IEEE Std 1687 specifies how to build a conformant on-chip network while not forcing a specific device interface to the network. IEEE Std 1687 has taken a more descriptive than prescriptive approach for specifying the on-chip network, thus allowing many existing architectures to conform.

This standardization effort is intended to address the *access* to on-chip instrumentation, not the instruments themselves. The elements of standardized access include the following:

a) The hardware architecture for connecting instruments to the TAP via a serial access network and/or to other interfaces via a serial or parallel access network,

b) A hardware description language (HDL) that documents this access network between the instruments and device interfaces, and

c) A procedure description language that documents the operation of the instruments.

Items that have been specifically included in the scope of the standard include the application of instrument-level procedures retargeted through various levels of hierarchy, the retargeting of intellectual property (IP)-level patterns to device-level TAP patterns, and the documentation of the connection of instruments to other (non-TAP) interfaces.

Items that have been excluded from the scope of the standard are the detailed off-chip protocols for instrument communication over non-TAP interfaces, synchronized interoperation between instruments, and the assurance of reusability of chip-level patterns at board or system levels.

The goal of IEEE Std 1687 is to facilitate the use and reuse of internal instrumentation by providing a standard yet flexible network architecture for accessing the instruments and standard descriptions of both the network and the operation of the instruments. Given the considerable number of devices containing internal instruments that are already available, this standard has made every reasonable attempt to provide support for the description of commonly used legacy implementations. The standard also supports forward-looking architectures that are scalable and uniform in their approach to accessing embedded instruments in a hierarchical manner.

1.4 Organization

This standard is organized as follows:

[2] The numbers in brackets correspond to those of the bibliography in Annex G.

Clause 1 provides background and context for this standard.

Clause 2 provides references necessary to understand this standard.

Clause 3 defines the terminology and nomenclature used in this standard.

Clause 4 is a tutorial that outlines the technologies addressed and utilized by this standard. This clause does not contain rules.

Clause 5 provides rules for the network architecture used by this standard.

Clause 6 provides rules for documenting the hardware architecture used by this standard.

Clause 7 provides rules for documenting the operation of the instruments used by this standard.

Clause 8 provides rules for manipulating the data produced by the instruments and for using Tcl as a programming language to manage instrument interactions.

Annex A (informative) specifies the language constructs used to describe the access network.

Annex B (informative) specifies the basic language constructs used to interact with the instruments.

Annex C (informative) specifies the advanced (Tcl) language constructs used to interact with the instruments.

Annex D (informative) identifies the subtle differences between the otherwise mostly common implementation of PDL.

Annex E (informative) presents many use cases showing applications of ICL and PDL to common circuits.

Annex F (informative) discusses practical issues for the implementation of commonly used circuits.

Annex G (informative) is a bibliography.

1.5 Context

The context in which this standard is applicable is illustrated in Figure 1, which shows a device under test (DUT) that includes a TAP interface, some pins and an I/O interface, and three (lightly shaded) access networks through which those interfaces are connected to four internal instruments. This standard defines the architecture of and a structured way to describe the hardware contents of the access network and the well-defined interfaces (darker shaded small rectangles) between the access networks and the instruments. This standard also describes the software content (e.g., test patterns and procedures) that is delivered over the access network to interact with the instruments.

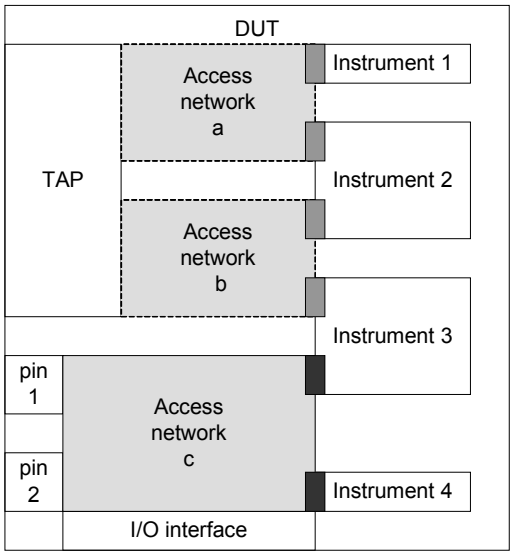


Figure 1—IEEE 1687 context

2. Normative references

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEEE Std 1149.1-2001, IEEE Standard Test Access Port and Boundary-Scan Architecture.[3, 4]

IEEE Std 1149.1-2013, IEEE Standard Test Access Port and Boundary-Scan Architecture.

IEEE Std 1500, IEEE Standard Testability Method for Embedded Core-Based Integrated Circuits.

3. Definitions, acronyms, and abbreviations

3.1 Definitions

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary Online* should be consulted for terms not defined in this clause. [5]

access network: *See:* **network.**

AccessLink: An **Instrument Connectivity Language (ICL)** keyword that is used to describe the details of the interface between the device pins and the **network**. The IEEE 1149.1 **test access port** (**TAP**) has a well-defined description built into **ICL**.

[3] IEEE publications are available from The Institute of Electrical and Electronics Engineers (http://standards.ieee.org/).
[4] The IEEE standards or products referred to in this clause are trademarks of The Institute of Electrical and Electronics Engineers, Inc.
[5] *IEEE Standards Dictionary Online* subscription is available at:
http://www.ieee.org/portal/innovate/products/standard/standards_dictionary.html.

active scan chain (or **Active shift path):** A set of serially connected shift register bits that advance by one bit per test clock cycle when enabled to shift. The network in the **module** (or modules) containing the segments of the scan chain is required to be selected in order to allow activity on the shift enable control signal.

Alias: An **Instrument Connectivity Language (ICL)** keyword that assigns a name to a given group of **ICL** objects (**register** bits or **ports**), which are then allowed to be referred to collectively by that name.

black-box module: A **module** description in **Instrument Connectivity Language (ICL)** that has only ports, but no contents. These modules may be used to hide proprietary intellectual property (**IP**). Such modules are often used in conjunction with **iScan** commands in **Procedural Description Language (PDL)** and thus have a precisely defined length at any given time, which is associated with each **ScanInterface** of the black-box module.

NOTE—See 6.4.5 and 7.3.5 for details.

boundary-scan testing: Testing of board or backplane interconnections between device pins as supported by IEEE Std 1149.1, IEEE Std 1149.4, and IEEE Std 1149.6. This testing technology is intended to find manufacturing defects such as open solder joints, shorted signal traces, broken bond wires, damaged drivers and receivers, etc.

built-in self-test (BIST), built-in test (BIT): Test of a component of a system by itself; more specifically, by test circuitry added to the mission circuitry that causes the component to exercise itself and report the results with no or minimal interaction with external sources. BIST may be applied on embedded components of a chip (such as memories, analog blocks, or I/O interfaces), on entire chips, or on entire boards or systems**.**

capture-shift-update (CSU): The sequence of operations applied to a **test data register** (TDR) to capture new data from the data-input port, shift the data through the TDR, then update the newly shifted data into the update stage of the TDR.

care bit: A port or register that has a non-X value that must be present in order to satisfy an objective (such as enabling a **scan chain** or maintaining a previously written value).

client interface: The **port functions** that attach a **module** to a host interface in a network. A client interface consumes control signals from the host interface and exchanges data with it.

client module: The successor of a **module** in a **network** organized in a host-client fashion. A **host module**'s client is closer to the **instrument**. A client module contains a **client interface** that connects to a **host interface**. *Contrast:* **host module**.

component: A hardware entity that is one of the primitive building blocks of the network, including storage elements, multiplexers, and logic.

controlling state: A representation in a software model of the set of logic values that have been written to **ports** and/or **storage elements** in the **network.**

NOTE—See description in 7.3.3.

core: A component of a **device** that may appear multiple times or may be reused in other devices. Also referred to as an IP (**intellectual property**) core or simply as IP.

data register: A set of one or more storage cells with a parallel input and a parallel output.

device: A collection of circuitry with a well-defined test boundary, generally including a **TAP** and physical **pins**.

device interface: The circuitry that connects the device **pins** to the instrument access **network**. A **TAP controller** is the most common example of a **device interface** for an instrument access network, but other interfaces (I^2C, a parallel microprocessor bus, a PCI-express port, etc.) could also serve as device interfaces if correctly adapted to the instrument access network. The device interface itself is not considered part of the network.

embedded test access port (eTAP): A captive version of a TAP thatthat acts as the interface to a portion of the network or a wrapped instrument rather than serving as the device-level **TAP**. An embedded **TAP** is typically accompanied by an embedded **TAP** controller (**eTAPC**).

embedded test access port controller (eTAPC): A finite state machine largely corresponding to that specified in IEEE Std 1149.1, which responds to the **embedded TAP** signals used as the interface to a sub-network or instrument. To force synchronization, there is a slight restriction from the IEEE 1149.1 **TAP** controller state diagram with respect to the arc traversed when the **eTAPC** is being re-selected.

NOTE—See 5.14.4.

expected state: A representation in a software model of the set of logic values that have been read from ports and/or storage elements in the network.

NOTE—See description in 7.3.3.

fan-in: The set of primitive elements encountered by tracing backwards through the network from the inputs of a specific element.

global reset: A signal that places all **modules** into their specified reset states.

handoff module: A **module** that is intended to be exchanged between parties [**intellectual property (IP) providers** and **integrators**] or tools (e.g., simulators or rule checkers) for which all access network **ports** must be declared and must match. A common example: a wrapped instrument delivered by an IP provider must have all the ports of the **Instrument Connectivity Language (ICL)** module present in the associated **register transfer language (RTL)** module. Note that the RTL may have additional ports that are unrelated to the access network; these need not be declared in the ICL module.

hardware description language (HDL): Any of a number of languages that describe the structural components, connectivity, behavior, or combinations thereof, of a circuit, including Verilog®, VHDL, **Instrument Connectivity Language (ICL)**, etc.

NOTE—Verilog is a registered trademark of Cadence Design Systems, Inc.

host interface*:* The **port functions** that attach a **module** to a **client interface**. A host interface provides control signals to a client interface and exchanges data with it.

host module: The predecessor of a **module** in a **network** organized in a host-client fashion. A **client module**'s host is closer to the **device interface**. A Host Module contains a **host interface** to which a **client interface** connects. *Contrast:* **client module**.

IEEE 1687 network: *See:* **network.**

instrument: Any on-chip circuit for test, debug, diagnosis, monitoring, characterization, configuration, or functional use that can be accessed by, configured from, or communicated with by an on-chip network described in **Instrument Connectivity Language (ICL)** that connects to an external device interface. An instrument is typically in a **module** that has only a **client interface**.

instrument access network: *See:* **network.**

Instrument Connectivity Language (ICL): The name of the language (both syntax and semantics) used to describe the structure of the **instrument access network** that connects **instruments** to **device interfaces**.

instrument interface: The connection point between the **network** and the **ports** of an **instrument**. When used in the context of a network, this point is conventionally referred to as the *right-hand side* of the network and may be compliant with the description of a scan host interface or a **test access port (TAP)** host interface. When used in the context of an instrument, this point describes the ports of the instrument, which may be compliant with the description of a scan client interface or a **TAP** client interface.

NOTE—See 5.16.

integrator: The consumer of **intellectual property (IP)** to be joined with other components to produce a device (or possibly another IP). The integrator's role and interest in IEEE Std 1687 is to construct a network that connects the IP, and use that network to **retarget** lower-level **Procedural Description Language (PDL)** procedures to be applied at the higher-level device that he has produced.

intellectual property (IP): A component of a device, potentially supplied by a third party, that would be accompanied by an **Instrument Connectivity Language (ICL)** description and one or more **Procedural Description Language (PDL)** procedures. *See:* **core**.

iScan command: A **Procedural Description Language (PDL)** command that specifies the bit values to both extract from and assign to the scan flip-flops comprising a **scan chain** of a known length.

NOTE—See 7.3.5 and 7.9.12.

justify (as in "**justify** a value on a signal"): The process of finding a valid setting of controllable values such that the target signal acquires the desired value.

left-hand side (LHS) ports: Module port functions that connect to **host modules**, which include ResetPort, SelectPort, CaptureEnPort, ShiftEnPort, UpdateEnPort, ScanInPort, ScanOutPort, TCKPort, DataInPort, DataOutPort, and others. Note that ScanInPort, ScanOutPort, DataInPort, and DataOutPort functions may connect to **peer modules** or may be **right-hand side (RHS) host ports** that connect to other **client modules**.

NOTE—See 5.11.

local reset: A signal generated internal to the **network** based on desired conditions that may be used to place a **module** into its specified reset state. A local reset event is desirable when a localized action is preferred to a **global reset** event (which might have consequences that are too broad).

local reset assert (**LRA**): A scan register bit whose purpose is to control the activation of a local reset signal.

logic state: A representation in a software model of in intermediate set of logic values that have been written to **ports** and/or **storage elements** in the **network.** *Contrast:* **controlling state**.

NOTE—See description in 7.3.3.

logical hierarchy: An arrangement of **modules** with parent-child relationships (i.e., a parent module instantiates child modules, with there being a **top-level module** at the highest level of the hierarchy).

module (IEEE 1687 module): A component containing a test access mechanism that provides communication with on-device instrumentation, an instrument, an interface to the pins of a device, or other module instantiations.

network (access network, instrument access network, IEEE 1687 network): The circuitry that connects the device interfaces to the instrument interfaces. An example of a device interface currently used for IEEE Std 1687 is the **test access port (TAP)**. The network consists of one or more **modules**. The **network interface** is the external access point to the network, conventionally referred to as the *left-hand side* (LHS) of the network. An instrument interface is the other end point of the network and is conventionally referred to as the *right-hand side* (RHS) of the network. Intermediate modules, if included, may be arranged in a host-client fashion and may contain logical hierarchy.

network hierarchy: A configuration of **modules** with host-client relationships (i.e., where one module is a host relative to a **client module** on its right and a client relative to a **host module** on its left, using the convention that the **device interface** is on the far left and the **instrument interface** is on the far right).

network interface: The connection point between the instrument access **network** and an external **device interface** [such as the **test access port (TAP)**]. *See:* **AccessLink**.

PDL Command Stream: A series of **Procedural Desription Language (PDL)** commands applied to an instrument or network or device. A PDL command stream may be contained within a file, or it may be generated on-the-fly by a **PDL** generator.

peer module: A **module** adjacent to a module in a **network** organized in a host-client fashion. A module and its peer both share the same **host module**.

pin: A physical connection point to a packaged **device**. Example: the solder bumps of a surface-mount device are pins.

plug-and-play: A descriptive term used to define a level of compliance of a **network interface**, indicating its type [either scan or **test access port (TAP)**] and its role (either host or client). The signals comprising a plug-and-play interface are sufficiently well specified such that corresponding interfaces (e.g., scan host and scan client) may be directly connected (i.e., "plugged") together to join sections of a **network**.

NOTE—See 5.16.

port: A connection point (terminal) of a collection of circuitry in a **module** that is intended to be wired to other circuitry. Example: the D, Q, and CK signals of a D-flip-flop module are ports; the internal signal that connects the master latch to the slave latch of the D-flip-flop is not a port.

port function: A **module** port with a specific function relevant to the operation of an instrument, network, or interface. The terminology "port function" indicates a property associated with a **port** that describes its operational purpose (i.e., its function). For IEEE Std 1687, the set of port functions are grouped into **left-hand side** (LHS) **ports** including ResetPort, SelectPort, CaptureEnPort, ShiftEnPort, UpdateEnPort, ScanInPort, ScanOutPort, TCKPort, DataInPort, and DataOutPort, as well as **right-hand side** (RHS) **ports** including ToResetPort, ToSelectPort, ToCaptureEnPort, ToShiftEnPort, ToUpdateEnPort, ToClockPort, ToTMSPort, ToTRSTPort, ToTCKPort, and ToIRSelectPort. Note that ScanInPort and ScanOutPort may be either left-hand or right-hand side ports; the same is true for DataInPort and DataOutPort.

primary input/primary output: A directly accessible input pin or output pin of a device in a chip context, or a DataInPort or DataOutPort in an instance context.

primitive component: A member of the set of lowest-level structural elements from which access networks are constructed; the set includes ScanRegister, ScanMux, OneHotScanGroup, DataRegister, DataMux, OneHotDataGroup, ClockMux, and LogicSignal.

NOTE—See 6.4.1.

Procedural Description Language (PDL): The name of the language (syntax and semantics) used to describe the operation of the instruments.

raw instrument: An instrument with only DataIn, DataOut, or Clock port functions in its client interface.

register transfer language (RTL): A **hardware description language (HDL)** that represents the behavior of a circuit at a high level by describing the storage elements and logical operations between them. The elements of the circuit are often organized using a hierarchy of **modules** (which define sub-circuits) and the connections between instances of those modules.

register: One or more **storage elements** that share a name and whose bits are arranged as a vector range (e.g., MyReg[31:0]).

retarget: The process of mapping a set of operations in **Procedural Description Language (PDL)** at the ports and/or registers of a **module** through the **network** to a desired module at a higher level of the logical hierarchy and/or closer to a **device interface** in the network hierarchy. A retargeted PDL will perform the same operations on the original module, but may also perform additional operations to configure the network between the original module and the new **target module**.

retargeting tool (or **retargeting software**): A tool that implements the process of **retargeting.**

right-hand side (RHS) ports: Module port functions that connect to **client modules**, which include ToResetPort, ToSelectPort, ToCaptureEnPort, ToShiftEnPort, ToUpdateEnPort, ScanInPort, ScanOutPort, ToTCKPort, DataInPort, DataOutPort, and others. Note that ScanInPort and ScanOutPort functions may connect to **peer modules** or may be **left-hand side (LHS) client ports** that connect to a **host module**; the same is true for DataInPort and DataOutPort.

NOTE—See 5.11.

run-test/idle (**RTI**): A state in the **test access port (TAP)** controller state diagram in which the controller can loop while other activity is occurring (e.g. a test is running or the chip is idling).

scan chain segment boundary: A connection between two portions of a **scan chain**. The segment boundary may occur within an IEEE 1687 **module** between two scan registers, or at a module boundary. A scan chain segment boundary may be used as the location for a scan chain reconfiguration circuit (e.g., a **scan multiplexer** or a **segment insertion bit**) or a retiming circuit (e.g., a lockup latch).

scan chain: One or more flip-flops connected into a serial shift register, whose shifting activity is controlled by test control signals (e.g., ShiftEn and TCK). A scan chain is modeled in **Instrument Connectivity Language (ICL)** as one or more ScanRegister. *Syn:* **scan path.**

NOTE—See 6.4.8.

scan client interface: A specific type of **Client Interface** that is comprised of the signals defined in a **ScanInterface** with a single **scan chain,** which allows plug-and-play attachment to a **serial access network**.

ScanInterface: An **Instrument Connectivity Language (ICL)** keyword used to identify a well-defined set of related signals of a **serial access network** that are ports of a **module** and comprise a logical interface to one or more **scan chains**. A ScanInterface may include multiple scan chains (with unique Scanin/Scanout pairs), and a module may have multiple ScanInterfaces.

segment insertion bit (SIB): A single-bit register (including an update stage) on a scan chain that controls whether or not a bypassable segment of that scan chain is included in the active shift path.

serial access network (scan network): A **network** or sub-network with one or more serial data communication paths and no parallel data signals.

shift path multiplexer (or **scan multiplexer):** A circuit element that may alter the configuration of a scan chain by selecting which shift storage elements are in the scan chain and/or in which order they appear.

shift storage element (or **shift path storage element**): A register on a scan chain.

signal: A connection between two or more **port functions** of **modules**.

simulation model: The circuit model utilized by a circuit simulation tool to predict the responses to stimuli, often represented in **register transfer language (RTL)**.

storage element: A bi-stable circuit element, such as a latch or a flip-flop, which retains state until certain enabling and/or clocking conditions are met that cause it to take on a new state.

target module: The **module** to which a **retarget**ing operation is destined; the **Procedural Description Language (PDL)** operations from an original module lower in the **logical hierarchy** or closer to the client side in a **network hierarchy** are mapped through the **network** to the target module. The target module is often the same as the **top-level module**.

test access port (TAP) controller (TAPC): The finite state machine specified in IEEE Std 1149.1 that responds to the **TAP** signals and drives some or all of the on-chip test logic.

test access port (TAP): The device interface pins defined in IEEE Std 1149.1: TDI, TMS, TCK, TDO, and optionally TRST*.

test data register (TDR): An IEEE 1149.1-compatible scan chain that can be connected between test data in (**TDI)** and test data out (**TDO)** when selected by an instruction in the **test access port (TAP) controller**.

test-logic-reset (**TLR**): A state in the **test access port (TAP) controller** state diagram that causes selected state elements to be loaded with their reset values.

top-level module: The **module** at the highest point of a **logical hierarchy**. A top-level module has only children and is not instantiated by any other modules.

variable-length scan chain: A scan chain whose connectivity (and thus length) can be altered as a function of the data within the scan chain, or from data contained in other scan chains or registers or pins.

wrapped instrument: An instrument whose parallel data ports have been connected to the data in and out ports of scan registers that comprise a scan wrapper. The interface to a wrapped instrument consists of the serial interface of the scan wrapper rather than the parallel interface of the original instrument.

3.2 Acronyms and abbreviations

BIST	built-in self-test
BSDL	Boundary Scan Description Language
CSU	capture-shift-update
DR	data register
EDA	electronic design automation
eTAPC	embedded TAP controller
FSM	finite state machine
IC	integrated circuit

ICL	Instrument Connectivity Language
IP	intellectual property
IR	instruction register
JTAG	Joint Test Action Group
LHS	left-hand side
LRA	local reset assert
PDL	Procedure Description Language
PI/PO	primary input / primary output
PLL	phase-locked loop
RHS	right-hand side
RTI	run-test/idle
RTL	register transfer language
SerDes	serializer-deserializer
SIB	segment insertion bit
TAP	test access port
TCK	test clock
TDI	test data in
TDO	test data out
TDR	test data register
TLR	test-logic-reset
TMS	test mode select
TRST*	test reset (* indicates active low)

4. Technology

4.1 Introduction

Access to embedded instrumentation involves two key on-chip hardware technologies: access networks and embedded instruments. Some essential hardware background material is provided here and in Clause 5 to provide the framework upon which IEEE Std 1687 is built. The language [Instrument Connectivity Language (ICL)] used to describe the hardware is covered in Clause 6.

The networks that provide access to embedded instruments have taken many forms, ranging from serially accessed wrappers to direct device pin connections. The former category has a well-established technology base enabled by other standards, the essential components of which are described in the next subclause.

4.2 Serial access networks

The signaling protocol used for an IEEE 1687 serial access network is consistent with IEEE Std 1500, and a common interface to a serial access network of instruments is the IEEE 1149.1 TAP and its associated

TAP controller. Since a basic knowledge of these portions of IEEE Std 1149.1 and IEEE Std 1500 is useful for a complete understanding of the operation and control of an IEEE 1687 serial access network, the relevant principles from those standards are summarized here. Though these fundamental principles are not expected to change in future version of those standards, it should be noted that IEEE Std 1687 utilizes the features as documented in the versions in existence solely at the time of this writing.

4.2.1 IEEE 1149.1 test access port

The interface specified by IEEE Std 1149.1 is referred to as the *test access port* (TAP), and consists of the following device pins:

- TMS: Test Mode Select (input to control the TAP state machine)
- TDI: Test Data In (input to provide test control and data to the device)
- TDO: Test Data Out (output to observe test status and data from the device)
- TCK: Test Clock [input to clock the TAP state machine and test data registers (TDRs)]
- (optional) TRST*: Test Reset (active low) (input to asynchronously reset the TAP controller)

This interface is connected to the TAP controller.

4.2.2 IEEE 1149.1 test access port controller

The IEEE 1149.1 TAP controller consists of a state machine with 16 states whose principal purposes are to load an instruction into the TAP Instruction Register (IR) and access the associated Test Data Register (TDR) between Test Data In (TDI) and Test Data Out (TDO). The state diagram for the TAP controller is shown in Figure 2. The diagram is dominated by two symmetrical paths that specify the sequence of operations for a register (a TDR on the left, and the instruction register on the right). A TDR consists of one or more serially connected flip-flops that can optionally capture data from a circuit at cell inputs, can serially shift data through the chain of cells, and/or optionally provide updated data to the circuit via cell outputs.

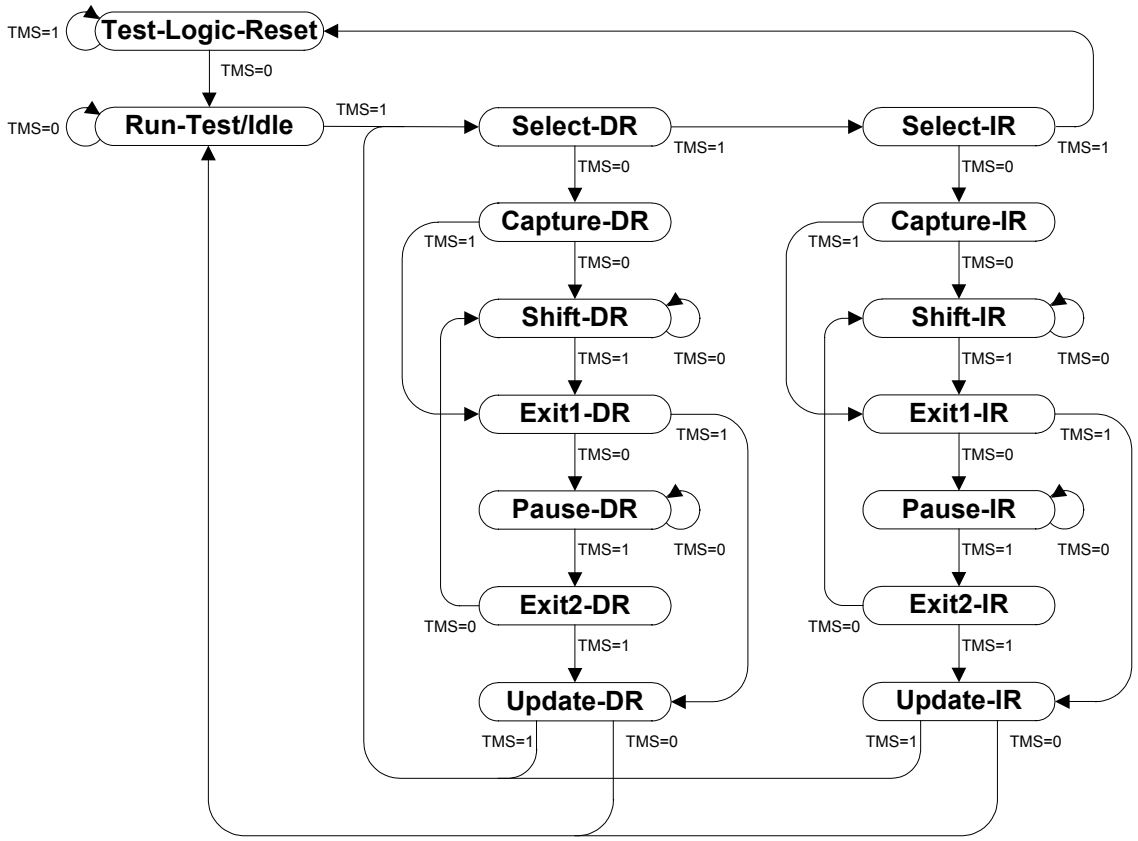
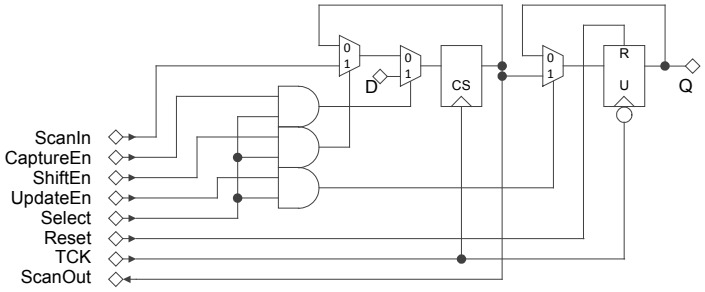


Figure 2—TAP Controller State Diagram

The capture, shift, and update operations for these registers occur during the corresponding states in the TAP state machine. Transitions between states are controlled by the value of the Test Mode Select (TMS) pin during the rising edge of Test Clock (TCK). By definition, a state is entered and exited on the rising edge of TCK; therefore, the first TCK edge seen in the new state is the falling edge. This is important in determining data transfers that may take place in a state. Figure 3 shows the relationship of the TAP states and the acceptable timing windows during which the signals that enable the capture, shift, and update operations are active. A circuit diagram of a single-bit TDR is shown for reference, but multiple shift cycles are included in the event-ordering diagram.

Two styles of TAP Controller designs exist with respect to clocking. The gated-clock style supplies separate Capture, Shift, and Update clocks by gating TCK with the Capture, Shift, and Update states of the TAP Controller. The enabled-clock style supplies TCK along with state signals indicating when the TAP Controller is in each of its states. While it is possible to connect an IEEE 1687 serial access network to either TAP clocking implementation, this document illustrates only the latter—a continuous TCK with separate state signals.



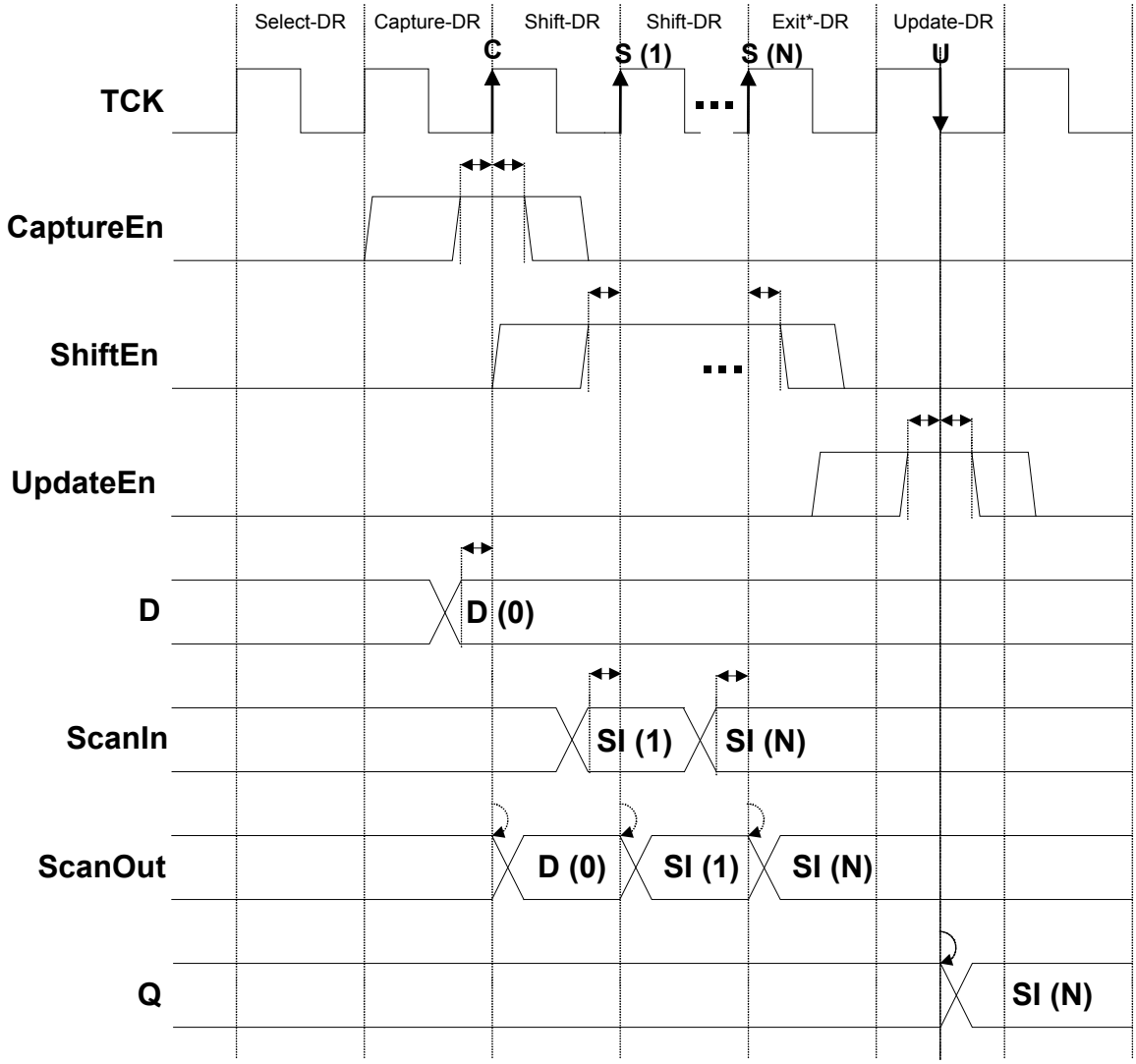


Figure 3—TAP state and enable signal timing

The three main register operations (capture, shift, and update) that are triggered during the corresponding TAP controller states are shown with arrows on the appropriate TCK edge. Space is included for multiple shift operations, with the first (1) and last (N) ScanInput (SI) data bits labeled. The corresponding ScanOut data stream begins with the data (D) captured into the flip-flop labeled "CS" and is followed by the shift data. Minimum setup time and hold time margins are shown with horizontal arrows before and after the appropriate clock edge, respectively. The clock edges after which output data may change are indicated with curved arrows.

4.2.2.1 Test data registers

A very common hardware mechanism through which information is exchanged to and from an instrument is the *test data register* (TDR). Each instruction loaded into the TAP Instruction register may place one, and only one, TDR between the TDI and TDO ports of the IEEE 1149.1 TAP Controller. The actions associated with the TDR that is selected by the current instruction are as follows:

— Data is captured into a TDR cell on the rising edge of TCK when the TAP is in the CAPTURE-DR state.

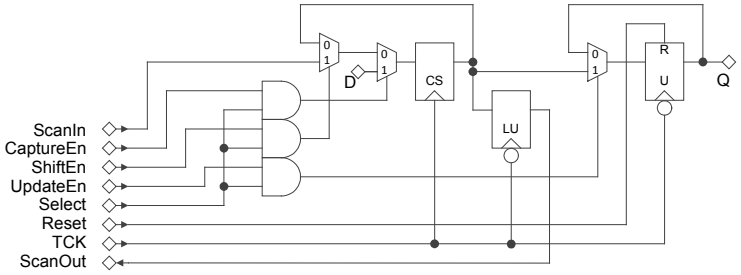
— Data is transferred from one TDR cell to the next in the serial chain on the rising edge of TCK when the TAP Controller is in the SHIFT-DR state.

— Data is transferred to the Update register of a cell on the falling edge of TCK when the TAP state machine is in the UPDATE-DR state. If a TDR cell is a "Status" cell (i.e., does not supply data to an instrument), then no Update register exists and nothing takes place on this falling edge.

One feature that deserves clarification is that for some single-cycle-long states, such as Capture-DR (and for Shift-DR in the case of a 1-bit long TDR), the edge on which the action occurs (capturing or shifting) is actually coincident with the edge that signifies the exit from the state (as shown by the up-arrows in Figure 3).

4.2.2.2 IEEE 1149.1 test access port and IEEE 1500 timing example

A simpler-to-understand version of the timing diagram in Figure 3, showing fixed timing on TCK edges rather than with the permissible setup and hold windows, is shown in Figure 4 and will be used as a reference throughout the remainder of this document. This timing is consistent with that of the Wrapper Serial Port (WSP) of IEEE Std 1500 and is also valid for IEEE Std 1687. One key difference from Figure 3 is that the TDR shown in the circuit diagram contains a retiming latch (also called a *lock-up latch*) on the ScanOut port. This is a requirement of the ScanOut port of an IEEE 1500-compliant wrapper. The only impact that the presence of this latch has is to delay the time at which the ScanOut data changes by one phase of TCK; logically, the circuits of Figure 3 and Figure 4 work the same way as seen from a TAP connected to them.



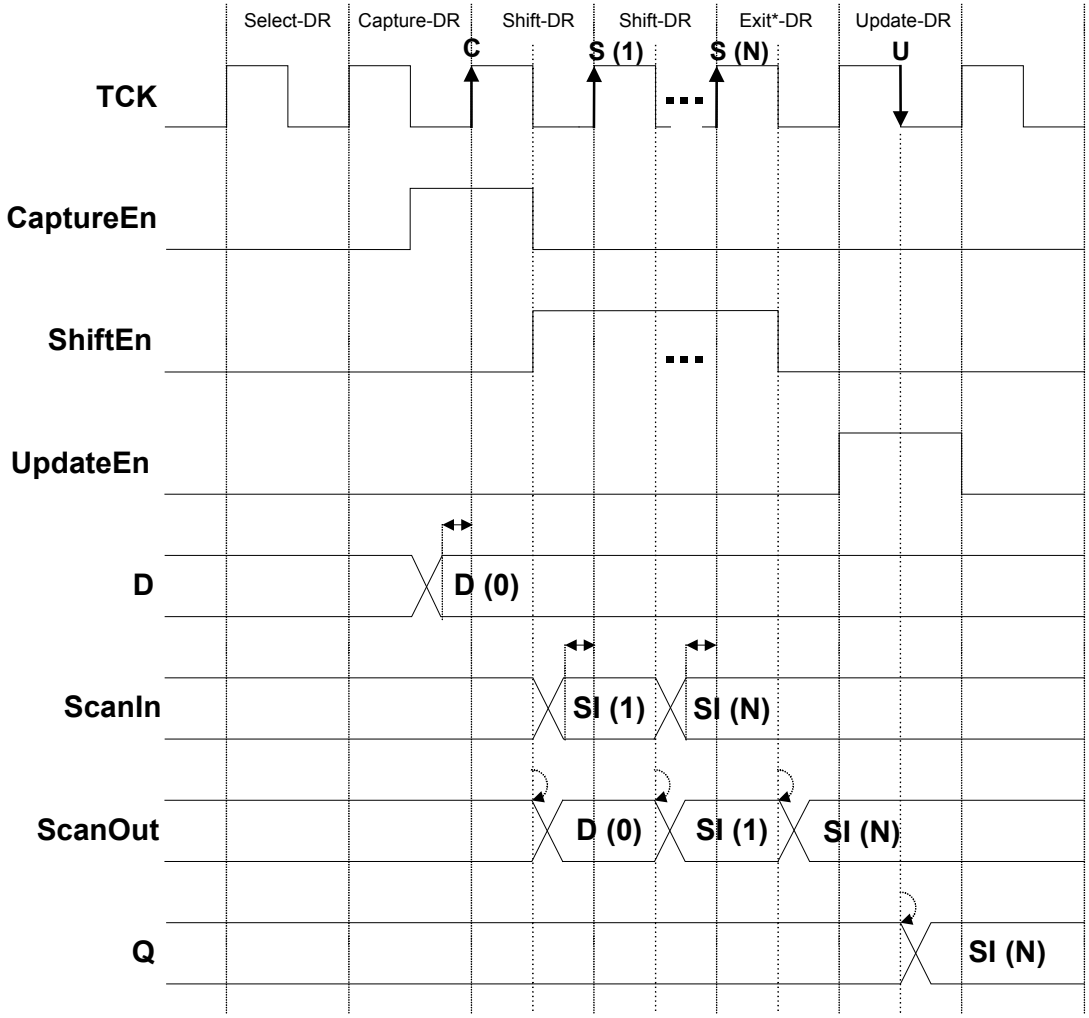


Figure 4—TAP state and enable signal timing, consistent with IEEE Std 1500

4.2.2.3 Reset mechanisms and timing

There are three distinct reset mechanisms in IEEE Std 1687 as follows:

— Asynchronous reset from the TRST* pin

— Synchronous reset that occurs when the TAP is in the test-logic-reset (TLR) state

— Synchronous local reset that may be produced by a TDR.

The timing diagrams in Figure 5 and Figure 6 illustrate the second and third reset mechanisms. The first (asynchronous) mechanism is omitted from the timing diagrams to avoid unnecessary clutter: no matter when TRST* is asserted, it causes an immediate reset.

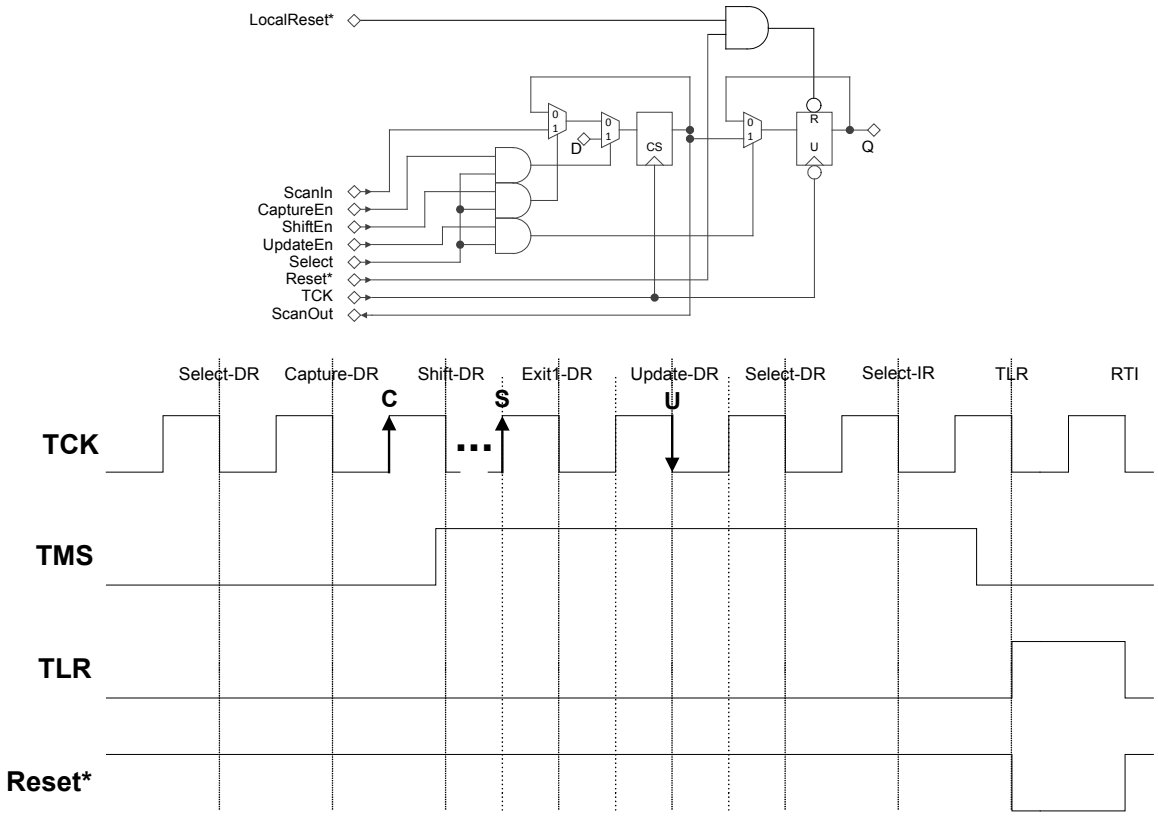


Figure 5—Test-Logic-Reset timing

The duration of the reset event shown in Figure 5 is dependent upon the TMS input; the figure shows a single-cycle spent in the test-logic-reset (TLR) state with an immediate transition to the run-test/idle (RTI) state based on TMS being set to 0.

When a local reset is used (it is optional), the timing of the events is as shown in Figure 6. The salient feature is that the local reset is a self-clearing signal: it returns to its de-asserted value prior to the next time the TDR generating it executes a capture operation.

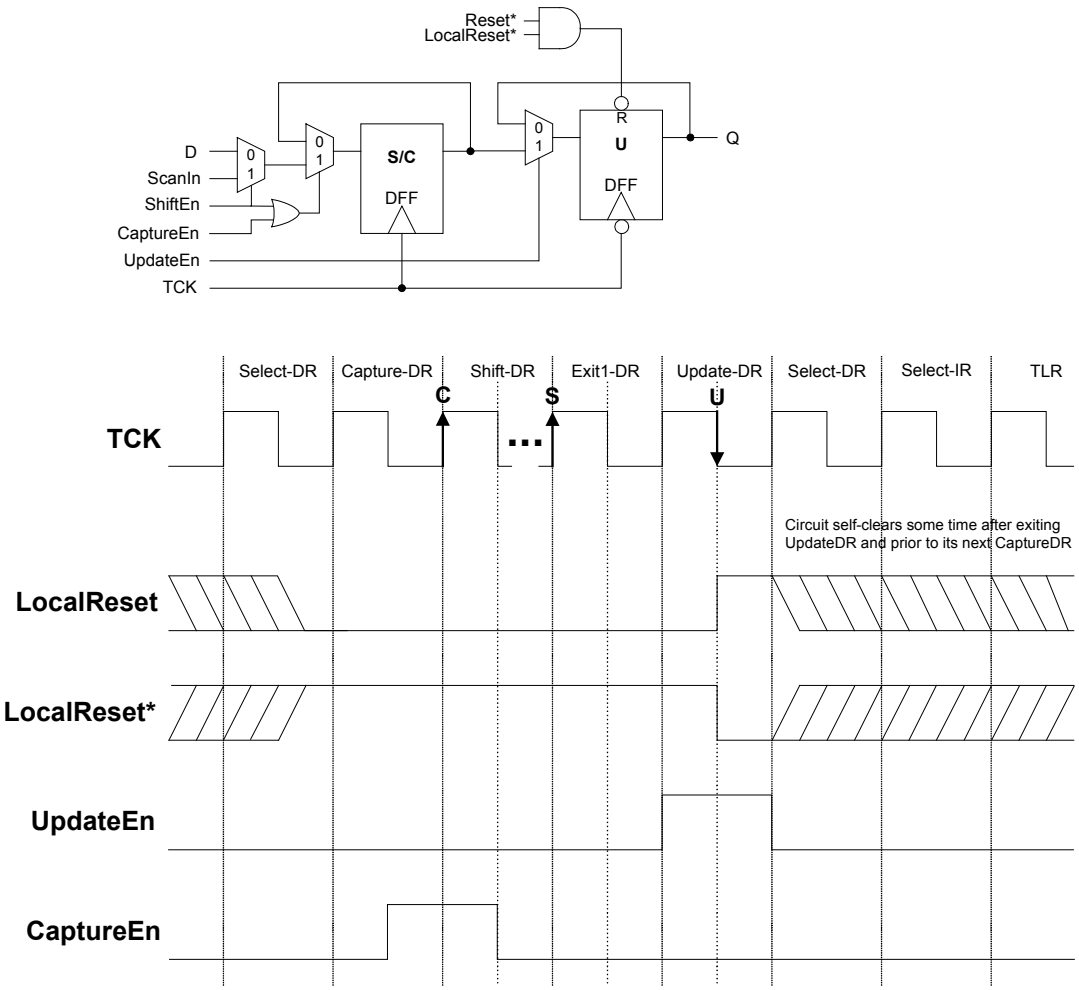


Figure 6—Local Reset Timing

4.3 On-chip instruments

Instruments are circuits embedded into a device intended for a specific control or data-collection purpose. Some examples of embedded instruments are as follows:

— Phase locked loop circuits: generate clocks at a specified frequency

— Clock control blocks: gate and control system clocks

— Memory BIST controllers: perform self-test of arrays of storage elements

— Logic BIST controllers: generate stimulus for and monitor response from logic

— Process monitoring and speed binning circuits: measure the performance of the fabricated device

— EFUSE blocks: program configurable elements of the device by selectively blowing electrical fuses

— I/O bit error rate counters: measure the rate at which transmission errors are received

— I/O eye diagram mapping engines: adjust voltage and timing parameters to measure signal integrity

These instruments interface to the device environment via their I/O ports. Those ports necessary for the operation, control, and data management for these instruments may be connected to an instrument access network via an instrument interface designed for that purpose. While the architecture of the instruments is not dictated, it is assumed that the instrument interface is static and event-driven (meaning that an instrument initiates actions when its inputs are changed and holds its resulting outputs until told to respond to new stimulus when its inputs change again). This document defines the requirements for the design of such instrument interfaces, including two types of "plug-and-play" standard interfaces to which instruments may be joined for compatible connection to the network. This document also defines a mechanism for describing the interconnections of the network, as well as a mechanism for procedurally controlling the network (i.e., sending data and commands to the individual instruments and/or receiving data from the individual instruments).

5. Hardware architecture

5.1 Introduction to the IEEE 1687 network

The basis of the hardware architecture is the instrument access network (simply "network" henceforth), which connects one or more device interfaces to one or more instruments. A device is said to contain one network (described in ICL); that network may contain multiple (possibly disjoint) portions (or sub-networks). A compliant network consists of at least one module, which may include logical hierarchy (i.e., the top-level module may have child instances, and these instances may themselves be parents of other children, and so on).

IEEE Std 1687 facilitates access from the chip boundary to instruments embedded within the device. Achieving this access requires at least three portions of an access architecture: (1) the device interface; (2) the instrument interface; and (3) the access network that resides between the device interface and the instrument interface. IEEE Std 1687 covers the mechanisms to build, describe, and operate those three architectural elements. IEEE Std 1687 can be connected to many types of controllers and interfaces, but pays particular attention to the very popular serial access networks used in IEEE Std 1149.1 and IEEE Std 1500. The signals available to operate the IEEE 1687 serial access network are shown in Table 1.

Table 1—Signals available to operate a serial access network

Signal name	Function
ScanIn	Serial data input
ScanOut	Serial data output
CaptureEn	Enables the shift storage elements to acquire data from their parallel inputs
ShiftEn	Enables the serial scan chain to shift
UpdateEn	Enables the update storage elements to copy data from the shift storage elementsand present it on their parallel outputs
Select	Activates the interface
Reset	Reset the update storage elements to the reset values (if specified)
TCK	Test clock to which the capture, shift, and update operations are synchronized

5.2 Hierarchical IEEE 1687 networks

The notion of hierarchy is important for IEEE 1687 networks and has two dimensions: network hierarchy (in the host-client sense) and logical hierarchy (in the parent-child sense). The components of an IEEE 1687 network are described in one or more modules. An example of a single-module network with endpoints included is shown in Figure 7.

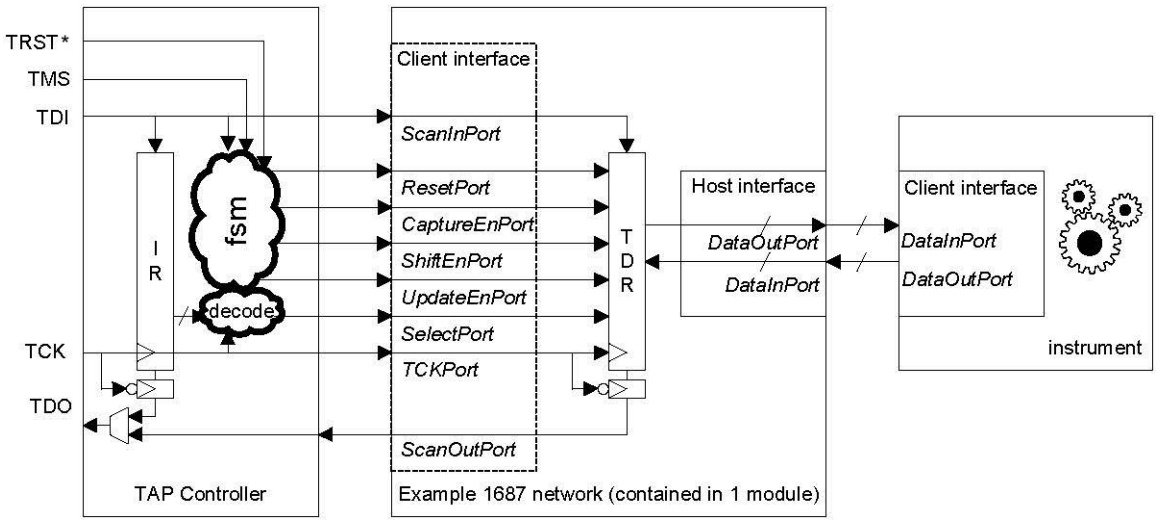


Figure 7—Single-module network architecture example

The convention used throughout this document is to represent the device interface (e.g., the TAP) to the left of the network and the instrument(s) to the right. Accordingly, the network has two symbolic interfaces: a client interface on the left side that interacts with the controller, and a host interface on the right side that interacts with the instrument(s). Likewise, each module within the network has a symbolic left-hand side (LHS, facing its host module, ultimately the controller), and a symbolic right-hand side (RHS, facing its client module, ultimately the instrument). The LHS interface of a module is called its *client interface*; the RHS interface of a module is called its *host interface*.

Two important concepts apply to host and client interfaces in IEEE Std 1687. First, instruments at the lowest level of integration usable in IEEE Std 1687 have only client interfaces, and those client interfaces consist of only "data" signals (though the actual function of those signals may be control or status or actual data). Second, some interfaces that contain specific sets of signals are given the label "plug and play" and called out as entities that establish specific types of compliance (see 5.16). These two concepts come together for an entity called a "wrapped instrument with a plug-and-play interface," which is a common method to deliver an instrument from an IP provider to an integrator.

Modules may be organized in a host-client fashion. A portion of a multiple-module network is shown in Figure 8, with its host and client interfaces highlighted. The internal contents of each of the modules (labelled "reconfiguration logic" in Figure 6) represent the components that comprise a portion of the network; these will be described in subsequent subclauses in Clause 5 and Clause 6.

From the perspective of logical hierarchy, all the modules in Figure 8 are siblings. It is also common to have logical hierarchy within an IEEE 1687 network, as shown in Figure 9. The boxes labeled A through I represent modules that contain elements of the IEEE 1687 network such as registers and muxes and wires (not shown).

The use of logical hierarchy in the network description supports IP reuse, partitioning, and, of course, modularity (which is widely used in modern HDLs as well as software languages).

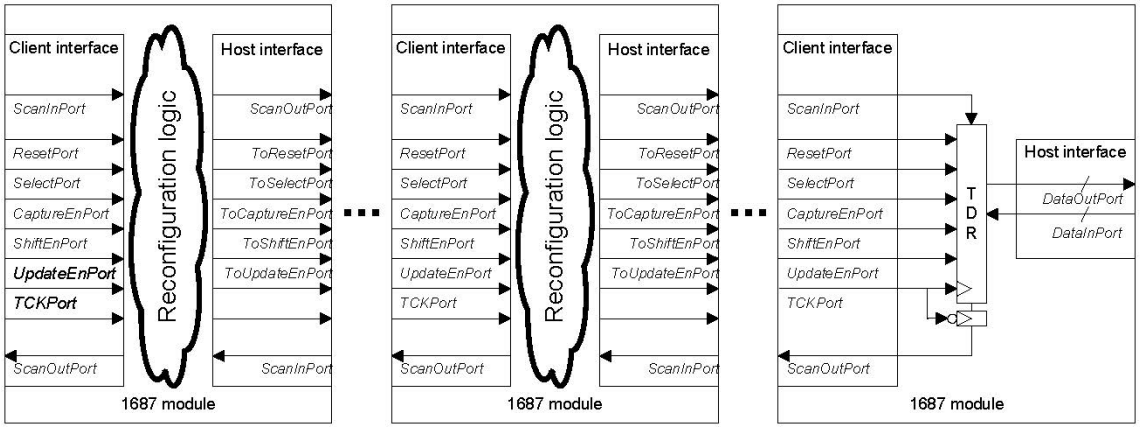


Figure 8—Multiple-module instrument access network

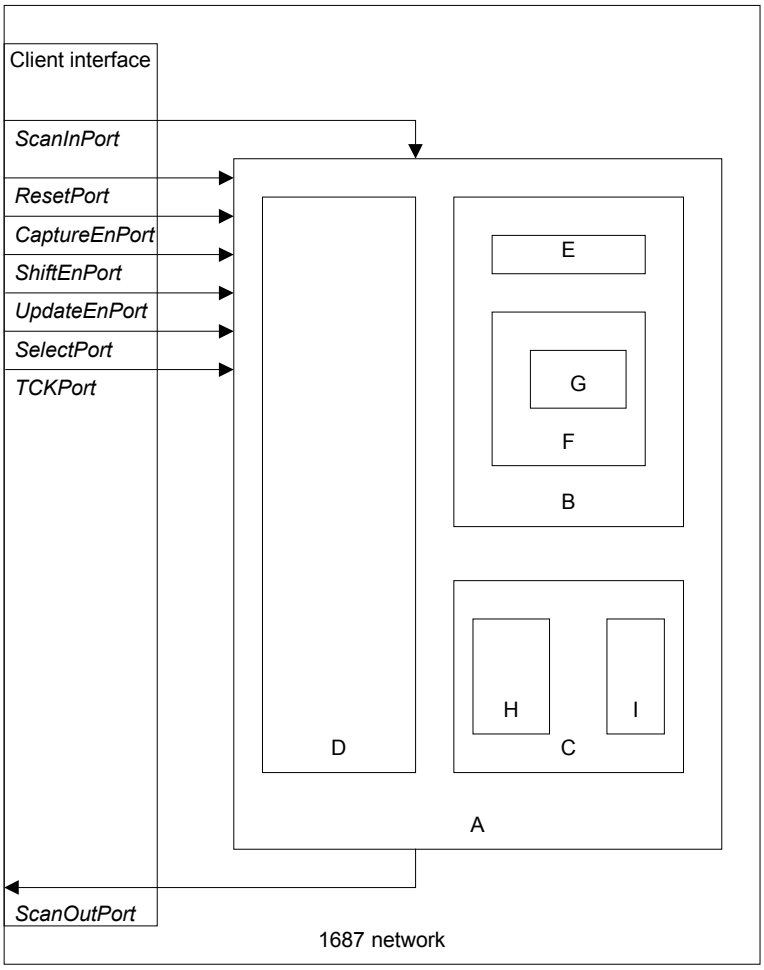


Figure 9—IEEE 1687 network with logical hierarchy

5.3 Controller

A controller generates the operation signals for the access network and may be as simple as direct connection of the operation signals of the access network to the chip's pin boundary, or may be as complex as a controller containing state machines and selection logic (such as the IEEE 1149.1 TAP and TAP Controller) or even a complete microcontroller or microprocessor.

IEEE Std 1687 defines and illustrates how to connect the access network to an IEEE 1149.1 Joint Test Action Group (JTAG) controller as in Figure 10, but also enables other controller types. The signals and protocols generated by the controller (independent of its type) are illustrated throughout Clause 4 and Clause 5 and are directly compatible with IEEE 1149.1.

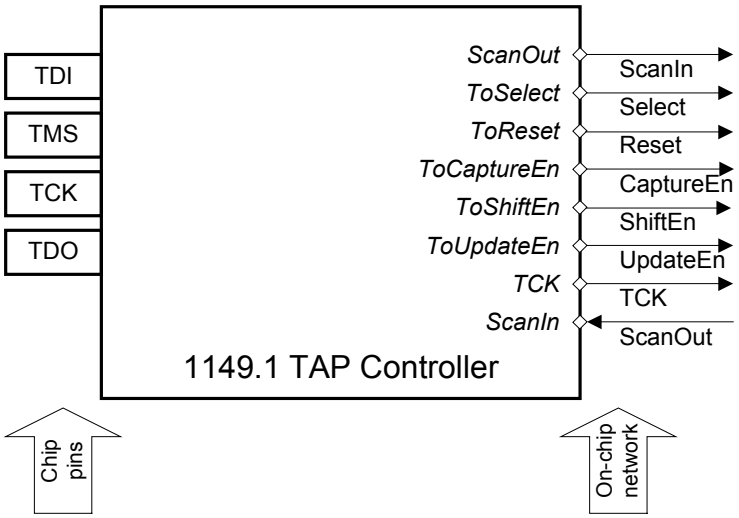


Figure 10—Example controller and controller signals to operate an access network

The IEEE 1149.1 Controller operates by converting clock (TCK) and control (TMS) inputs and a serial data (TDI) input at the chip's pin boundary into signals that can be used to select, access, and operate embedded registers and logic inside the chip. Data from inside the chip can be observed outside of the chip through a serial data (TDO) output pin. In general, an instruction encoding installed in the IEEE 1149.1's Instruction Register selects a TDR to be in the serial scan chain and provides all of the signals necessary to configure and operate the TDR. For the case of IEEE Std 1687, an instruction, known as the AccessLink Instruction, selects a portion of the network to be the active scan chain connected between TDI and TDO; and the Controller provides operation signals that sequence in accordance with the IEEE 1149.1 finite state-machine (FSM) for the selected scan chain. More than one AccessLink Instruction may exist to select independent portions of the network, and more than one AccessLink statement may exist to associate different controllers with the network.

5.4 Instrument interface

The instrument interface for IEEE 1687 purposes is defined as the signals on the instrument that are required to operate and control the instrument. For example, Figure 11 shows a Memory BIST instrument and its target, a memory array – and note that the memory array has a selection option of connecting to the Memory BIST instrument or to functional operation signals. There are many signals shown in the diagram of the instrument system, but for this example, only the Reset, Start, Stop, Done, and Fail signals are required to configure, operate and observe the instrument. The other chip-level signals, the functional data and address bus, the functional read/write, the functional clock; and the internal "instrument"-to-

"instrument's target" signals, the BIST_R/W, BIST_En, BIST_Data, and BIST Address – do not need to be "accessed by an IEEE 1687 instrument access network" in order to operate the instrument.

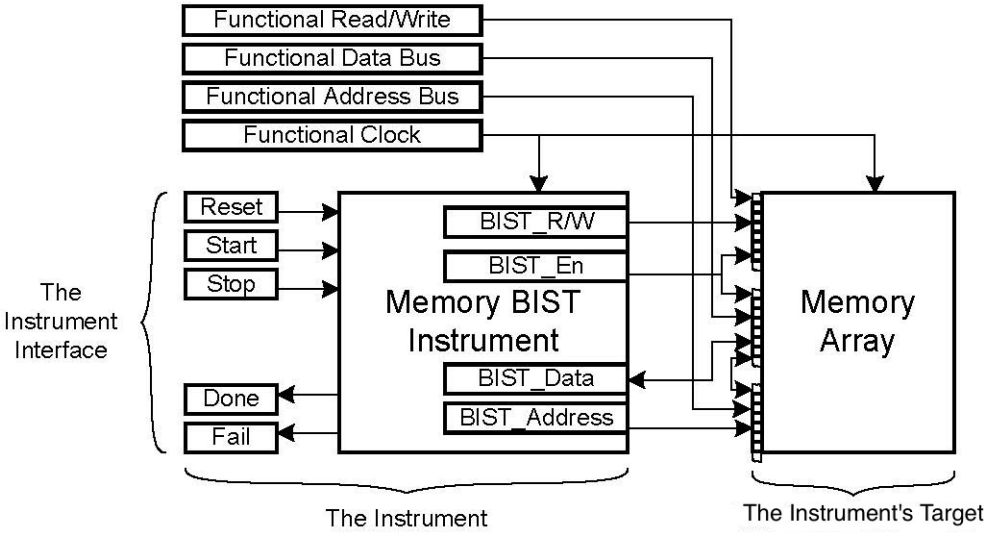


Figure 11—Instrument and target of instrument

In many cases, these other non-accessed signals are meant to remain hidden and may be considered proprietary. In addition, the internal structure of the instrument itself may be considered proprietary. When an instrument is delivered from a provider in a standalone manner (often called a *raw instrument*), it is delivered with documentation that identifies all of the signals and their manner of connection (which may include, among other items, a Verilog RTL model, a CTL file, timing/voltage/power specifications, etc.) to the chip integrator. After integration into a device, the detailed information about the instrument and its connections may be considered proprietary. Therefore, for IEEE Std 1687, only the limited subset of signals required to configure, operate, and observe the instrument needs to be separately and specifically identified and documented in ICL. The signals on a raw instrument (an instrument with no IEEE 1687 network interface) are represented by generic input signals known as DataIn and generic output signals known as DataOut.

5.5 Access network

Given the defined endpoints of the controller on one side and the instrument interface on the other, the access network is defined as everything in between. This network consists of architectural elements including scan chains, registers, muxes, control signals, etc. The network may be represented by many possible configurations to be of optimal value for any given implementation. The IEEE 1687 access network is not meant to be an overly restrictive architecture that limits the possible connections to instruments. It is meant to solve the problems of access optimization, instrument scheduling, and implementation in accordance with engineering budgets. Different network configurations allow for active scan chain length management, flexible access to instruments, and minimization of the registers needed or routes involved with the operation signals. The following paragraphs describe common network configurations: static, locally reconfigurable, remotely reconfigurable, and configurable via segment insertion or path selection.

Figure 12 shows the concept of a single instrument IEEE 1687 network where the controller is on the left, the Instrument Interfacing Test Data Register (TDR) is in the middle, and the Instrument is on the right. The data path from the controller to the TDR is a serial scan chain that, for this example, operates in accordance with the IEEE 1149.1 FSM in the TAP Controller. The data is delivered to the TDR on ScanIn signals that ultimately source from the IEEE 1149.1 TAP's TDI, and the data is returned to the TAP

Controller through ScanOut signals that ultimately connect to the TAP's TDO. By convention in this document, scan data is presented with the least significant bit (LSB) as the rightmost bit that is physically closest to the output scan port. Therefore, scan data always shifts visually from left-to-right .The TDR is a set of IEEE 1149.1 Scan Cells that contain both a shift-side, and a ripple-free update side. A "shift-in and update" writes to the instrument's parallel DataIn signals, and a "capture and shift-out" reads the instrument's parallel DataOut signals. This instrument access network is selected by the installation of an instruction in the IEEE 1149.1 Instruction Register (IR), which is documented by the AccessLink statement in ICL.

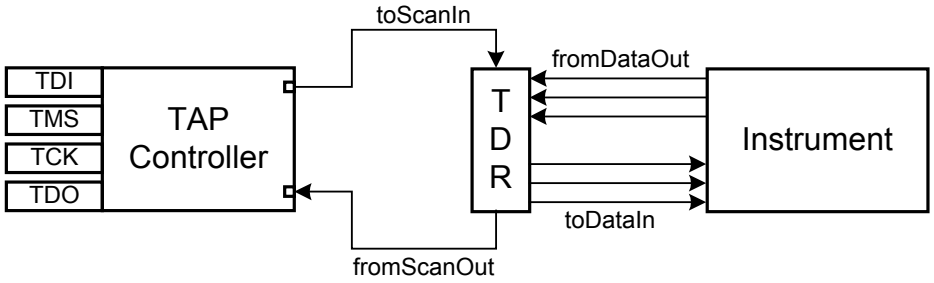


Figure 12—Conceptual example of a single instrument IEEE 1687 network

Figure 13 shows another conceptual IEEE 1687 network configuration that includes two instruments accessible by the serial scan chain from the controller. The TDR shown has two register fields, each of which communicates with a different instrument (A and B). Since both register fields are on the same serial scan chain they may be viewed as one contiguous TDR, which is consistent with the definition of a TDR as being the entire collection of the scan flops connected between TDI and TDO. However, the point of view typically taken by the IEEE 1687 architecture separates TDRs into their component register fields (conventionally referred to as ScanRegisters or simply registers) as a more natural method to interface with each instrument independently.

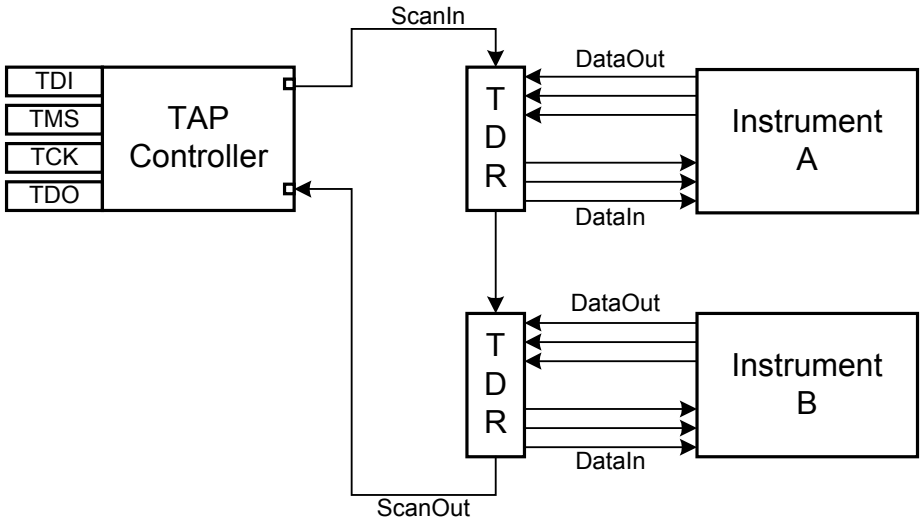


Figure 13—Conceptual multiple instrument single scan chain IEEE 1687 network

5.5.1 Reconfigurable scan chains

It is expected that some instruments may be provided as IP units from different suppliers. The IP may be delivered wrapped and complete with the TDR that contains the scan chain and interface signals to and from the instrument, or the IP may be delivered raw with only its DataIn and DataOut ports. Either an automation tool or a human integrator will incorporate the different IP into an overall chip-level scan chain.

In some cases, multiple complex networks containing tens or hundreds of instruments may need to be incorporated into the overall chip-level IEEE 1687 network. In order to avoid having extremely long scan-paths resulting from the daisy-chaining of the interface TDRs of all the instruments, other connection options are also supported.

One such connection option is to allow registers to be added to and subtracted from the active scan chain without requiring tens or hundreds of instruction encodings in the IEEE 1149.1 TAP Controller. Not only is this inefficient, but the TAP Controller may not exist at the time the instruments and network are created. In this case, the IP provider may need to create an optimized network architecture with a flexible configuration control mechanism. There are multiple methods that can be used to accomplish this.

Figure 14 shows a conceptual IEEE 1687 network configuration that supports multiple instruments and includes an element known as a Segment-Insertion-Bit (SIB) to allow the overall scan chain to be of variable length. In this example, when asserted, the SIB causes the scan chain to expand to include (and allow hierarchical access to) the register that provides the instrument interface to Instrument C. When de-asserted, the SIB will return to its closed configuration, which bypasses this register and instead provides a shortcut (as short as one bit) to this segment of the scan chain. In this de-asserted mode, the SIB can be thought of as being a single-bit bypass register for a wrapped instrument (e.g., the combination of the bottom-most TDR with the instrument labeled C in Figure 14) or even an entire network of instruments. For illustrative purposes, the convention of the SIB opening the associated segment when asserted will be used in these examples, but a SIB could be built with the opposite behavior.

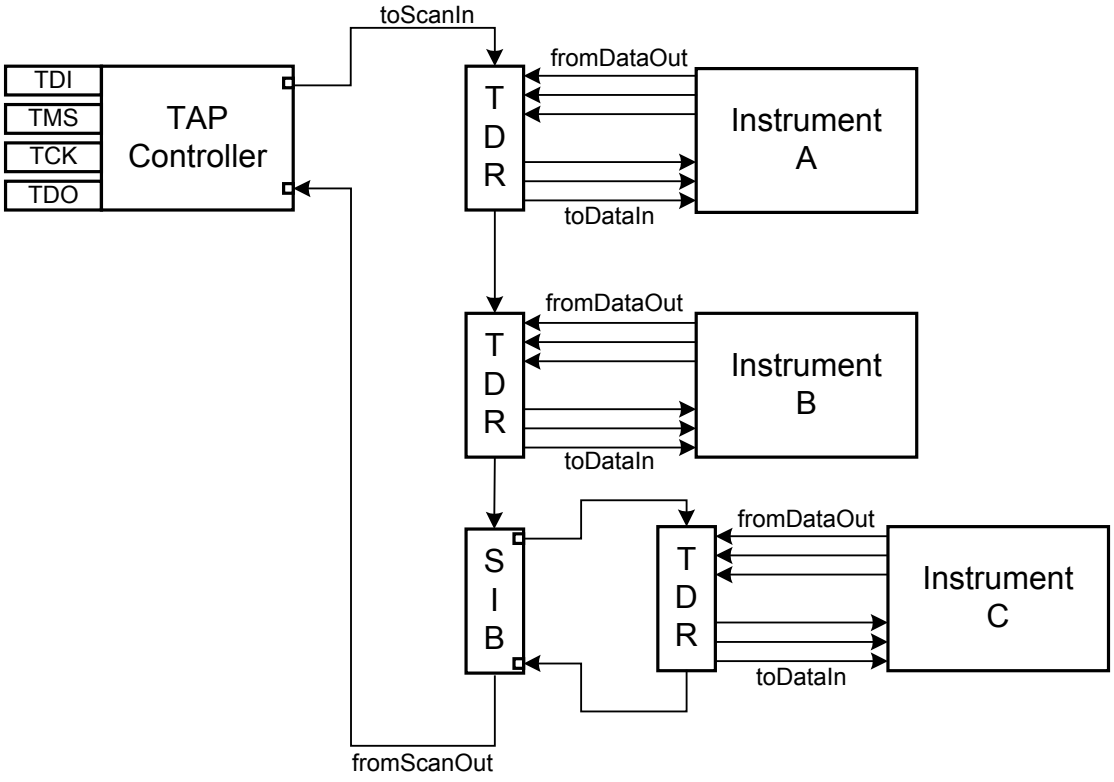


Figure 14—Conceptual multiple instrument variable-length scan chain IEEE 1687 network

An expanded view of an example SIB configuration is shown in Figure 15. This shows how a shift-update cell can be coupled in-line with and adjacent to a multiplexer to allow the selection of an alternate scan chain. The normal scan chain when the SIB is closed is from the TDI of the TAP to the shift-bit of the SIB and from the shift-bit of the SIB to Instrument A's TDR and then back to the TDO. The update-bit of the

SIB defaults to the de-assert value (e.g., Logic-0), which propagates through the ToSelect output signal from the SIB to disable the operation control signals of the TDRs attached to instruments B and C. If an assert value is shifted into the SIB's shift-bit and an update operation is conducted, then the ToSelect signal enables the Instrument B and Instrument C TDR control signals. When the IEEE 1149.1 state machine next enters the Capture and then Shift states, the Instrument B and C TDRs will operate and they will take the position of being in front of the SIB's shift-bit in the active scan chain. Note that placing the multiplexer in front of the shift register in the SIB configuration is a best practice in that it prevents the formation of long combinational paths with deep hierarchy, which could happen if the multiplexer were placed after the shift-bit. Also note that placing a Delay Bit after the Update Bit for the generation of the ToSelect signal is a best practice to prevent a race condition where the SIB receives its update and (prematurely) deselects the downstream TDRs before they have a chance to update. Please consult Annex F for detailed explanations and design guidance for implementation of SIBs.

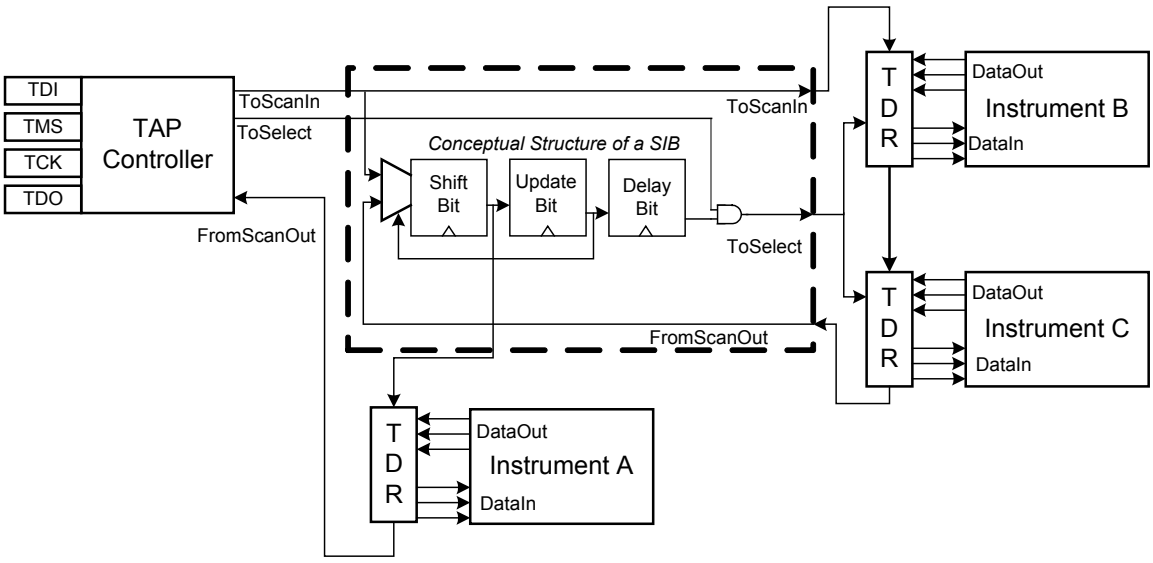


Figure 15—SIB configuration accessing an alternate scan chain

The in-line SIB contains both the scan multiplexer and the register bit connected to the control input of that mux, and this register bit is included in the same scan chain as the segment that it inserts (i.e., it is "in-line" with the segment). Figure 16 presents an alternative approach, where the SIB is exchanged with a mux only. The control input to the mux is connected to a TDR that is not part of the same scan chain as the elements it is selecting. This is referred to as remotely controlled scan mux architecture. The source of the control bit can be either another TDR (shown at the top of the figure where TDR0 produces Select1A to place TDR1A into scan chain 1) or a decoded instruction from the TAP (shown at the bottom of the figure where Select2A places TDR2A into scan chain 2).

Both the locally controlled (SIB) and remotely controlled scan mux configurations illustrate the concept of "opening" a network. The SIB configuration only adds a scan segment after the appropriate scan mux control value has been shifted in and then updated. For example, in the IEEE 1149.1 FSM, the sequence from SelectDR-to-UpdateDR (known as a ScanDR) would be used. In that context, an initial ScanDR is required to open a SIB, then subsequent ScanDRs can operate the newly opened scan chain. Opening nested SIBs is a sequential process: a SIB nested deep in the network cannot be operated until all the previous SIBs gating its access have been opened. The act of opening or closing a SIB is known as a "configuration operation" and is referred to as "reconfiguring the network."

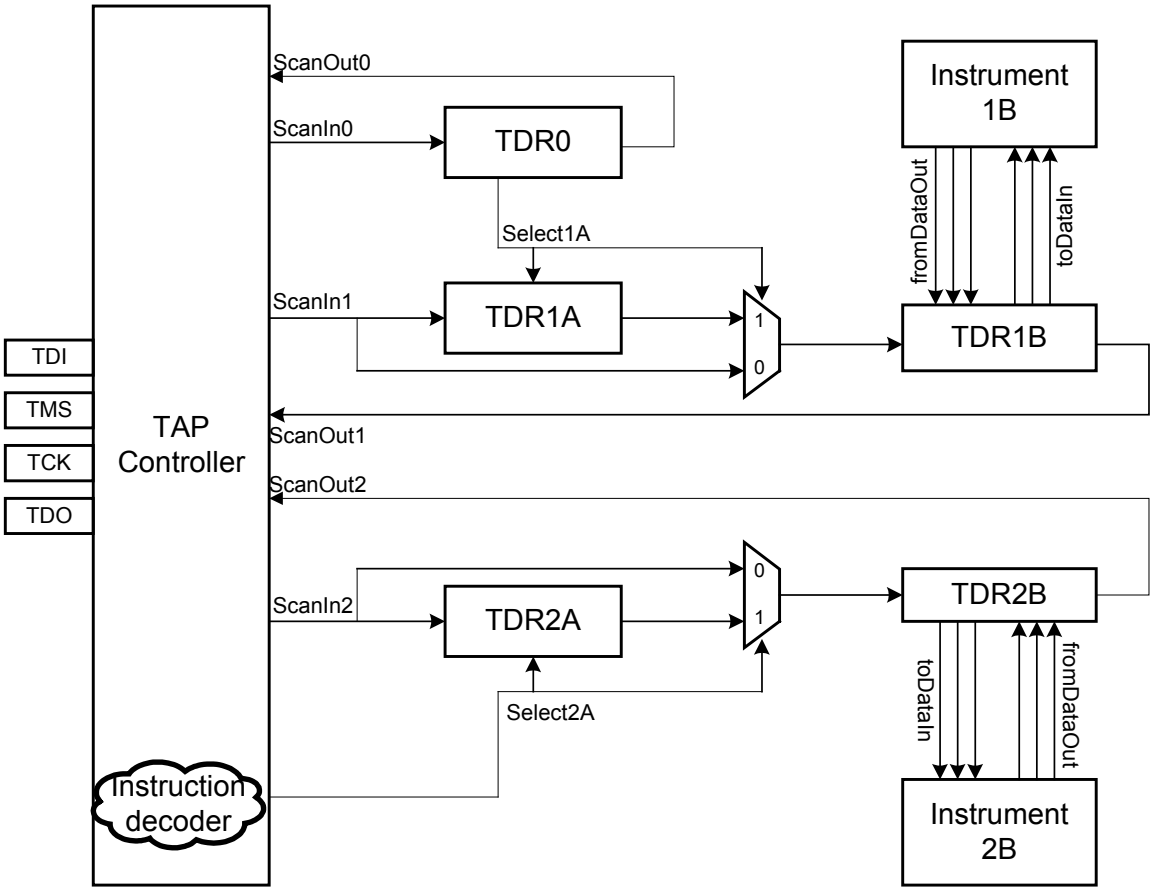


Figure 16—Remotely controlled scan muxes: two variants

Figure 17 shows an example of an IEEE 1687 network that has two levels of hierarchy—there are two SIBs in level one that make a 2-bit scan chain from the TAP Controller's TDI to TDO. When the first SIB from TDI is opened, then the TDR Interface to instrument A is accessible as well as the SIB that provides access to instrument B. This SIB is in level two and can be opened when another ScanDR is conducted after the level-one SIB is opened. Closing a SIB can also be done by conducting a configuration operation, and placing the de-assert value in the SIB after an update, or it can be done by asserting a reset operation. The SIBs will revert to the closed configuration when a reset occurs. Note that when a SIB closes, but not in conjunction with a reset, it de-asserts the ToSelect signal from the network segment to which it provides access. This in turn "freezes" that segment of the scan chain to its current value (making the segment persistent). To change the value in the segment of the scan chain requires either a reset or opening the SIB and changing the value through a capture or scan-in operation followed by an update.

Another connection option is to swap one segment of an active scan chain for another. Each scan chain segment may be of the same length or they may be of different lengths; there is simply a trade-off of access time versus access length. This concept can be expanded to allow many mutually exclusive selectable scan chains to exist in parallel with, for example, one or more instruments contained within the active scan chain at any given time.

Figure 18 shows a conceptual example of an IEEE 1687 network configuration that supports a split, mutually-exclusive, selectable scan chain. An in-line ScanMux control bit (S1) may be used to select the TDR associated with either Instrument A or Instrument B; note that the default configuration is required to be defined for when the network is first accessed or after a reset operation occurs.

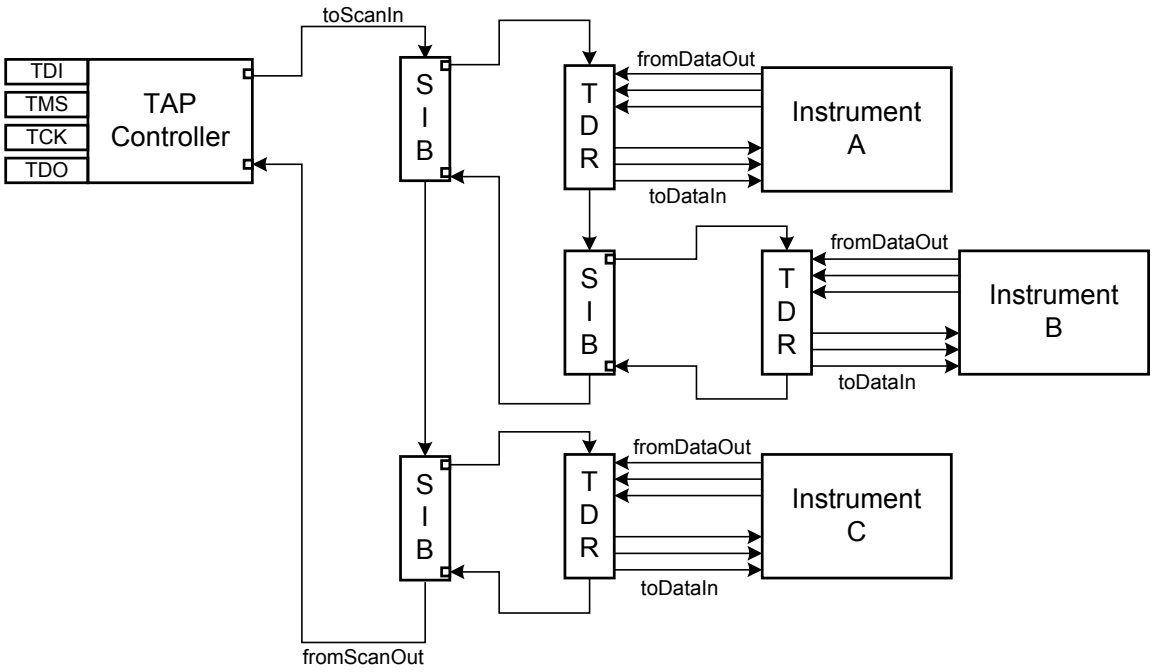


Figure 17—Multiple SIB configuration with hierarchical depth

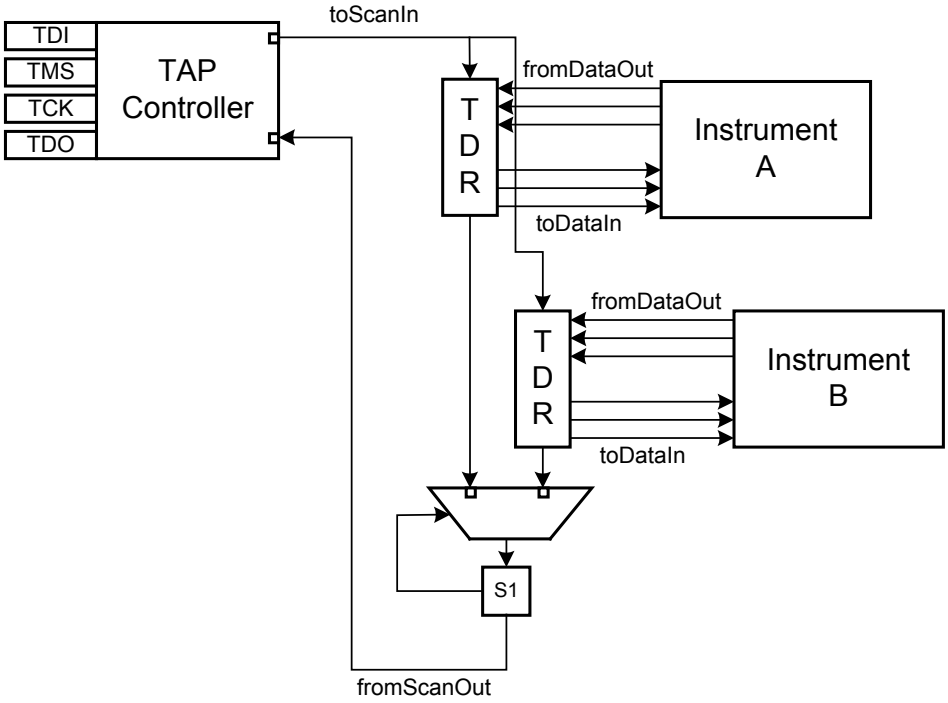


Figure 18—Conceptual multiple instrument split scan chain IEEE 1687 network

Figure 19 shows a combination of both techniques that allows instrument C to be included with either instrument A or instrument B depending on the state of the update-bit in the ScanMux control bit S1 and in the SIB configuration.

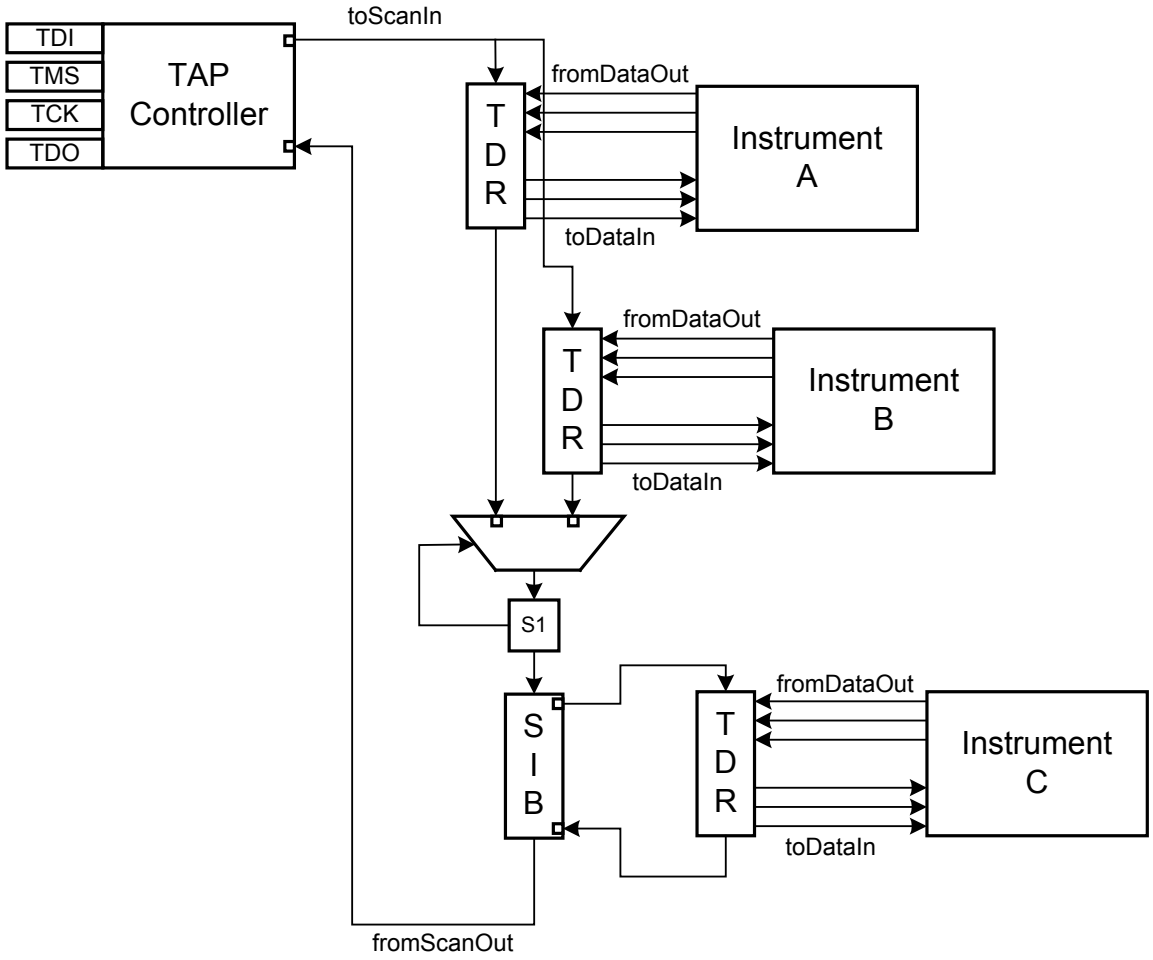


Figure 19—Conceptual multiple instrument swap and add IEEE 1687 scan chain network

5.5.2 Broadcast scan

Scan chains need not be limited to point-to-point serial connections. The concept of broadcast scan, where a point in the scan chain fans out to drive multiple downstream scan segments in parallel, is also supported in IEEE Std 1687. This configuration allows a reduction in the time spent loading scan segments intended to receive identical data. Figure 20 shows an example of a scan broadcast configuration. The structure involves a group of instances, each of which has a uniquely identified scan interface, which share a common scan data input and can be selected simultaneously. That common scan input happens to come from a ScanRegister (marked “BC” in the figure) whose update stage controls the scan multiplexers, but that ScanRegister could just as easily have been at the end of the scan chain as the beginning, or even been on a completely different scan chain.

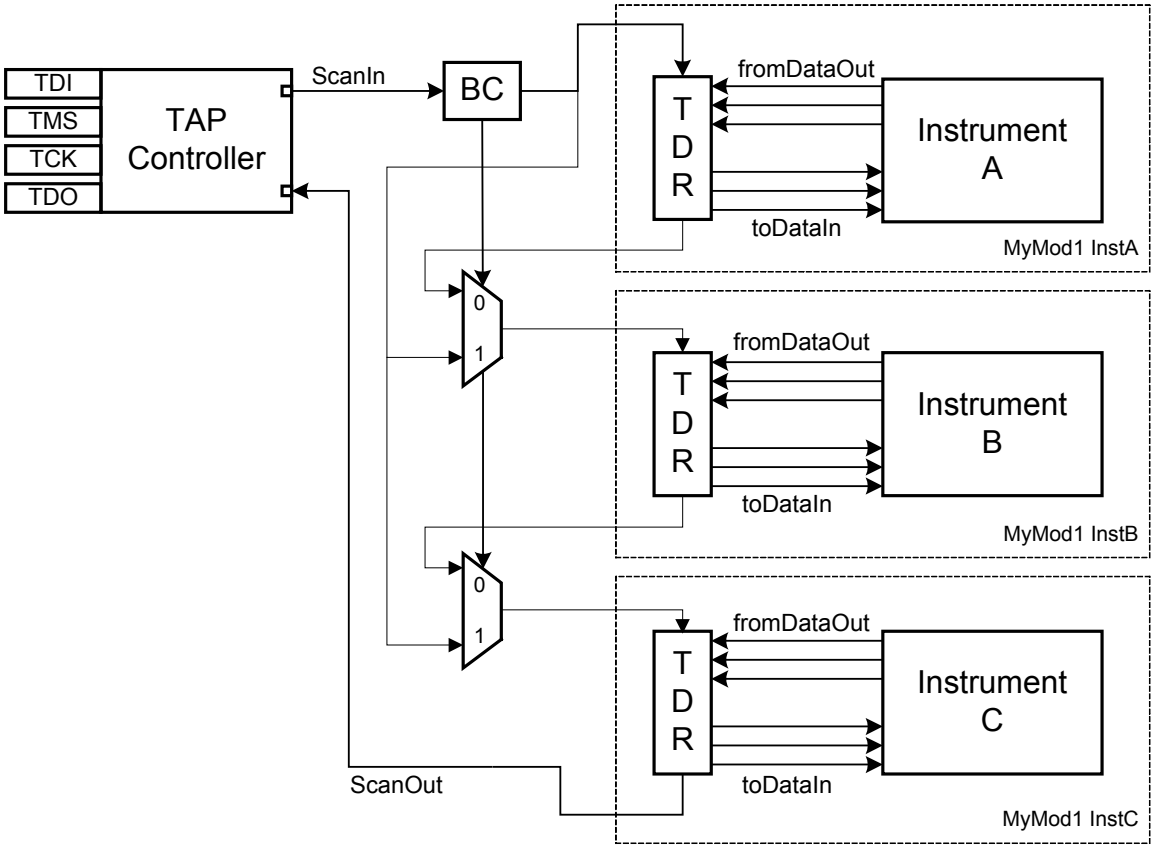


Figure 20—Broadcast scan example

It is important to note that the notion of scan broadcast applies only to data being written to (i.e., serially loaded into) scan chains. While broadcast mode is active, the data being read from (i.e., serially unloaded from) a scan broadcast group comes from one and only one of the instances in the broadcast group (Instrument C in the example in Figure 20, since it happens to be the one connected to be on the active scan chain when the broadcast control bit BC is active). When broadcast mode is not active, each of the instances is accessible for read operations (in one long chain of Instrument A, Instrument B, Instrument C in Figure 20). Note that the scan broadcast configuration in this example has some unusual features (e.g., there is no capability to read each instrument individually). A more common broadcast scan configuration is shown in Annex E.

Each of these IEEE 1687 network configurations presented allow different engineering trade-offs to be made and different scheduling options to be taken. These examples illustrate just some of the IEEE 1687 network configurations that are allowable IEEE 1687 networks.

5.6 Test data register

The IEEE 1149.1 Test Controller (TAP Controller) is just one possible controller that may operate an IEEE 1687 network. If used, it naturally generates the signals required to control, configure, operate, and observe the TDRs as originally shown in Figure 10. The TDRs provide elements that conduct both the network control and configuration through ScanMux elements and the control and observation of instruments through the instrument interface element. These TDRs are very similar to some of the defined IEEE 1149.1 cells but are a much simpler subset in that they only need to conduct the Read, Write, or Select operations.

An example of a conceptual 3-bit Write-Only TDR, with only the data-path elements and data signals shown to indicate how the serial data flows through the system and is applied to the instrument, is represented in Figure 21. Note that the scan chain numbering is generally from zero and is indexed with zero being at the ScanOut end and incrementing 1 integer per bit until the ScanIn is reached. The ScanIn signal shown feeding Shift-Bit-2 is ultimately sourced from the TDI signal from the chip's TAP when an appropriate IEEE 1687 network instruction is installed and asserted in the IEEE 1149.1 Instruction Register (IR). In a conventional IEEE 1149.1 operation, the scan data flows from the ScanIn toward the ScanOut by one bit on each rising-edge of TCK when the IEEE 1149.1 State Machine is in the ShiftDR state. The scan operation creates data ripple on the output of the Shift-Cells as the data moves through the serial scan register; however, the Update-Cell prevents the rippling scan data from being seen on the parallel toDataIn signals. The data in the Shift-Cells is transferred to the Update-Cells on the falling-edge of TCK when the IEEE 1149.1 State Machine is in the UpdateDR state. When this data transfer occurs, the toDataIn signals reflect the logic values contained within the respective Update-Cells. The sequence of a shift-in operation followed by an update operation constitutes a write operation if the update bits are connected to an instrument interface, and constitutes a select operation if the update bits are connected to other network elements such as multiplexers or gating functions

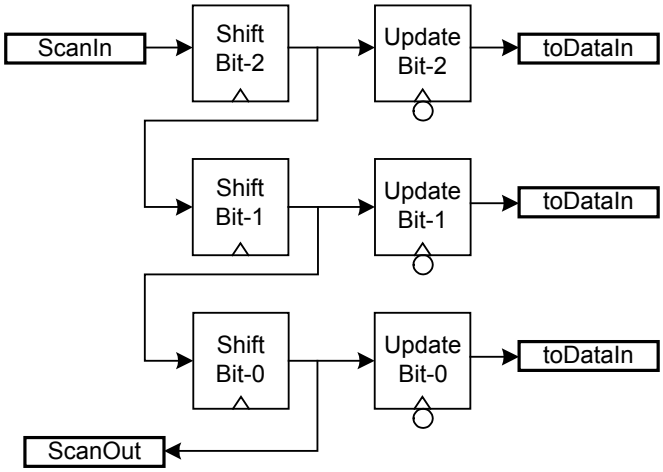


Figure 21—Example conceptual diagram of a Write-Only TDR

A conceptual example of a 3-bit Read-Only TDR is shown in Figure 22. Only the scan data path elements and instrument data signals shown to indicate how the serial data flows through the circuit and is captured from the instrument. The Read-Only TDR is comprised only of Capture-Shift bits with no Update bits. The Read-Only TDR conducts a read operation in two stages. In the first stage, instrument data that is present on fromDataOut signals is captured on the rising-edge of TCK when the state machine is leaving the Capture state. In the second stage, the captured data is shifted out for observation. The ScanIn signal shown is ultimately sourced from the TDI signal from the TAP when an appropriate IEEE 1687 network instruction is installed in the IEEE 1149.1 Instruction Register (IR). The capture operation occurs before the shift operation in the IEEE 1149.1 state machine. In a conventional IEEE 1149.1 TDR scan operation, the scan data flows from the toScanIn toward the fromScanOut by one bit on each rising-edge of TCK while the IEEE 1149.1 State Machine is in the ShiftDR state.

A conceptual example of a 3-bit Read-Write TDR is shown in Figure 23. Only the scan data path elements and instrument data signals shown to indicate how the serial data flows through the circuit and is captured from or applied to the instrument. The Read-Write TDR conducts a read by capturing instrument data presented on fromDataOut signals and then shifting the captured data out—this same shift operation can bring in new data to be written as a simultaneous shift-in. Figure 23 also shows that the concept that Write-Only bits, Read-Only bits, and Read-Write bits may be mixed in the TDR.

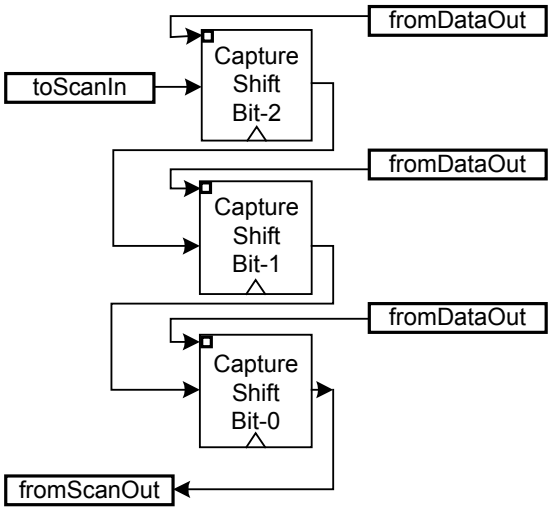


Figure 22—Example conceptual diagram of a Read-Only TDR

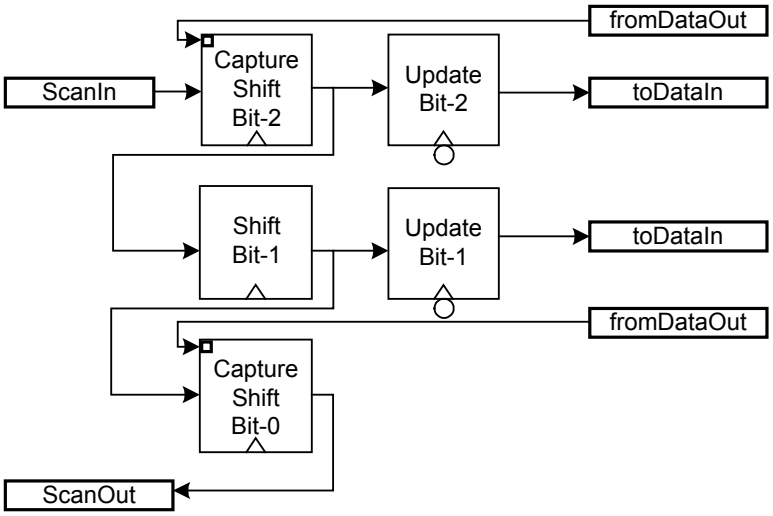


Figure 23—Example diagram of a Read-Write TDR

An instrument interface does not need to be the exclusive controller of an instrument. Functional signals may also access and operate instruments. A TDR that supports a functional connection is shown in Figure 24 where either bit-2 can drive a data input to the instrument or a functional signal ("Func_In") can drive the data input. In IEEE 1149.1 terms, there would be an instruction encoding in the instruction register that selects whether a TDR addresses the instrument or allows functional signals to pass through (e.g., the boundary scan register). An IEEE 1687 network allows in-line data-multiplexer select bits so the data passing through the scan chain may configure the very same scan chain. This is one of the main differences between older versions of IEEE Std 1149.1 (-1990, a-1993, b-1994, -2001) and both IEEE Std 1687 and the new IEEE Std 1149.1-2013. The older versions of IEEE Std 1149.1 always required traversal of the IR side of the TAP state machine to reconfigure the active scan chain.

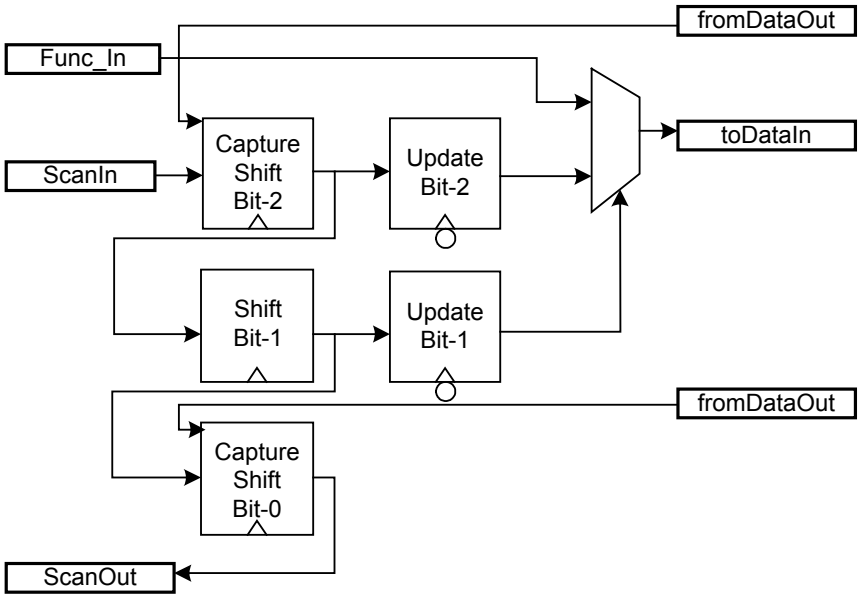


Figure 24—Example 3-bit Read-Write TDR that allows direct functional access

The TDR can be made of Read-Only bits, Write-Only bits, and Read-Write bits. TDRs supporting such bits are required to be able to conduct the capture, shift, and update operations when directed to do so from either the operation of the raw control signal or the TAP state-machine. The ability to support these functions is facilitated by the delivery to each TDR bit of both data, such as the ScanIn and ScanOut discussed earlier, and the control signals, as shown in Figure 25. The control signals line up with the various required operations: the ability to shift data from ScanIn to ScanOut is enabled by a shift-enable control signal (ShiftEn); the ability to capture data is enabled by a capture-enable control signal (CaptureEn); the ability to present and apply data is enabled by an update-enable control signal (UpdateEn); and the ability to reset the apply portion of the bit is enabled by a reset control signal (and that same signal can be used to optionally reset the shift-capture bit as well as the mandatory reset of the update bit). The actions are all synchronized to the rising or falling edge of the test clock.

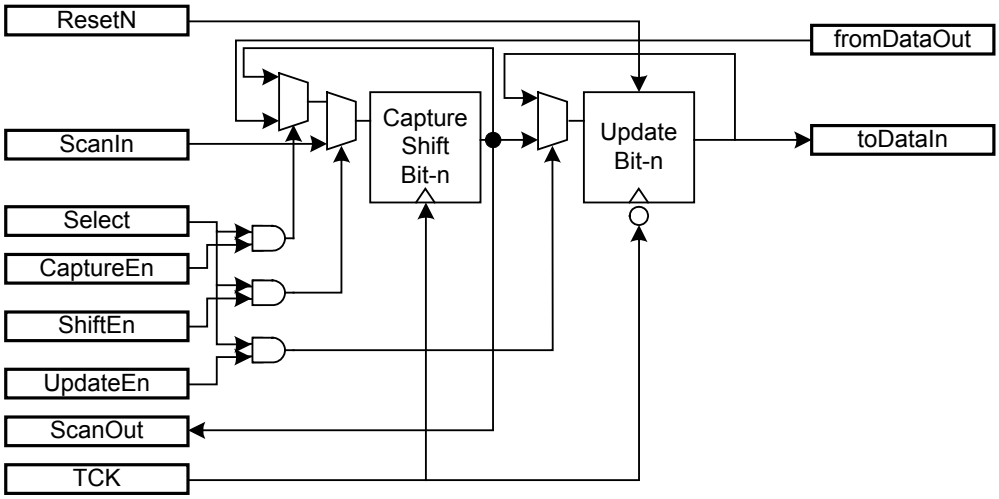


Figure 25—Example Read-Write TDR bit with control signals shown

One of the main requirements of operation of a TDR is that the data output bits of the TDR may change only when the TDR is part of the active scan chain—when not actively selected, but not in reset, the Update stage of the TDR should be "frozen" (resulting in no changes). The Capture/Shift stage may also freeze when not selected, but is not required to do so—the requirement is only for the Update stage. This requirement is implemented by the use of a selection signal (Select) that when asserted allows the control signals of the TDR to become active. There are many different methods to ensuring that the TDR is enabled when the Select signal is asserted and also ensuring the TDR is frozen when the Select signal is de-asserted. One typical implementation is to gate the control signals for shift, capture, and update (effectively gating the data operations, as shown by the AND gates in Figure 25), or to gate the TCK clock signal. In addition, the gating and distribution of the clock and control signals may be placed at a very high level such as the source of the selection signal (a WIR instruction, an IR instruction, a SIB, or a scan register driving a ScanMux control bit), or the gating may be applied local to the TDR or TDR bits where the control signals or clock are distributed as a tree across the chip.

5.7 Local reset

Many devices include a feature that performs what is known as a Global Reset of the test infrastructure. For example, in many IEEE 1149.1-controlled devices, assertion of the IEEE 1149.1 TRST* signal or entry into the TLR state of the TAP state machine generates a reset, which places a known state in the update-stage of some set of user-chosen TDR bits, optionally places a known state in the shift-side of some set of user-chosen TDR bits, and connects the IDCODE Register in between TDI and TDO. This type of reset is used to place a known state into the IEEE 1149.1 TAP Controller and a portion (perhaps the entirety) of the attached register architecture and is thus referred to as a Global Reset. The two different reset mechanisms (TRST* or TLR) may have different effects on the TDRs, depending on how the associated signals are connected to the reset ports of the TDRs.

A Global Reset may not be optimal when using the IEEE 1149.1 TAP Controller for reset control of an IEEE 1687 network. An IEEE 1687 network may be made of several different sub-networks that may be delivered by different instrument and IP providers. Each sub-network may optionally be placed behind an IEEE 1687 SIB construct so it may be accessed and treated independently. For example, multiple cores may be delivered with IEEE 1500 wrapper serial ports (WSPs) and each of these IEEE 1500 WSPs may be accessed by a unique SIB on a scan chain in a serial access network, as illustrated in Figure 26. To further facilitate independence and portability, each IEEE 1500 WSP's SelectWIR signal may be generated by an in-line ScanMux control bit (a shift-update TDR bit), labeled SWIR in Figure 26.

In this context, the impact of applying a Global Reset can be disruptive. Specifically, a Global Reset event from the TAP would close the SIBs that access the IEEE 1500 units and would force all IEEE 1500 wrapper architectures to a known reset state, regardless of whatever active state was in operation (assuming that reset forces the SIB configuration to a closed state and that the IEEE 1500 implementation defaults to the bypass instruction). In other words, a Global Reset may be useful for avoiding a scan operation that installs a default value into a particular (resettable) register, but it comes at the cost of also placing the associated default states into all the other (resettable) network registers.

A better and more useful configuration is to allow the IEEE 1687 network to include the ability to reset portions of the active or inactive scan chain without resorting to the Global Reset. This type of function is known as a Local Reset. The following discussion describes the use of circuitry to generate a local reset, but it is important to note that other types of circuits could be designed to suppress a global reset (see 6.4.6.13).

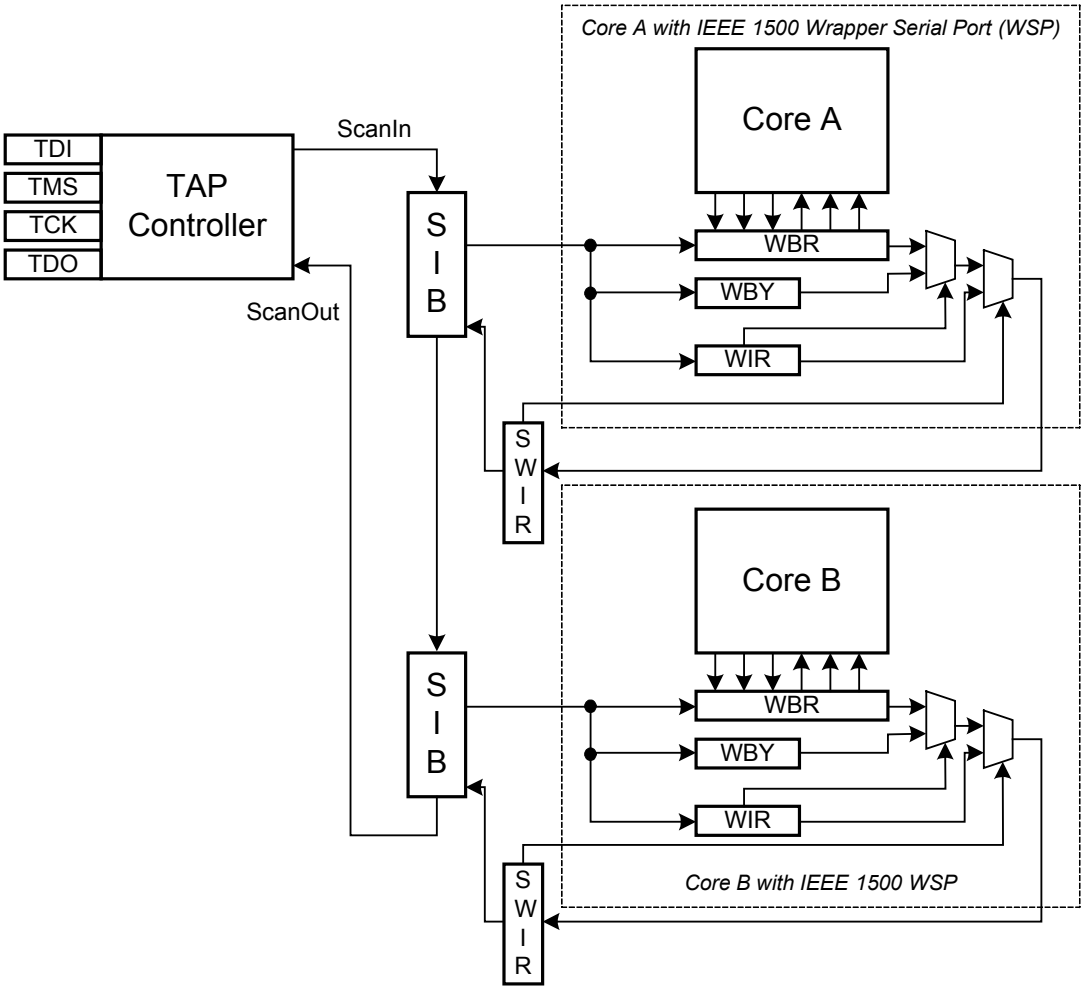


Figure 26—Example 2-SIB architecture with inline ScanMux control bit for SWIR

For example, the two-SIB architecture shown in Figure 27 may include another pair of control bits (Scan Registers labeled "LRA" for "Local Reset Assert" downstream from each of the SIBs) that, when asserted, would create the Local Reset signal via logic (labeled "LR gen" near each LRA bit). The Local Reset assertion would occur when an assert value is in the shift stage of the LRA register and the IEEE 1687 control signals generate an update operation. Since the update-side of a TDR-Bit normally holds state from one update assertion to the next update assertion (the next pass through the DR side of the IEEE 1149.1 FSM, for example), it is stipulated that the assertion of a local reset is required to not prevent the capture or shift operations of any scan registers with which it is associated. This stipulation enables the local reset to be de-asserted by shifting in new data and performing another update operation. It is common for reset to affect only the update stage of a scan register, so this stipulation is usually trivially satisfied. However, if the shift stage of a scan register is also affected by reset, there is a potential for the active scan chain to be broken by being held in reset (since both capture and shift operations occur between any two update events). To prevent this, a reset affecting a shift element is required to resolve itself to a de-assert state before the next shift event. There are several physical implementations, using several different possible methods that allow this capability, and since any implementation is valid, this type of shift-stage reset can simply be specified as "self-clearing." One method of self-clearing, for example, is to permanently place the update stage of the local reset register in its reset state and selectively block the reset during the update operation; if the local reset register is written with the Local Reset de-assertion value during update, then nothing changes. However, if the local reset register is written with the Local Reset assertion value during update, then a Local Reset pulse will be created and this pulse can then place the target subset of IEEE 1687 network into a known defined state.

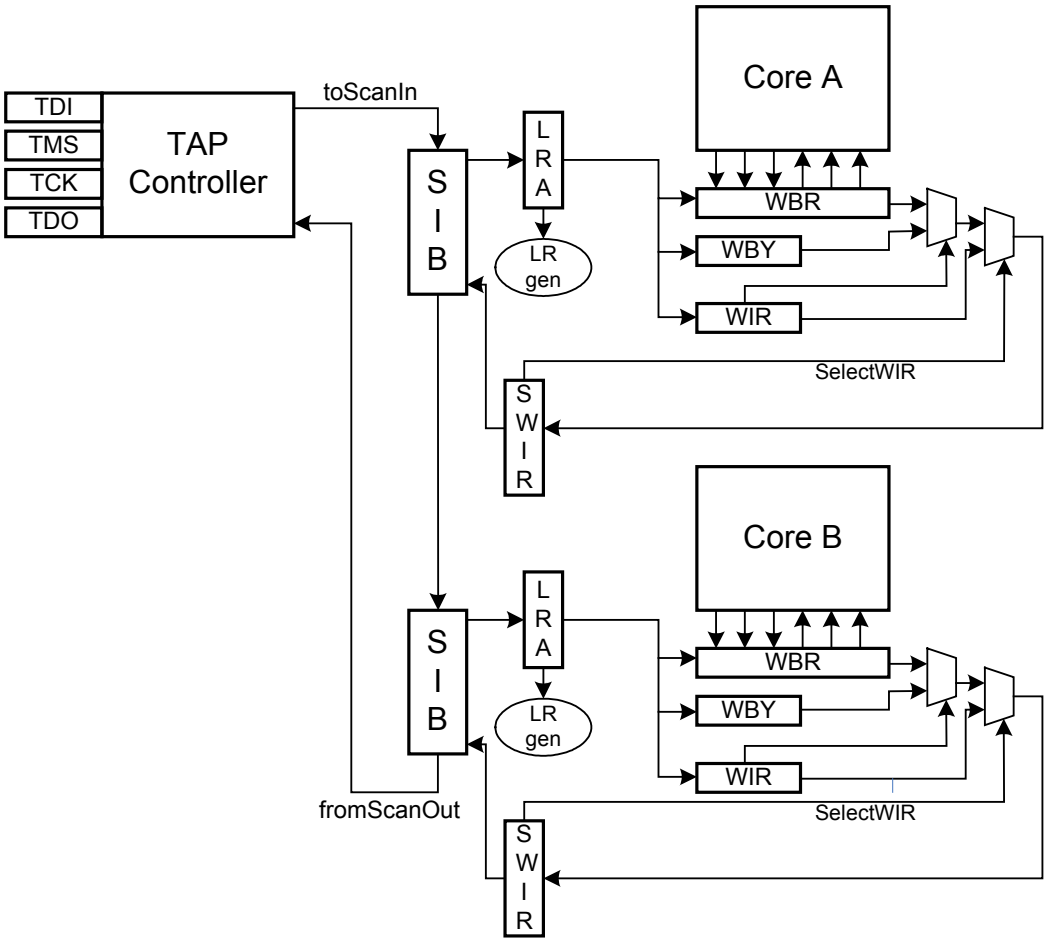


Figure 27—Local reset signals from inline control bits

Note that all IEEE 1687 network elements react to their module's Global Reset when it asserts, and those with a defined Local Reset also react when the Local Reset asserts; therefore, the reset input to the individual elements that make up a TDR that are also targeted by a Local Reset is the OR (assuming active-high signaling) of the module's Global Reset and the Local Reset (Figure 28).

In a complex IEEE 1687 network, one or more Local Resets may be generated to control different sections of the IEEE 1687 network.

Figure 29 shows an in-line control bit that can create the Local Reset for the previous bit in the scan chain. The Local Reset is generated on the falling edge of TCK as the IEEE 1149.1 FSM passes through the UpdateDR state. This local reset propagates to the target bit immediately, but arrives to the control bit's update stage through a delay operation. The only criterion for the delay is that its effect is complete prior to the next capture event. This feedback of the local reset signal results in a self-clearing pulse.

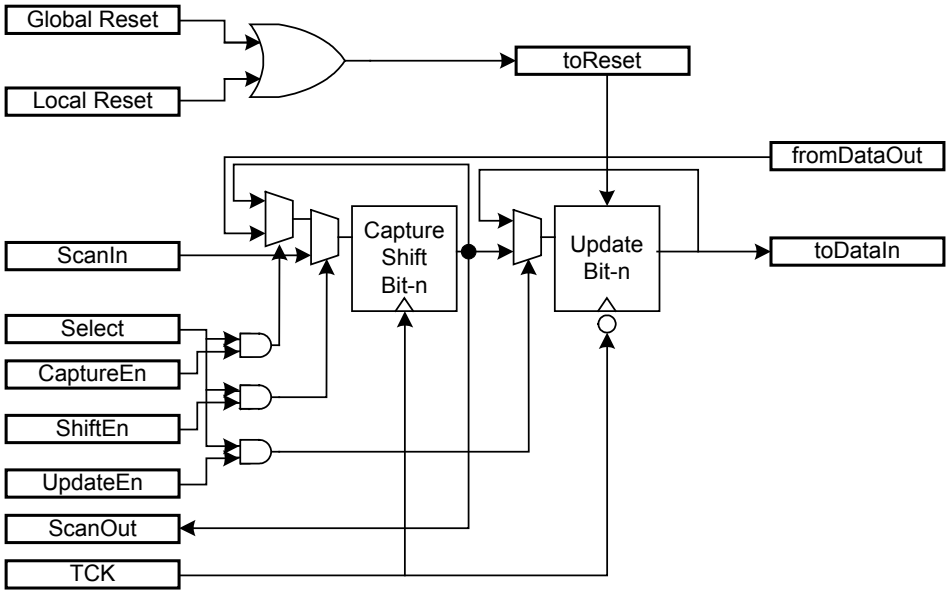


Figure 28—Example of merging Local Reset with Global Reset

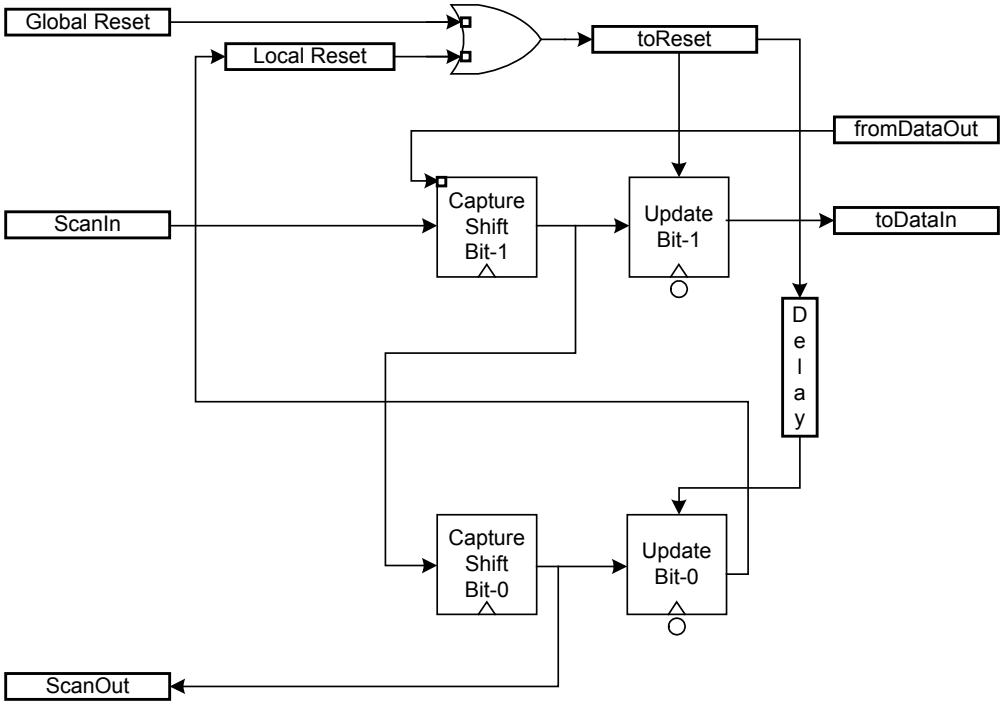


Figure 29—Control bit that generates and distributes a Local Reset

5.8 Delivery and integration of instruments

There are two basic methods for acquiring and including instruments within an IC design—one method is to receive IP cores that represent instruments; the other method is to design instruments as part of the chip development effort. One of the main purposes of IEEE Std 1687 is to enable the design and delivery of portable and reusable instruments that are easy to sign off locally and then integrate into an overall chip design.

IP Cores may be developed in-house or by third-party providers and may be delivered in various stages of completeness. Generally, IP Core providers will deliver the core and make it optional as to whether there is an access and test interface or test wrapper. A "raw" instrument is delivered without a test wrapper, a "wrapped"" instrument already includes a test wrapper, and a "tapped" instrument already includes an embedded IEEE 1149.1 TAP controller (see Figure 30). During integration, an instrument is connected to its control structure (the access, configuration, operation, and observation interface). The appropriate control structure is a function of how the instrument was delivered (raw, wrapped, or tapped) and the type of network and device interface desired by the integrator. The integration process may follow the methodology of wrapping all of the raw instruments first and then connecting them together as wrapped instruments, or the integration may require the creation of an on-chip access network and then plugging the instrument into this network.

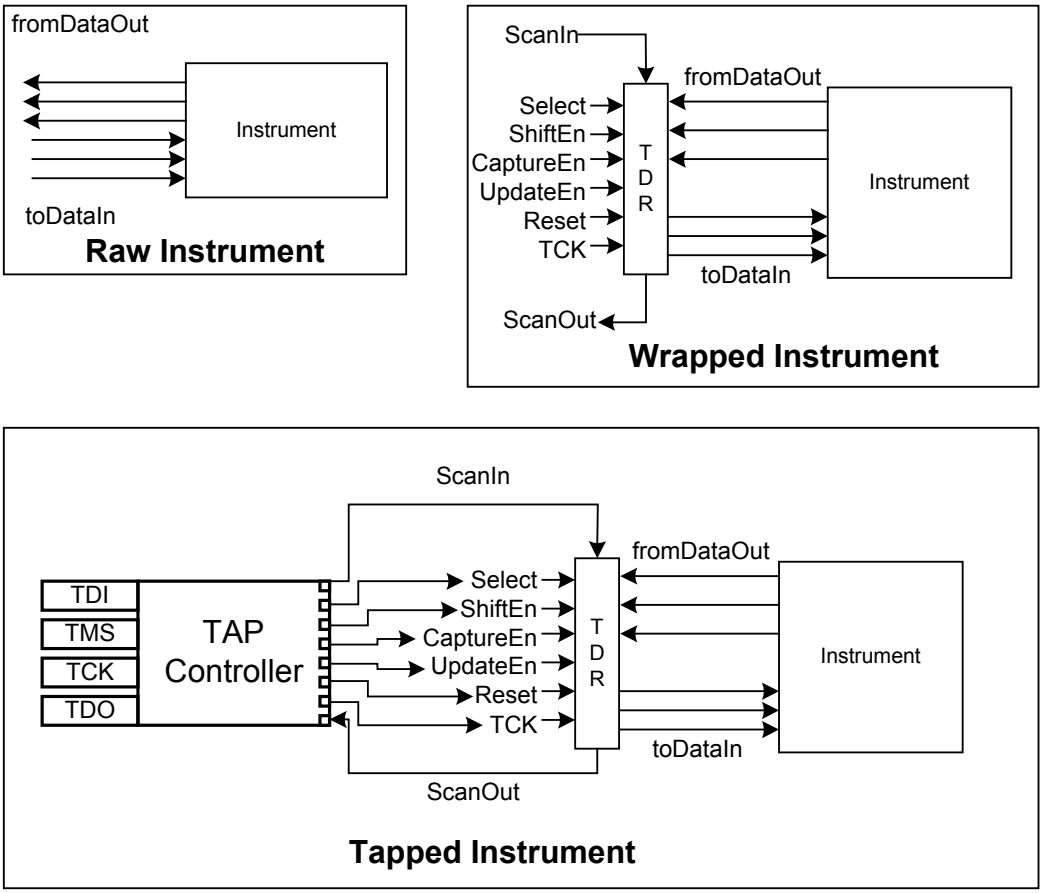


Figure 30—Examples of instrument delivery options

If the instrument core is delivered with an existing test wrapper, then this test wrapper is required to be integrated into the overall IEEE 1687 scan chain by connecting the operation signals (the signals that represent ShiftEn, CaptureEn, UpdateEn, Reset, and TCK) to a control source and by connecting the scan chain data signals (ScanIn and ScanOut) to the chip's pin interface through access selection logic. The test wrapper interfacing to any given instrument may be any IEEE 1149.1 compatible test wrapper such as a simple IEEE 1149.1 TDR, a complex IEEE 1500 wrapper with multiple instruction choices, or a specifically designed IEEE 1687 wrapper that includes scan chain management features.

In some cases, it may be advantageous for an instrument provider to deliver the instrument, or multiple instruments, in a configuration where a complete portion of an IEEE 1687 network may be provided and already organized. For example, a very complex structure such as a high-speed SerDes-to-LVDS PHY with multiple adjustment features could be delivered with the multiple PHY adjustment instruments already organized into a specific test management structure. In the cases of complex instrumentation, creation of the entire subsystem and including an embedded controller, such as an embedded IEEE 1149.1 TAP Controller (eTAPC) provides the most portable method and requires the least amount of specific instrument knowledge from the end user.

Each of these basic delivery configurations is described in ICL to meet IEEE 1687 integration needs. For a raw instrument, only the signals that are required to be wrapped need to be documented for IEEE 1687 integration purposes. It is expected that the instrument integrator will have all of the documentation (e.g., electrical specifications) needed to integrate and connect the instrument into the design, but to generate the IEEE 1687 network description, only the access, configuration, operation, and observation signals are needed. These signals are usually a subset of all of the signals that comprise an instrument. For a wrapped instrument, the documentation may only include the IEEE 1687 signals needed to access the instrument interface TDR; the instrument interface may be obfuscated by transferring the information about the raw instrument signals to the bits in the TDR.

Another delivery mechanism represents an instrument as a black-box where the IEEE 1687 instrument wrapper is identified, but there may be embedded scan chains within the instrument that are not publicly divulged. Any patterns delivered would be required to operate these scan chains by conducting iScan operations (see 7.9.12).

The delivery of a tapped instrument for inclusion in a network is addressed in the next subclause.

5.9 Embedded TAP controller

In some cases, an IP provider may deliver an instrument access network complete with an IEEE 1149.1 TAP Controller. When such a network is to be included within another chip, then the TAP Controller becomes an embedded TAP Controller (eTAPC). Figure 31 shows an IEEE 1500 wrapper architecture (which is a valid IEEE 1687 network) driven by TAP signals that have been changed to eTDI, eTDO, eTCK, and eTMS to represent that they are embedded.

The eTAPC may not be a complete and compliant IEEE 1149.1 TAP Controller that includes a bypass register, boundary scan register, instruction register and mandatory instructions, but instead may simply be the Finite State Machine (FSM) that operates using the IEEE 1149.1 TMS and TCK TAP signals brought to the eTMS and eTCK connections, respectively, with the purpose of generating the instruction-side operation signals to facilitate operation of an embedded instruction register, as shown in Figure 32.

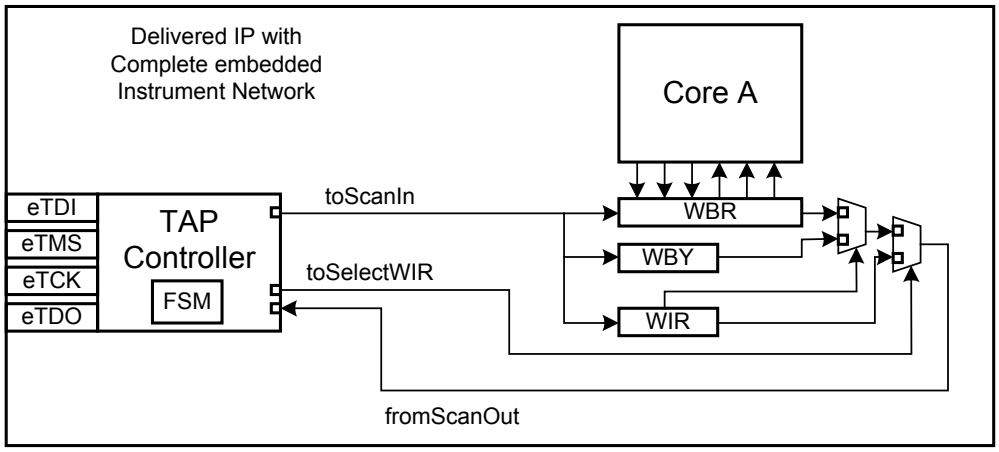


Figure 31—Example embedded TAPC to drive a network Instruction Register

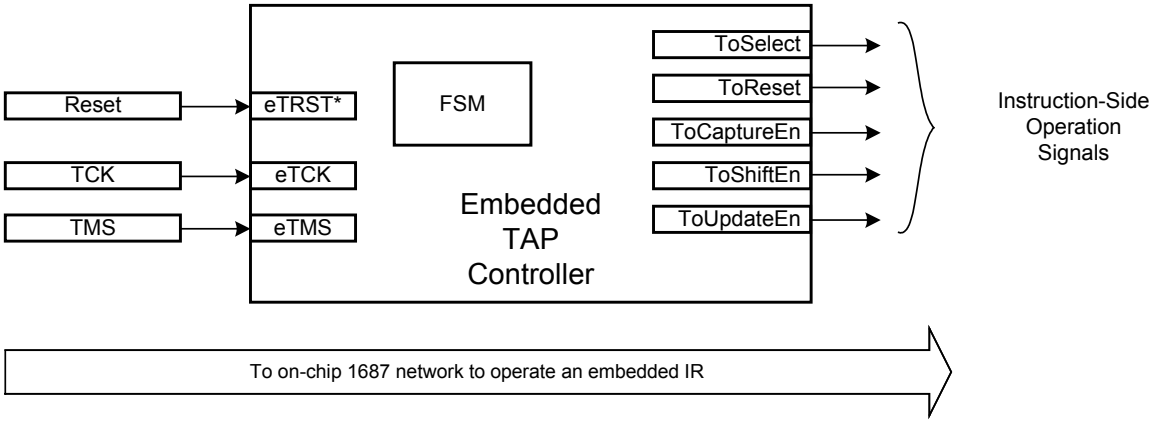


Figure 32—Example eTAPC to drive a network Instruction Register

The actual signals needed to drive the eTAPC and for the eTAPC to drive any embedded network instruction register depends on the type of physical implementation applied to the instrument access network. If the TCK and all of the Reset, CaptureEn, ShiftEn, and UpdateEn signals are routed as buses from the main TAP, and the TDI and TDO inputs are provided separately by the IEEE 1687 network, then the minimum signal connections required by the eTAPC are the eTCK and eTMS on the input-side (and possibly an eTRST* to synchronize the eTAPC's reset to the main TAP Controller); and the ToIRSelect on the output side. If the physical implementation is created locally using the eTAPC, then the ShiftEn, CaptureEn, UpdateEn, and Reset signals may be generated by the eTAPC (as in Figure 32). Note that the eTAPC needs to be connected to the IEEE 1687 network on the input side and so the IEEE 1687 signals of Reset, TCK, and TMS would be used as eTAPC inputs; this configuration has the advantage of needing to route and close timing on fewer signals (the three on the left side, versus the five on the right side) on the possible long connections from the device-level TAP to the TDRs.

An embedded TAP controller is required to exist as a distinct module in ICL so that it may be represented with the proper port functions. The top-level TAP controller for the device usually has BSDL and is usually not described in ICL; instead, it is only referred to in the AccessLink statement present in the top-level module of the ICL. This reference consists of identifying the BSDL entity name, the name of the instruction that selects the TDR associated with the IEEE 1687 network, and the name of the ScanInterface in that network. There is an exception where the top-level TAP controller does need to be described in ICL:

when it shares the TDO pin with one or more embedded TAP controllers via a scan multiplexer. In such cases, the AccessLink information is omitted.

The key concern with the eTAPC (sometimes also referred to as a *shadow-TAPC*) is that its FSM stay synchronized to the main TAP Controller's FSM that is used as the AccessLink. If the eTAPC is not following the same exact state mappings, then the shift function may not work (blocking the scan chain) and the capture and update functions will happen at the wrong times. By providing a reset input, the TAPC can be instantly synchronized to the main TAP Controller by conducting a reset of the main TAP Controller (place the main TAP Controller's FSM in the TLR state). If a reset input is not provided, then the chip-level TMS signal is required to be driven to a logic-1 for five TCK clock cycles to force the eTAPC into the TLR state simultaneously to the main TAP Controller entering into the TLR state.

Another configuration of an embedded TAPC is one that fully uses and generates signals to completely manage the access and use of an embedded instruction register (Figure 33). The complete IR-only eTAPC may support this configuration when an embedded core with a complete instrument access network or multiple IEEE 1500 wrappers, is delivered as a hard core with a frozen (non-modifiable) routing architecture. This type of eTAPC may include the TDI and TDO signals as well as the TMS and TCK, with an optional TRST* on the input-side. On the output-side, the complete IR-only eTAPC will generate all of the signals necessary to select and operate one or more embedded instruction registers whenever the eTAPC FSM is on the instruction-side. The minimum set of output signals would be: the ScanIn, ScanOut data signals; the TCK clock; the ToIRSelect or SelectWIR signals; and the ToShiftEn, ToCaptureEn, ToUpdateEn, and ToReset signals.

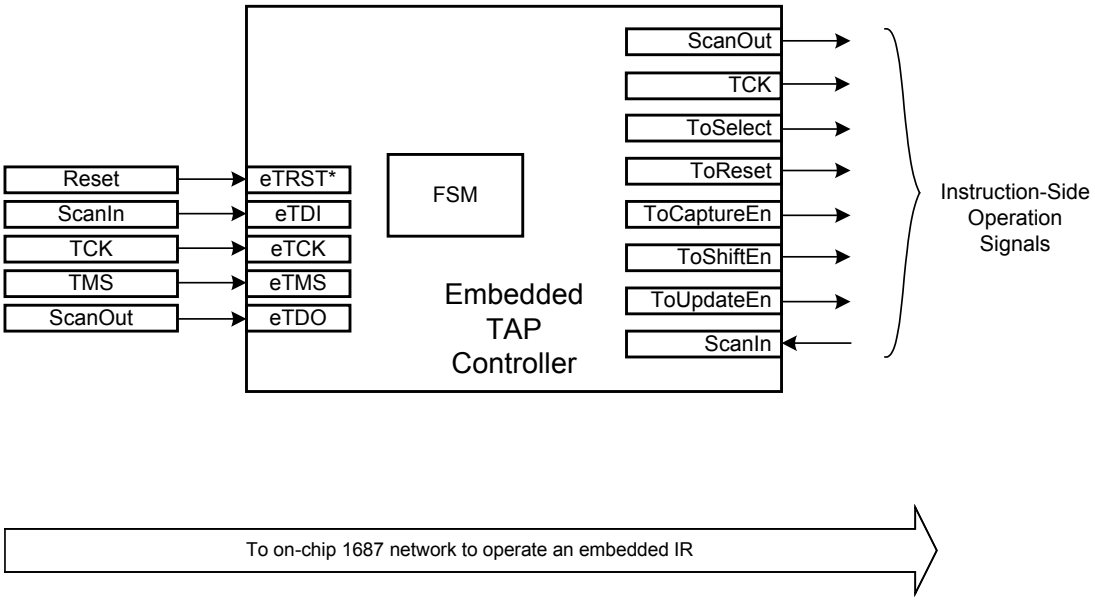


Figure 33—Another example eTAPC to drive an Instruction Register

An eTAPC may also be used as a complete TAP Controller that provides access, configuration, operation, and observation of a whole and complete embedded IEEE 1687 network; this situation may be required if the embedded IEEE 1687 network is delivered as a whole complete and portable hard IP core. For this case, the eTAPC may deliver IR-side signals and one or more groupings of DR-side signals to select one or more independent IEEE 1687 networks. As shown in Figure 34, this eTAPC can be connected to the IEEE 1687 network in the rest of the chip by providing the input signals: Reset (if supported); the serial data signals ScanIn and ScanOut; the test clock, TCK; and the FSM control signals TMS from the main TAP Controller that is the AccessLink for the IEEE 1687 network that accesses this eTAPC.

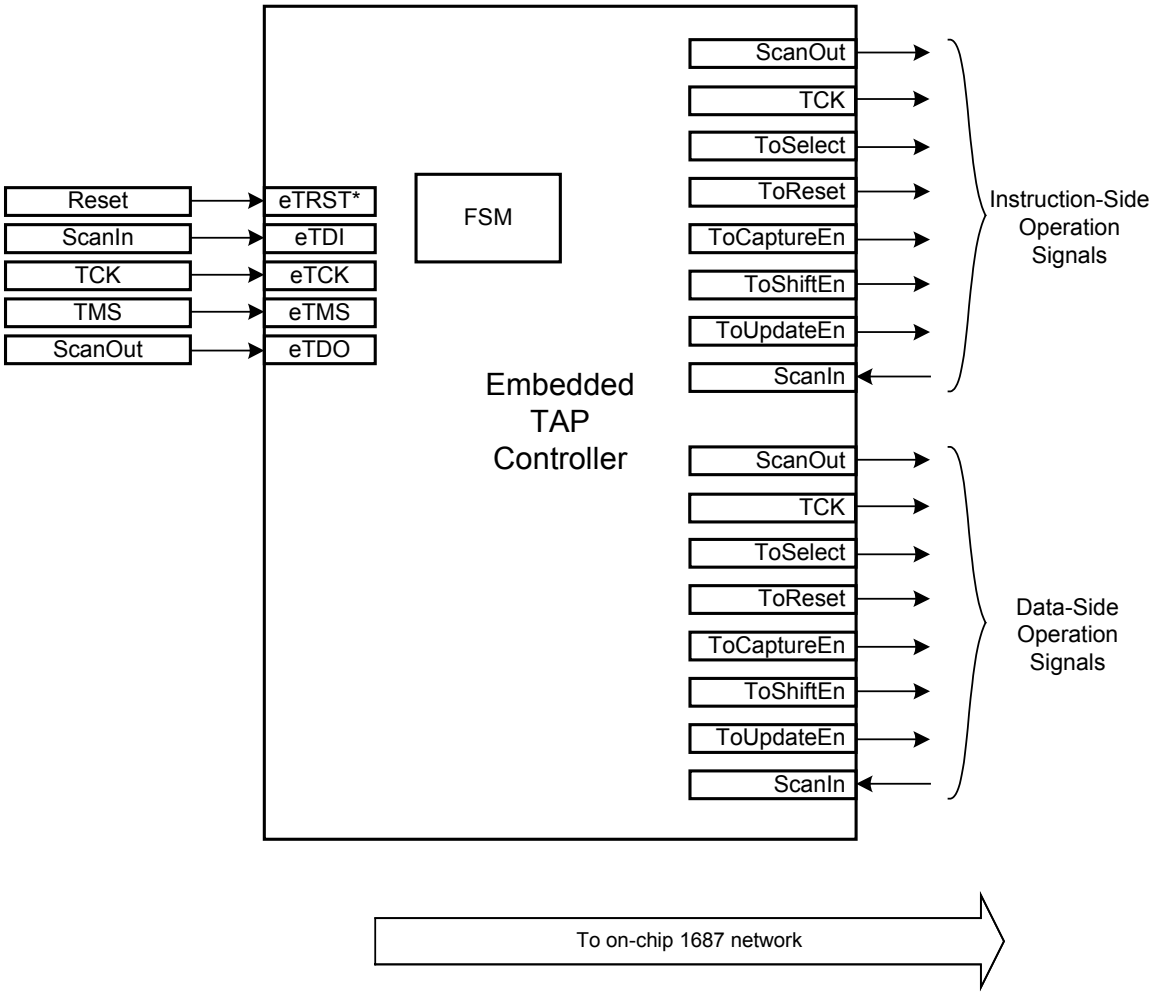


Figure 34—An eTAPC for driving a subset of an embedded IEEE 1687 network

All of the eTAPCs shown so far are always active and continually synchronized to the main TAP Controller. However, one of the advantages of the IEEE 1687 network is that portions of the network can be taken out of the active scan chain to manage the scan chain; or instruments and instrument interfaces that do not need to be operating can be left out of the active scan chain and can be in a quiescent configuration. As was explained with the individual TDR that may be accessed with a SIB configuration or a swap configuration, when not actively selected, the TDR freezes and holds state until it is either activated with an asserted Select signal or until it is set to a default state with a Reset signal (local or global). The same is true if an entire sub-network is sequestered behind a selection device. If an eTAPC is placed behind either a SIB or swap configuration, then it also remains frozen.

The most common method of freezing an eTAPC is to force the state machine into a parking state such as run-test/idle (RTI). The RTI-method is implemented by modifying the eTAPC signal interface with an AND-gate to allow a Select signal to enable or disable the eTMS signal (Figure 35). When the Select signal is asserted (logic-1), then the eTMS will directly take on the value of the main TAP Controller's TMS signal; when the Select signal is deasserted (logic-0), then the eTMS will remain at a logic-0. Since the Select signal changes state when the FSM is leaving the UpdateDR state, and if the logic value on the TMS signal is a logic-0 when the Select transitions to a logic-0, then the FSM will enter the RTI state and will remain there until the Select transitions back to a logic-1. The Select may be asserted by the main TAP Controller when leaving the UpdateDR state, and there should be a logic-0 on the main TAP Controller's TMS so that its FSM will also enter the RTI state—this will synchronize the eTAPC's FSM with the main TAP Controller FSM.

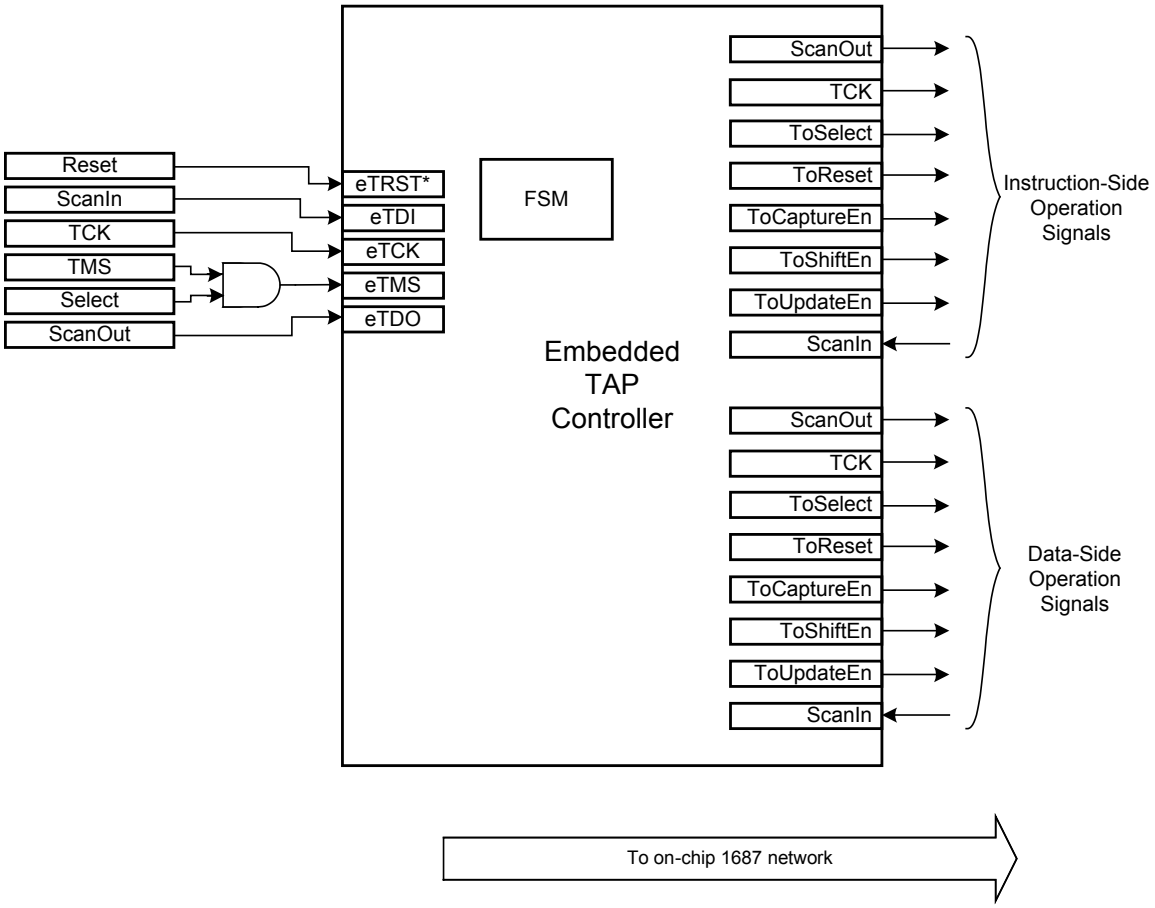


Figure 35—A parkable eTAPC for driving a subset embedded IEEE 1687 network

An alternate methodology is to force the eTAPC into a Reset state when not actively selected to be part of the active scan chain (Figure 36). This methodology is useful when the eTAPC may be part of a core that includes a low power mode. The Reset state requires parking the eTAPC FSM in the TLR state. For eTAPCs that support an asynchronous reset signal, the Select signal may be connected directly to the eTRST* signal, so that when the Select is logic 0, the asynchronous reset is asserted. For eTAPCs that do not support an asynchronous reset, the de-assertion of the Select signal is required to force the eTMS port to be held at logic 1; the IEEE 1149.1 FSM rules dictate that the FSM will enter the TLR state within five TCK cycles. Since the Select will transition when leaving the UpdateDR state, and if the leaving state is the RTI, then the TLR state should be entered in three TCK transitions.

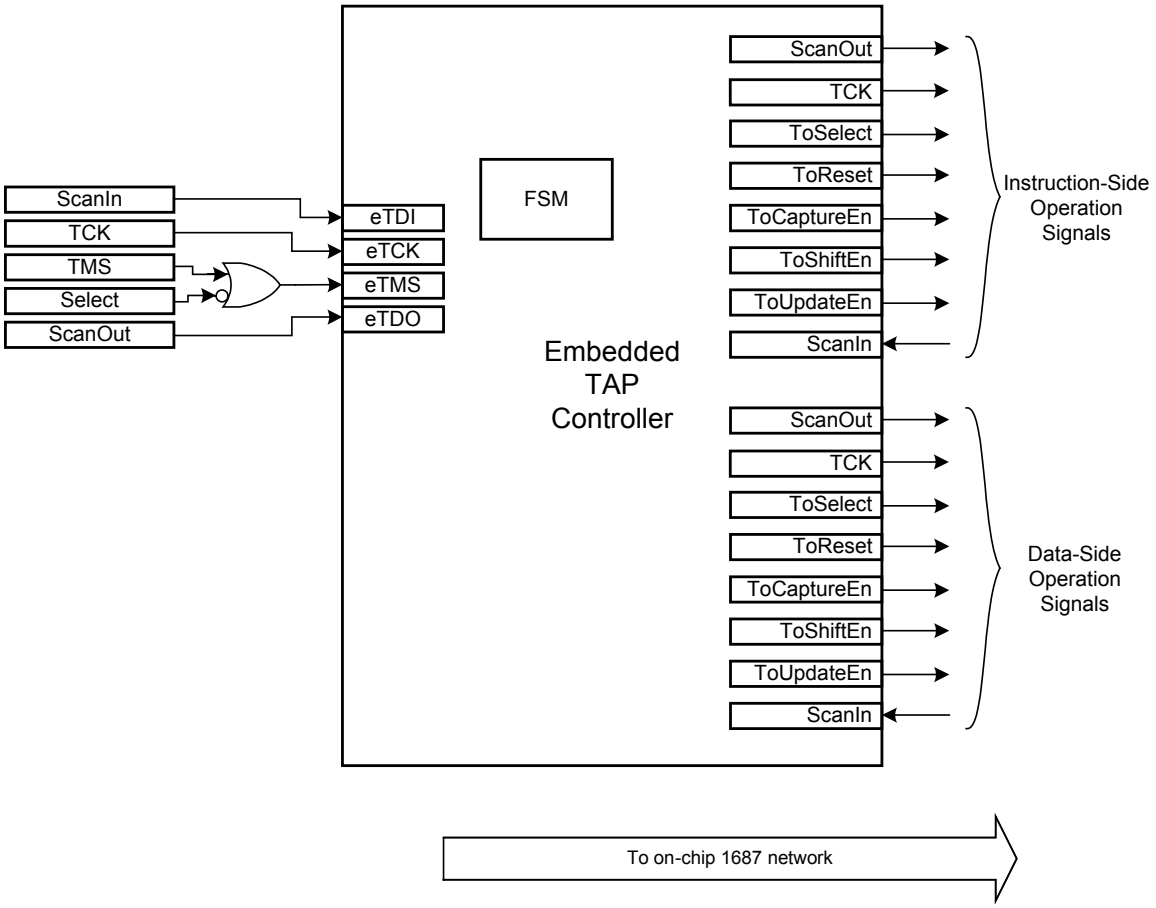


Figure 36—An eTAPC that parks in Reset when not part of active scan chain

5.10 Definitions of the structure of IEEE 1687 hardware architecture

The key terms used in the hardware architectural rules are condensed as follows for convenience.

— ***Component:*** A hardware entity that comprises the primitive building blocks of the network, including storage elements, multiplexers, and logic.

— ***Client interface***: The port functions that attach a module to a host interface in a network. The client interface consumes control signals from the host interface and exchanges data with it.

— ***Host interface:*** The port functions that attach a module to a client interface. The host interface provides control signals to the client interface and exchanges data with it.

— ***Instrument***: A module that has only a client interface.

— ***Module (IEEE 1687 module):*** A component containing a test access mechanism that provides communication with on-device instrumentation, an interface to the pins of a device, or other modules.

— ***Network:*** An interconnected set of modules bounded by instruments and external interfaces.

— ***Port function:*** A module port with a specific function relevant to the operation of an instrument, network, or interface; there are 26 such port functions in IEEE Std 1687 described in Table 2.

— ***Raw instrument:*** An instrument with only DataIn, DataOut, or Clock port functions in its client interface.

— ***Signal:*** A connection between two or more port functions of modules.

— ***Storage element:*** A bi-stable circuit element, such as a latch or a flip-flop, which retains state until certain enabling and/or clocking conditions are met that cause it to take on a new state.

There are two terms ("associated" and "controls") that are frequently used in the hardware rules later in this clause that are defined as follows:

— ***Associated:*** In the context of IEEE 1687 normative hardware rules, "associated" establishes that a signal or port controls or is controlled by another signal or port. A rule using this word will then describe a behavior resulting from that association.

— ***Controls:*** Contributes to establishing the data value or asserted or deasserted state of a port or signal.

The IEEE 1687 architecture is composed of port functions, signals, and components; these elements will be described in detail in the next three subclauses.

5.11 Port functions of a module

The access network is comprised of one or more modules, each module has one or more interfaces, and each interface has one or more ports. Each port has a specific function (henceforth referred to as a *port function*) that is pertinent to the access network. Table 2 lists the port functions associated with the client and host interfaces. The direction arrows indicate the flow of information with respect to the interface port: an arrow pointing toward the port name indicates that it is an input to the interface, while an arrow pointing away from the port name indicates an output from the interface. Note that there are four port functions (ScanIn, ScanOut, DataIn, and DataOut) that may appear on either the host or the client interface; all other port functions are uniquely associated with either a host or a client interface. Also note that, while not pertinent for this subclause, the active values of each of these port functions may be found in Table 5.

Table 2—IEEE 1687 module client and host interface ports

IEEE 1687 module				
Client interface		**Functions**	**Host interface**	
→	ScanInPort	Scan data to TDR	ScanOutPort	→
←	ScanOutPort	Scan data from TDR	ScanInPort	←
→	TCKPort	Test clock	ToTCKPort	→
→	SelectPort	Select this module	ToSelectPort	→
→	ResetPort	Reset this module	ToResetPort	→
→	CaptureEnPort	Capture control to TDR	ToCaptureEnPort	→
→	ShiftEnPort	Shift control to TDR	ToShiftEnPort	→
→	UpdateEnPort	Update control to TDR	ToUpdateEnPort	→
→	DataInPort	Parallel data to instrument	DataOutPort	→
←	DataOutPort	Parallel data from instrument	DataInPort	←
→	ClockPort	Functional clock	ToClockPort	→
→	TMSPort	Test Mode Select	ToTMSPort	→
→	TRSTPort	Test Reset	ToTRSTPort	→
		Select Instruction Register	ToIRSelectPort	→
→	AddressPort	Address		
→	WriteEnPort	Write Enable		
→	ReadEnPort	Read Enable		

These port functions connect to signals (described in the next subclause) according the following rules:

Rules

a) A module shall have at least one port function or AccessLink statement.

b) A ScanInPort port function is an input that shall take its source from a ScanData signal.

c) A ScanOutPort port function is an output that shall present the value of a ScanData signal.

d) A ShiftEnPort port function is an input that shall take its source from a ShiftEn signal.

e) A CaptureEnPort port function is an input that shall take its source from a CaptureEn signal.

f) An UpdateEnPort port function is an input that shall take its source from an UpdateEn signal.

g) A SelectPort port function is an input that shall take its source from a Select signal.

h) A ResetPort port function is an input that shall take its source from a Reset signal.

i) A TCKPort port function is an input that shall take its source from the network's TCK signal.

j) A ToShiftEnPort port function is an output that shall present the value of a ShiftEn signal.

k) A ToShiftEnPort port function shall be associated with ScanInPort port functions.

l) A ToCaptureEnPort port function is an output that shall present the value of a CaptureEn signal.

m) A ToCaptureEnPort port function shall be associated with ScanInPort port functions.

n) A ToUpdateEnPort port function is an output that shall present the value of an UpdateEn signal.

o) A ToUpdateEnPort port function shall be associated with ScanInPort port functions.

p) A ToSelectPort port function is an output that shall present the value of a Select signal.

q) A ToSelectPort port function shall be associated with ScanInPort port functions.

r) A ToSelectPort shall assert when Select is asserted and the associated ScanInPort function is selected as a result of UpdateEn becoming de-asserted.

s) A ToResetPort port function is an output that shall present the value of a Reset signal.

t) A DataInPort port function is an input that shall take its source from a Data signal.

u) A DataOutPort port function is an output that shall present the value of a Data signal.

v) A ToTCKPort port function is an output that shall present the value of a TCKPort signal.

w) An AddressPort port function is an input that shall take its source from a Data signal.

x) A WriteEnPort port function is an input that shall take its source from a Data signal.

y) A ReadEnPort port function is an input that shall take its source from a Data signal.

Explanations

Rule a) requires that a module contains at least one port with relevance to IEEE Std 1687 (which is indicated when a module port has a port function as defined in Table 2) or an AccessLink (see 6.4.17) through which an IEEE 1687 network may be accessed. Different types of module interfaces will naturally have different collections of port functions. For example, the module for a read-only instrument such as a temperature alarm may have only a single port with the port function DataOutPort, while the module for an instrument wrapped in a multiple TDRs may have a number of scan interfaces, each of which would include ports with port functions ScanInPort, ScanOutPort, SelectPort, ShiftEnPort, CaptureEnPort, and UpdateEnPort.

Rules b) through v) define the relationships among port functions and between port functions and signals. Signals, as defined in 5.10, connect port functions to each other and have specific types (Select, Data, ScanData, ShiftEn, CaptureEn, UpdateEn, Reset), which are described in the next subclause.

Rules for the five port functions related exclusively to embedded TAP controllers (in three rows of Table 2, three up from the bottom) are found in 5.14.2.

5.12 Signals between and within IEEE 1687 modules

The signals used in the definitions of the rules for port functions (5.11) serve to connect IEEE 1687 modules to each other or to connect components within an IEEE 1687 module. Note that some of the rules refer to the embedded TAP controller state names, which are defined in Table 3.

Rules

a) A Select signal shall be a combinational function of one or more of the following:
   1) DataIn port functions that, once connected in a network, ultimately take their value from Data Storage Elements or Update Storage Elements or device input pins
   2) DataOut port functions of instantiated modules that ultimately take their value from Data Storage Elements or Update Storage Elements
   3) Data Storage Elements
   4) Update Storage Elements
   5) Select port functions
   6) IRSelect signals
   7) ToSelect port functions of instantiated modules

b) Select signals shall only change as a result of UpdateEn becoming de-asserted, meaning that Select shall not change while TCK is low and UpdateEn is high.

c) A Data signal shall be multiplexed from one or more of the following:
   1) DataIn port functions
   2) DataOut port functions of an instantiated module
   3) Data Storage Elements
   4) Update Storage Elements

d) The source of a data signal shall be disjoint from the signals that select it.

e) A ScanData signal shall take its value from the value or the inverse value of one of the following:
   1) ScanInPort function
   2) ScanOutPort function of an instantiated module
   3) Shift Storage element
   4) Shift Path Multiplexer

f) A ShiftEn signal shall assert when either of the following conditions is met:
   1) All associated Select signals are asserted and its associated ShiftEn port function is asserted.
   2) An associated embedded TAP controller is in either of the iShiftIR or iShiftDR states.

   Otherwise it shall be de-asserted.

g) A CaptureEn signal shall assert when either of the following conditions is met:

   1) All associated Select signals are asserted and its associated CaptureEn port function is asserted and a path is sensitized through any data multiplexer which may exist between that port function and the signal.

   2) An associated Embedded TAP controller is in either of the iCaptureIR or iCaptureDR states.

   Otherwise it shall be de-asserted.

h) An UpdateEn signal shall assert when either of the following conditions is met:

   1) All associated Select signals are asserted and its associated UpdateEn port function is asserted and a path is sensitized through any data multiplexer that may exist between that port function and the signal.

   2) An associated Embedded TAP controller is in either of the iUpdateIR or iUpdateDR states.

   Otherwise it shall be de-asserted.

i) A Reset signal shall assert whenever the associated Reset port function is asserted and a path is sensitized through any data multiplexer that may exist between that port function and the signal.

j) The assertion of a Reset signal shall not interfere with the next capture or shift operations of the selected scan register.

Permissions

k) A Reset signal may assert with the falling edge of TCK when an UpdateEn signal is asserted and a Select signal is asserted.

Explanations

Rules a), c), and d) describe the port functions and/or components that may be used to create the Select, Data, and ScanData signals, respectively. The port functions were described in 5.11, and the components will be described in the next subclause.

Rules f), g), and h) define the conditions under which the ShiftEn, CaptureEn, and UpdateEn signals assert. There are two such conditions: when the interface containing the associated port function is selected and that port function is asserted, or when an embedded TAP controller (which will be described in 5.14) associated with the signal is in one of the corresponding active states.

Rule i) disallows the assertion of reset from preventing scan data to be captured into and/or shifted through an associated scan chain. Note that this rule does not prevent the assertion of reset from blocking the update operation; it is left to the user to de-assert the reset signal. The pathological case of a local reset signal blocking its own ability to update should be avoided.

5.13 Components comprising a module

The building blocks of an IEEE 1687 module include instances of other IEEE 1687 modules and primitive components such as storage elements, multiplexers, and combinational logic, but in restricted architectural arrangements.

Rules

a) There shall be no sequential or combinational loops through collections of primitive components consisting of Update Storage Elements, Data Storage Elements, or combinational logic that can be sensitized when Update Storage Elements and top-level module ports with functions SelectPort and DataInPort are stable.

b) A Shift Path Storage Element shall take on one of the following values on the rising edge of the TCKPort Function when its associated CaptureEn signal is asserted :

1) The value present on an associated Data Signal.

2) The value already present in the Shift Storage Element.

c) A Shift Storage Element shall take on the value of a ScanData signal on the rising edge of the TCKPort Function when its associated ShiftEn signal is asserted.

d) A serial shift path between a ScanInPort function and a ScanOutPort function of an IEEE 1687 interface shall uniquely include all Shift Storage Elements that are selected at the time when the ShiftEn port function is asserted.

e) A Shift Storage Element shall not be assumed to hold a known value after the UpdateEn signal is de-asserted and before either of CaptureEn signal or ShiftEn signal is asserted or the UpdateEn signal is re-asserted.

f) If a Shift Storage Element has a defined reset state, it shall take on that defined state when the associated Reset signal is asserted.

g) A Shift Storage Element shall retain its value in any other case than those described.

h) An Update Storage Element shall take on the value of a Shift Storage Element on the falling edge of the TCKPort Function when an associated UpdateEn signal is asserted.

i) If an Update Storage Element has a defined reset state, it shall take on that defined state when the associated Reset signal is asserted.

j) An Update Storage Element shall retain its value in any other case than those described.

k) A Data Storage Element shall take on the value of a Data signal when an associated UpdateEn signal is asserted and the TCKPort function is low.

l) If a Data Storage Element has a defined reset state, it shall take on that defined state when the associated Reset signal is asserted.

m) A Data Storage Element shall retain its value in any other case than those described.

n) Sufficient time shall be provided for all storage elements to resolve and propagate their value prior to the next assertion of the CaptureEn signal, the ShiftEn signal or the UpdateEn signal.

o) A Scan Multiplexer (ScanMux) shall select and present the value of one of two or more of ScanData signals.

p) The value selected and presented by a Scan Multiplexer shall be determined by one or more Select signals.

q) The Select signals of a Scan Multiplexer shall have their fan-in cones bounded by Update Storage Elements and/or device primary inputs.

r) The Update Storage Elements in the fan-in of the Select signals of a Scan Multiplexer shall be associated with Reset signals.

s) Scan Multiplexers shall not be capable of deselecting their controlling Storage Elements; an exception can be made if re-selection can be accomplished by performing a Reset.

Permissions

t) An IEEE 1687 module may contain one or more IEEE 1687 components.

u) A Shift Storage Element may utilize a negative-edge flip-flop on the ScanOut path.

Explanations

Rule a) disallows loops through network elements that may toggle arbitrarily. The rule is necessary to allow the network to capture stable values, in keeping with the fundamental assumption that instrument interfaces are static and event-driven (as described in 4.3). The storage elements in the network are expected to obey the same property: they only change in response to network stimulus. Such network stimulus is regulated by the values in Update Storage Elements and by the SelectPort and DataInPort functions of the top-level module, so when those ports are at their stable final values, there should be no active loops through storage elements or logic in the network. Rules b) through m) identify the conditions where the storage elements may change their state [rules b) c), f), h) i), l)], where they must hold their state [rules g), j), m)], and when their state cannot be assumed to be known [rule e)].

Rules f), i), and l) require that those Shift, Update, and Data storage elements in the network that can be reset to a specified state do so when their associated ResetPort is asserted. All three types of storage elements are covered, though it is worth mentioning that the action of resetting a Shift Storage Element is not a common use model (since the reset value will be overwritten by the next Capture event before it can be shifted out, and even if the Capture event were suppressed, a new value would be shifted in before the reset state could be updated).

Rule b2) means the Shift Storage Element may optionally not respond to capture operations. As made explicit in rule e), this does not mean that the value is the last shifted value.

5.14 TAP finite state machine embedded in an IEEE 1687 module

5.14.1 Overview of embedded TAP (eTAP)

Some IEEE 1687 modules provide direct port access to the signals that operate the network (Select, CaptureEn, ShiftEn, UpdateEn, etc.). Other IEEE 1687 modules, however, use as their interface an embedded TAP, along with the associated state machine in the embedded TAP controller to generate the signals that operate the network within that module. For this reason, IEEE Std 1687 defines an embedded TAP controller state-machine that is identical in state and transition arc topology to the IEEE 1149.1 TAP controller state machine defined in all versions of that standard from inception through IEEE Std 1149.1-2013, but which may be connected to an embedded TAP interface (as opposed to connecting to external pins, as is required by IEEE Std 1149.1). For the purposes of making this document capable of standing alone, the TAP controller state machine definition is included here, with the state names prefixed with the letter “i” to indicate that this controller may be present on “internal” networks.

Several signals are required to make use of an embedded TAP controller. These signals include TMS (to control the state transitions), TRST* (to return the state-machine to a known state), TCK (to clock the state machine), and Select (to include the interface containing the state-machine in the selected network and permit the state-machine to respond). Typically, the Select signal is not explicitly present when an embedded TAP is used as a network interface; rather, the Select function is used to gate the ToTMSPort or ToTCKPort functions that drive the embedded TAP to hold it idle when deselected. Alternatively, the Select function is used to gate the ToTRST port function that drives the embedded TAP to hold it in reset when deselected.

Embedding one or more eTAPC state machines in an IEEE 1687 network comes with the restriction that the associated eTAPC finite state machines (FSMs) remain synchronized when selected. Without this provision, the state of the eTAPC, and thus the network, would not be known. It is thus necessary to park this type of state machine when de-selected in either the RunTestIdle or TestLogicReset state. This restriction enables, after the Update event (from UpdateIR or UpdateDR) that renders them reselected, the newly selected state machines to always pass through the RunTestIdle state and to thus resume synchronized operation with the other state machines.

5.14.2 Port functions

Rules

a) A TMSPort function is an input that shall take its value from a TMS signal or a top-level TMS pin or a data signal when a path is sensitized through any data multiplexer that may exist between that port function and the signals or pin.

b) A ToTMSPort port function is an output that shall present the value of a TMSPort signal or a data signal when a path is sensitized through any data multiplexer that may exist between that port function and the signals.

c) A TRSTPort function is an input that shall take its value from a reset signal, but only when that reset signal originates from an associated TRST* pin of the device or an internal power-up detector, or a data signal when a path is sensitized through any data multiplexer that may exist between that port function and the signal or pin.

d) A ToTRSTPort port function is an output that shall present the value of a reset signal, but only when that reset signal originates from an associated TRST* pin of the device or an internal power-up detector, or a data signal when a path is sensitized through any data multiplexer that may exist between that port function and the signal or pin.

e) A ToIRSelectPort function is an output that shall take its value from an IRSelect signal.

Explanations

These rules define the relationships between the eTAPC-related port functions and the associated signals, which are defined in the next subclause. Rules c) and d) distinguish between a TAP reset caused by the assertion of the TRST* pin and the entry into the TLR state, which can be accomplished by holding the TMS pin high for five TCK cycles; only the former method causes the TRSTPort function to assert.

5.14.3 Signals

Rules

a) A TMS signal shall take its value from a TMSPort function or a ToTMSPort function and may be gated by a data multiplexer.

b) An IRSelect signal shall take its value from either a ToIRSelectPort or an Embedded TAP controller.

c) The CaptureEn, ShiftEn, and UpdateEn signals associated with an Embedded TAP controller shall behave in accordance with the event ordering diagram shown in Figure 4 for the associated DR and IR events.

d) An IRSelect signal shall be asserted when the Select signal associated with its Embedded TAP controller is asserted and that Embedded TAP controller is in the iSelectIR, iCaptureIR, iShiftIR, iExit1IR, iPauseIR, iExit2IR, or iUpdateIR states.

Explanations

Rules a) and b) are the reflexive definitions relating the signals to the port functions defined in the previous subclause. Rule c) defines the windows during which the control signals (CaptureEn, ShiftEn, UpdateEn) are active (with reference to Figure 4). Rule d) defines when IRSelect is active.

5.14.4 Components

The behavior of an Embedded TAP Controller and the context in which it may be used in a network are defined in the following rules.

Rules

a) If a network interface includes an embedded TAP connected to an embedded TAP controller for the purpose of locally generating signals, the host interface from the embedded TAP controller shall include ToSelect, ToCaptureEn, ToShiftEn, and ToUpdateEn port functions.

b) If a module includes an embedded TAP, then it and its parents in the hierarchy shall have associated TMSPort and TCKPort functions and optionally TRSTPort functions, along with the Select function either as an explicit port or gated into any of the TMSPort or TCKPort or TRSTPort functions.

c) An embedded TAP controller shall update its state with the rising edge of TCK.

d) An embedded TAP controller shall only change state when its associated Select signal is asserted.

e) When its associated Select signal is asserted, an embedded TAP controller shall update its state in response to the state of TMS at the rising edge of TCK in accordance with Table 3.

f) An embedded TAP controller that is deselected shall be parked in either the iRTI state or the iTLR state.

NOTE—This means that when selecting a currently parked embedded TAP controller, the operation of an embedded TAP controller shall not cause a transition to iSelectDR from any state except iRTI.

g) An embedded TAP controller shall transition immediately to the iTLR state when its associated Reset signal is asserted.

h) A module with an embedded TAP interface shall not be in series on the same scan chain as a module with a non-TAP interface (i.e., a scan interface).

Table 3—TAP FSM state transition table

Current state	Next state	
	TMS = 0	TMS = 1
iTLR	iRTI	iTLR
iRTI	iRTI	iSelectDR
iSelectDR	iCaptureDR	iSelectIR
iCaptureDR	iShiftDR	iExit1DR
iShiftDR	iShiftDR	iExit1DR
iExit1DR	iPauseDR	iUpdateDR
iPauseDR	iPauseDR	iExit2DR
iExit2DR	iShiftDR	iUpdateDR
iUpdateDR	iRTI	iSelectDR, see rule f
iSelectIR	iCaptureIR	iTLR
iCaptureIR	iShiftIR	iExit1IR
iShiftIR	iShiftIR	iExit1IR
iExit1IR	iPauseIR	iUpdateIR
iPauseIR	iPauseIR	iExit2IR
iExit2IR	iShiftIR	iUpdateIR
iUpdateIR	iRTI	iSelectDR, see rule f

Permissions

i) If a network interface includes an embedded TAP connected to an embedded TAP controller for the purpose of locally generating signals, the host interface of that embedded TAP controller may include ToResetPort and ToIRSelectPort functions.

Explanations

Rules a) and b) define the required port functions on the host and client interfaces of an Embedded TAP Controller module. Rules c) through e) describe the state transition operation of the Embedded TAP Controller, which is analogous to that of IEEE 1149.1. Rules f) and g) define the behavior associated with the operations of selecting, resetting, and deselecting an Embedded TAP Controller. Rule f) in particular allows parking of embedded TAP Controllers in only two of the states, which is necessary to maintain synchronization with the top-level TAP controller (and all other embedded TAP Controllers). Rule h) is necessary to prevent the mixing of Embedded TAP Controllers with modules that react to only DR-side operations of the device TAP Controller: such mixtures would not pass the IR-side operations to and from an Embedded TAP Controller in series.

5.15 Access network behavior

The basic unit of the IEEE 1687 hardware is the access network. An access network routes data and control signals from one or more device interfaces to one or more instrument interfaces.

The top-level controller is responsible for selecting a portion of the network to access and for supplying signals and data. The accessed portion of the network consists of a path through the module(s) between the controller and the instruments through which data and control are supplied and results are extracted. This path may be serial, in which case a well-defined protocol with port timing that matches that of IEEE Std 1500 is employed, or it may be parallel, in which case a protocol-independent sequence of transactions are used.

A storage element is defined to be “accessible” when it is supposed to be capable of taking on new values. For a shift path storage element, this implies that it is on an active scan chain that will respond to ShiftEn. For an update storage element, this implies that it is attached to a shift path register that is accessible (and will thus respond to UpdateEn). For a data storage element, this implies that the source of the signal that enables it to capture data (e.g., a write enable) is asserted and that the register driving it is accessible. For example, the source of the write enable signal for a data storage element could be connected to a TDR that is on an active scan chain.

5.16 Plug-and-play interfaces

5.16.1 Overview of plug-and-play interfaces

The hardware rules described in the preceding subclauses have been kept deliberately abstract in order to support a variety of legacy implementations; this is consistent with the philosophy of IEEE Std 1687 being descriptive rather than prescriptive. However, in acknowledgement of the facts that there is considerable value in a standard providing a clearly defined mechanism for establishing compliance and that many legacy implementations utilize IEEE 1149.1 protocol, this clause includes additional rules that, when followed, confer one of four possible types of plug-and-play compliance to the associated interfaces. An interface is either a host (which provides control signals) or a client (which consumes control signals), and there are two types of plug-and-play compliance for each; these are termed “scan” and “TAP” in order of increasing restriction on the port functions included in the instrument. As the name implies, plug-and-play compliance indicates that the corresponding types of interfaces can be “plugged” together to satisfy the

requirements of a properly connected network. Specifically, a plug-and-play scan client interface will connect to a plug-and-play scan host interface, and a plug-and-play TAP client interface will connect to a plug-and-play TAP host interface. Network integration is significantly eased when all interfaces are plug-and-play compatible. The details and rules for each of these four interfaces will be described in the following subclauses, but first the context common to all four will be reviewed.

All four types of plug-and-play interfaces are consistent with the right-hand portion of the generic iApply protocol described in Figure 37 (which is discussed in more depth in Clause 7 at Figure 53).

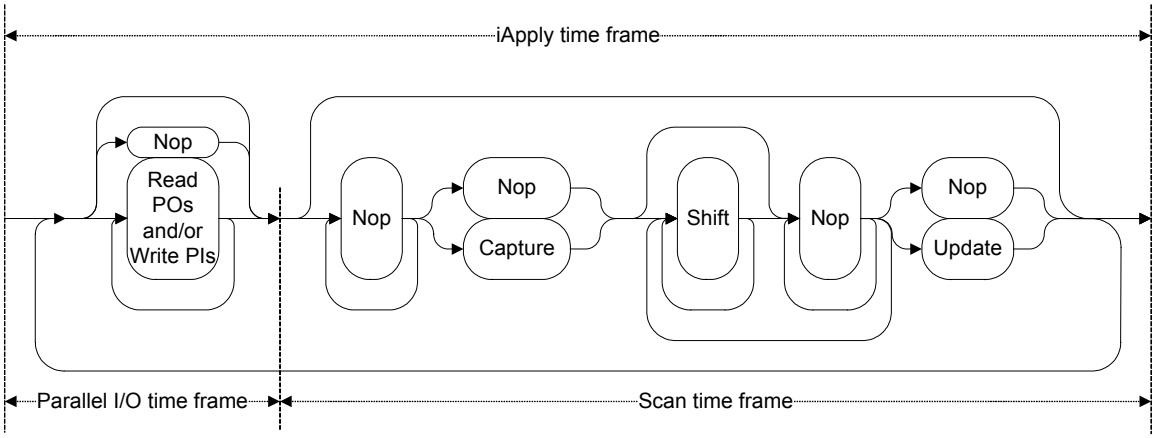


Figure 37—iApply protocol at interfaces

Each bubble in Figure 37 represents the action (or actions) that apply to each stage of the sequence comprising the protocol. Each bubble represents one cycle of the clock that is driving the IEEE 1687 network (i.e., TCK). The CaptureEn, ShiftEn, and UpdateEn signals are only active during the bubbles with corresponding names (i.e., Capture, Shift, Update); in all other bubbles, those signals are inactive. Bubbles in which none of the three signals are active are labeled “Nop” (for “no operation”). There are paths through the diagram that bypass certain bubbles or sets of bubbles (these paths are shown above the bubbles), and there are other paths in the diagram that loop backwards to a given bubble or set of bubbles (these paths are shown below the bubbles).

Note that Figure 37 is separated into two portions: a parallel I/O time frame and a scan time frame during which each of the respective port functions will operate. Note that operations may traverse these two portions in either order and may do either or both portions multiple times. The parallel I/O time frame includes several traversal options, including (from top to bottom in the diagram) skipping the operation entirely, spending one clock cycle with no activity, spending one clock cycle performing any combination of reading primary outputs (POs) and/or writing primary inputs (PIs), or performing either the Nop or PO/PI operations or both over multiple clock cycles. The scan time frame (where the four types of plug-and-play interfaces with IEEE 1149.1 lineage operate) has several traversal options as well, including (from left to right in the diagram) the ability to skip the scan time frame entirely, loop doing nothing, suppress the Capture event during its assigned clock cycle, bypass the Shift event entirely or spend one or more clock cycles in Shift, suppress the Update event during its assigned clock cycle, and repeat another parallel/scan sequence. Note the requirement of having at least one clock cycle between the Update and Capture events, which matches the behavior of an IEEE 1149.1 TAP controller.

5.16.2 Scan host plug-and-play interface

A scan plug-and-play instrument operates on the scan signals first described in Table 1 (i.e., {ScanIn, ScanOut, CaptureEn, ShiftEn, UpdateEn, Select, Reset, TCK}). A scan host plug-and-play interface provides ScanIn data to the instrument, sources the scan control signals, and consumes the ScanOut data

coming from the instrument. The relevant portion of the protocol is shown in Figure 38, which is functionally equivalent to the scan time frame portion of Figure 37. The outside arcs of Figure 37 have been adjusted to indicate the scan-only nature of this interface: the top-most arc in Figure 37 that bypasses the scan time frame is omitted in Figure 38, and the bottom-most arc in Figure 37 that allows repetition is shortened in Figure 38 to omit the non-existent parallel I/O time frame.

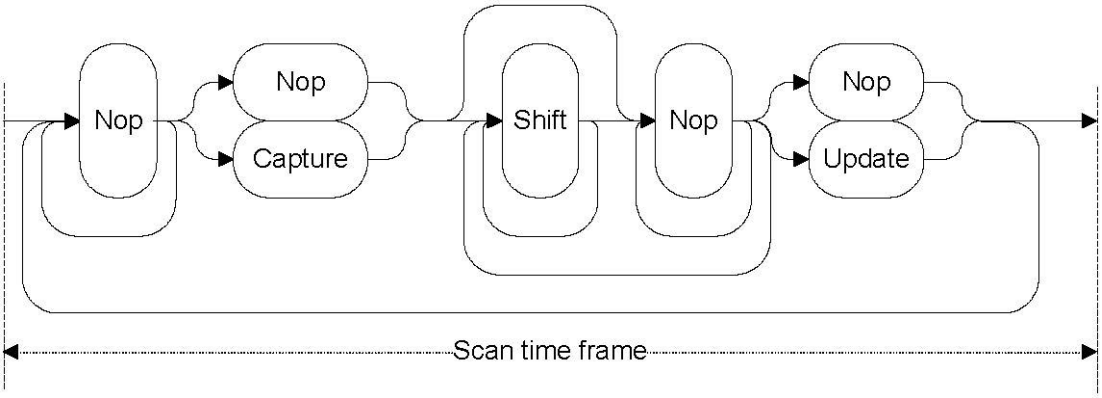


Figure 38—Protocol for Scan plug-and-play instruments

This scan-only protocol is consistent with the operation of the IEEE 1149.1 TAP state machine, so when a TAP is controlling the IEEE 1687 network, any scan plug-and-play instruments connected to it will operate as intended. Likewise, if some other controller is in charge of the network, as long as that controller obeys this protocol, the scan plug-and-play instruments will operate as intended.

Rules

a) A scan host plug-and-play interface shall include exactly one each of the ScanInPort, ScanOutPort, ToCaptureEnPort, ToShiftEnPort, ToUpdateEnPort, ToSelectPort, ToResetPort, and ToTCKPort functions.

b) A scan host plug-and-play interface shall obey the protocol in Figure 38.

Permissions

c) Except for the ScanInPort and ToSelectPort functions, all other port functions may be shared by multiple scan host plug-and-play interfaces.

5.16.3 Scan client plug-and-play interface

A scan client plug-and-play interface may use a subset of the port functions available in the associated scan host interface. Specifically, in order to shift, a scan client plug-and-play interface needs to have ScanIn and ScanOut data ports along with Select and ShiftEn control ports and a TCK clock. It may optionally have the capability to capture new data (CaptureEn), update recently shifted data into locked storage (UpdateEn), and reset to a defined state (Reset). The scan client plug-and-play interface is required to obey the protocol in Figure 38.

Rules

a) A scan client plug-and-play interface shall include exactly one each of the ScanInPort, ScanOutPort, ShiftEnPort, SelectPort, and TCKPort functions.

b) A scan client plug-and-play interface shall obey the protocol in Figure 38.

Permissions

c) A scan client plug-and-play interface may contain at most one each of the CaptureEnPort, UpdateEnPort, and ResetPort functions.

d) Except for the ScanOutPort and SelectPort functions, all other port functions may be shared by multiple scan client plug-and-play interfaces.

5.16.4 TAP host plug-and-play interface

An even more restrictive level of plug-and-play compliance is achieved by a module that contains an embedded TAP controller (eTAPC) finite state machine. The signals involved for this type of interface are ScanIn, TMS, TCK, TRST*, and ScanOut. By virtue of using the (IEEE 1149.1-compliant) eTAPC FSM embedded inside the module, the serial protocol shown in Figure 38 will apply.

Rules

a) A TAP host plug-and-play interface shall include exactly one each of the ScanInPort, ScanOutPort, ToTMSPort, ToTRSTPort, and ToTCKPort functions.

b) A TAP host plug-and-play interface shall be compatible with the protocol in Figure 38.

5.16.5 TAP client plug-and-play interface

A TAP client plug-and-play interface connects to a TAP host interface and thus has the complementary set of port functions.

Rules

a) A TAP client plug-and-play interface shall include exactly each of the ScanInPort, ScanOutPort, TMSPort, TRSTPort, and TCKPort functions.

b) A TAP client plug-and-play interface shall be compatible with the protocol in Figure 38.

6. Instrument Connectivity Language (ICL)

6.1 ICL introduction

The purpose of the Instrument Connectivity Language (ICL) is to describe the elements that comprise the instrument access network as well as their logical (though not necessarily their physical) connections to each other and to the instruments at the endpoints of the network. The language bears a loose resemblance to a hierarchical netlist: it is organized by modules that may contain instances of other (child) modules, and it describes the attachment points of the ports of each module to the ports of other modules and/or to ICL primitives. However, it is important to note that ICL is not a complete netlist: connections are port-to-port rather than through nets, and it freely uses abstraction in order to omit the detailed physical construction of the circuitry—only the behavioral operation of the network needs to be represented.

Beyond merely documenting the structure of the network, ICL enables an operation called "retargeting," whereby the sequences of operations written in PDL (see Clause 7) that are to be applied at the instrument boundary (or any module boundary) are mapped through the instrument access network to a higher level of hierarchy (and ultimately to the ports of the top-level module that connect to device pins). The expectation is that automated software will read and store the ICL information in a data structure, then use that information to map the instrument-level operations into corresponding actions that will be applied at a higher level. Note that while the lowest-level instrument interface is described in ICL, the internal logic of the instrument itself is not part of this standard.

This clause will present the ICL language from three different perspectives: an information-centric context, a syntactic detail context, and a semantic context. The information-centric approach is useful to teach the concepts behind the language without diving too deeply into the details of the syntax, although the examples given use syntactically correct format.

6.2 ICL overview

6.2.1 ICL information

The fundamental entity in ICL is called a *module.* The ICL description for a device will consist of one or more module entities that may reside in one or more files. Modules may be instantiated inside other modules, with the module at the root of the hierarchy referred to as the *top-level module.* A device may have multiple independent access networks described in the same or in different modules.

The ICL for a module consists of the following two types of information:

— Hierarchical network structure and instrument interface (required)

— Parameters, aliases, enumerations, and attributes (optional)

The required network information in ICL reflects the structure of the architecture as described in Clause 5. It therefore consists of interconnections between the host and client interfaces of modules that may themselves be constructed from primitive building blocks (such as storage elements or shift-path multiplexers, for example). The network may connect to one or more controllers on one end and one or more instruments on the other end. The network may include embedded TAPs and black-box elements whose contents are not exposed but may themselves contain shift-path elements (e.g., shift storage elements or shift path multiplexers). The structure is described in such a way as to allow a retargeting tool to navigate the network in order to control and observe any instrument that resides in the network.

The optional information is included to improve reusability and readability. The parameter information is used to make the ICL definitions somewhat flexible, by allowing a module to utilize variables that may be passed into it from its parent to specify values. The alias and enumeration information in ICL is included to simplify the process of writing PDL by allowing the user to group signals and define mnemonics in the instrument interface module. Though these mnemonics are intended for use in the PDL, they are centrally defined in the ICL to allow them to be referenced to the actual structural names (in ICL) and to allow them to serve as a common library across multiple PDLs. The attribute information is included as a placeholder for the user to communicate details otherwise not described about the module.

6.2.2 Hierarchical network structure

Each module in the network is given a unique name (a module_name with an optional **`name_space::`** prefix). There is at least one module in a network. A module in a network is defined by its interface(s) and its contents. The interface is described in terms of its constituent ports, and the contents may be any combination of registers, combinational logic, or other child modules. The use of logical hierarchy (i.e.,

modules that contain instances in a parent/child relationship) is common in netlisting applications and facilitates both component reuse and compact representation of networks.

It is important to note that the ICL network does not need to exactly describe the hierarchical structure of the actual netlist (RTL or gate-level) with which it is associated. While it is certainly possible that the hierarchical partitioning of the ICL may correspond to the logical or physical netlist, it may be far more expedient to use a different level of abstraction for the ICL, since ICL needs to represent only the behavior of the network. Taken to the limit, the ICL for the network could consist merely of the interconnection of the flattened ICL primitive components with all the logical hierarchy removed. In fact, it is envisioned that retargeting tools will create just such a model of the network. However, for human readability, ease of debug, and to allow simulation at specific hierarchical boundaries, the expectation is that the use of hierarchy in ICL will be commonplace.

6.2.3 Module content: primitive building blocks and organizational elements

There is a small set of components that may be used to define modules: ports, registers of two styles, multiplexers of five styles, simple combinational logic, scan interfaces, access links, as well as instances of other modules. These primitives are accompanied by some syntactical elements to simplify descriptions: aliases, enumerations, parameters, and attributes. This set of primitives and organizational elements comprise the keyword set in ICL:

```
Module <moduleName> {

# building block primitives
  <func>Port  <PortName>                          { ... }
  ScanInterface <ScanInterfaceName>               { ... }
  Instance <InstanceName> Of  <moduleName>        { ... }
  ScanRegister <ScanRegName>                      { ... }
  ScanMux      <MuxName> SelectedBy <Selector>    { ... }
  OneHotScanGroup <SignalName>                    { ... }
  LogicSignal  <SignalName>                       { ... }
  DataRegister <DataRegName>                      { ... }
  DataMux      <MuxName> SelectedBy <Selector>    { ... }
  ClockMux     <MuxName> SelectedBy <Selector>    { ... }
  OneHotDataGroup <SignalName>                    { ... }
  AccessLink  <InstanceName> Of <linkType>        { ... }

# organizational elements
  Alias <AliasName> = <Element>                   { ... }
  Enum <EnumName>                                 { ... }
  Parameter  <parName> = <parValue>;
  Attribute  <attName> = <attValue>;
}
```

There are two pieces of shorthand used in the preceding description, as follows:

— "**<func>Port**" is a placeholder for a group of keywords for which each identifies a unique port function (see Table 4 for the full list).

— There is additional information ("…") that may be provided between the curly braces ("**{…}**") located after most of the statements (see 6.4 for each statement type to view this information). If no extra information is required to be present for a statement, the curly braces may be omitted and the statement terminated with a semicolon. Alternatively, a pair of empty curly braces may be used to terminate the statement.

6.2.4 ICL abstraction

ICL is not a literal duplication of the device netlist; rather, it is an abstract representation of the device that faithfully allows navigation of the instrument access network. Only those components inside a module that are necessary to describe the network need to be included. For example, the functional logic in the design does not appear in the ICL (unless parts of it comprise portions of the instrument access network).

While the notion of signals is present in the formulation of the hardware rules, as well as in the formal grammar descriptions of the ICL, there are no ICL keywords that describe signals per se. ICL can fully describe a network using only instances and ports, without signals. One pleasant consequence of this approach is that only the source of information needs to be noted (and not the destination, as would also be required if a signals-based approach had been used).

6.3 ICL lexical conventions and definitions

6.3.1 Purpose

This subclause describes the lexical tokens used in ICL source text and their conventions.

NOTE—If there is a difference in grammar presented in this subclause and Annex A, the grammar in this subclause prevails.

6.3.2 ICL lexical tokens

ICL source text files define a stream of lexical tokens. A lexical token shall consist of one or more characters. The layout of tokens in a source file shall be free format. Spaces, carriage returns, and newlines shall not be syntactically significant other than as token separators. In general, ICL utilizes curly braces ({}) as major block separators that act as a *wrapper*, a semicolon (`;`) as an element list terminator, an OR separator (`|`) to indicate a bitwise OR operation, and a comma (`,`) to indicate concatenation.

If a specified block has no elements, either a semicolon (`;`) or empty braces ({}) shall be used as the termination character.

The types of lexical tokens in ICL are as follows:

- Keywords
- Generic tokens
- White space
- Comments
- Numbers
- Strings
- Identifiers

Rules

a) A lexical token in ICL shall consist of one or more characters.

b) The semicolon character (“;”) or empty curly braces (“{}”) shall be used as the termination character.

6.3.3 ICL keywords

ICL keywords and identifiers are case sensitive and *not* reserved. Keywords can be used for identifier names. For example, the following ICL statement:

```
Module Module { <... more statements...> }
```

is valid syntax where the first **`Module`** is a case-sensitive keyword and the second **`Module`** is the case-sensitive user name for the module.

Keywords shall match both spelling and case. All ICL keywords begin with a capitalized letter. If the keyword consists of multiple words, then the first letter for *each* word is capitalized (e.g., **`ScanInPort`**).

Within a module, the statements can appear in any order. There is no requirement to define before use. Within a pair of braces, the statement elements can appear in any order as well. The only exceptions to this rule are related to the NameSpace and UseNameSpace keywords (which take effect from the place of declaration onward) and the assembly of DataInPort (or DataOutPort) lists to create the logical data port for an addressable instance (which proceeds left-to-right, top-to-bottom).

Rules

a) ICL keywords and identifiers shall match both spelling and case unless otherwise indicated.

Permissions

b) ICL keywords are not reserved and may therefore be reused as identifiers.

Recommendations

c) Reusing keywords as identifiers is confusing and should be avoided.

6.3.4 ICL grammar syntax convention

The grammar used to describe ICL is written using the following syntax convention, which is in ANTLR4 [B1] format:

1) The colon (**:**) delimits the token name and should be read as "*is defined as*"
2) A semicolon (**;**) terminates a definition
3) Single quotes (**'**) surround one or more literal characters that appear exactly as shown
4) Parenthesis surround a group that can be an alternative choice or repeated
5) A question mark (**?**) signifies that the grouping can be repeated "zero or one times"
6) An asterisk (*****) signifies that the grouping may be repeated "zero or more times"
7) A plus (+) signifies that the group shall be repeated "one or more times"
8) The vertical bar (|) separates alternatives
9) The ellipsis (**..**) between two characters means "these two alternatives and all alternatives in the set bounded by them"
10) A backslash (\) means that the next character is a literal, with the exception of \t (which signifies a tab), \r (which signifies a carriage return), \n (which signifies a newline)

11) An expression specified with no spaces between the delimiting quote marks indicates that no white space is allowed; if a space is specified then any number of white space characters is allowed including tabs, carriage returns and comments.

12) White space can be omitted before and after the following characters: '(', ')', '[', ']', '{', '}', ':', '=', ';', ',', '|' and the single quote (').

13) Tokens are case sensitive unless otherwise indicated.

14) Comments in the grammar start with // and continue to the end of the line.

Rules

a) ICL shall be defined by the Grammar sections in the various subclauses of Clause 6 using the conventions listed in this subclause.

6.3.5 Generic ICL tokens

There are a common set of tokens used to form the identifiers in ICL, as described in the following Grammar:

Grammar

```
SCALAR_ID : ('a'..'z' | 'A'..'Z')('a'..'z' | 'A'..'Z' | DEC_DIGIT | '_')* ;
pos_int : '0' | '1' | POS_INT ;
POS_INT : DEC_DIGIT('_'|DEC_DIGIT)* ;
size : pos_int | '$' SCALAR_ID ;
UNKNOWN_DIGIT : 'X' | 'x';
DEC_DIGIT : '0'..'9' ;
BIN_DIGIT : '0'..'1' | UNKNOWN_DIGIT ;
HEX_DIGIT : '0'..'9' | 'A'..'F' | 'a'..'f' | UNKNOWN_DIGIT ;
DEC_BASE : '\'' ('d' | 'D') (' ' | '\t')*;
BIN_BASE : '\'' ('b' | 'B') (' ' | '\t')*;
HEX_BASE : '\'' ('h' | 'H') (' ' | '\t')*;
UNSIZED_DEC_NUM : DEC_BASE POS_INT ;
UNSIZED_BIN_NUM : BIN_BASE BIN_DIGIT('_'|BIN_DIGIT)* ;
UNSIZED_HEX_NUM : HEX_BASE HEX_DIGIT('_'|HEX_DIGIT)* ;
sized_dec_num : size UNSIZED_DEC_NUM ;
sized_bin_num : size UNSIZED_BIN_NUM ;
sized_hex_num : size UNSIZED_HEX_NUM ;
```

Rules

a) When ('$'SCALAR_ID) is used to define SIZE, the token shall reference a parameter_name with a POS_INT value.

6.3.6 ICL white space

White space consists of spaces, tabs, newlines, and carriage return (if there is one) characters. These characters shall be ignored except when they serve to separate other lexical tokens. Spaces and tabs shall be considered significant characters within double-quoted strings.

Rules

a) White space (spaces, tabs, carriage returns, new lines) shall not be syntactically significant other than as token separators.

b) Spaces and tabs shall be considered significant characters within double-quoted strings.

6.3.7 ICL comments

ICL has two forms to introduce comments. A one-line comment shall start with the two characters slash-slash (**//**) and end with a new-line character or a pair of carriage return–linefeed characters. A block comment can span multiple lines and shall start with the two characters slash-asterisk (**/***) and end with the two characters asterisk-slash (***/**). Block comments shall not include additional (i.e., nested) block comments. Within a block comment, the one-line comment token **//** shall have no special meaning.

Examples

```
// this is a valid single-line line comment

Module Demo { // this is a valid end-of-line comment

/* this is a
valid block comment */

/* this is /* not */ a valid comment
   because it contains a nested block comment */
```

Rules

a) A one-line comment shall start with the two characters slash-slash (//) and end with a new-line character or a pair of carriage return–linefeed characters.

b) A block comment may span multiple lines and shall start with the two characters slash-asterisk (/*) and end with the two characters asterisk-slash (*/).

c) Block comments shall not include additional (i.e., nested) block comments.

d) Within a block comment, the one-line comment token // shall have no special meaning.

6.3.8 ICL numbers

Numbers are specified as unsigned integer constants in decimal, hexadecimal, or binary format. There are two forms that express integer constants. The first form is a simple decimal number that shall be specified as a sequence of digits **0** through **9**. The second form consists of three tokens that define a number: (1) an optional size constant, (2) a specific number base format, and (3) digits in the number base representing the value of the number.

The first token, the size constant, is specified as an unsigned, non-zero decimal number, and may be delivered as a parameter reference. It denotes the exact number of binary digits that shall be assumed for the binary representation of the number. It is possible that the size constant defines more or fewer binary digits than the number's binary representation would entail. In this case either padding or truncation will occur in accordance with procedures described as follows.

The second token, a base format, shall consist of a case-insensitive letter specifying the base for the number preceded by the single quote character (**'**). Legal base specifications are **d** or **D**, **h** or **H**, and **b** or **B** for the decimal, hexadecimal, and binary bases, respectively. The single quote and the base format character (e.g., **'b**) shall not be separated by any white space.

The third token, an unsigned number, shall consist of digits that are legal for the specified base format. The unsigned number token shall follow the base format, optionally preceded by white space. The hexadecimal digits **a** to **f** shall be case-insensitive.

The use of **'x'** in defining the value of a number is case-insensitive. An **'x'** represents the unknown value in hexadecimal and binary constants. For a number represented in the hexadecimal base, an **'x'** shall

specify 4 unknown bits in the binary representation of this hexadecimal number; an **'x'** for a number in the binary base shall specify 1 unknown bit. An **'x'** cannot be specified in the decimal base.

If the number of binary digits of a number's binary representation is less than the number of binary digits defined by the size constant, then the number's binary representation will be padded on the left by either **'0'** or **'x'** until the number of digits equals the number of binary digits defined by the size constant. If the leftmost digit of the specified number is not **'x'**, then **'0'** shall be used as the padding digit; otherwise **'x'** shall be used as the padding digit.

If the number of binary digits of a number's binary representation is larger than the number of binary digits defined by the size constant, then the number's binary representation shall be truncated, removing the leftmost digits of the number's binary representation, until the number of binary digits in the number equals the number of binary digits defined by the size constant. Such truncation is allowed only when removing digits whose values are **'x'** or **'0'**; an error is generated if a digit with value 1 is truncated.

If the number is unsized (i.e., it is in simple decimal notation, like 42, or it is has a base but lacks a size constant, like **'h 5AB2** or **'d 123)**, then it shall inherit the size of the object to which it is being applied.

The underscore character (_) shall be legal anywhere in a number except as the first character and is ignored. The underscore character is used for readability purposes, as in **16'b0101_1010_1011_0010**.

———

Example 1—Unsized constant numbers

```
1_3_4   // a valid decimal number (134)
0235    // a valid decimal number
'd 134  // a valid decimal number
'h 23ff // a valid hexadecimal number
3fa     // Error: invalid hex format (requires 'h)
0xa     // Error: wrong hex format
```

Example 2—Sized constant numbers

```
4'b0101  // a 4-bit binary number
5'D 3    // a 5-bit decimal number
3'b0_1x  // a 3-bit number with the lsb an unknown number
7'hx     // a 7-bit unknown number (same as 7'bxxxxxxx)
Parameter SIZE = 2;
$SIZE'h3 // Use of a parameter to size a number (which will be 2'b11)
$SIZE'hF // Error: cannot truncate digits which are 1
7' hx    // Error: invalid hex number (no space allowed between ' and h)
0'b      // Error: zero size is invalid
5'h30    // Error: significant digits above the specified size
```

Example 3—Automatic left padding

Assume **rega** is defined (with **ScanRegister** statement) as 9 bits in width:

```
rega[8:0] == 'h x   // same as rega[8:0] == 9'bxxxxxxxxx
rega[8:0] == 'h 3x  // same as rega[8:0] == 9'b00011xxxx
rega[8:0] == 'hx3   // same as rega[8:0] == 9'bxxxxx0011
```

NOTE—The **ScanRegister** statement explicitly defines both register size and indices.

Rules

a) Integer numbers with a base shall be specified using an optional size constant immediately followed by a single quote character (') immediately followed by a case-insensitive letter specifying the base: b or B for binary, h or H for hexadecimal, and d or D for decimal.

b) The size constant shall be an unsigned, non-zero, decimal number that denotes the exact number of binary digits that shall be used to represent the integer number, regardless of the actual base of that integer number.

c) The single quote and the base format character (e.g., `'b`) shall not be separated by any white space.

d) The digits following the base specification shall be legal for that base: decimal includes digits 0 through 9; binary includes 0, 1, x; hexadecimal includes 0–9, a–f, and x.

e) Hexadecimal digits `'a'` through `'f'` and `'x'` shall be case insensitive.

f) Binary digit `'x'` shall be case insensitive.

g) The digit `'x'` shall represent the unknown value for a bit in the binary representation.

h) Hexadecimal digit `'x'` shall be equivalent to 4 unknown bits in the binary representation.

i) If the number of binary digits of a number's binary representation is less than the number of binary digits defined by the size constant, the number's binary representation shall be padded on the left by either 0 or x until the number of digits equals the number of binary digits defined by the size constant. If the leftmost digit of the specified number is not x, then 0 shall be used as the padding digit, otherwise x shall be used as the padding digit.

j) If the number of binary digits of a number's binary representation is larger than the number of binary digits defined by the size constant, then the number's binary representation shall be truncated, removing the leftmost digits of the number's binary representation, until the number of binary digits in the number equals the number of binary digits defined by the size constant. Such truncation shall result in an error if a digit with binary value 1 is truncated.

k) The underscore character shall be ignored when it occurs anywhere between digits in a number.

l) Unsized numbers shall inherit the size of the ICL object with which they are associated.

Permissions

m) Numbers with bases may omit the sizing constant.

n) Decimal numbers may be specified as a simple sequence of digits 0 through 9 without the sizing constant and **`'d`** or **`'D`** base prefix.

o) The sizing constant may be delivered as a parameter reference.

p) The base specification and the following integer number may be separated by white space.

q) The size constant may define more or fewer binary digits than the integer number's binary representation entails.

r) The underscore character may appear anywhere in a number except before the first digit.

6.3.9 ICL strings

A string is a sequence of visible or white space characters contained within a matching pair of double quotes ("). Within the string any character preceded by a backslash escape character is seen as that following character without the backslash escape. This escaping mechanism allows the double quote (") character with a preceding backslash escape character to be considered as part of the string and not the string termination character. To get a single backslash character within the string requires the use of an

escaped backslash (\\). For example the string USE THE "\" CHARACTER would have to be written as the string "USE THE \"\\\" CHARACTER".

Example of strings

```
"this is a double-quoted string"
"a string with \\ and \" within the string"
"a string with an escaped name \\a/b/c within the string"
"a multi-line
 string"
```

Rules

a) The backslash character ("\") inside a string shall serve as the escaping mechanism to allow an immediately following double-quote character that would normally terminate the string to instead be considered part of the string, and an immediately following backslash character that would normally be an escape character to be considered part of the string.

6.3.10 ICL identifiers

6.3.10.1 Types of ICL identifiers

There are four types of identifiers: scalar, parameter, vector, and hierarchical port. There are two types of vector identifier: single index and range. All identifiers are case sensitive. There is no length limit for identifiers.

Rules

a) All identifiers shall be case sensitive.

b) There shall be no length limit imposed on identifiers.

6.3.10.2 ICL scalar identifier

A scalar identifier has a width of exactly one (1) bit and shall begin with a letter and, optionally, followed by a sequence of letters, digits, and underscore characters (_). A scalar identifier is distinguished from a single-indexed vector identifier by the fact that a scalar identifier shall not include an index enclosed in square brackets [].

Examples of scalar identifiers

```
bus_index
n657
_bus3           // Error: cannot start with underscore.
5bitbus         // Error: cannot start with integer
bit-bus         // Error: cannot include special character
A$suffix        // Error: parametric reference not allowed
```

Rules

a) A scalar identifier shall have a width of 1 bit.

b) A scalar identifier shall begin with a letter and may be followed by a sequence of letters, digits, and underscore characters.

6.3.10.3 ICL parameter identifiers

A parameter identifier is used to reference a parameter definition and is defined as a scalar identifier prefixed with a dollar (**$**) sign.

Rules

a) A parameter identifier shall be a scalar identifier prefixed with a dollar sign ($).

6.3.10.4 ICL vector identifiers

A vector identifier defines one or more bits of a bus. Vector identifiers can be described with a single index or a single range of two index values. These two types of vector identifiers are referred to as: a *single-indexed vector identifier* and a *range-indexed vector identifier*.

6.3.10.4.1 ICL single-indexed vector identifier

The single-index vector identifier is used to describe a single element of a bus. The index is represented by an unsigned decimal integer expression. The syntax for a single-indexed vector identifier is a scalar identifier followed by an unsigned decimal integer (or an arithmetic expression that resolves to an unsigned decimal integer) or a parameter reference equivalent enclosed in brackets.

Examples of single-indexed vector identifiers

```
A[7]          // refers to bit 7 of identifier A

B[0]          // refers to bit 0 of identifier B
C[+1]         // Error: index cannot be a signed integer
D[$REG_SIZE] // REG_SIZE is a parameter
E[$width-1]  // simple arithmetic expression allowed
F-G[2]        // Error: scalar identifier includes invalid character (-)
H$suffix[2]  // Error: parameter substitution not allowed
```

Rules

a) A single-indexed vector identifier shall have a width of 1 bit.

b) A single-indexed vector identifier shall be composed of a scalar identifier followed by a pair of square brackets containing an index value.

c) The index value for a single-indexed vector identifier shall be one of the following:

   1) An unsigned decimal integer.

   2) An arithmetic expression that evaluates to an unsigned decimal integer.

   3) A parameter reference that resolves to an unsigned decimal integer.

6.3.10.4.2 ICL range-indexed vector identifier

The range-indexed vector identifier is used to describe a vector (i.e., a bus) with a contiguous sequence of elements. The range is described with two unsigned decimal integer expression separated by a colon. The order of elements is implied by the ordering of the two indices.

Examples of range indexed vector identifiers

```
A[7:0]         // MSB is 7, LSB is 0
B[0:7]         // MSB is 0, LSB is 7
C[6:6]          // same as C[6]
D[$MSB:$LSB]   // parameter references are unsigned integers
E[$size-1:0]   // parameter expressions returns unsigned integers
F[+4:0]        // Error: a signed number cannot be an index
```

Rules

a) The width of a range-indexed vector identifier shall be the absolute value of the difference between the indices, plus one.

b) A range-indexed vector identifier shall be composed of a scalar identifier followed by a pair of square brackets containing two index values separated by a colon character (:).

c) The index values used in a range-indexed vector identifier shall each be one of the following:

   1) An unsigned decimal integer.

   2) An arithmetic expression that evaluates to an unsigned decimal integer.

   3) A parameter reference that resolves to an unsigned decimal integer.

d) The two index values used in a range-indexed vector identifier shall form a contiguous sequence of elements with no gaps.

e) The order of the elements in a range-indexed vector identifier shall correspond to the ordering of the two indices.

Explanations

Rule e) allows the distinction between elements based on the ordering of the index values to the left and right of the colon in the range. For example, if vec[2:0] consists of elements A, B, C, then a connection of vec[0:2] will place the elements in order C, B, A.

6.3.10.5 ICL hierarchical port identifiers

A hierarchical port identifier is used to refer to the instance path of an object that is composed of one or more instance names followed by a port name, each separated by a dot (**.**). No spaces are allowed within an instance identifier. The instance must be the name of an instance defined in the same module as the hierarchical port identifier being used. The only allowed exception is to define an alias name (see 6.5.2) inside a module for an object one or many levels down within the module. Such alias names cannot be referenced by any ICL elements except within other alias name definitions.

Examples of hierarchical port identifiers

```
pinA          // pinA is a port name of the current module
U2.pinA       // U2 is an instance name and pinA is a port name
U1.U2.pinA    // more than one level of instance name. Can only be used in an
              // alias name and the alias name can only be used as the target
              // of an iWrite or iRead command.
```

Rules

a) Hierarchical port identifiers shall not contain white space characters.

b) Hierarchical port identifiers shall contain one or more instance names.

c) Instance names in a hierarchical port identifier shall be terminated by a dot (.) character.

d) Instances named in a hierarchical port identifier shall exist in the module in which the hierarchical port identifier is being used.

e) Hierarchical port identifiers shall end in a port name.

6.3.11 Inversion and concatenation

There are two operations, inversion and concatenation, which can be applied to certain elements that represent values. These operations are performed using the tilde ("~") and comma (",") operators, respectively. Inversion complements the values; concatenation joins elements into larger collections.

When an identifier is preceded by a tilde (~), each bit in the binary representation of the value of that identifier is complemented.

Examples of inversion

```
~4'b1000 // means 4'b0111
~8'hAA   // means 8'h55
~42      // means ~'d42 = ~'b101010 = 'b010101
~ABC     // the value of ABC will be bitwise inverted before being used
```

It is valid in some places to define a value that is the result of concatenating (i.e., joining) elements defined as members of a list separated by a comma (**,**). Aliases can be created by concatenating objects like ports or register bits. Strings may be concatenated only with other strings. Sized numbers may be concatenated with other sized numbers, and at most one unsized number can be included with a group of sized numbers. Since the size of all targets is known, the "size" of an unsized number in this situation is inferred to be the difference between the target size and the sum of the sized numbers' widths and is required to be equal to or greater than the number of bits required for the unsized number.

Parameter references may be used as concatenation elements provided that they meet the typing restrictions that apply. An element such as a **ScanRegister** output may also appear in a concatenation list as a sized element.

Examples of unrestricted concatenation

```
Parameter abc = 4'h7;
"abc", "def", "ghi"  // equivalent to: "abcdefghi"
1'b0, 4'h7, 5'h1a    // equivalent to: 10'b0_0111_11010
1'b0, $abc, 5'h1a    // equivalent to: 10'b0_0111_11010
```

Examples of known size (10 bits) concatenation

```
Parameter $val1 = 0;
Parameter $val2 = 32;
1'b0, 4'h7, 5'h1a     // equivalent to: 10'b0_0111_11010
1'b0, ~0, 5'h1a       // also equivalent to: 10'b0_1111_11010
1'b0, $val2, 5'h1a    // Error: 32 is too large for total, cannot be
                      // expressed with remaining 4 bits
1'b0, 0, $val1, 5'h1a // Error: two unsized numbers
```

The size of a concatenation can be known in advance based on the declared size of the object (e.g., a scan register) for which the concatenation is being performed. Note on the fourth line of the second group of preceding examples, the software is required to determine the implied size of the unsized 0 so that it may perform known-size concatenation. Since the total size is known (10 bits, presumably because the size of the associated object in the ICL is 10 bits) and the sizes of the first and last numbers are known (1 bit and 5 bits, respectively), the size of the 0 is therefore 10 – 6 = 4 bits. In the third example, the unsized number, 32, cannot be expressed with 4 bits so the concatenation is an error.

Examples of unknown size concatenation

```
1'b0, ~4'h7, 5'h1a    // equivalent to: 10'b0_1000_11010
1'b0, 4               // Error: any unsized number is an error
1'b0, ~4              // Error: unsized numbers cannot be inverted
```

Rules

a) When an identifier is preceded by a tilde (~), each bit in the binary representation of the value of that identifier shall be complemented.

b) Strings shall be concatenated only with other strings.

c) At most one unsized number shall be permitted in a concatenation of numbers. The unsized number shall be sized by taking the difference between the size of the ICL object with which the concatenation is being associated and the sum of the sizes of the other (sized) numbers in the concatenation.

6.3.12 Generic ICL identifiers

There are several types of generic identifiers. Though many of them reduce to the same generic tokens (like numbers or signals or ports or registers), presenting them as unique identifiers clarifies the semantic rules for each usage. The generic identifiers are listed as follows.

Grammar

```
vector_id : SCALAR_ID '[' (index | range) ']' ;
index : integer_expr ;
range : index ':' index ;
number : unsized_number | sized_number | integer_expr ;
integer_expr : integer_expr_lvl1 ;
integer_expr_lvl1 : integer_expr_lvl2 ( ('+' | '-') integer_expr_lvl1 )? ;
integer_expr_lvl2 : integer_expr_arg (('*' | '/' | '%') integer_expr_lvl2 )? ;
integer_expr_paren : '(' integer_expr ')'; // Parentheses
integer_expr_arg : integer_expr_paren | pos_int | parameter_ref ;
parameter_ref : '$'(SCALAR_ID) ;
unsized_number : pos_int | UNSIZED_DEC_NUM | UNSIZED_BIN_NUM |
UNSIZED_HEX_NUM ;
sized_number : sized_dec_num | sized_bin_num | sized_hex_num;
concat_number : '~'? number (',' '~'? number)* ;
//semantic rules prevents inverting unsized numbers and having more than one
//unsized number within a concat_number. See section 6.4.10.
concat_number_list : concat_number ( '|' concat_number )* ;
hier_port : (instance_name '.')+ port_name ;
port_name : SCALAR_ID | vector_id ;
register_name : SCALAR_ID | vector_id ;
instance_name : SCALAR_ID;
namespace_name : SCALAR_ID;
module_name : SCALAR_ID;
reg_port_signal_id : SCALAR_ID | vector_id;
signal : (number | reg_port_signal_id | hier_port | alias_name ) ;
reset_signal : '~'? signal ;
scan_signal : '~'? signal ;
data_signal : '~'? signal ;
clock_signal : '~'? signal ;
tck_signal : signal ;
tms_signal : signal ;
trst_signal : signal ;
shiftEn_signal : signal ;
captureEn_signal : signal ;
```

```
updateEn_signal : signal ;

concat_reset_signal : (reset_signal | data_signal)
                      ( ',' reset_signal | data_signal )*;
concat_scan_signal : (scan_signal | data_signal)
                     ( ',' scan_signal | data_signal )*;
concat_data_signal : data_signal ( ',' data_signal)*;
concat_clock_signal : (clock_signal | data_signal)
                      ( ',' clock_signal | data_signal)*;
concat_tck_signal : (tck_signal | data_signal)
                    ( ',' tck_signal | data_signal)*;
concat_shiftEn_signal : (shiftEn_signal | data_signal)
                        ( ',' shiftEn_signal | data_signal)* ;
concat_captureEn_signal : (captureEn_signal | data_signal)
                          ( ',' captureEn_signal | data_signal )*;
concat_updateEn_signal : (updateEn_signal | data_signal)
                         ( ',' updateEn_signal | data_signal )*;
concat_tms_signal : (tms_signal | data_signal)
                    ( ',' tms_signal | data_signal )*;
concat_trst_signal : (trst_signal | data_signal)
                     ( ',' trst_signal | data_signal )*;
```

Rules

a) Referencing a vectored object without an index shall be equivalent to referencing the vectored object with its complete index range.

b) For *reset_signal*:
   1) *reg_port_signal_id* shall be one of the following:
      i) *resetPort_name*
      ii) *alias_name* of any item allowed in its *reg_port_signal_id* or *hier_port* lists
   2) *hier_port* shall be either *toResetPort_name* or *toTrstPort_name* on a local instance.

c) For *scan_signal*:
   1) *reg_port_signal_id* shall be one of the following:
      i) *scanInPort_name*
      ii) the rightmost bit of a *scanRegister_name*
      iii) *scanMux_name*
      iv) *oneHotScanGroup_name*
      v) *alias_name* of any item allowed in its *reg_port_signal_id* or *hier_port* lists
   2) *hier_port* shall be *scanOutPort_name* on a local instance.

d) For *data_signal*:
   1) *reg_port_signal_id* shall be one of the following:
      i) *dataInPort_name*
      ii) *selectPort_name*
      iii) *scanRegister_name*
      iv) *dataRegister_name*
      v) *logicSignal_name*

vi) *dataMux_name*

vii) *OneHotDataGroup_name*

viii) *resetPort_name* when and only when it is used as the data input of a *DataMux* and the output of the mux drives a *reset_signal*

ix) *captureEnPort_name* when and only when it is used as the data input of a *DataMux* and the output of the mux drives a *captureEn_signal*

x) *updateEnPort_name* when and only when it is used as the data input of a *DataMux* and the output of the mux drives a *updateEn_signal*

xi) *tmsPort_name* when and only when it is used as the data input of a *DataMux* and the output of the mux drives a *tms_signal*

xii) *trstPort_name* when and only when it is used as the data input of a *DataMux* and the output of the mux drives a *trst_signal*

xiii) *alias_name* of any item allowed in its *reg_port_signal_id* or *hier_port* lists

2) *hier_port* shall be one of the following on a local instance:

i) *dataOutPort_name*

ii) *toSelectPort_name*

iii) *toResetPort_name* when and only when it is used as the data input of a *DataMux* and the output of the mux drives a *reset_signal*.

iv) *toCaptureEnPort_name* when and only when it is used as the data input of a *DataMux* and the output of the mux drives a *captureEn_signal*

v) *toUpdateEnPort_name* when and only when it is used as the data input of a *DataMux* and the output of the mux drives a *updateEn_signal*

vi) *toTmsPort_name* when and only when it is used as the data input of a *DataMux* and the output of the *mux drives a tms_signal*

vii) *toTrstPort_name* when and only when it is used as the data input of a *DataMux* and the output of the mux drives a *trst_signal*

e) For *clock_signal*:

1) *reg_port_signal_id* shall be one of the following:

i) *clockPort_name* not associated with a *differentialInvOf_def*

ii) *tckPort_name*

iii) *clockMux_name*

iv) *alias_name* of any item allowed in its *reg_port_signal_id* or *hier_port* lists

2) *hier_port* shall be one of the following on a local instance:

i) *toClockPort_name* not associated with a *differentialInvOf_def*

ii) *toTCKPort_name*

f) For *tck_signal*:

1) *reg_port_signal_id* shall be one of the following:

i) *tckPort_name*

ii) *alias_name* of any item allowed in its *reg_port_signal_id* or *hier_port* lists

2) *hier_port* shall be *toTCKPort_name* on a local instance.

g) For *captureEn_signal*:
   1) *reg_port_signal_id* shall be one of the following:
      i) *captureEnPort_name*
      ii) *alias_name* of any item allowed in its *reg_port_signal_id* or *hier_port* lists
   2) *hier_port* shall be *toCaptureEnPort_name* on a local instance.

h) For *shiftEn_signal*:
   1) *reg_port_signal_id* shall be one of the following:
      i) *shiftEnPort_name*
      ii) *alias_name* of any item allowed in its *reg_port_signal_id* or *hier_port* lists
   2) *hier_port* shall be *toShiftEnPort_name* on a local instance.

i) For *updateEn_signal*:
   1) *reg_port_signal_id* shall be one of the following:
      i) *updateEnPort_name*
      ii) *alias_name* of any item allowed in its *reg_port_signal_id* or *hier_port* lists
   2) *hier_port* shall be *toUpdateEnPort_name* on a local instance.

j) For *tms_signal*:
   1) *reg_port_signal_id* shall be one of the following:
      i) *tmsPort_name*
      ii) *alias_name* of any item allowed in its *reg_port_signal_id* or *hier_port* lists
   2) *hier_port* shall be *toTmsPort_name* on a local instance.

k) For *trst_signal*:
   1) *reg_port_signal_id* shall be one of the following:
      i) *trstPort_name*
      ii) *alias_name* of any item allowed in its *reg_port_signal_id* or *hier_port* lists
   2) *hier_port* shall be *toTrstPort_name* on a local instance.

l) There may be at most one unsized element in a concatenation. When the unsized element in a concatenation is sized using the size of the object for which the concatenation applies, the size of the unsized element shall be one or larger. It shall not be zero or negative.

m) Unsized elements shall not be inverted.

Explanations

The preceding semantic rules prescribe the valid sources of the different signal types. When the grammar of an ICL statement uses one of those signal types in its definition, the previous semantic rule automatically applies to the ICL statement that uses it.

6.4 ICL primitive element keywords and statements

6.4.1 Building blocks

The first collection of ICL keywords describes the primitive building block components that are used to describe the instrument access network inside a module. Figure 39 represents the components that may comprise a module. These primitive building blocks make manifest the hardware components described in 5.13. Of all the components shown in Figure 39, a module shall have at least one port function or one AccessLink; it may have zero or more of the other components.

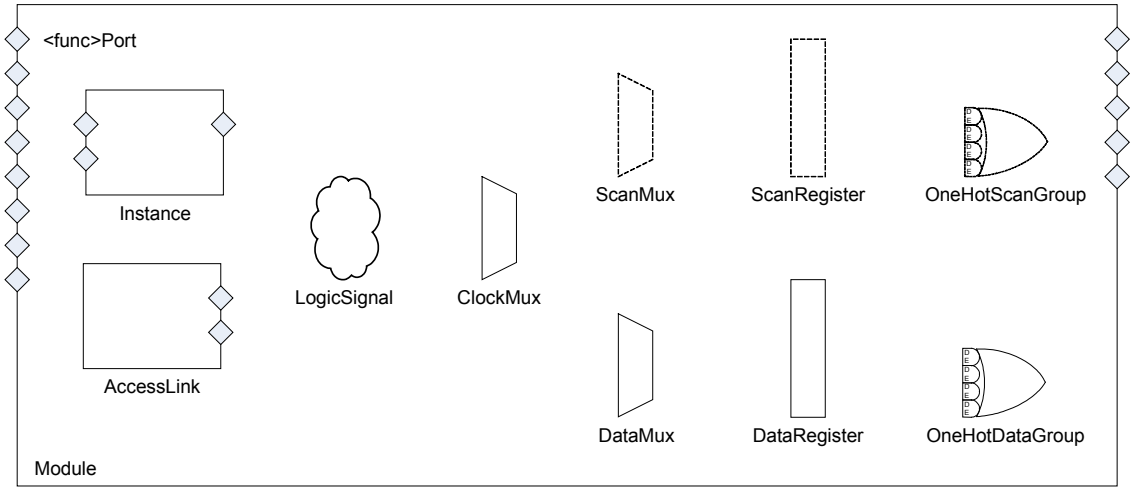


Figure 39—Primitive building blocks for module content

6.4.2 ICL file

An ICL source file consists of at least one ICL source item.

Grammar

```
icl_source : iclSource_items+ ;
iclSource_items : nameSpace_def | useNameSpace_def | module_def;
```

6.4.3 NameSpace statement

A `NameSpace` statement is used to optionally specify the name space in which forthcoming module definitions will exist. If the nameSpace_name argument is empty, the name space becomes the default (root) name space, which is equivalent to not specifying any name space at all.

Grammar

```
nameSpace_def : 'NameSpace' namespace_name? ';'
```

Generic example

```
NameSpace myNameSpace;    // modules defined below will exist in myNameSpace
```

Rules

a) An empty *namespace_name* shall cause the name space to be the root name space.

b) A specified NameSpace remains active until the end of the ICL source file or until a new NameSpace statement is encountered.

6.4.4 UseNameSpace statement

A UseNameSpace statement is used to optionally specify the name space in which forthcoming instance definitions will exist. It is a shorthand method for using the namespace_name as part of the prefix to every instance. If the namespace_name argument is empty, the default name space used is the root name space. Inside a file, until a UseNameSpace statement is encountered, the default name space to use for instance definitions is the namespace_name in which the module is defined.

Grammar

```
useNameSpace_def : 'UseNameSpace' namespace_name? ';'
```

Generic example

```
UseNameSpace myNameSpace;    // following instances will refer to modules in myNameSpace
```

Rules

a) A UseNameSpace directive found outside a module statement shall supersede any prior UseNameSpace directive declared outside a module statement.

b) A UseNameSpace directive found outside a module statement shall only have effect until the end of the ICL source file.

c) A UseNameSpace directive found inside a module shall supersede prior UseNameSpace directive declared inside or outside the module statement.

d) A UseNameSpace directive found inside a module shall only have effect until the end of the module statement.

e) An empty namespace_name shall cause the default name space to be the root name space.

Recommendations

f) Providers of IP should utilize namespaces that maximize the likelihood that their IP can be uniquely identified.

Explanations

ICL supports the use of module name spaces to help prevent module name collision in complex designs, particularly when multiple IP providers may be contributing content. Simply defined, a *name space* is a tag that identifies the context in which module definitions are made. For example, VendorA and VendorB may both (unbeknownst to each other) define Module MBIST, creating the possibility for collision if they are both present in the same device. ICL makes it simple to disambiguate such situations through the use of the "NameSpace" declaration and either of two methods to associate an instance with a particular name space.

The default mode of ICL is for every definition to be implicitly present in the root name space. Items may be explicitly placed in the root name space, as well. If preferred, this feature may be safely ignored so long as any instantiated modules do not use name spaces. However, name spaces are a powerful tool for managing libraries and IP, preventing the need for broadly invasive uniquification of definitions.

ICL has two active name spaces at all times; one that is active for module definitions and one that is active for module instantiations. Name spaces apply to all module definitions and module instantiations.

Definition name spaces are declared using the NameSpace declaration:

```
NameSpace myNameSpace ;
```

After that statement, all definitions will be placed in the myNameSpace name space. This will remain in effect until the end of the ICL source file is reached (causing the name space to revert to the root name space) or until another NameSpace declaration is made. The root name space may be made explicitly active, also using the NameSpace command:

```
NameSpace ;
```

NameSpace declarations may only be made at the file level, outside of a module definition. There may be multiple NameSpace declarations in an ICL source file and each is active until the next is encountered. The active name space is reset to the root name space at the end of an ICL source file. Consider an example where two different vendors have provided ICL defining their product, both of which happen to be named "MBIST":

```
NameSpace VendorA;
Module MBIST { … }

NameSpace VendorB;
Module MBIST { … }
```

Instantiating items declared in a name space is done explicitly, and there are two ways to accomplish this.

The first method is to call out the active instantiation name space on an instance-by-instance basis. This is accomplished by prefixing an identifier with the name of the name space in which the proper module definition will be found, followed by two colons ("::").

```
Instance MBIST_A Of VendorA::MBIST;
Instance MBIST_B Of VendorB::MBIST;
```

Note that any module ***not*** preceded by a "<namespace>::" is part of the default name space, which is the root name space unless either the UseNameSpace keyword or the NameSpace keyword described next is specified.

```
// module from the name space called "VendorA":
Instance i1 Of VendorA::myModule;
// different module with the same name from the default (root) name space:
Instance i2 Of myModule;
```

The second method, which is more convenient if many instantiations are to be made, is to declare which name space definition will be referenced by default using the "UseNameSpace" declaration.

```
NameSpace; // set the root name space as default for module declarations
UseNameSpace  NS1; // this is for instances, not module definitions
Module ABC {  // defined in the root name space
  // module from NS1 (since it was set to be default via UseNameSpace):
  Instance i1 Of myModule;
  Instance i2 Of myModule;    // another instance of same module from NS1
  Instance i3 Of myModule;    // another instance of same module from NS1
  // instance of a different module (though with the same name "myModule")
  // from the root name space:
  Instance i4 Of ::myModule;
}
```

Without the presence of a "UseNameSpace" command, the NameSpace declaration also has the effect of declaring the instantiation name space from which definitions will be drawn. To associate an instance with a module from a different namespace, that other namespace is used to prefix the module type of the instance. Note that an empty namespace name preceding the "::" refers to the root namespace.

```
NameSpace NS1 ;
Module ABC {                    // now defined in the NS1 name space
  Instance i1 Of myModule;      // module from NS1
  // instance of a different module (though with the same name "myModule")
  // from the root name space:
  Instance i2 Of ::myModule;
}
```

This can be overridden, of course, with the UseNameSpace command:

```
NameSpace NS1 ;
UseNameSpace NS2;
Module ABC {   // now defined in the NS1 name space.
  Instance i1 Of myModule; // module from NS2
  // instance of a different module (though with the same name "myModule")
  // from the root name space:
  Instance i2 Of ::myModule;
}
```

Instantiation UseNameSpace declarations may exist at the ICL source file level or inside the scope of a module. There may be multiple instantiation name space declarations in an ICL source file. Each is active until the end of the ICL source file or until another UseNameSpace declaration is encountered at the ICL source file level. If a UseNameSpace is declared inside a module, then it is active until the end of the module or another UseNameSpace declaration is encountered inside the module. At the end of the module, the instantiation name space returns to the one that was active immediately prior to the module being defined. An explicit "<namespace>::" prefix for a given instance always overrides any active UseNameSpace declaration.

A more complete, though intentionally obtuse, example follows. Assume that File 1 is meant to be read before File 2:

File 1:

```
Module ex1 { // defined in root name space
  Instance ex2_inst Of ex2; // definition take from root name space
}

NameSpace ex1ns;
Module ex1 { // defined in ex1ns name space.
  Instance ex2_inst Of ex2; // definition taken from ex1ns
  UseNameSpace; // instantiation name space set to root
  Instance ex2_inst_2 Of ex2; // definition taken from root name space
  Instance ex2 Of ex2ns::ex2; // definition taken from ex2ns name space
}

Module ex2 {} // defined in ex1ns name space
Module ex3 {
  Instance ex2_inst Of ex2; //definition taken from ex1ns name space
}
```

File 2:

```
Module ex2 {} // defined in root name space
  NameSpace ex2ns;
  Module ex2 {} // defined in ex2ns name space
```

Note that NameSpace ex2ns was used in File 1 before it was defined in File 2. Like all of ICL, name spaces are order-independent; consistency checking is done at elaboration time.

Finally, though the previous examples show that name spaces can require sophisticated reasoning to resolve, the intention is for them to be used to simplify things. Typically, as noted in Recommendation f), an IP provider should provide all their ICL in a name space that identifies the provider, the version and other pertinent information. This way, if two IP providers each use identical names for modules, both of which are to be instantiated in the same network, then they can easily both be used without having to go through laborious and error-prone uniquification efforts. Also, occasionally ICL from different versions of the same library may be instantiated in the same network. Simple application of the UseNameSpace command resolves this without the need for uniquification. Note that an IP integrator may choose to add name spaces to ICL obtained from IP providers who have chosen to omit them.

6.4.5 Module statement

A **`Module`** statement is the container for a collection of primitive building blocks and possibly instances of other modules that comprise a network and/or instrument.

A module may be categorized as one of two types: handoff or internal. Handoff modules are intended for exchange between parties and thus declare the complete list of all port functions related to the access network. Internal modules may declare only a subset of their port functions (omitting those whose presence and connectivity may be inferred, see 6.7) and thus may not be able to be processed standalone because they lack sufficient context. Internal modules are useful shorthand representations to simplify ICL models, but must always be instantiated in the context of a fully-specified parent module in order to be part of a handoff.

A *black-box* module is a *handoff* or *internal* module with only the ports and optionally the scan interfaces defined. When they define one or many ScanInterfaces, black-box modules are operated with iScan PDL commands. Otherwise, a black-box module typically only includes ports with function DataInPort, DataOutPort, and ClockPort. A black-box module may also include alias definitions, enumeration tables, and parameters associated with the ports.

Grammar

```
module_def : 'Module' module_name '{' module_item+ '}' ;

module_item : useNameSpace_def |
              port_def |
              instance_def |
              scanRegister_def |
              dataRegister_def |
              logicSignal_def |
              scanMux_def |
              dataMux_def |
              clockMux_def |
              oneHotDataGroup_def |
              oneHotScanGroup_def |
              scanInterface_def |
              accessLink_def |
              alias_def |
              enum_def |
              parameter_def |
              localParameter_def |
              attribute_def ;

port_def : scanInPort_def |
           scanOutPort_def |
```

```
        shiftEnPort_def |
        captureEnPort_def |
        updateEnPort_def |
        dataInPort_def |
        dataOutPort_def |
        toShiftEnPort_def |
        toUpdateEnPort_def |
        toCaptureEnPort_def |
        selectPort_def |
        toSelectPort_def |
        resetPort_def |
        toResetPort_def |
        tmsPort_def |
        toTmsPort_def |
        tckPort_def |
        toTckPort_def |
        clockPort_def |
        toClockPort_def |
        trstPort_def |
        toTrstPort_def |
        toIRSelectPort_def |
        addressPort_def |
        writeEnPort_def |
        readEnPort_def;
```

Generic example

```
Module module_name {
  Attribute att_name = att_value;
  Parameter param_name = param_value;
  ScanInterface interface_name { … }
  ScanInPort port_name { … }
  // … all the other port types
  ReadEnPort port_name { … }
  Instance instance_name Of another_module_name { … }
  LogicSignal signal_name { … }
  ScanRegister register_name { … }
  DataRegister register_name { … }
  ScanMux mux_name SelectedBy selector { … }
  DataMux mux_name SelectedBy selector { … }
  ClockMux mux_name SelectedBy selector { … }
  OneHotDataGroup name { … }
  OneHotScanGroup name { … }
  Enum enum_name { … }
  Alias alias_name = element_list { … }
  AccessLink instance_name Of link_type { … }
}
```

Examples

```
Module my_module { … }
```

Rules

a) A redefinition of a *module_def* shall overwrite a previously defined *module_def* having the same module_name within the same nameSpace_name.

b) A *module_def* shall have at least one *port_def* unless it is the top module and contains an AccessLink statement.

c) The following objects shall have unique names within a *module_def*:

- *port_name*
- *scanRegister_name*
- *dataRegister_name*
- *one_hot_scan_group_name*
- *scanMux_name*
- *dataMux_name*
- *clockMux_name*
- *one_hot_data_group_name*
- *logicSignal_name*
- *alias_name*

d) The following objects shall have unique names within a *module_def*:

- *instance_name*
- *scanInterface_name*

e) An *inputPort_name* shall be one of the following:

- *scanInPort_name*
- *shiftEnPort_name*
- *captureEnPort_name*
- *updateEnPort_name*
- *dataInPort_name*
- *selectPort_name*
- *resetPort_name*
- *tmsPort_name*
- *tckPort_name*
- *clockPort_name*
- *trstPort_name*
- *addressPort_name*
- *writeEnPort_name*
- *readEnPort_name*

f) An *outputPort_name* shall be one of the following:

- *scanOutPort_name*
- *dataOutPort_name*
- *toShiftEnPort_name*
- *toUpdateEnPort_name*

— *toCaptureEnPort_name*

— *toSelectPort_name*

— *toResetPort_name*

— *toTckPort_name*

— *toTmsPort_name*

— *toClockPort_name*

— *toTrstPort_name*

— *toIRSelectPort_name*

g) Multiple *inputPort_name* port functions that are of type *vector_id* may share the same *SCALAR_ID* as long as the indexes form a contiguous range and every index range is of the same either ascending or descending type.

h) Multiple *outputPort_name* port functions that are of type *vector_id* may share the same *SCALAR_ID* as long as the indexes form a contiguous range and every index range is of the same either ascending or descending type.

i) Within each *module_def* excluding the *black-box* module, each *scan_signal* shall have a path from at least one *scanInPort_name* to at least *one scanOutPort_name* if they are to be read or written to during the PDL operations.

j) A *handoff* module with at least one *scanInPort_def* shall have at least one *shiftEnPort_def*.

k) A *handoff* module shall explicitly list all port functions related to the access network, and their names shall match the names of the ports in the corresponding simulation model. The simulation model may contain (functional) ports not present in the ICL model; in other words, the port list in the simulation model is a superset of the port list of the ICL model.

l) The *target module* as defined in 7.1 shall always be a *handoff* module and meet all the requirements imposed on that module type.

Explanations

Rule a) allows a module name to be redefined in the same namespace, with the effect that the most recent definition overwrites any previous definition. This is useful to facilitate debugging by quick replacement of specific modules without requiring the original ICL source file to be edited, but just augmented. Note that this usage of duplication within a namespace is unique to the Module statement; all other ICL statements must have unique names. Rules c) and d) enforce unique naming of objects that could otherwise not be differentiated based on context. The ten items listed in Rule c) can occur in the same context and must therefore have unique names. Likewise, for the two items listed in Rule d). However, the items in Rule c) cannot occur in the context of the items in Rule d) and vice versa, so it is acceptable for items from one list to share names with items from the other list. For example, a port and an instance may have the same name. For an example of an allowed set of port declarations that satisfies semantic rule g) and h), consider **`sigI[5:0],`** which is a contiguous 6-bit bus (i.e., it skips no bits in its range) and is connected only to input ports, as well as a second 6-bit bus, **`sigO[5:0],`** which also is contiguous and is connected only to output ports. The following assignments show how both buses can be utilized in ICL statements:

```
CaptureEnPort   sigI[2];
ShiftEnPort     sigI[1];
ShiftEnPort     sigI[0];
DataInPort      sigI[4:3];
ScanInPort      sigI[5];
ToCaptureEnPort sigO[2];
ToShiftEnPort   sigO[1];
```

```
ToShiftEnPort   sigO[0];
DataOutPort     sigO[4:3];
ScanOutPort     sigO[5];
```

6.4.6 <func>Port: port functions

The interface between a module and the outside world consists of a set of well-defined ports, each of which is uniquely identified by a name (and a bit range indicating the width of the port if appropriate) after a keyword indicating its function. The term "port function" is introduced for the keywords that explicitly identify the particular function of those ports that are relevant to the operation of the network. Some port functions include certain additional information, and all of the ports may include additional attributes. The complete set of port functions, along with the optional fields, is summarized in Table 4. The precise definition of each port function follows in the subsequent subclauses. The active values of each of these port functions may be found in Table 5.

Table 4—Port function keywords and properties summary

Port function	Properties [optional properties shown in square brackets; * indicates repeatability]
ScanInPort	[Attribute]*
ScanOutPort	[Attribute]* [Source] [Enable]
ShiftEnPort	[Attribute]*
CaptureEnPort	[Attribute]*
UpdateEnPort	[Attribute]*
DataInPort	[Attribute]* [RefEnum] [DefaultLoadValue]
DataOutPort	[Attribute]* [Source] [Enable] [RefEnum]
ToShiftEnPort	[Attribute]* [Source]
ToCaptureEnPort	[Attribute]* [Source]
ToUpdateEnPort	[Attribute]* [Source]
SelectPort	[Attribute]*
ToSelectPort	[Attribute]* [Source]
ResetPort	[Attribute]* [ActivePolarity]
ToResetPort	[Attribute]* [Source] [ActivePolarity]
TMSPort	[Attribute]*
ToTMSPort	[Attribute]* [Source]
TCKPort	[Attribute]*
ToTCKPort	[Attribute]*
ClockPort	[Attribute]* [DifferentialInvOf]
ToClockPort	[Attribute]* [Source] [FreqMultiplier] [FreqDivider] [DifferentialInvOf] [Period]
TRSTPort	[Attribute]*
ToTRSTPort	[Attribute]* [Source]

Table 4—Port function keywords and properties summary *(continued)*

Port function	Properties [optional properties shown in square brackets; * indicates repeatability]
ToIRSelectPort	**[Attribute]***
AddressPort	**[Attribute]***
WriteEnPort	**[Attribute]***
ReadEnPort	**[Attribute]***

6.4.6.1 ScanInPort statement

A **ScanInPort** is used to load serial scan data into a module. The only required field is the name of the port. Optional attribute properties may also be included.

The declaration of the ScanInPort, when it exists, is mandatory. This is required even for an internal (non-handoff) module when the ScanInterface it is associated with is to be used during PDL retargeting (see 7.4).

Grammar

```
scanInPort_def : 'ScanInPort' scanInPort_name (';' |
                 ( '{' attribute_def* '}' ) ) ;
scanInPort_name : port_name ;
```

Generic example

```
ScanInPort port_name {
  Attribute att_name = att_value;
}
```

Examples

```
ScanInPort SI1;
ScanInPort SI2 { Attribute target = "PLL_IP"; }
ScanInPort SI3 {} // empty braces equivalent to semicolon
```

Rules

a) At least one *scanInPort_def* is required whenever a *scanOutPort_def* is present in the *module_def*.

6.4.6.2 ScanOutPort statement

A **ScanOutPort** is used to unload serial scan data from a module. The name of the port is required to be specified, and if the module is not a black-box module, the source of the scan data signal is also required. No assumption is made about the TCK edge that causes the ScanOutPort to change. Optional attributes may also be added.

The enable signal (identified by the “Enable” property keyword), if listed, is required to be made active in order to connect the ScanOut source (identified by the “Source” property keyword) to this ScanOutPort. The use of the enable signal is described in the OneHotScanGroup subclause (6.4.14).

The declaration of the ScanOutPort, when it exists, is mandatory. This is required even for an internal (non-handoff) module when the ScanInterface it is associated with is to be used during PDL retargeting.

Grammar

```
scanOutPort_def : 'ScanOutPort' scanOutPort_name ( ';' |
                  ( '{' scanOutPort_item* '}' ) ) ;
scanOutPort_name : port_name;
scanOutPort_item : attribute_def |
                   scanOutPort_source |
                   scanOutPort_enable ;
scanOutPort_source : 'Source' concat_scan_signal ';';
scanOutPort_enable : 'Enable' data_signal ';';
```

Generic example

```
ScanOutPort port_name {
  Source source;
  Enable source;

  Attribute att_name = att_value;
}
```

Examples

```
ScanOutPort SO1 {
   Source RegA[0];

   Attribute LaunchEdge = "Rising";
}
```

Rules

a) At least one *scanOutPort_def* is required whenever a *scanInPort_def* is present in the *module_def*.

b) When the *scanOutPort_source* points to a ScanRegister element, it shall refer to the rightmost bit.

c) There shall be at most one of each of *scanOutPort_source or scanOutPort_enable* statement for a given *scanOutPort_def*.

d) When the *scanOutPort_source* property is present, the width of the *concat_scan_signal* used in the *scanOutPort_source* shall match the width of the *scanOutPort_name*.

6.4.6.3 ShiftEnPort statement

A ShiftEnPort is used to control when data is shifted into the shift storage elements within a module. The only required field is the name of the port. Optional attribute properties may also be included. Between the CaptureEn cycle and the UpdateEn cycle, the ScanRegister holds its value when the ShiftEnPort is low. Like an IEEE 1149.1 TDR, a ScanRegister shall tolerate the presence of an arbitrary number of pause cycles inserted in the middle of the shift operation.

Grammar

```
shiftEnPort_def : 'ShiftEnPort' shiftEnPort_name (';' |
                  ( '{' attribute_def* '}' ) ) ;
shiftEnPort_name : port_name;
```

Generic example

```
ShiftEnPort port_name {
  Attribute att_name = att_value;
}
```

Examples

```
ShiftEnPort SE;
```

Rules

a) Between the CaptureEn cycle and the UpdateEn cycle, the ScanRegister shall hold its value when the ShiftEnPort is low.

b) A ScanRegister shall tolerate the presence of an arbitrary number of pause cycles inserted during the shift operation.

Permissions

c) The declaration of the ShiftEnPort may be omitted for an internal (non-handoff) module even if the internal module includes ScanRegister elements.

6.4.6.4 CaptureEnPort statement

A CaptureEnPort is used to control when data is captured into the storage elements within a module. The only required field is the name of the port. Optional attribute properties may also be included. Gating of the signal driving a CaptureEnPort is performed only by a specifically configured DataMux.

Grammar

```
captureEnPort_def : 'CaptureEnPort' captureEnPort_name (';' |
                    ( '{' attribute_def* '}' ) ) ;
captureEnPort_name : port_name ;
```

Generic example

```
CaptureEnPort port_name {
  Attribute att_name = att_value;
}
```

Examples

```
CaptureEnPort CE;
```

Rules

a) If the action of the CaptureEn signal driving the CaptureEnPort of an instance of a module may be logically suppressed, the source of the concat_captureEn_signal driving that instance input port shall be modelled as a DataMux element whose select line is a data_signal and whose two data inputs are a constant zero value and another captureEn_signal.

b) If a DataMux drives a captureEn_signal, the **`data_signal`** driving the select input of that DataMux shall be justified to the value that sensitizes the path to the input captureEn_signal signal (not the constant zero value) in order to enable the ScanRegisters using the **`CaptureEnPort`** to capture and observe their specified **`CaptureSource`**.

Permissions

c) The declaration of the CaptureEnPort may be omitted for an internal (non-handoff) module even if the internal module includes ScanRegister elements.

Explanations

Rule a) explains how to model a suppressible CaptureEnPort of an instance: drive the CaptureEnPort of the instance from a 2-input DataMux, where one input is a logic zero and the other input is a captureEn_signal (e.g., the CaptureEnPort of the parent module or the ToCaptureEnPort from a sibling module). The select input of the DataMux a data_signal. In the example shown in Figure 40, that data_signal is labeled "cap_dis."

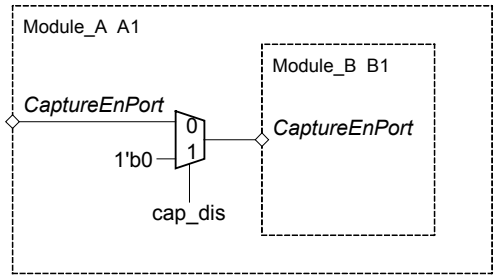


Figure 40—CaptureEnPort gating

When this configuration is used, rule b) requires that the data_signal driving the select input of the DataMux be justified to select the captureEn_signal (e.g., the parent module's CaptureEnPort), not the constant zero, in order to enable ScanRegisters using the CaptureEn_signal to capture.

6.4.6.5 UpdateEnPort statement

An UpdateEnPort is used to control when data is updated into the update storage elements within a module. The only required field is the name of the port. Optional attribute properties may also be included. Gating of the signal driving an UpdateEnPort is performed only by a specifically configured DataMux.

Grammar

```
updateEnPort_def : 'UpdateEnPort' updateEnPort_name (';' |
                   ( '{' attribute_def* '}' ) ) ;
updateEnPort_name : port_name ;
```

Generic example

```
UpdateEnPort port_name {
  Attribute att_name = att_value;
}
```

Examples

```
UpdateEnPort UE;
```

Rules

a) If the action of the UpdateEn signal driving the UpdateEnPort of an instance of a module may be logically suppressed, the source of the concat_updateEn_signal driving that instance input port shall be modelled as a DataMux element whose select line is a data_signal and whose two data inputs are a constant zero value and another updateEn_signal.

b) If a DataMux drives an updateEn_signal, the **`data_signal`** driving the select input of that DataMux shall be justified to the value that sensitizes the path to the input updateEn_signal (not the constant zero value) in order to enable the ScanRegisters using the **`UpdateEnPort`** to perform an update operation.

Permissions

c) The declaration of the UpdateEnPort may be omitted for an internal (non-handoff) module even if the internal module includes ScanRegister elements.

Explanations

Rule a) explains how to model a suppressible UpdateEnPort of an instance: drive the UpdateEnPort of the instance from a 2-input DataMux, where one input is a logic zero and the other input is an updateEn_signal (e.g., the UpdateEnPort of the parent module or the ToUpdateEnPort from a sibling module). The select input of the DataMux a data_signal. In the example shown in Figure 41, that data_signal is labeled "upd_dis."

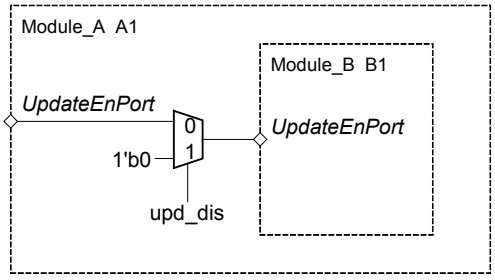


Figure 41—UpdateEnPort gating

When this configuration is used, Rule b) requires that the data_signal driving the select input of the DataMux be justified to select the updateEn_signal (e.g., the parent module's UpdateEnPort), not the constant zero, in order to enable ScanRegisters using the updateEn_signal to update.

6.4.6.6 DataInPort statement

A DataInPort is used to write parallel (i.e., not scan) data to a module. The only required field is the name of the port. Optional attributes may be added, as may a default load value as well as a reference to an enumeration table that may contain mnemonics for data values.

The declaration of the DataInPort, when it exists, is mandatory even for an internal (non-handoff) module if it is to be used during PDL retargeting.

Grammar

```
dataInPort_def : 'DataInPort' dataInPort_name ( ';' |
                 ( '{' dataInPort_item* '}' ) ) ;
dataInPort_name : port_name ;
dataInPort_item : attribute_def |
```

```
                  dataInPort_refEnum |
                  dataInPort_defaultLoadValue ;
dataInPort_refEnum : 'RefEnum' enum_name ';' ;
dataInPort_defaultLoadValue : 'DefaultLoadValue' (concat_number |
                                enum_symbol) ';' ;
```

Generic example

```
DataInPort port_name {
  Attribute attName = att_value;
  RefEnum enum_name;
  DefaultLoadValue load_value;
}
```

Examples

```
DataInPort DI;
DataInPort CNTL[7:0];
DataInPort ABC { RefEnum enum1; }
DataInPort D { DefaultLoadValue 8'b01011010; }
```

Rules

a) There shall be at most one dataInPort_refEnum element per dataInPort_def statement.

b) The width of enum_value within the dataInPort_RefEnum shall match the width of dataInPort_name.

c) When connected to an instance or data register with the AddressValue statement, the *logicalDataIn_signal* shall be created by concatenating all dataInPort_name items from top to bottom as listed in the module. The left most bit (n–1) of the logicalDataIn_signal is the leftmost bit of the first dataInPort_name, and the right most bit (0) of the logicalDataIn_signal is the rightmost bit of the last dataInPort_name.

d) When a path is sensitized to a DataInPort, any of its bits without a value specified in a current write operation shall be written to their previously written value (if they have been previously written), their corresponding dataInPort_defaultLoadValue (if specified and non-X), or 0, in that order of priority.

Explanations

Rule c) applies to the situation where there are multiple DataInPort statements in the definition of a module that is referred to by an addressable instance or DataRegister. To form the (single) logicalDataIn_signal for this addressable instance, those multiple DataInPort statements are simply concatenated top-to-bottom and left-to-right. For example, consider a module that defines (in order) these DataInPorts:

```
DataInPort A[7:0];
DataInPort B[3:0];
DataInPort C[15:0];
DataInPort D[3:0];
```

The resulting logicalDataIn_signal would consist of the 32 bits {A[7:0], B[3:0], C[15:0], D[3:0]} in that order.

Rule d) defines the policy for filling unspecified bits of a multi-bit DataInPort for which a write operation is underway but only contains values for a subset of the bits. This rule also applies when no deliberate write operation is being performed, but a sensitized path to a DataInPort exists nonetheless. In either case, the first priority for filling unspecified bits is to rewrite the previously written value. If there was no previously written value, the second priority is to use the DefaultLoadValue. If there was no DefaultLoadValue

specified, or if it was X, the third priority is to simply write a 0. Note that this rule adds to the list of objectives that must be satisfied by the current iApply group (see 7.3).

6.4.6.7 DataOutPort statement

A **DataOutPort** is used to read parallel data from a module. Specification of the port name is required. Optional additions include a source definition, attributes, and a reference to an enumeration table that may contain mnemonics for data values.

The enable signal (identified by the "Enable" property keyword), if listed, is required to be made active in order to connect the data source (identified by the "Source" property keyword) to this **DataOutPort**. The use of the enable signal is described in the OneHotDataGroup subclause (6.4.15).

The declaration of the DataOutPort, when it exists, is mandatory even for an internal (non-handoff) module if it is to be used during PDL retargeting.

Grammar

```
dataOutPort_def : 'DataOutPort' dataOutPort_name (';' |
                  ( '{' dataOutPort_item* '}' ) ) ;
dataOutPort_name : port_name ;
dataOutPort_item : attribute_def |
                   dataOutPort_source |
                   dataOutPort_enable |
                   dataOutPort_refEnum ;
dataOutPort_source : 'Source' concat_data_signal ';' ;
dataOutPort_enable : 'Enable' data_signal ';' ;
dataOutPort_refEnum : 'RefEnum' enum_name ';' ;
```

Generic example

```
DataOutPort port_name {
  Source source;
  Enable source;   // only mandatory when part of a one-hot group
  Attribute att_name = att_value;
  RefEnum enum_name;
}
```

Examples

```
DataOutPort DO;
DataOutPort status[15:0]  {
  Source data[15:0];
}
```

Rules

a) There shall be at most one each of **dataOutPort_source**, **dataOutPort_enable**, and **dataOutPort_refEnum** for a given **dataOutPort_def.**

b) If the **dataOutPort_source** is present, the width shall match the **dataOutPort_name** width.

c) The width of **dataOutPort_enable** shall be 1 bit.

d) The width of **enum_value** within the **dataOutPort_refEnum** shall match the width of **dataOutPort_name**.

e) A dataOutPort_enable element, when present, shall be justified high when the DataOutPort is read in PDL. This requirement shall be met even if the iRead statement is issued on the port itself or on an object that is observed through the DataOutPort.

f) When on an instance with the AddressValue statement, the *logicalDataOut_signal* shall be created by concatenating all *dataOutPort_name* items from top to bottom as listed in the *module_def* of the instance. The left most bit (n–1) of the *logicalDataOut_signal* is the left most bit of the first *dataOutPort_name,* and the right most bit (0) of the *logicalDataOut_signal* is the right most bit of the last *dataOutPort_name*.

Explanations

Rule f) applies to the situation where there are multiple DataOutPort statements in the definition of a module, which is instantiated inside a OneHotDataGroup with an AddressValue. To form the (single) logicalDataOut_signal for this addressable instance, those multiple DataOutPort statements are simply concatenated top-to-bottom and left-to-right. For example, consider a module that defines (in order) these DataOutPorts:

```
DataOutPort A[7:0];
DataOutPort B[3:0];
DataOutPort C[15:0];
DataOutPort D[3:0];
```

The resulting logicalDataOut_signal would consist of the 32 bits {A[7:0], B[3:0], C[15:0], D[3:0]} in that order.

6.4.6.8 ToShiftEnPort statement

A **`ToShiftEnPort`** is an output from a module that is used to control when downstream modules shift data into their shift storage elements. The only required field is the name of the port. Optional attribute and source elements may also be included.

A ToShiftEnPort can be referenced by a ScanInterface with at least one ScanInPort or ScanOutPort. The ToShiftEnPort is active when the associated ScanInPort and ScanOutPort in this ScanInterface are part of the currently active scan chain.

The declaration of the ToShiftEnPort is optional for an internal (non-handoff) module.

Grammar

```
toShiftEnPort_def : 'ToShiftEnPort' toShiftEnPort_name (';' |
                    ( '{' toShiftEnPort_items* '}' ) ) ;
toShiftEnPort_name : port_name ;
toShiftEnPort_items : attribute_def |
                      toShiftEnPort_source ;
toShiftEnPort_source : 'Source' concat_shiftEn_signal ';' ;
```

Generic example

```
ToShiftEnPort port_name {
  Attribute att_name = att_value;
  Source source;
}
```

Examples

```
ToShiftEnPort toSE;
ToShiftEnPort toSe2 {
  Source tap_fsm.toSe;
}
```

Rules

a) There shall be at most one *toShiftEnPort_source* statement for a given *toShiftEnPort_def*.

b) The width of *toShiftEnPort_source* shall match the width of *toShiftEnPort_name*.

c) A *toShiftEnPort_def* with a *toShiftEnPort_source* pointing to a *shiftEn_signal* that evaluates to a 0 shall be assumed to be tied off and any ScanInterface associated with the ToShiftEnPort shall never be used as part of the active scan chain.

d) A *toShiftEnPort_def* with a *toShiftEnPort_source* that is driven by a *shiftEn_signal* that evaluates to a 1 shall generate an error.

Explanations

The two semantic rules c) and d) cover the cases where the source of a ToShiftEnPort may evaluate to a constant (which can happen when it is sourced by a ShiftEnPort and this ShiftEnPort is tied off during instantiation). Tying the ShiftEnPort to either a constant 0 or a constant1 makes the ScanInterface that depends on it unusable during PDL retargeting. Despite these rules, it is legal to drive a ToShiftEnPort with a constant 0 as long as the associated ScanInterface will not ever be used during retargeting of PDL. This allows an integrator to customize a generic instrument (by constrained instantiation) to use only a subset of the features available from the instrument provider.

6.4.6.9 ToCaptureEnPort statement

A **`ToCaptureEnPort`** is an output from a module that is used to control when downstream modules capture data into their storage elements. The only required field is the name of the port. Optional attribute and source elements may also be included. Gating of the signal driving a ToCaptureEnPort is performed only by a specifically configured DataMux.

The declaration of the ToCaptureEnPort is optional for an internal (non-handoff) module.

Grammar

```
toCaptureEnPort_def : 'ToCaptureEnPort' toCaptureEnPort_name (';' |
                      ( '{' toCaptureEnPort_items* '}' ) ) ;
toCaptureEnPort_name : port_name ;
toCaptureEnPort_items : attribute_def |
                        toCaptureEnPort_source ;
toCaptureEnPort_source : 'Source' concat_captureEn_signal ';' ;
```

Generic example

```
ToCaptureEnPort port_name {
  Attribute att_name = att_value;
  Source source;
}
```

Examples

```
ToCaptureEnPort toCE;

ToCaptureEnPort toCE2 {
  Source enable2; //Capture suppressed when enable2 evaluates to 0
}
```

Rules

a) There shall be at most one *toCaptureEnPort_source* statement for a given *toCaptureEnPort_def*.

b) The width of *toCaptureEnPort_source* shall match the width of the *toCaptureEnPort_name*.

c) If the action of the CaptureEn signal on a ToCaptureEnPort of a module may be logically suppressed, the source of the signal driving that port shall be modelled as a DataMux element whose select line is a data_signal and whose two data inputs are a constant zero value and a captureEn_signal.

d) If a DataMux drives a ToCaptureEnPort, the data_signal driving the select input of that DataMux shall be justified to the value that sensitizes the path to the captureEn_signal in order to enable the ScanRegisters downstream from the CaptureEnPort to capture and observe their specified CaptureSource.

Explanations

Rule c) explains how to model a suppressible ToCaptureEnPort of an instance: drive the ToCaptureEnPort of the instance from a 2-input DataMux, where one input is a logic zero and the other input is a captureEn_signal. The select input of the DataMux a data_signal. In the example shown in Figure 42, that data_signal is labeled "cap_dis."

```
Module Module_A {
  DataInPort cap_dis;
  CaptureEnPort CE;
  ToCaptureEnPort to_CE {Source CE_mux;}
  ...

  DataMux  CE_mux SelectedBy cap_dis {
    1'b0 : CE;
    1'b1 : 1'b0;
  }
…
}
```

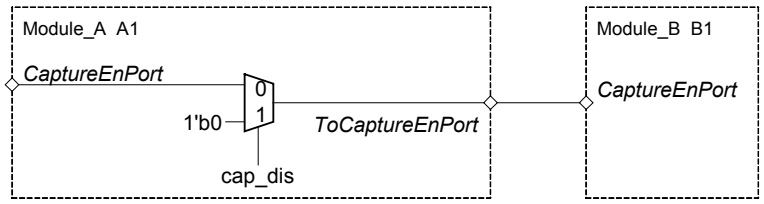


Figure 42—ToCaptureEnPort gating

When this configuration is used, Rule d) requires that the data_signal driving the select input of the DataMux be justified to select the captureEn_signal (not the constant zero) to enable downstream ScanRegisters to capture.

6.4.6.10 ToUpdateEnPort statement

A ToUpdateEnPort is an output from a module that is used to control when downstream modules update data into their update storage elements. The only required field is the name of the port. Optional attribute or source elements may also be included. Gating of the signal driving a ToUpdateEnPort is performed only by a specifically configured DataMux.

The declaration of the ToUpdateEnPort is optional for an internal (non-handoff) module.

Grammar

```
toUpdateEnPort_def : 'ToUpdateEnPort' toUpdateEnPort_name (';' |
                     ( '{' toUpdateEnPort_items* '}' ) ) ;
toUpdateEnPort_name : port_name ;
toUpdateEnPort_items : attribute_def | toUpdateEnPort_source ;
toUpdateEnPort_source : 'Source' concat_updateEn_signal ';' ;
```

Generic example

```
ToUpdateEnPort port_name {
  Attribute att_name = att_value;
  Source source;
}
```

Examples

```
ToUpdateEnPort toUE;
ToUpdateEnPort toUE2 {
  Source enable2; //Update suppressed when enable2 evaluates to 0
}
```

Rules

a) There shall be at most one *toUpdateEnPort_source* statement for a given *toUpdateEnPort_def*.

b) The width of *toUpdateEnPort_source* shall match the width of the *toUpdateEnPort_name*.

c) If the action of the UpdateEn signal on a ToUpdateEnPort of a module may be logically suppressed, the source of the signal driving that port shall be modeled as a DataMux element whose select line is a data_signal and whose two data inputs are a constant zero value and an updateEn_signal.

d) If a DataMux drives a ToUpdateEnPort, the data_signal driving the select input of that DataMux shall be justified to the value that sensitizes the path to the updateEn_signal in order to enable the ScanRegisters downstream from the ToUpdateEnPort to perform an update operation.

Explanations

Rule c) explains how to model a suppressible ToUpdateEnPort of an instance: drive the ToUpdateEnPort of the instance from a 2-input DataMux, where one input is a logic zero and the other input is an updateEn_signal. The select input of the DataMux a data_signal. In the example shown in Figure 43, that data_signal is labeled “upd_dis.”

```
Module Module_A {
 DataInPort upd_dis;
 UpdateEnPort UE;
 ToUpdateEnPort to_UE {Source UE_mux;}
 ...

 DataMux  UE_mux SelectedBy upd_dis {
  1’b0 : UE;
  1’b1 : 1’b0;
 }
```

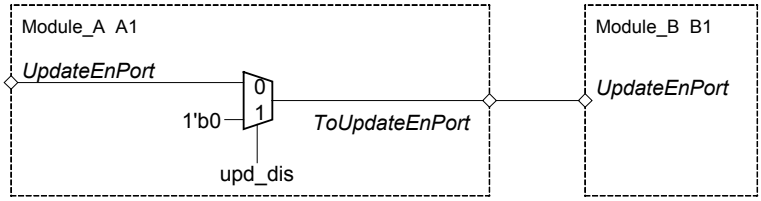


Figure 43—ToUpdateEnPort gating

When this configuration is used, Rule d) requires that the data_signal driving the select input of the DataMux be justified to select the updateEn_signal to enable downstream ScanRegisters to update.

6.4.6.11 SelectPort statement

A **SelectPort** is typically used to enable the scan interface of a module. When the SelectPort is active, the scan interface is active. When the SelectPort is inactive, the scan interface does not respond to the other control signals. The only required field is the name of the port. Optional attribute elements may also be included. A SelectPort is associated with a scan interface when it is referenced within a ScanInterface statement or when the module has a single scan client interface but omits the ScanInterface statement and the module only includes a single SelectPort with a width of 1. When not associated with a ScanInterface, a SelectPort is equivalent to a DataInPort.

The declaration of the SelectPort is optional for an internal (non-handoff) module even if the module includes a ScanRegister element. It is only mandatory when the SelectPort is part of a ScanInterface and the ScanInterface is part of the AllowBroadcastOnScanInterface statement during the module instantiation.

Grammar

```
selectPort_def : 'SelectPort' selectPort_name (';' |
                 ( '{' attribute_def* '}' ) ) ;
selectPort_name : port_name ;
```

Generic example

```
SelectPort port_name {
  Attribute att_name = att_value;
}
```

Examples

```
SelectPort SEL;
```

Rules

a) A *selectPort_name* shall be associated with a ScanInterface if it is referenced using a Port statement inside the ScanInterface statement, or when there is a single undeclared scan client interface and a single *selectPort_name*.

b) Each SelectPort associated with a ScanInterface and having a defined source (i.e., having no undriven signals in its fan-in anywhere in the hierarchy) shall be justified active when the ScanInterface is part of the active scan chain. If any SelectPort associated with an active ScanInterface does not have a defined source, that SelectPort shall be assumed to be active.

c) When a ScanInterface is not part of the active scan chain, and each SelectPort associated with it has a defined source (i.e., has no undriven signals in its fan-in anywhere in the hierarchy), then at least one of the SelectPorts associated with that ScanInterface shall be justified inactive. If any SelectPort associated with an inactive ScanInterface does not have a defined source, that SelectPort shall be assumed to be inactive.

d) There shall be at most one toSelectPort_source statement for a given toSelectPort_def.

Explanations

Rules b) and c) both include assumptions in their final sentences that require explanation and are based on the notion of an active scan chain. The behavior of a scan interface without a SelectPort definition, or with a SelectPort that has no source definition, is assumed to be consistent with the status of the scan chain through that interface: if the scan chain is active, these undefined SelectPorts are assumed to be active; if the scan chain is not active, then neither are these undefined SelectPorts, This policy is based on the fact that ICL is not a detailed netlist listing every connection, but an abstraction that allows navigation of the instrument access network. In this case, when there is an active path between the ScanInPort and the ScanOutPort of a chain, ICL may safely assume that along that path any scan interfaces without explicit SelectPorts have been selected. This inference allows the simplest description of the scan chain connectivity and avoids duplication of scan multiplexer selection logic and SelectPort logic. It is important to note that if there are SelectPorts explicitly defined along that path, then each must be justified active for the scan chain to be active.

Rule c) enforces that a ScanInterface does not remain active when it is not part of the active scan chain by forcing at least one SelectPort associated with that ScanInterface to be turned off. It is typical for the retargeting software to take responsibility for controlling SelectPort signals.

6.4.6.12 ToSelectPort statement

A ToSelectPort is an output from a module and is typically used to activate the scan interface of downstream modules. The name of the port is required. Optional source and attribute elements may also be included. A ToSelectPort is associated with a scan interface when it is referenced within a ScanInterface statement.

The declaration of the ToSelectPort is optional for an *internal* (non-handoff) module.

Grammar

```
toSelectPort_def : 'ToSelectPort' toSelectPort_name (';' |
                   ( '{' toSelectPort_item+ '}' ) ) ;
toSelectPort_name : port_name ;
toSelectPort_item : attribute_def | toSelectPort_source ;
toSelectPort_source : 'Source' concat_data_signal ';' ;
```

Generic example

```
ToSelectPort port_name {
  Source source;
  Attribute att_name = att_value;
}
```

Examples

```
ToSelectPort toSEL  {
  Source mySEL;
}
```

Rules

a) A ToSelectPort shall be considered associated with a ScanInterface when it is referenced by a Port statement inside the ScanInterface statement.

b) When specified, the width of the *toSelectPort_Source* element shall match the width of *toSelectPort_name*

c) The *toSelectPort_Source* element shall be specified when the *ToSelectPort* fans out to a ScanInterface referenced by an *AllowBroadcastingOnScanInterface* property anywhere in the hierarchy.

6.4.6.13 ResetPort statement

A **ResetPort** is used to place certain storage elements within a module into a defined (reset) state. When the Reset port is asserted to its active value, all update and data storage elements inside a module with a specified reset value will take on that specified value. Storage elements without a specified reset value do not change state. The only required field is the name of the port. The active polarity of the reset port may be specified; if not specified, it defaults to active high. Optional attribute elements may also be included.

There are three general categories of signals that may drive a ResetPort: a data_signal (for local reset driven by a DataMux), a ToTRSTport (for an asynchronous reset from a pin), or a ToResetPort that ultimately comes from a state machine. The difference between the last two options is explained in rule d)) and is also discussed in 6.4.6.22. Any of these sources may be delivered via the ResetPort of an intervening parent module in the logical hierarchy. The reset port is allowed to reset the shift elements of ScanRegisters; however, as mentioned in the semantic rules of the ScanRegister element, the effect of the active value must be self-clearing and not affect the future scan operation of the ScanRegister even when the ResetPort is kept active.

The declaration of the ResetPort is optional for an internal (non-handoff) module even if the module contains a storage element with a ResetValue.

Grammar

```
resetPort_def : 'ResetPort' resetPort_name (';' |
                ( '{' resetPort_item* '}' ) ) ;
resetPort_name : port_name ;
resetPort_item : attribute_def |
resetPort_polarity ;
resetPort_polarity : 'ActivePolarity' ('0'| '1') ';' ;
```

Generic example

```
ResetPort port_name {
  ActivePolarity 0 |1;
  Attribute att_name = att_value;
}
```

Examples

```
ResetPort RST {
  ActivePolarity 0;
}
```

Rules

a) If the **`resetPort_name`** is driven by a **`TrstPort_name`** or a **`reset_signal`**, the polarity of the driving element and the **`resetPort_name`** shall be consistent.

b) When the *resetPort_polarity* is not specified and not dictated by rule a), the active polarity of the *resetPort_name* shall be 1.

c) If the **`ResetPort_name`** is driven by a **`toTrstPort_name`**, **`the ResetPort`** shall respond to only activity on the pin or the internal power-up detector that ultimately drives the **`toTrstPort_name`**.

d) When the **`concat_reset_signal`** includes a **`data_signal`**, any bits of type **`data_signal`** shall be directly driven by a **`DataMux`** element whose select line is a **`data_signal`** and whose two data inputs are another **`data_signal`** and a **`reset_signal`**.

e) If a DataMux drives the ResetPort of an instance, the data_signal driving the select input of the DataMux shall be justified to select the reset_signal (not the other data signal) whenever the downstream registers are performing their normal (non-reset) operations.

f) If a DataMux drives the ResetPort of an instance, the data_signal driving the select input of the DataMux shall select the reset_signal (not the data_signal) upon assertion of the trst_signal.

Explanations

Rule c) indicates that when a ResetPort is driven only from a pin (such as the IEEE 1149.1 TRST* pin, perhaps through the TRSTPort and/or toTRSTPort of one or more intervening modules) that it will respond only to that pin, not any other source of reset (such as the IEEE 1149.1 TLR state, for example),

Rule d) explains how to model a local reset of an instance: drive the ResetPort of the instance from a 2-input DataMux, where one input is a local reset value and the other input is the reset_signal (e.g., the ResetPort of the parent module). The select input of the DataMux a data_signal. In the example shown in Figure 44, that data_signal is labeled “reset_force_en.” The other data_signal that provides the actual value of the local reset is labeled “reset_force_val.”

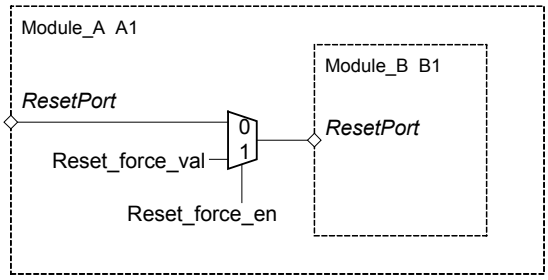


Figure 44—ResetPort forcing

When this configuration is used, Rule e) requires that the data_signal driving the select input of the DataMux be justified to select the reset_signal (e.g., the parent module’s ResetPort or a sibling module’s ToResetPort) to enable downstream ScanRegisters to perform their normal (non-reset) operations without being affected by the local reset value, and to allow a global reset to affect them.

6.4.6.14 ToResetPort statement

The **ToResetPort** is used to source the ResetPort of other modules. The source of the ToResetPort may be a trivial connection of the ResetPort of the same module, or it may be a data_signal (driven by a DataMux) to model a local reset. Optional attribute elements may also be included.

The declaration of the ToResetPort is optional for an *internal* (non-*handoff*) module.

Grammar

```
toResetPort_def : 'ToResetPort' toResetPort_name (';' |
                  ( '{' toResetPort_item+ '}' ) ) ;
toResetPort_name : port_name ;
toResetPort_item : attribute_def |
                   toResetPort_source |
                   toResetPort_polarity;
toResetPort_source : 'Source' concat_reset_signal ';' ;
toResetPort_polarity : 'ActivePolarity' ('0'| '1') ';' ;
```

Generic example

```
ToResetPort port_name {
  Source source_name;
  ActivePolarity 0|1;
  Attribute att_name = att_value;
}
```

Examples

```
ToResetPort local_reset_out {
  Source myreset_signal;
  ActivePolarity 0;
}
```

Rules

a) There shall be at most one *toResetPort_source* statement and at most one ToResetPort_polarity statement for a *toResetPort_def*.

b) When specified, the width of the *toResetPort_source* shall match the width of *toResetPort_name*.

c) When the *toResetPort_source* statement is not specified, the *toResetPort_def* shall be associated with the module's single *reset_signal* when present; otherwise (i.e., if there is no *reset_signal* present or if there are multiple *reset_signals* present), the *toResetPort_def* shall be associated with the global reset function.

d) The *toResetPort_source* statement shall be specified if the *module_def* includes more than one *reset_signal*.

e) The ToResetPort shall always be active when its associated *reset_signal* is active.

f) When the *toResetPort_polarity* statement is not specified, the active polarity of the *toResetPort_name* shall be 1.

g) When the concat_reset_signal in the toResetPort_source statement includes a data_signal, any bits of type data_signal shall be directly driven by a DataMux element whose select line is a data_signal and whose two data inputs are another data_signal and a reset_signal.

h) If a DataMux drives the ToResetPort of a module, the data_signal driving the select input of the DataMux shall be justified to select the reset_signal (not the other data signal) whenever the downstream registers are performing their normal (non-reset) operations.

i) If a DataMux drives the ToResetPort of a module, the data_signal driving the select input of the DataMux shall select the reset_signal (not the data_signal) upon assertion of the trst_signal.

Explanations

Rule g) explains how to model a local reset generated within a module: drive the ToResetPort of the module from a 2-input DataMux, where one input is a local reset value and the other input is the reset signal (e.g., the ResetPort of the module). The select input of the DataMux a data_signal. In the example shown in Figure 45, that data_signal is labeled "reset_force_en." The other data_signal that provides the actual value of the local reset is labeled "reset_force_val."

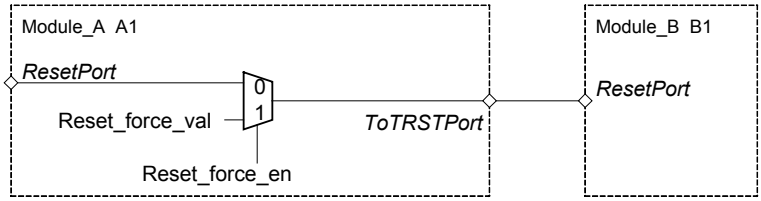


Figure 45—ToResetPort forcing

When this configuration is used, Rule h) requires that the data_signal driving the select input of the DataMux be justified to select the reset_signal (e.g., the module's ResetPort) to enable downstream ScanRegisters to perform their normal (non-reset) operations without being affected by the local reset value, and to allow a global reset to affect them.

6.4.6.15 TMSPort statement

There may be embedded TAP controllers (eTAPCs, but sometimes referred to as wrapper TAPs, child TAPs, or slave TAPs) within the hierarchy of the serial access network. The **`TMSPort`** function is used to identify the TMS port on a module that is or contains an embedded TAP controller. Only one **`TMSPort`** is allowed within a module that also defines a ToIRSelectPort. Any module that includes a ToIRSelectPort function will also include a TMSPort function, since the module obviously contains an embedded TAP controller that generates the ToIRSelectPort output and thus that module associates the TMSPort input to the eTAPC to the ToIRSelectPort output from the eTAPC.

Grammar

```
tmsPort_def : 'TMSPort' tmsPort_name (';' |
              ( '{' attribute_def*'}' ) ) ;
tmsPort_name : port_name ;
```

Generic example

```
TMSPort port_name {
   Attribute att_name = att_value;
}
```

Examples

```
Module TAP_FSM {
  TCKPort tck;
  TRSTPort trst;
```

```
  TMSPort tms;
  ToIRSelectPort irSel;
  ToCaptureEnPort ce;
  ToShiftEnPort se;
  ToUpdateEnPort ue;
}
```

Rules

a) There shall be exactly one TMSPort for a module that has a ToIRSelectPort.

b) When the port exists, the **`TMSPort`** declaration shall not be omitted (even for non-handoff modules) if the ScanInterface with which it is associated is to be used during PDL retargeting.

c) When the concat_tms_signal includes a data_signal, any bits of type data_signal shall be directly driven by a DataMux element whose select line is a data_signal and whose two data inputs are another data_signal and a tms_signal.

d) If a DataMux drives the TMSPort of an instance, the data_signal driving the select input of the DataMux shall be justified to select the tms_signal (not the other data signal) whenever the downstream eTAPCs are performing their normal operations.

Explanations

Rule c) explains how to model gating of the TMSPort of an instance: drive the TMSPort of the instance from a 2-input DataMux, where one input is a local TMS value and the other input is the TMSPort of the parent module. The select input of the DataMux is a data_signal. In the example shown in Figure 46, that data_signal is labeled "TMS_force_en." The other data_signal that provides the actual value of the local TMS is labeled "TMS_force_val."

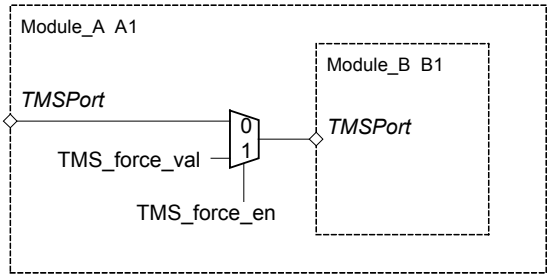


Figure 46—TMSPort forcing

When this configuration is used, Rule d) requires that the data_signal driving the select input of the DataMux be justified to select the tms_signal (e.g., the parent module's TMSPort or a sibling module's ToTMSPort) to enable downstream eTAPCs to perform their normal operations without being affected by the local TMS value, and to allow the global TMS to affect them.

6.4.6.16 ToTMSPort statement

The ToTMSPort function is used to pass the TMSPort to another module that either is an embedded TAP controller or contains an embedded TAP controller. The ToTMSPort may be sourced directly from a TMSPort in the same module, or it may be driven by a data_signal (which is the output of a DataMux).

The declaration of the ToTMSPort, when it exists, is mandatory. This is required even for an *internal* (non-*handoff*) module when the ScanInterface it is associated with is to be used during PDL retargeting.

Grammar

```
toTmsPort_def : 'ToTMSPort' toTmsPort_name (';' |
                ( '{' toTmsPort_item* '}' ) ) ;
toTmsPort_name : port_name ;
toTmsPort_item : attribute_def |
                 toTmsPort_source ;
toTmsPort_source : 'Source' concat_tms_signal ';' ;
```

Generic example

```
ToTMSPort port_name {
   Source source;
   Attribute att_name = att_value;
}
```

Examples

```
Module bottom_die {
  ToTMSPort tms_to_upper_die { Source escalator_active; }
  …
}
```

Rules

a) A module that includes a *toTmsPort_def* shall also include a *tmsPort_def*.

b) The *toTmsPort_source* shall be specified when the module includes more than one *tmsPort_def*.

c) When the *toTmsPort_source* is not specified, the *toTmsPort_source* shall be assumed to refer to the single *tmsPort_def* of the module.

d) When the *concat_tms_signal* in the *toTmsPort_source* statement includes a data_signal, any bits of type data_signal shall be directly driven by a DataMux element whose select line is a data_signal and whose two data inputs are another data_signal and a tms_signal.

e) If a DataMux drives the ToTMSPort of a module, the data_signal driving the select input of the DataMux shall be justified to select the tms_signal (not the other data signal) whenever the downstream eTAPCs are performing their normal operations.

f) When the port exists, the ToTMSPort declaration shall not be omitted (even for non-handoff modules) if the ScanInterface with which it is associated is to be used during PDL retargeting.

Explanations

Rule d) explains how to model the gating of a TMS signal within a module: drive the ToTMSPort of the module from a 2-input DataMux, where one input is a local TMS value and the other input is the TMSPort of the module. The select input of the DataMux a data_signal. In the example shown in Figure 47, that data_signal is labeled “TMS_force_en.” The other data_signal that provides the actual value of the local TMS is labeled “TMS_force_val.”

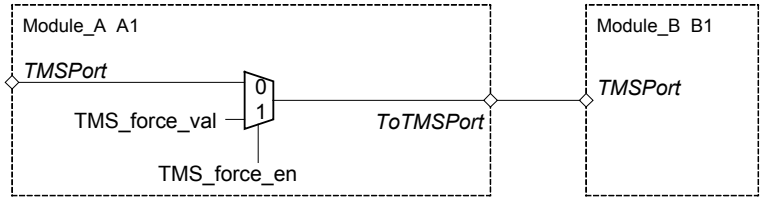


Figure 47—ToTMSPort forcing

When this configuration is used, Rule e) requires that the data_signal driving the select input of the DataMux be justified to select the module's TMSPort to enable downstream eTAPCs to perform their normal operations without being affected by the local TMS value, and to allow the global TMS to affect them.

6.4.6.17 TCKPort statement

The **TCKPort** is used to deliver the test clock to a module.

The declaration of the TCKPort is optional for an *internal* (non-handoff) module even if the module includes ScanRegister elements.

Grammar

```
tckPort_def : 'TCKPort' tckPort_name (';' |
              ( '{' attribute_def* '}' ) ) ;
tckPort_name : port_name ;
```

Generic example

```
TCKPort port_name {
   Attribute att_name = att_value;
}
```

Examples

```
Module M1 {
  TCKPort TCK;
  …
}
```

Rules

a) If a module has more than *one tckPort_def* element, then all shall be assumed to be equivalent with the same polarity.

Explanations

The delivery of the control signals (CaptureEn, ShiftEn, and UpdateEn) assume the concept of one global free running test clock. This is a behavioral requirement only. The actual clocking may be implemented differently. For a handoff module associated with a physical design with more than one power domain, it is possible that more than one TCKPort actually enters the module such that different parts of the design can be powered up and down independently. The assumption is that when a portion of the network is used, it is actually powered up and that its test clock behaves as the global free running test clock.

6.4.6.18 ToTCKPort statement

The ToTCKPort function is used to pass the TCKPort to another module. If the module contains one or more ToTCKPorts, each is equivalent to any of the TCKPorts and with the same polarity.

The declaration of the ToTCKPort is optional for *internal* (non-handoff) module even if it sources the TCK clock to another module that includes ScanRegister elements.

Grammar

```
toTckPort_def : 'ToTCKPort' toTckPort_name (';' |
                ( '{' attribute_def* '}' ) ) ;
toTckPort_name : port_name ;
```

Generic example

```
  ToTCKPort port_name {
     Attribute att_name = att_value;
  }
```

Examples

```
 Module bottom_die {
   ToTCKPort tck_to_upper_die;
   …
 }
```

Rules

a) A module that includes a *toTckPort_def* shall also include a *tckPort_def*.

b) Each *toTckPort_def* shall be equivalent to any *tckPort_def* in the module.

6.4.6.19 ClockPort statement

The **ClockPort** is used when a functional clock is delivered to a module. If a clock is delivered to the module differentially, there will be two ClockPort functions, and they will be linked by the **DifferentialInvOf** property on the negative clock.

Grammar

```
clockPort_def : 'ClockPort' clockPort_name
                (';' | ( '{' clockPort_item* '}' ));
clockPort_name : port_name ;
clockPort_item : attribute_def |
                 clockPort_diffPort ;
clockPort_diffPort : 'DifferentialInvOf' concat_clock_signal ';' ;
```

Generic example

```
  ClockPort port_name {
    Attribute att_name = att_value;
    DifferentialInvOf other_port_name;
  }
```

Examples

```
ClockPort PCK;
ClockPort NCK { DifferentialInvOf PCK; }
```

Rules

a) When the port exists, the **ClockPort** declaration shall not be omitted (even for a non-handoff module) if it is to be referenced by the iClock PDL command (see 7.9.15).

b) If a clock is delivered to the module differentially, there shall be two *clockPort_def* functions, and they shall be linked by the *clockPort_diffPort* property on the negative clock.

6.4.6.20 ToClockPort statement

The **ToClockPort** is used when a module modifies a functional clock and sends it to other modules. The modification is specified in terms of multiplying and/or dividing the frequency of the source clock to produce the output clock. Conversion from single-ended to differential or vice versa is also permitted anywhere in the hierarchy. When the Period element is used, the ToClockPort models an embedded oscillator.

The declaration of the ToClockPort, when it exists, is mandatory. This is required even for an internal (non-handoff) module when it is the source of a ClockPort referenced by an iClock PDL command.

Grammar

```
toClockPort_def : 'ToClockPort' toClockPort_name (';' |
                  ( '{' toClockPort_item+ '}' ) ) ;
toClockPort_name : port_name ;
toClockPort_item : attribute_def |
                   toClockPort_source |
                   freqMultiplier_def |
                   freqDivider_def |
                   differentialInvOf_def |
                   period_def ;
toClockPort_source : 'Source' concat_clock_signal ';' ;
freqMultiplier_def : 'FreqMultiplier' pos_int ';' ;
freqDivider_def : 'FreqDivider' pos_int ';' ;
differentialInvOf_def : 'DifferentialInvOf' concat_clock_signal ';' ;
period_def : 'Period' pos_int ('s' | 'ms' | 'us' | 'ns' | 'ps' | 'fs' | 'as')?
';' ;
```

Generic example

```
ToClockPort port_name {
  Attribute att_name = att_value;
  Source source_name;
  FreqMultiplier mult_integer;
  FreqDivider div_integer;
  DifferentialInvOf other_port_name;
  Period per_integer units;
}
```

Examples

```
ToClockPort FastCLK {
  // produce a 1.5X frequency clock
  Source SCK;
  FreqMultiplier 3;
  FreqDivider 2;
}

ToClockPort SlowCLK {
  // produce a half-speed clock
  Source SCK;
  FreqDivider 2;
}
```

```
ToClockPort CLK {
  Period 3400ps;
}
```

Rules

a) There shall be at most one of each of *differentialInvOf_def*, *toClockPort_Source*, *freqMultiplier_def*, *freqDivider_def* or *period_def* elements within a given *toClockPort_def*.

b) The width of *toClockPort_Source* shall match the width of *toClockPort_name*.

c) The width of *differentialInvOf_def* shall match the width of *toClockPort_name*.

d) The use of *period_def* is mutually exclusive with the usage of the *differentialInvOf_def*, *toClockPort_Source*, *freqMultiplier_def* and *freqDivider_def* element within a *toClockPort_def*.

e) The use of *differentialInvOf_def* is mutually exclusive with the usage of the *period_def*, *toClockPort_Source*, *freqMultiplier_def* and *freqDivider_def* element within a *toClockPort_def*.

f) The units of the specified *period_def* shall be **'ns'** when omitted.

g) If either *freqMultiplier_def* or *freqDivider_def* is present within a *toClockPort_def*, then a *toClockPort_Source* element shall also be present.

h) The presence of *freqMultiplier_def* shall indicate that the frequency of the ToClockPort_Source is multiplied by the associated value.

i) The presence of *freqDivider_def* shall indicate that the frequency of the ToClockPort_Source is divided by the associated value.

j) The presence of the *period_def* element is used to indicate that an embedded oscillator drives the *ToClockPort_def*.

Explanations

Rule d) indicates that if a ToClockPort statement contains a period definition that it cannot contain anything else besides an attribute property. Likewise, rule e) indicates that if a ToClockPort statement identifies a differential pair element that it cannot contain anything else besides an attribute property.

6.4.6.21 TRSTPort statement

The **TRSTPort** is used when the TAP Reset signal is delivered to a module that either is or contains an embedded TAP controller. The TRSTPort is distinct from the ResetPort. The declaration of the TRSTPort is optional for *internal* (non-handoff) module, even if the module includes a TMSPort. The presence of a TMSPort indicates the presence of an embedded TAP controller, but for an internal module the presence of the TRSTPort is inferable and thus may be omitted.

Grammar

```
trstPort_def : 'TRSTPort' trstPort_name (';' |
               ( '{' attribute_def* '}' ) ) ;
trstPort_name : port_name ;
```

Generic example

```
TRSTPort port_name {
  Attribute att_name = att_value;
}
```

Examples

```
TRSTPort TRST_L;
```

Rules

a) The active polarity of the *trstPort_name* shall be low.

b) When the concat_trst_signal includes a data_signal, any bits of type data_signal shall be directly driven by a DataMux element whose select line is a data_signal and whose two data inputs are another data_signal and a trst_signal.

c) If a DataMux drives the TRSTPort of an instance, the data_signal driving the select input of the DataMux shall be justified to select the trst_signal (not the other data signal) whenever the downstream eTAPCs are performing their normal (non-reset) operations.

d) If a DataMux drives the TRSTPort of an instance, the data_signal driving the select input of the DataMux shall select the trst_signal (not the data_signal) upon assertion of the trst_signal.

Explanations

Rule b) explains how to model a local reset of an instance containing an embedded TAP controller: drive the TRSTPort of the instance from a 2-input DataMux, where one input is a local reset value and the other input is the trst_signal (e.g., the TRSTPort of the parent module). The select input of the DataMux a data_signal. In the example shown in Figure 48, that data_signal is labeled "TRST_force_en." The other data_signal that provides the actual value of the local reset is labeled "TRST_force_val."

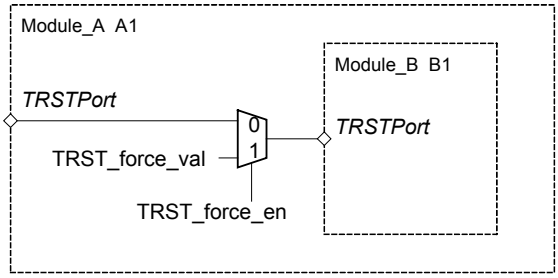


Figure 48—TRSTPort forcing

When this configuration is used, Rule c) requires that the data_signal driving the select input of the DataMux be justified to select the trst_signal (e.g. the parent module's TRSTPort) to enable downstream eTAPCs to perform their normal (non-reset) operations without being affected by the local reset value, and to allow a global reset to affect them.

6.4.6.22 ToTRSTPort statement

The ToTRSTPort is used when the TAP Reset signal is sent to another module. If the module contains one or many ToTRSTPort, they are all assumed to be equivalent to the TRSTPort when present. When the module contains no TRSTPort, the ToTRSTPort is assumed to be driven by a power-up detector that cannot be reasserted low for as long as the circuit remains powered up. The ToTRSTPort may be driven by a data signal (which is the output of a DataMux) in order to qualify the TRST signal to the downstream TAP client. This is particularly useful to hold a downstream embedded TAP controller in the iTLR state.

The declaration of the ToTRSTPort is optional for internal (non-handoff) module even if the module includes a ToTMSPort.

Grammar

```
toTrstPort_def : 'ToTRSTPort' toTrstPort_name (';' |
                 ( '{' toTrstPort_item+ '}' ) ) ;
toTrstPort_name : port_name ;
toTrstPort_item : attribute_def |
                  toTrstPort_source ;
toTrstPort_source : 'Source' concat_trst_signal ';' ;
```

Generic example

```
ToTRSTPort port_name {
  Source source;
  Attribute att_name = att_value;
}
```

Examples

```
ToTRSTPort TRST_L;
```

Rules

a) A module that includes a toTrstPort_def but no trstPort_def shall include an embedded power-up detector or equivalent circuit that temporarily activates the ToTRSTPort after the device has been powered up and deactivates the ToTRSTPort by the time the device is used.

b) There shall be at most one *toTrstPort_source* statement for each *toTrstPort_def*.

c) When specified, the width of the *toTrstPort_source* shall match the width of *toTrstPort_name*.

d) When the *toTrstPort_source* statement is not specified, the *toTrstPort_def* is associated with the single *trst_signal* when present otherwise it is associated with the global reset function.

e) The *toTrstPort_source* statement shall be specified if the *module_def* includes more than one *trst_signal*.

f) The ToTRSTPort shall always be active when its associated *trst_signal* is active.

g) The active polarity of the *toTrstPort_name* shall be low.

h) When the concat_trst_signal in the toTrstPort_source statement includes a data_signal, any bits of type data_signal shall be directly driven by a DataMux element whose select line is a data_signal and whose two data inputs are another data_signal and a trst_signal.

i) If a DataMux drives the ToTRSTPort of a module, the data_signal driving the select input of the DataMux shall be justified to select the trst_signal (not the other data signal) whenever the downstream eTAPCs are performing their normal (non-reset) operations.

j) If a DataMux drives the ToTRSTPort of an instance, the data_signal driving the select input of the DataMux shall select the trst_signal (not the data_signal) upon assertion of the trst_signal.

Explanations

Rule h) explains how to model a local reset for embedded TAP controllers generated within a module: drive the ToTRSTPort of the module from a 2-input DataMux, where one input is a local reset value and the other input is the trst_signal (e.g., the TRSTPort of the module). The select input of the DataMux a data_signal. In the example shown in Figure 49, that data_signal is labeled “TRST_force_en.” The other data_signal that provides the actual value of the local reset is labeled “TRST_force_val.”

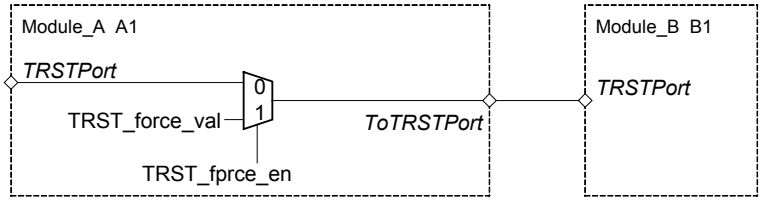


Figure 49—ToTRSTPort forcing

When this configuration is used, Rule i) requires that the data_signal driving the select input of the DataMux be justified to select the trst_signal (e.g., the module's TRSTPort) to enable downstream eTAPCs to perform their normal (non-reset) operations without being affected by the local reset value, and to allow a global reset to affect them.

6.4.6.23 ToIRSelectPort statement

The ToIRSelectPort is used when a module contains an instruction register (IR) of a TAP controller. The signal is used to drive the select of a 2-to-1 multiplexer that selects between the instruction scan register and other scan registers. The ToIRSelectPort is always associated with a TMSPort and optionally to a TRSTPort.

The declaration of the ToIRSelectPort, when it exists, is mandatory. This is required even for an *internal* (non-*handoff*) module when the scan chain that goes through the ScanMux it controls is to be used during PDL retargeting.

Grammar

```
toIRSelectPort_def : 'ToIRSelectPort' toIRSelectPort_name (';' |
                     ( '{' attribute_def* '}' )) ;
toIRSelectPort_name : port_name ;
```

Generic example

```
ToIRSelectPort port_name {
  Attribute att_name = att_value;
}
```

Examples

```
ToIRSelectPort irEn;
```

Rules

a) A **`module_def`** having a **`toIRSelectPort_def`** port shall include one and only one **`tmsPort_def`**.

b) A **`module_def`** with a **`toIRSelectPort_def`** shall have at most one **`trstPort_def`**.

c) A ScanRegister that is part of the active scan chain when any ToIRSelectPort associated with the active ScanInterface is high shall not be on the active scan chain when any ToIRSelectPort associated with that ScanInterface is low.

d) The ToIRSelectPort shall only be used as the select of a two-input ScanMux.

Explanations

Rule c) requires that TAP controllers be in series only with other TAP controllers. Connecting a TDR in series with a TAP would fail this rule as the TDR would remain in the active scan chain independent of the value of ToIRSelectPort.

The association of the ToIRSelectPort with the active ScanInterface mentioned in semantic rule c) is through the TMSPort associated with the ScanInterface.

6.4.6.24 AddressPort statement

The **`AddressPort`** is used when a module contains addressable instruments and/or data registers. This port serves as the address. An AddressPort on an instance or DataRegister with an AddressValue (see 6.4.7) is assumed to be connected to the logicalAddress_signal present in the parent module of that instance.

The declaration of the AddressPort, when it exists, is mandatory. This is required even for an internal (non-handoff) module when the module contains addressable instance or data registers.

Grammar

```
addressPort_def : 'AddressPort' addressPort_name (';' |
                  ( '{' attribute_def*'}' ) ) ;
addressPort_name : port_name ;
```

Generic example

```
AddressPort port_name {
  Attribute att_name = att_value;
}
```

Examples

```
AddressPort my_addr[7:0];
```

Rules

a) The *logicalAddress_signal* used to compare a specified AddressValue shall be created by concatenating all *addressPort_name* items from top to bottom as listed in the module. The left most bit (n–1) of the *logicalAddress_signal* is the left most bit of the first *addressPort_name,* and the right most bit (0) of the *logicalAddress_signal* is the right most bit of the last *addressPort_name*.

b) The source of the AddressPort shall be an AddressPort of the parent module or bits of a ScanRegister with no intervening logic in either case.

6.4.6.25 WriteEnPort statement

The **`WriteEnPort`** is used when a module contains writable addressable instruments and/or registers. This port serves as the write enable.

The declaration of the WriteEnPort, when it exists, is mandatory. This is required even for an internal (non-handoff) module when the module contains writable addressable instance or data registers.

Grammar

```
writeEnPort_def  : 'WriteEnPort' writeEnPort_name (';' |
                   ( '{' attribute_def* '}' ) ) ;
writeEnPort_name : port_name ;
```

Generic example

```
WriteEnPort port_name {
  Attribute att_name = att_value;
}
```

Examples

```
WriteEnPort WE;
```

Rules

a) There shall be zero or one *writeEnPort_def* per *module_def*.

b) There shall be exactly one *writeEnPort_def* for a *module_def* that contains at least one writable addressable data register or at least one writable addressable instance.

c) The source of the WriteEnPort is a WriteEnPort of the parent module or a bit of a ScanRegister with no intervening logic in either case.

6.4.6.26 ReadEnPort statement

The **`ReadEnPort`** is used when a module contains readable addressable instruments and/or registers. This port serves as the read enable. A read operation requires two steps: the first to activate the ReadEnPort and supply an address value on the *logicalAddress_signal*, and the second to capture the actual data value being read. During that second step, the ReadEnPort is automatically deactivated.

The declaration of the ReadEnPort, when it exists, is mandatory. This is required even for an internal (non-handoff) module when the module contains readable addressable instances or data registers.

Grammar

```
readEnPort_def : 'ReadEnPort' readEnPort_name (';' |
                 ( '{' attribute_def* '}' ) ) ;
readEnPort_name : port_name ;
```

Generic example

```
ReadEnPort port_name {
  Attribute att_name = att_value;
}
```

Examples

```
ReadEnPort RE;
```

Rules

a) There shall be zero or one **`readEnPort_def`** per module.

b) There shall be exactly one *readEnPort_def* for a *module_def* that contains at least one readable addressable data register or at least one readable addressable instance.

c) The source of the ReadEnPort shall be assumed to have an iApplyEndState of 0.

d) The source of the ReadEnPort shall be a ReadEnPort of the parent module or a bit of a ScanRegister with no intervening logic in either case.

Explanations

Rule c) makes a forward reference to the iApplyEndState property of the Alias statement (see 6.5.2), which is a mechanism to force the final value of a register or port to be a desired state. It is used here as the mechanism to instruct the retargeting software to enforce that the read enable port is disabled after the read operation is performed.

6.4.7 Instance statement

The **`Instance`** statement is used to create hierarchy by instantiating modules.

A module may be part of a hierarchical structure such that it may contain one or more instances (children) of other modules (for which it is the parent). Each of these child instances is identified by both name (i.e., the name of the instance) and type (i.e., the name of the referenced module), and the sources of its input ports are specified (though they may be omitted in some cases). In addition to the usual optional attributes, there are two other properties that may apply to an instance: an address value and an allowance for broadcast scan on a set of named scan interfaces. The address value is present when the instance is the object of a write or read address. The allowance for broadcast indicates that one or more scan interfaces for the module type being instantiated may participate in a broadcast scan action, making it mandatory to specify the source of its associated SelectPorts.

Grammar

```
instance_def : 'Instance' instance_name 'Of' (namespace_name? '::')?
               module_name (';' | ( '{' instance_item* '}' ) ) ;
instance_item : inputPort_connection |
                allowBroadcast_def |
                attribute_def |
                parameter_override |
                instance_addressValue ;
inputPort_connection : 'InputPort' inputPort_name '=' inputPort_source ';' ;
allowBroadcast_def : 'AllowBroadcastOnScanInterface' scanInterface_name ';' ;
inputPort_name : port_name ;
inputPort_source : concat_reset_signal |
                   concat_scan_signal |
                   concat_data_signal |
                   concat_clock_signal |
                   concat_tck_signal |
                   concat_shiftEn_signal |
                   concat_captureEn_signal |
                   concat_updateEn_signal |
                   concat_tms_signal |
                   concat_trst_signal ;
parameter_override : parameter_def;
instance_addressValue : 'AddressValue' number ';' ;
```

Generic example

```
Instance instance_name Of namespace::module_name {   // optional namespace::
  Attribute att_name = att_value;     // repeatable
  Parameter param_name = param_value; // repeatable
  InputPort port_name = source;       // repeatable
  AllowBroadcastOnScanInterface name1,name2, ...;
  AddressValue addr_value;            // for addressable instances
```

```
                                        //   and OneHotDataGroups
}
```

Examples

```
Instance Inst1 Of MyMod1;      // no input ports

Instance Inst2 Of NS1::MyMod1; // different MyMod1 inst from name space NS1

Instance Inst3 Of clock_gen {
   Parameter MSB = 4;            // a parameter override
   InputPort REFCLK = CK_IN;
   InputPort BYPASS = CK_BYP;
}

Module MyMod2 {
   ScanInPort SIa; ScanOutPort SOa; SelectPort Sela;
   ScanInterface A { Port SIa; }
   ScanInPort SIb; ScanOutPort SOb; SelectPort Selb;
   ScanInterface B { Port SIb; }
   ScanInPort SIc; ScanOutPort SOc; SelectPort Selc;
   ScanInterface C { Port SIc; }
…
}

Instance Inst4 Of MyMod2 {
   AllowBroadcastOnScanInterface A,B,C;
   InputPort SIa  = SI[0];
   InputPort SIb  = SI[1];
   InputPort SIc  = SI[2];
   InputPort Sela = Sel[0];
   InputPort Selb = Sel[1];
   InputPort Selc = Sel[2];
}
```

Rules

a) All *instance_def* elements shall have a corresponding module definition in the associated module *nameSpace_name*.

b) An *instance_def* defined within a *OneHotDataGroup_def* shall have an *instance_addressValue* statement.

c) An instance_def not defined within a *OneHotDataGroup_def* shall not have an *instance_addressValue* statement.

d) All *scanInPort_name* and *dataInPort_name* within the module being instantiated shall be referenced by an *inputPort_connection* statement.

e) All *selectPort_name* associated with a given *scanInterface_name* within the module being instantiated shall be referenced by an *inputPort_connection* statement if the associated *scanInterface_name* is listed in the *allowBroadcast_def*.

f) If the *inputPort_name* refers to a *scanInPort_name* within the module being instantiated, the *inputPort_source* shall be a *concat_scan_signal* of the same width.

g) If the *inputPort_name* refers to a *dataInPort_name*, *addressPort_name*, *writeEnPort_name*, *readEnPort_name* or *selectPort_name* in the module being instantiated, the *inputPort_source* shall be a *concat_data_signal* of the same width.

h) If the *inputPort_name* refers to a *clockPort_name* within the module being instantiated, the *inputPort_source* shall be a *concat_clock_signal* of the same width.

i) If the *inputPort_name* refers to a *tckPort_name* within the module being instantiated, the *inputPort_source* shall be a *concat_tck_signal* of the same width.

j) If the *inputPort_name* refers to a *shiftEnPort_name* within the module being instantiated, the *inputPort_source* shall be a *concat_shiftEn_signal* of the same width.

k) If the *inputPort_name* refers to a *captureEnPort_name* within the module being instantiated, the *inputPort_source* shall be a *concat_captureEn_signal* of the same width.

l) If the *inputPort_name* refers to an *updateEnPort_name* within the module being instantiated, the *inputPort_source* shall be a *concat_updateEn_signal* of the same width.

m) If the *inputPort_name* refers to a *resetPort_name* within the module being instantiated, the *inputPort_source* shall be a *concat_reset_signal* of the same width.

n) If the *inputPort_name* refers to a *tmsPort_name* within the module being instantiated, the *inputPort_source* shall be a *concat_tms_signal* of the same width.

o) If the *inputPort_name* refers to a *trstPort_name* within the module being instantiated, the *inputPort_source* shall be a *concat_trst_signal* of the same width.

p) Each *inputPort_name* shall be specified only once among all *inputPort_names* of the specified *inputPort_connection* statements. The order of the *inputPort_name* specified in the *inputPort_connection* statements is not relevant. For an indexed element with a *vector_id*, the same *SCALAR_ID* can be repeated as long as the indices do not overlap.

q) The *parameter_def* included in a *parameter_override* element shall match both the name and type of a *parameter_def* within the module being instantiated.

r) Control port (CaptureEnPort, ShiftEnPort, UpdateEnPort) activity on a scan register shall occur when the ScanOut of the scan register is on the active scan chain or when the ScanOut of another scan register in the broadcast group is on the active scan chain.

s) If not explicitly connected in the instantiation, an AddressPort on an instance with an AddressValue shall be assumed to be connected to the *logicalAddress_signal* of the parent module of that instance if it exists in the parent.

t) If not explicitly connected in the instantiation, a WriteEnPort on an instance with an AddressValue shall be assumed to be connected to the WriteEnPort of the parent module of that instance if it exists in the parent.

u) If not explicitly connected in the instantiation, a ReadEnPort on an instance with an AddressValue shall be assumed to be connected to the ReadEnPort of the parent module of that instance if it exists in the parent.

v) A DataInPort on an instance with an AddressValue shall be assumed to be connected to the logicalDataInPort of the parent module of that instance.

w) A module shall not contain any instances of itself anywhere in its child hierarchy.

x) If a **`scanInterface_name`** in an **`allowBroadcast_def`** contains multiple scan chains, then the broadcast action applies to all scan chains of that **`scanInterface_name`**.

Explanations

Both parameters and attributes are optional. Any parameter present in an instance is required to match the definition in the instantiated module so that it can be substituted wherever it is found in the module.

The convention for connectivity declaration is to identify the source of an input rather than the destination of an output (since an input source is unique while there could be many fan-out destinations). This source-based approach is also used for module outputs. Figure 50 and the associated ICL illustrate source identification.

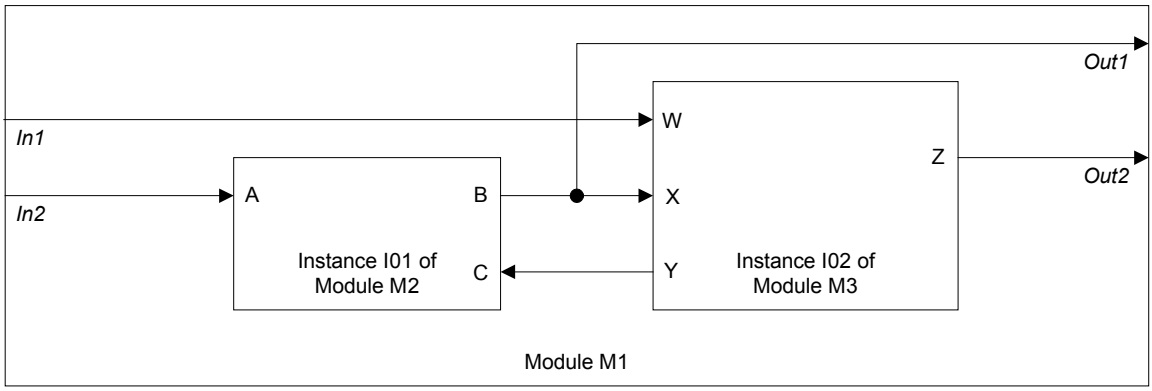


Figure 50—Instance and module port source identification

Example ICL for instance and module port source identification:

```
Module M1 {
  DataInPort In1;
  DataInPort In2;
  DataOutPort Out1 { Source I01.B; }
  DataOutPort Out2 { Source I02.Z; }
  Instance I01 Of M2 {
    InputPort A = In2;
    InputPort C = I02.Y;
  }
  Instance I02 Of M3  {
    InputPort X = I01.B;
    InputPort W = In1;
  }
}
```

6.4.8 ScanRegister statement

A **`ScanRegister`** statement is used to define a shift register operated by a ShiftEn signal when selected. The register consists of a shift cell with optional capture capability and an optional update cell. The register may contain multiple bits, in which case it will have a corresponding index range. When such a multi-bit register shifts, it does so from left to right, meaning that the bit associated with the index to the left of the ":" in the range moves toward the index to the right of the ":". The rightmost bit (i.e., the one associated with the index to the right of the ":" in the range) is defined as the ScanOut signal of the register.

The source of the scan input data for a ScanRegister is required to be identified to enable the connectivity of the scan chain to be established. Optionally, the capture source, default load value, and reset value may also be specified.

A ScanRegister is considered part of the active scan chain when it responds to activity of the ShiftEnPort functions and thus shifts serial data starting from a ScanInPort to a ScanOutPort of the active top level ScanInterface. When broadcasting is enabled, only the ScanRegisters through which the serial data flows are considered part of the active scan chain. The other ScanRegisters receiving a copy of the serial data are said to *shadow* a portion of the active scan chain.

Grammar

```
scanRegister_def : 'ScanRegister' scanRegister_name (';' |
                   '{' scanRegister_item* '}') ;
scanRegister_name : register_name ;
scanRegister_item : attribute_def |
                    scanRegister_scanInSource |
```

```
                    scanRegister_defaultLoadValue |
                    scanRegister_captureSource |
                    scanRegister_resetValue |
                    scanRegister_refEnum ;
scanRegister_scanInSource : 'ScanInSource' scan_signal ';' ;
scanRegister_defaultLoadValue : 'DefaultLoadValue' (concat_number |
                                enum_symbol) ';' ;
scanRegister_captureSource : 'CaptureSource' (concat_data_signal |
                             enum_symbol) ';';
scanRegister_resetValue : 'ResetValue' (concat_number |
                          enum_symbol)';' ;
scanRegister_refEnum : 'RefEnum' enum_name ';' ;
```

Generic example

```
  ScanRegister register_name {
    Attribute att_name = att_value;
    ScanInSource scan_in_source;
    CaptureSource capture_source;
    DefaultLoadValue default_load_value;
    ResetValue reset_value;
    RefEnum enum_name;
}
```

Examples

```
   ScanRegister TDR[42:0] { ScanInSource TDI; }
```

Rules

a) The *scanRegister_name* shall shift from left to right, meaning that the bit associated with the index to the left of the ":" in the index range shifts toward the index to the right of the ":".

b) The *scanRegister_scanInSource* shall be included in the *scanRegister_item* list.

c) The *scanRegister_scanInSource* referring to a multi-bit *scanRegister_name* shall always specify the right most bit, meaning the bit referenced by the index to the right of the ":" in the range.

d) Except for the *attribute_def*, there shall be at most one occurrence of each element type in a *scanRegister_item*.

e) If used, the width of *scanRegister_DefaultLoadValue*, *scanRegister_CaptureSource*, and *scanRegister_ResetValue* shall match the width of the *scanRegister_name*.

f) The width o*f enum_value re*ferenced by *scanRegister_refEnum* shall match the width of *scanRegister_name*.

g) The *enum_symbol* optionally used in the *scanRegister_defaultLoadValu*e, *scanRegister_captureSource*, or *scanRegister_resetValue* statement shall exist in the referenced *scanRegister_refEnum*.

h) A scanRegister_name that reacts to reset shall include the corresponding scanRegister_resetValue specification.

i) A *module_def* having at least one ScanRegister with a specified ResetValue shall have at most a single *reset_signal* so that the source of the reset signal for the ScanRegister is unambiguous.

j) For read data to be predictable at the scan register, the scanRegister_CaptureSource element shall be specified as the source of the captured data.

k) A ScanRegister that is part of the active scan chain or is shadowing parts of the active scan chain shall hold its data when ShiftEn is low between the CaptureEn and the UpdateEn cycle.

l) The data that enters on the ScanInSource shall exit unaltered (excepting possible inversion) *n* TCK shift cycles later, where *n* is the width of the scanRegister_name and a shift cycle is a TCK pulse with ShiftEn active.

m) The first time a ScanRegister is on the active scan chain, any bit that is not a care bit to satisfy the requirements of the current iApply shall be loaded with the *scanRegister_defaultLoadValue* if it is specified, else the *scanRegister_resetValue* if it is specified and not X, and if neither is specified, it shall be loaded with 0.

n) The first time a ScanRegister is on the active scan chain after it has been reset, any bit with a non-X reset value that is not a care bit to satisfy the requirements of the current iApply shall be loaded with the specified *scanRegister_resetValue.* Any bit with an X reset value shall be loaded as expressed in rule m).

o) When the ScanRegister is on the active scan chain, any bit that is not a care bit to satisfy the requirements of the current iApply (including the iApplyEndState, if present, and the values of any downstream DataRegister bits and DataInPort bits that must be justified) shall be loaded with the same value as the previous time it was on the active scan chain.

p) Any bit of a ScanRegister fanning out to the select of ScanMux or to a SelectPort or ToSelectPort of a ScanInterface anywhere in the hierarchy shall have a 0 or 1 value specified in the *scanRegister_ResetValue*.

q) The update stage of the ScanRegister shall assume the specified *scanRegister_resetValue* when its associated reset_signal is activated.

r) Any bit of a ScanRegister driving a *data_signal* anywhere in the hierarchy shall have an update stage (or a mechanism to cause equivalent behavior) that only changes when the ScanRegister is active and its associated *updateEn_signal* is high.

s) When both *scanRegister_defaultLoadValue* and *scanRegister_resetValue* exist, the values in the bits of *scanRegister_defaultLoadValue* shall match the non-X values in the corresponding bits of *scanRegister_resetValue*.

t) If *scanRegister_CaptureSource* references a name that is both present as a *reg_port_signal_id* as well as an *Enum* of the supplied *RefEnum,* then precedence shall go to the *Enum*.

Permissions

u) A scanRegister_scanInSource referring to a multi-bit scanRegister_name may specify only the SCALAR_ID portion of the name; the rightmost bit as defined in rule c) shall be the source.

Explanations

The only required element for a scan register is the **`scanRegister_ScanInSource`** that enables the scan chain connectivity to be traced.

If the scan register is resettable, then the **`scanRegister_ResetValue`** is required to be specified (as 0, 1, or X). The reset is typically applied only to the update cell of the scan register. The behavior of the update cell is well-defined over time and may be observed by feeding the output of the update cell back into the capture input of the shift cell. The reset could also be applied to the shift cell of the register. Although there is no ICL language support to describe a reset of the shift cell, the effect of resetting a shift cell may be observed if the pulse intended for the CaptureEnPort of a resettable shift cell is suppressed (the capability for which can be described in ICL). Scan registers that are not resettable are assumed to hold state during a reset operation (as noted in 6.4.6.13).

In the atypical case that a module has more than one *reset_signal*, the ScanRegisters associated with different resets must be pushed down into sub-modules, each having a single ResetPort sourced by the proper signal. This allows satisfying semantic rule h) while precisely describing that reset_signal resets each ScanRegister.

Rule j) indicates that a CaptureSource must be specified and its value must be known at the time of the Capture event in order to be able to predict the value being read from a ScanRegister. If a CaptureSource is not specified, observation of the ScanRegister will return whatever it did in fact capture, but that value will not be predictable based on the ICL representation.

Rules n), o), and p) state the policy for filling bits on a newly activated, or a newly reset and now active, or a simply active scan chain, respectively. If a bit on such a scan chain is not specified by the user (i.e., it is not a care bit in the current pattern being applied, see 7.3), and it is not required to be set in order to justify a value on a downstream DataRegister or a DataInPort for which there is a sensitized path (see 6.4.6.6), then the value of that bit will be specified as the default load value (if defined), the reset value (if defined), or 0, in that order of priority.

A scan register may or may not include an update latch. Rule s) states that the update stage (or equivalent behavior) is required to exist in the hardware (and is assumed to be in the ICL implicitly) when the output of a scan register sources a *data_signal* anywhere in the hierarchy. It sources a *data_signal* when it is connected in the following configurations (as illustrated in Figure 51 by the shaded latches):

— When a scan register drives the select line of a ScanMux, DataMux, or a ClockMux.

— When a scan register drives the data inputs of a DataMux.

— When a scan register drives the WriteDataSource, WriteEnSource, or ReadDataSource of a data register.

— When a scan register drives a DataInPort, SelectPort, ResetPort, TRSTPort, TMSPort, CaptureEnPort, or UpdateEnPort of another instance.

— When a scan register drives a DataOutPort, ToSelectPort, ToResetPort, ToTRSTPort, ToTMSPort, ToCaptureEnPort, or ToUpdateEnPort of the module in which it is located.

— When a scan register drives the CaptureSource of another ScanRegister.

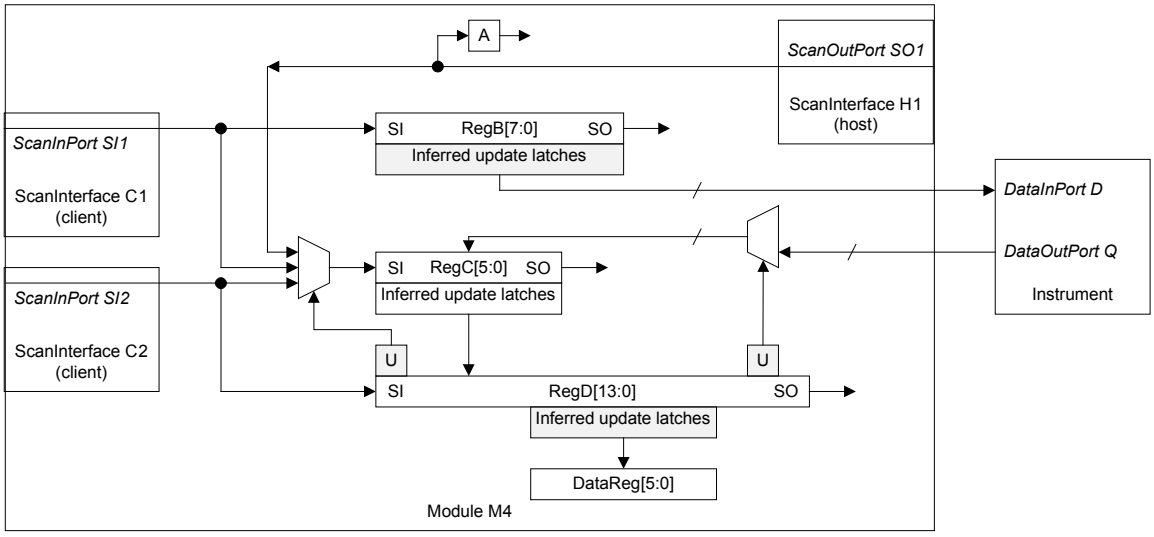


Figure 51—Inferred update latches

A behavior equivalent to a multi-bit update stage may be provided with the use of a single update stage on a "valid" bit in a handshake interface, such that the multiple bits may ripple during shift but are only sampled when the valid bit is high. The user assumes the responsibility for both sides of the interface in such cases, not just the driving side when update stages are used on the multiple bits.

6.4.9 DataRegister statement

A **DataRegister** statement is used to define a data register rather than a scan register. A data register consists of a set of one or more storage cells with a parallel input port and a parallel output port. A data register has a fixed length and is accessible by a register name and an associated index number or range. As with scan registers, the "rightmost bit" refers to the bit associated with the index to the right of the ":" in the range. Data registers are not on a shift path.

There are three (3) types of data registers: selectable, addressable, and callback.

The *selectable* data register requires that the source of the write data and the write enable signal be identified. The **WriteDataSource** indicates the data_signal that acts as the source for the register. The **WriteEnSource** identifies the signal source that must be equal to 1 to allow the register to be written. Retargeting software is expected to be able to navigate through selectable data registers. Note that a selectable register can be used to supply written data and observe read data.

The *addressable* data register is assumed to be accessible by presenting the data register with an address to trigger a select along with a read or write enable to execute the appropriate action. The addressable data register is a convenient abstraction that can be used to represent many forms of encoded access. Retargeting software is expected to be able to navigate through addressable data registers during the retargeting of the iWrite and iRead commands.

The *callback* register exists outside the logical connectivity described in ICL and instead relies on prewritten PDL procedures (see the description of an "iProc" in 7.9.6) in order to be accessed. The retargeting software simply executes the PDL procedure to access the callback register via some start- or end-point registers or ports that are present in the ICL model. There are separate read and write *callback* register types. Each requires the name of an associated PDL procedure to be provided. The iProc associated with the write *callback* register is called when the data register is written to (or needs to be written to supply a value on the data_signal it sources). The iProc associated with the read callback register is called when the data register is read. The *iProc_args* '<D>' (for "Data") is replaced by the actual data value to be written or observed when invoking the procedure. The <R> (for "Register") is replaced by the instance name of the register relative to the ***iProc_namespacemodule*** without the index. The actual procedures are written in PDL and located in the specified ICL module namespace.

Grammar

```
dataRegister_def : 'DataRegister' dataRegister_name (';' |
                   ( '{' dataRegister_item+ '}' ) ) ;
dataRegister_name : register_name ;
dataRegister_item : dataRegister_type |
                    dataRegister_common ;
dataRegister_type : dataRegister_selectable |
                    dataRegister_addressable |
                    dataRegister_readCallBack |
                    dataRegister_writeCallBack ;

// Common to all types:
dataRegister_common : dataRegister_resetValue |
                      dataRegister_defaultLoadValue |
                      dataRegister_refEnum |
                      attribute_def ;
dataRegister_resetValue : 'ResetValue' (concat_number |
                          enum_symbol) ';';
dataRegister_defaultLoadValue : 'DefaultLoadValue' (concat_number |
                                enum_symbol)';' ;
dataRegister_refEnum : 'RefEnum' enum_name ';' ;

//For Selectable Data Register:
```

```
dataRegister_selectable : dataRegister_writeEnSource |
                          dataRegister_writeDataSource;
dataRegister_writeEnSource : 'WriteEnSource' '~'? data_signal ';' ;
dataRegister_writeDataSource : 'WriteDataSource' concat_data_signal ';' ;

// For Addressable Data Register:
dataRegister_addressable : dataRegister_addressValue;
dataRegister_addressValue : 'AddressValue' number ';' ;

// For CallBack Data Register:
dataRegister_readCallBack : dataRegister_readCallBack_proc |
                            dataRegister_readDataSource ;
dataRegister_readCallBack_proc : 'ReadCallBack' iProc_namespace
                                   iProc_name iProc_args* ';';
dataRegister_readDataSource : 'ReadDataSource' concat_data_signal ';' ;
dataRegister_writeCallBack : 'WriteCallBack' iProc_namespace
                             iProc_name iProc_args* ';' ;
iProc_namespace : (namespace_name? '::')? ref_module_name
                  ( '::' sub_namespace )? ;
iProc_name : SCALAR_ID | parameter_ref ;
iProc_args : '<D>' |
             '<R>' |
             number |
             STRING |
             parameter_ref ;
             sub_namespace : SCALAR_ID |
             parameter_ref ;
             ref_module_name : SCALAR_ID |
             parameter_ref ;
```

Generic example

```
DataRegister register_name {
  //Common for all types
  Attribute att_name = att_value;
  ResetValue reset_value;
  DefaultLoadValue load_value;
  RefEnum enum_name;

  //Selectable
  WriteDataSource data_source;
  WriteEnSource enable_source;

  //Addressable
  AddressValue addr_value;

  //write callback
  WriteCallBack write_function_call;

  //read callback
  ReadCallBack read_function_call;
  ReadDataSource data_source;

}
```

Examples

//*Selectable* type:

//Note that all data registers (even when cascaded as shown below) evaluate immediately after the UpdateEn cycle.

```
DataRegister Reg1[31:0] {
  WriteDataSource databus[31:0];
}
DataRegister local_sel {
  WriteDataSource local_sel;
  WriteEnSource block_sel;
}

//Addressable type

Module M0 {
  //logicalDataIn_signal is byte1[7:0], byte0[7:0]
  DataInPort byteIn1[7:0];
  DataInPort byteIn0[7:0];
  // logicalAddress_signal is AddHigh[3:0], AddLow[7:0]
  AddressPort AddHigh[3:0];
  AddressPort AddLow[7:0];
  ReadEnPort re;
  WriteEnPort we;
  // data out source is explicit so order is not relevant
  DataOutPort byteOut0[7:0] { Source M1[7:0];}
  DataOutPort byteOut1[7:0] { Source M1[15:8];}

  OneHotDataGroup M1[15:0] {
    // computed_address of M2.R1 = 1000 + 001 = 1001
    Instance I1 Of M2 { AddressValue 'h1000; }
    Instance I2 Of M2 { AddressValue 'h2000; }
    DataRegister R1   { AddressValue 'h3001; }
    DataRegister R2   { AddressValue 'h3002; }
  }
}
Module M2 {
  DataInPort dataIn[11:0];
  AddressPort addIn [11:0];
  ReadEnPort re;
  WriteEnPort we;
  DataOutPort dataOut[11:0] { Source M1;}
  OneHotDataGroup M1[11:0] {
    DataRegister R1   { AddressValue 'h001; }
    DataRegister R2   { AddressValue 'h002; }
  }
}

// callback type
// Need to have defined an iProc named read_proc and write_proc following
// iProcForModule XXX where XXX match the value of the callback_namespace
// parameter in ICL
Module M3 {
  // omit port definitions for brevity
  Parameter callback_namespace = M3;
  DataRegister R1 {
    ReadDataSource u1.dout[7:0];
    ReadCallBack $callback_namespace read_proc <D> <R>;
  }
  DataRegister W1 {
```

```
    WriteCallBack $callback_namespace write_proc <D> <R>;
  }
}
```

Rules

a) Except for attribute_def, there shall be at most one occurrence of each element in a dataRegister_item.

b) If used, the width of dataRegister_defaultLoadValue, dataRegister_writeDataSource, dataRegister_resetValue, the enum_value referenced by the dataRegister_RefEnum shall match the width of the dataRegister_name.

c) The width of dataRegister_writeEnSource shall be 1 bit.

d) Elements from the dataRegister_selectable, dataRegister_addressable, dataRegister_readCallBack, and dataRegister_writeCallBack elements are mutually exclusive.

e) A *dataRegister_def* defined within a *OneHotDataGroup_def* statement shall only be of type addressable and dataRegister_def of type *addressable* shall only be defined inside a *OneHotDataGroup_def* statement.

f) The first time a *selectable* or *addressable* DataRegister is selected (*dataRegister_writeEnSource* is 1 or AddressValue match *logicalAddress_signal*), any bit that is not a care bit to satisfy the requirements of the current iApply shall be loaded with the *dataRegister_defaultLoadValue* if it is specified and not X, else the *dataRegister_resetValue* if it is specified and not X, and if neither is specified, it shall be loaded with 0.

g) The first time a *selectable* or *addressable* DataRegister is selected (*dataRegister_writeEnSource* is 1 or AddressValue match *logicalAddress_signal*) after it has been reset, any bit with a non-X reset value that is not a care bit to satisfy the requirements of the current iApply shall be loaded with the specified *dataRegister_resetValue.* Any bits whose reset value is not specified shall be loaded as expressed in rule f).

h) When the *selectable* or *addressable* DataRegister is selected (*dataRegister_writeEnSource* is 1 or AddressValue matches *logicalAddress_signal*), any bit that is not a care bit to satisfy the requirements of the current iApply (including the iApplyEndState, if present, and the values of any downstream DataRegister bits and DataInPort bits that must be justified) shall be loaded with the same value as the previous time it was enabled.

i) The DataRegister shall assume the specified *dataRegister_resetValue* when an associated reset_signal is activated.

j) When both dataRegister_defaultLoadValue and dataRegister_resetValue exist, the values in the dataRegister_defaultLoadValue shall be consistent with the non-X values of the dataRegister_resetValue.

k) For a *selectable* DataRegister, the WriteDataSource shall be sourced by either all DataInPorts of the top level, a single DataRegister or a single ScanRegister.

l) For an *addressable* DataRegister, the *logicalAddress_signal*, the ReadEnPort and the WriteEnPort shall be sourced by either all DataInPorts of the top level, or a single ScanRegister.

m) The source of the AddressPort shall be the logicalAddressPort of the parent module or bits of a ScanRegister with no intervening logic in either case.

n) The source of the WriteDataSource shall be the logicalDataInPort of the parent module or bits of a ScanRegister with no intervening logic in either case.

o) All *selectable* or *addressable* DataRegister shall be independently selectable from all other *selectable* or *addressable* DataRegisters.

p) The enum_symbol optionally used in the dataRegister_defaultLoadValue or dataRegister_resetValue statement shall exist in the referenced dataRegister_refEnum.

q) A *module_def* having at least one DataRegister with a specified ResetValue shall have at most a single *reset_signal* so that the source of the reset signal for the DataRegister is unambiguous.

r) An addressable data register shall be loaded with the right most bits of the *logicalDataIn_signal* vector of the module when the effective *logicalAddress_signal* matches its computed *address_value* and the WriteEnPort of the module is active. The *address_value* is computed by adding the specified AddressValue on the DataRegister with the AddressValue found on successive parent instances of the DataRegister and that also include an AddressValue, until a ScanRegister sourcing the address is encountered within a parent instance. The effective *logicalAddress_*signal is the *logicalAddress_signal* of the instance highest in that direct parental hierarchy of the DataRegister.

s) An addressable data register shall be the active source of a OneHotDataGroup when the *logicalAddress_signal* of the module matches its computed *address_value* and the ReadEnPort of the module is active. The *address_value* is computed by adding the specified AddressValue on the DataRegister with the AddressValue found on instances that are successive parent instances of the DataRegister and that also include an AddressValue, until a ScanRegister sourcing the address is encountered within a parent instance.

t) The elements of the logicalDataIn_signal in a module_def containing any addressable DataRegister or Instance shall only be used by the addressable DataRegisters or Instances.

u) The computed *address_value* of addressable DataRegister contributing into a OneHotDataGroup, either directly or through DataOutPorts of instances, shall be unique and its width shall be smaller than or equal to the width of the *logicalAddress_signal.*

v) The width of the register named *dataRegister_name* must be smaller than or equal to the width of the *logicalDataIn_signal* vector of the module.

w) The *iProc_namespace* shall refer to an ICL module with an optional sub-namespace in the logical hierarchy above the DataRegister.

x) The <D> argument in *iProc_args* of a ReadCallBack register shall be substituted by the data to be read on the ReadDataSource before calling the iProc.

y) The <D> argument in *iProc_args* of a WriteCallBack register shall be substituted by the data to be written into the register before calling the iProc.

z) The <R> argument in *iProc_args* of a ReadCallBack or WriteCallBack register shall be substituted by the instance name of the register relative to the ICL module defined in *iProc_namespace* without the range.

Explanations

Rule d) dictates that a DataRegister be of a specified type (i.e., it is either a selectable register, an addressable register, a read callback register, or a write callback register).

Rules f), g), and h) indicate how to specify the values of non-user-specified bits of newly activated, newly reset and activated, or simply activated data registers, respectively. If a bit in such a data register is not specified by the user (i.e., it is not a care bit in the current pattern being applied, see 7.3), and it is not required to be set in order to justify a value on a downstream DataRegister or a DataInPort for which there is a sensitized path (see 6.4.6.6), then the value of that bit will be specified as the default load value (if defined), the reset value (if defined), or 0, in that order of priority.

Rules l), m), r), s) and u) refer to the formal definition of *logicalAddress_signal* in the semantic rules of the AddressPort section. Rules n) and t) refer to the formal definition of *logicalDataIn_signal* in the semantic

rules of the DataInPort section. Rule w) requires that <R> exists below *iProc_namespace* when doing the substitution mentioned in Rule z).

Rules x), y), and z) apply to callback registers and explain how to perform the argument substitution for the placeholder arguments "<R>" (for Register) and "<D>" (for Data).

6.4.10 LogicSignal statement

A **`LogicSignal`** statement defines a control signal described as a Boolean expression that produces a single-bit result based on the values contained in specified network elements.

One of the key features of a network is that it may be designed to be reconfigurable, either in its scan connectivity or in its interface to the instruments. The control of the configuration is a function of the values contained in certain designated elements in the network itself. The actual logic functions that combine these values are described so that the retargeting software can identify the necessary assignments to be made to the network in order to configure it properly.

The **`LogicSignal`** statement uses a subset of the standard Boolean operators (unary and binary) to combine **`data_signals`** (perhaps through other logic) into a single-bit value that can be used for network control. The supported operators (in decreasing order of precedence) are as follows:

- Unary operators:
  - Boolean NOT (!)
  - Bitwise NOT (~)
- Binary operators:
  - Boolean logical AND (&&) and logical OR (||)
  - Bitwise AND (&), OR (|), XOR (^)
  - Logical equality (==) and inequality (!=)

All operators are left-to-right associative, and operators with equal precedence are evaluated left-to-right. Associativity and precedence may be expressly specified with parentheses ().

Valid operands are as follows:

- Registers (ScanRegisters and DataRegisters)
- Ports
- Numbers
- Enum references
- LogicSignals
- Parameter references
- ICL parameters

Grammar

```
logicSignal_def : 'LogicSignal' logicSignal_name
                  '{' logic_expr ';' '}' ;
logicSignal_name : reg_port_signal_id;
```

```
logic_expr : logic_expr_lvl1 ;
logic_expr_lvl1 : logic_expr_lvl2 ( ('&&' | '||') logic_expr_lvl1 )? ;
logic_expr_lvl2 : logic_expr_lvl3 ( ('&' | '|' | '^') logic_expr_lvl2 )? |
                  ( ('&' | '|' | '^') logic_expr_lvl2 );
logic_expr_lvl3 : logic_expr_lvl4 ( ('==' | '!=') logic_expr_num_arg )?;
logic_expr_lvl4 : logic_expr_arg (',' logic_expr_lvl4 )?;
logic_unary_expr : ('~'|'!') logic_expr_arg;
logic_expr_paren : '(' logic_expr ')';
logic_expr_arg : logic_expr_paren |
                 logic_unary_expr |
                 concat_data_signal ;
logic_expr_num_arg : concat_number |
                     enum_name |
                     '(' logic_expr_num_arg ')' ;
```

Generic example

```
LogicSignal signal_name {
  expression
}
```

Examples

```
// implements ctl = (e1 == 2'b10 || e2 == 2'b11)
LogicSignal ctl {
  e1 == 2'b10 || e2 == 2'b11;
}
// alternative implementation of the same function
LogicSignal ctl {
  e1[1] & ~e1[0] | e2[1] & e2[0];
}
```

Rules

a) The width of **logicSignal_name** shall be 1 bit.

b) In the **logic_expr_lvl3** statement, the width of **logic_expr_lvl4** shall match the width of **logic_expr_num_arg** when present.

c) In the **logic_expr_lvl3** statement, when **logic_expr_num_arg** is an **enum_name, then logic_expr_lvl4** shall be a single object with a **RefEnum** statement and **enum_name** shall exist inside the referenced **enum.**

d) Parameter references shall be substituted before evaluating the expression.

e) Comparison and Boolean logic operators shall return a 1-bit value: 1'b1 if the expression is true, 1'b0 if the expression is false.

f) Boolean logic operands and the condition operand of the ternary operator shall be 1-bit numbers.

g) Unknowns ('x') shall be only allowed in **logic_expr_num_arg**.

h) All operators shall honor sized values. The size of unsized numbers shall be able to be unambiguously inferred. Consequently, a concatenated expression may only contain at most one unsized number.

i) If the size of an unsized number in the expression needs to be extended, the remaining bits shall either be 0 filled if the most significant bit (MSB) of the unsized number is a 0 or 1; otherwise the remaining bits will filled with unknown (x).

j) Both selection operands of the ternary operator shall be evaluated and shall be of equal size.

k) The use of an enum name shall be possible only where the logic_expr_lvl4 resolves uniquely to the name of an object with a corresponding enumeration definition and with no range constraint.

Explanations

Rule c) deals with the case where a subrange is misapplied to an enum defined for a larger range. For example, consider a 4-bit data port defined and associated with an enumeration table as follows:

DataInPort A[3:0] { RefEnum A_names;} // where A_names includes "blue = 4'b0101"

A LogicSignal element such as "(A == blue)" is perfectly fine, while "(A[1:0] == blue)" is invalid because "blue" applies to the entire A object and not to a subrange.

6.4.11 ScanMux statement

A **ScanMux** statement is used to define a scan multiplexer. Scan multiplexers are the primitive mechanism used to reconfigure shift paths (either in length or in order).

Grammar

```
scanMux_def : 'ScanMux' scanMux_name 'SelectedBy' scanMux_select
              '{' scanMux_selection+ '}' ;
scanMux_name : reg_port_signal_id ;
scanMux_select : concat_data_signal ;
scanMux_selection : concat_number_list ':' concat_scan_signal ';' ;
```

Generic example

```
ScanMux mux_name SelectedBy mux_select_signals {
   mux_select_value : data_source_name; // Repeatable
}
```

Examples

```
// implements SIB_out = (SIBREG) ? aux[0] : base[0]
   ScanMux SIB_out SelectedBy SIBREG  {
     1'b0 : base[0];
     1'b1 : aux[0];
   }
// implements compare_out = (check_mismatch) ? different : same
   ScanMux compare_out SelectedBy check_mismatch[1:0]  {
     1'b0,1'b1 | 1'b1,1'b0 : different;
     1'b1,1'b1 | 1'b0,1'b0 : same;
   }
```

Rules

a) The width of all concat_numbers within a *scanMux_selection* vector shall match the width of the *scanMux_select* vector.

b) The width of a *concat_scan_signal* vector within a *scanMux_selection* shall match the width of the *scanMux_name* vector.

c) The *scanMux_selection* entries shall be evaluated in the order of their appearance.

Permissions

d) The scanMux_select vector may be driven by a ToIRSelectPort.

Explanations

The **ScanMux** statement maps one or more input values onto the output (which are of the same width) when a certain specified select condition (the one listed after the **'SelectedBy'** keyword) is met. There are keywords for the output (i.e., **scanMux_name**) and the select (**scanMux_select**), followed by the body of the statement inside the curly braces (**{}**). Each entry in the body lists the pairings of a select condition value with the associated input value. The select condition value in each of these pairings can be thought of as an opcode, and Rule a) requires that the width of this opcode matches the width of the select condition. Rule b) similarly requires that the width of the input value matches the width of the ScanMux output. Rule c) specifies that the order in which the opcodes appear is the priority order in which they will be evaluated. Bits of the opcode may be denoted as **'x'** to match multiple select conditions.

Both the select condition and the opcodes may use concatenation to assemble the full width of the field. Multiple opcodes can also be associated with a single input value by enumerating all matching opcodes on the same line. The select condition may be assembled from register bits (ScanRegister or DataRegister) or **DataIn**Port functions; the opcodes are likewise assembled and may also include constants. No default (fall-through) condition is included if no opcode matches the select condition; in such a case, the output is undefined.

6.4.12 DataMux statement

The **DataMux** statement is used to describe a data multiplexer, which differs from a scan multiplexer in that it is not used as part of a shift path. Instead, it is used to steer data signals. The **DataMux** definition is analogous to that for the **ScanMux** and differs only in context.

Grammar

```
dataMux_def : 'DataMux' dataMux_name 'SelectedBy' dataMux_select
              '{' dataMux_selection+ '}' ;
dataMux_name : reg_port_signal_id ;
dataMux_select : concat_data_signal ;
dataMux_selection : concat_number_list ':' concat_data_signal ';' ;
```

Generic example

```
DataMux mux_name SelectedBy mux_select_signals {
   mux_select_value : data_source_name; // Repeatable
}
```

Examples

```
// implements C_S_REG = (select_bit) ? command[7:0] : status[7:0];
// note: C_S_REG is 8 bits wide: it could be listed as C_S_REG[7:0]
DataMux C_S_REG SelectedBy select_bit  {
  1'b0 : status[7:0];
  1'b1 : command[7:0];
}
```

Rules

a) The width of all *concat_numbers* within a *dataMux_selection* vector shall match the width of the *dataMux_select* vector.

b) The width of *concat_data_signal* vector within *dataMux_selection* shall match the width of the *dataMux_name* vector.

c) The *dataMux_selection* entries shall be evaluated in the order of their appearance.

Explanations

The DataMux statement maps one or more input values onto the output (which are of the same width) when a certain specified select condition (the one listed after the "SelectedBy" keyword) is met. There are keywords for the output (i.e., dataMux_name) and the select (dataMux_select), followed by the body of the statement inside the curly braces ({}). Each entry in the body lists the pairings of a select condition value with the associated input value. The select condition value in each of these pairings can be thought of as an opcode, and Rule a) requires that the width of this opcode matches the width of the select condition. Rule b) similarly requires that the width of the input value matches the width of the DataMux output. Rule c) specifies that the order in which the opcodes appear is the priority order in which they will be evaluated. Bits of the opcode may be denoted as `'x'` to match multiple select conditions.

Both the select condition and the opcodes may use concatenation to assemble the full width of the field. Multiple opcodes can also be associated with a single input value by enumerating all matching opcodes on the same line. The select condition may be assembled from register bits or DataInPort functions; the opcodes are likewise assembled and may also include constants. No default (fall-through) condition is included if no opcode matches the select condition; in such a case, the output is undefined.

6.4.13 ClockMux statement

The **`ClockMux`** statement identifies a multiplexer with one or more select inputs, and one or more clock inputs. A clock multiplexer, called a *clock mux*, is the third and final unique type of multiplexer. As the name implies, it is used to steer clock signals instead of scan or data signals. The **`ClockMux`** definition is exactly analogous to the definition for the **`ScanMux`** and **`DataMux`** and differs only in the context.

Grammar

```
clockMux_def : 'ClockMux' clockMux_name 'SelectedBy' clockMux_select
               '{' clockMux_selection+ '}' ;
clockMux_name : reg_port_signal_id ;
clockMux_select : concat_data_signal ;
clockMux_selection : concat_number_list ':' concat_clock_signal ';' ;
```

Generic example

```
ClockMux mux_name SelectedBy mux_select_signals {
   mux_select_value : data_source_name; // Repeatable
}
```

Examples

```
// implements ck=(md == 2'b01) ? sck1 : (md = 2'b10) ? sck2 : sck3;
ClockMux ck SelectedBy md[1:0] {
   2'b01 : sck1;
   2'b10 : sck2;
   2'bxx : sck3;
}
```

Rules

a) The width of each *concat_number* within the *clockMux_selection* vector shall match the width of the *clockMux_select* vector.

b) The width of *concat_clock_signal* vector within the *clockMux_selection* shall match the width of the *clockMux_name* vector.

c) The *clockMux_selection* entries shall be evaluated in the order of their appearance.

Explanations

See the explanation for the DataMux statement immediately preceding.

6.4.14 OneHotScanGroup statement

The **OneHotScanGroup** statement supports a scan multiplexer based on one-hot control signals. It is logically equivalent to a scan multiplexer. The reason for including it in the language is to cope with the problem of a distributed scan multiplexer (i.e., one in which the inputs to the multiplexer are resident in many different modules). Employing a traditional scan multiplexer in that situation would require routing all of the signals and related control signals upward in the hierarchy to a common module. Conversely, if all of the inputs are available in a module, this statement can still be used but it offers no advantages over the traditional multiplexer.

The **OneHotScanGroup** statement consists of a list of the names of the scan sources. The order in which the sources are listed is not significant. Each source is either a **ScanOutPort** function with a **scanOutPort_enable** element or another **OneHotScanGroup** somewhere in its fan-in.

When a scan source is selected, its enable signal is active while all the other enable signals for the other scan source contributors are held inactive. The presence of the '~' indicates an inversion between the source and **oneHotScanGroup_name.**

It is permissible to construct OneHotScanGroups hierarchically; i.e., a OneHotScanGroup may refer to ScanOutPorts sourced by other OneHotScanGroups. This also implies that a OneHotScanGroup may be in a “zero-hot” state (where none of its enable signals are active) so that it does not violate the “at-most-one-hot” condition at the next level.

Grammar

```
oneHotScanGroup_def : 'OneHotScanGroup' oneHotScanGroup_name
                      '{' oneHotScanGroup_item+ '}' ;
oneHotScanGroup_name : reg_port_signal_id ;
oneHotScanGroup_item : 'Port' concat_scan_signal ';' ;
```

Generic example

```
OneHotScanGroup group_name {
  Port source_name_0;
  Port source_name_1;
  …
  Port source_name_n;
}
```

Examples

```
//Illustrates the cascading of OneHotScanGroups
```

```
Module M1 {
  ScanOutPort SO { Source M1; }
  OneHotScanGroup M1 {
    Port child1.SO;
    Port child2.SO;
    Port child3.SO;
  }
  Instance child1 Of M2 {...}
  Instance child2 Of M2 {...}
  Instance child3 Of M2 {...}
}
Module M2 {
  ScanOutPort SO { Source M1; }
  OneHotScanGroup M1 {
    Port child1.SO;
    Port child2.SO;
    Port child3.SO;
  }
  Instance child1 Of M3 {...}
  Instance child2 Of M3 {...}
  Instance child3 Of M3 {...}
}

Module M3 {
  ScanOutPort SO {
    Source R1[0];
    Enable En1;
  }
}
```

Rules

a) The width of *concat_scanOut_port* shall match the width of *oneHotScanGroup_name*.

b) Each *scan_signal* used in *concat_scan_signal* shall have a ScanOutPort with an associated *scanOutPort_Enable* element or another OneHotScanGroup element somewhere in its controlling fan-in.

c) A *OneHotScanGroup_name* shall have one and only one of its *concat_scan_signal* activated when it is part of the active scan chain.

d) The value of a *oneHotScanGroup_name* shall be undefined if more than one *concat_scan_signal* is active.

Explanations

The order of the sources has no impact. Source ports all have an Enable somewhere in the hierarchy, all but one of which is deactivated to uniquely control the OneHotScanGroup, where “deactivation” means that the signal source of the Enable property is set to 0.

6.4.15 OneHotDataGroup statement

The **`OneHotDataGroup`** statement supports a data multiplexer based on one-hot control signals. It is logically equivalent to a data multiplexer. The reason for including it in the language is to cope with the problem of a distributed data multiplexer, one in which the inputs to the multiplexer are resident in many different modules. Employing a traditional data multiplexer in that situation would require routing all of the signals and related control signals upward in the hierarchy to a common module. Conversely, if all of the inputs are available in a module, this statement can still be used but it offers no advantages over the traditional multiplexer.

The **OneHotDataGroup** statement consists of a list of the names of the data sources. The order in which the sources are listed is not significant. Each source has a **dataOutPort_enable** element or a **dataOutPort_**addressValue element that determines which element is sourcing the **OneHotDataGroup** statement.

When a data source is selected, its enable signal is active while all the other enable signals for the other data source contributors are held inactive.

The **OneHotDataGroup** includes three possibilities to describe the multiplexer sources: instance, data register, or port.

Grammar

```
oneHotDataGroup_def : 'OneHotDataGroup' oneHotDataGroup_name
                      '{' oneHotDataGroup_item+ '}' ;
oneHotDataGroup_name : reg_port_signal_id ;
oneHotDataGroup_item : instance_def |
                       dataRegister_def |
                       oneHotDataGroup_portSource ;
oneHotDataGroup_portSource : 'Port' concat_data_signal ';' ;
```

Generic example

```
  OneHotDataGroup group_name {
    Port source_name_0; //ex: ABC[1:0];
    Port source_name_1; //ex: A,B;
    …
    Port source_name_n; //ex u2.ABC[1:0]
    Instance instance_name Of module_type {
      AddressValue inst_addr_value;
    }
    DataRegister reg_name {
      AddressValue reg_addr_value;
    }
  }
```

Examples

```
Module M1 {
  DataInPort MSBS[3:0];
  DataInPort LSBS[3:0];
  OneHotDataGroup M1 { //Addressable elements
    Port A;
    Port M2_I1.A;
    Port M2_I1.B;
    Instance A Of Mod1  { AddressValue 8'h00; }
    Instance B Of Mod1  { AddressValue 8'h10; }
    DataRegister A[7:0] { AddressValue 8'h20; }
    DataRegister B[7:0] { AddressValue 8'h21; }
  }
  Instance M2_I1 Of M2 {...}
}
Module M2 {
  DataOutPort A {
    Source Reg1;
    Enable En1;
  }
}
```

Rules

a) The width of *concat_data_signal* shall be smaller than or equal to the width of *oneHotDataGroup_name*.

b) The width of *logicalDataOut_signal* within the *instance_def* shall be smaller than or equal to the width of oneHotDataGroup_name.

c) The width of the *dataRegister_name* within the *dataRegister_def* shall be smaller than or equal to the width of *oneHotDataGroup_name*.

d) The *oneHotDataGroup_portSource* shall reference other OneHotDataGroup_names or *dataOutPort_names* with an enabling condition. An enabling condition is defined as one of the following three choices:

   1) A dataOutPort_enable element is included in the dataOutPort_def.

   2) The dataOutPort_source element included in the dataOutPort_def refers to dataOutPort_names with an enabling condition as defined in this rule.

   3) An instance_AddressValue is included in the instance_def of the port.

e) A OneHotDataGroup shall take on the value of its uniquely enabled source. Its value shall remain unknown when more than one of its sources is enabled.

Explanations

The definition of the **`OneHotDataGroup`** is analogous to the **`OneHotScanGroup`** but differs in the usage context. Note that the order of the sources has no impact. Sources all have an enabling condition, all but one of which are deactivated to uniquely control the OneHotDataGroup, where "deactivation" means that the signal source of the Enable property is set to 0 or the AddressValue is not matched.

The formal definition of *logicalDataOut_signal* of *addressable* instance is defined in the DataOutPort subclause.

6.4.16 ScanInterface statement

The purpose of the ScanInterface statement is to define the list of ports that comprise a scan interface. A scan interface consists of a ScanInPort and/or a ScanOutPort with the related control signals depending on the type of the ScanInterface. There are four types of ScanInterfaces: scan client, scan host, TAP client, and TAP host. The control signals associated with a client ScanInterface may include SelectPort, CaptureEnPort, ShiftEnPort, and UpdateEnPort. The control signals associated with a host ScanInterface may include ToSelectPort, ToCaptureEnPort, ToShiftEnPort, and ToUpdateEnPort. The client-TAP ScanInterface include TMSPort and may include TRSTPort. The host-TAP ScanInterface include ToTMSPort and may include ToTRSTPort. If there is only a single interface in a module, the ScanInterface statement is not required. If there are multiple scan interfaces in a module, there will be as many ScanInterface statements as there are scan interfaces in the module.

The ScanInterface statement assigns a unique group name to each collection of scan-related port functions. This unique name is used to identify the set of signals associated with a given scan interface so that the retargeting tool knows which port functions to use when retargeting PDL patterns at the boundary of a module which contains more than one client or client-TAP scan interface. The ScanInterface name is also used by the ICL AccessLink statement and the PDL iScan statement.

A ScanInterface may also contain multiple scan chains, each of which is named and specified by its individual ports and default load value, but all of which react simultaneously to the set of associated control signals.

As described in the SelectPort subclause, a SelectPort is different from a DataInPort only when it is associated with a scan interface using the ScanInterface statement.

Grammar

```
scanInterface_def : 'ScanInterface' scanInterface_name '{' scanInterface_item+
                    '}' ;
scanInterface_name : SCALAR_ID;
scanInterface_item : attribute_def | scanInterfacePort_def | defaultLoad_def |
                     scanInterfaceChain_def ;
scanInterfacePort_def : 'Port' reg_port_signal_id ';';
scanInterfaceChain_def : 'Chain' scanInterfaceChain_name '{'
                         scanInterfaceChain_item+ '}' ;
scanInterfaceChain_name : SCALAR_ID;
scanInterfaceChain_item : attribute_def | scanInterfacePort_def |
                          defaultLoad_def ;
defaultLoad_def : 'DefaultLoadValue' concat_number ';' ;
```

Generic example

```
  ScanInterface interface_name {
    Attribute att_name = att_value;
    // if there is a single scan chain in the interface, use next two lines:
    Port port_name;               //Repeat for all ports in the interface
    DefaultLoadValue load_value;
    // else, if there are multiple scan chains in the interface, for each use:
    Chain chain_name {
      Attribute att_name = att_value;
      Port port_name;             // List SI and SO ports
      DefaultLoadValue load_value;
    }
  }
```

Examples

```
ScanInterface Client {
  Port SEL;
  Port SI;
  Port SO;
}
ScanInterface Host {
  Port ToSel;
  Port FromSO;
}
ScanInterface Par_client2 {
  Port Sel;
  Port SE;
  Chain c1 {
    Port SI1A;
    Port SO1A;
    DefaultLoadValue 8'b11011010;
  }
  Chain c2 {
    Port SI1B;
    Port SO1B;
    DefaultLoadValue 6'b00100101;
  }
}
ScanInterface TAP {
  Port tms;
  Port tdi;
  Port tdo;
}
```

Rules

a) A *handoff* module with more than one port of function ScanInPort or more than one port of function ScanOutPort shall define as many ScanInterface statements as there are scan interfaces in the module. ScanInterface statements shall remain optional for *internal* modules.

b) A *blackbox* module with at least one port of function ScanInPort and one port of function ScanOutPort shall define as many ScanInterface statements as there are client scan interfaces in the module. Each ScanInterface statement shall also include a valid DefaultLoad value that reflects the length of the scan chain behind each ScanInterface after a global reset.

c) All ScanInterfaces found on *blackbox* modules shall be non-inverting.

d) A ScanInterface shall be one and only one of the following types: client, host, client-TAP, or host-TAP.

e) A client ScanInterface shall have the following:

   1) One or many pairs of ports with function ScanInPort and ScanOutPort
   2) One port with function ShiftEnPort and/or one or many ports with function SelectPort
   3) Zero or one port with function CaptureEnPort
   4) Zero or one port with function UpdateEnPort
   5) Zero or one port with function ResetPort
   6) Zero or one port with function TCKPort

f) A host ScanInterface shall have the following:

   1) One port with function ScanInPort and/or one port with function ScanOutPort
   2) One port with function ToShiftEnPort and/or one or many ports with function ToSelectPort
   3) Zero or one port with function ToCaptureEnPort
   4) Zero or one port with function ToUpdateEnPort
   5) Zero or one port with function ToResetPort
   6) Zero or one port with function ToTCKPort

g) A client-TAP ScanInterface shall have the following:

   1) One or many pairs of ports with function ScanInPort and ScanOutPort
   2) One port with function TMSPort
   3) Zero or one port with function TRSTPort
   4) Zero or one port with function TCKPort

h) A host-TAP ScanInterface shall have the following:

   1) One port with function ScanInPort and/or one port with function ScanOutPort
   2) One port with function ToTMSPort
   3) Zero or one port with function ToTRSTPort
   4) Zero or one port with function ToTCKPort

i) When there is a single port of function ShiftEnPort, CaptureEnPort, or UpdateEnPort, it shall be implicitly associated with every client scan interface that does not already have a port of that particular port function among its members.

j) When there is a single port of function ToShiftEnPort, ToCaptureEnPort, or ToUpdateEnPort, it shall be implicitly associated with every host scan interface that does not already have a port of that particular port function amongst its members.

k) If a scan client interface has more than one ScanInPort/ScanOutPort pair, then each pair shall be declared in a *scanInterfaceChain_def* element.

l) If a *scanInterfaceChain_def* element is present, there shall be neither *scanInterfacePort_def* (with function ScanInPort or ScanOutPort) nor *defaultLoad_def* statements outside the *scanInterfaceChain_def*.

m) Each scan interface of a module shall be uniquely selectable (with all other scan interfaces of that module disabled).

n) The *scanInterfaceChain_name* shall be unique within a *scanInterface_def*.

o) The *scanInterfacePort_def* inside a *scanInterfaceChain_def* shall only include ports with function **`ScanInPort`** and **`ScanOutPort`**.

p) The *scanInterfaceChain_def* is only allowed in scanInterface_def of type client or client-TAP.

q) A *scanInterfaceChain_def* element shall include one *scanInterfacePort_def* with function **`ScanInPort`** and one with function **`ScanOutPort`** and each port shall only be referenced by a single *scanInterfaceChain_def* element.

Explanations

Rule c) disallows inversions for ScanInterfaces on black-box modules, meaning that there is no net inversion from the ScanInPort to the ScanOutPort for each and every scan chain in the interface.

6.4.17 AccessLink statement

The **`AccessLink`** statement is the mechanism to indicate the interface between a controller and an instrument access network. The **`AccessLink`** statement is similar to an **`Instance`** statement since it indicates a named object to which other instances are connected, but it differs from an **`Instance`** statement in that it is uniquely used to indicate the interface location to be used by the software that will be exercising the network. The **`AccessLink`** statement supports either an IEEE 1149.1 TAP controller described in BSDL or a generic controller. IEEE Std 1687 specifies the details for only the former type of controller; the detailed description of the latter is left open to the user community or future revisions of or amendments to IEEE Std 1687. The purpose of including the generic controller type in the AccessLink statement is to identify the attachment point of a network (e.g., a ScanInterface) to a device interface. This mechanism may be used by the provider of a network to indicate to the integrator what signals need to be connected to an external controller, for example.

As shown in Figure 7, the “left-hand side” of a network is linked to a controller of some type, often to an IEEE 1149.1 TAP controller. Though the top-level TAP controller could be described in ICL, the *AccessLink* statement is preferred when interfacing to the TAP already described via the BSDL file for the device. The **`AccessLink`** statement is allowed only in the top-level ICL module for this purpose. If there are other TAP controllers embedded as part of the on-chip network, those are required to be described in ICL if they are to be used for downstream instrument access.

Grammar

```
accessLink_def : accessLink1149_def | accessLinkGeneric_def ;

accessLinkGeneric_def : 'AccessLink' accessLink_name 'Of' accessLink_genericID
                        accessLinkGeneric_block;
accessLink_genericID : SCALAR_ID;
accessLinkGeneric_block :'{' ( accessLinkGeneric_block | AccessLinkGeneric_text
                             | STRING )* '}' ;
AccessLinkGeneric_text : [^{}"\t\n\r ]+;


accessLink1149_def : 'AccessLink' accessLink_name 'Of'
                     ('STD_1149_1_2001' | 'STD_1149_1_2013')
                     '{' 'BSDLEntity' bsdlEntity_name ';'
                     bsdl_instr_ref+ '}' ;
accessLink_name : SCALAR_ID;
bsdlEntity_name : SCALAR_ID ;
bsdl_instr_ref : bsdl_instr_name '{' bsdl_instr_selected_item+ '}' ;
bsdl_instr_name : SCALAR_ID ;
bsdl_instr_selected_item : 'ScanInterface'
                           '{' (accessLink1149_ScanInterface_name ';')+ '}' |
                           ('ActiveSignals'
                           '{'(accessLink1149_ActiveSignal_name ';')+ '}' ) ;
accessLink1149_ActiveSignal_name : reg_port_signal_id ;
accessLink1149_ScanInterface_name : instance_name( '.' scanInterface_name)? ;
```

Generic example

```
AccessLink instance_name Of link_type {
  // content
}
```

Generic example for IEEE 1149.1-based controller

```
AccessLink instance_name Of STD_1149_1_2001 {
  BSDLEntity entity_name;
  instruction_name {
    ScanInterface { scan_interface_name1; scan_interface_name2; …}
    ActiveSignals { signal_name1; signal_name2; … }
  }
}
```

Example 1: IEEE 1149.1 access

```
AccessLink dot1 Of STD_1149_1_2001 {
   BSDLEntity chip2542;
   ijtag_en { // Name of instruction
      ScanInterface { InstPath.MyScanInterface; }  // defines the ScanInterface
                                                   // selected
   }
}
```

Example 2: Generic access

```
AccessLink mylink Of Generic { MyScanInterface; }
```

Rules

a) An *accessLink1149_def*, when present, shall only be in the top-level module.

b) An *instruction_name* shall be unique among all *instruction_name* entries and shall exist in the *INSTRUCTION_OPCODE* attribute of the referenced *BSDLEntity*.

c) There shall be only one *accessLink1149_def* per device-level IEEE 1149.1 interface, and at most one *BSDLEntity* element within each *accessLink1149_def*.

d) Each *accessLink_name* shall be unique among all *accessLink_name entries*.

e) If an *accessLink1149_ScanInterface_name* is specified, it shall match a *scanInterface_name* of the module type corresponding to *instance_def* at the top level, and that *scanInterface_name* shall have exactly one scan chain associated with it.

f) When the instruction register described in the BSDL file contains the opcode associated with the specified *instruction_name*, the listed *accessLink1149_ScanInterface_name* shall be concatenated top-to-bottom and left-to-right between the *TAP_SCAN_IN* and *TAP_SCAN_OUT* ports described in the BSDL. The ScanInPort of the first *accessLink1149_ScanInterface_name* shall be closest to TAP_SCAN_IN and the ScanOutPort of the last *accessLink1149_ScanInterface_name* shall be closest to TAP_SCAN_OUT.

g) When the instruction register described in the BSDL file is loaded with the specified *instruction_name*, the listed *accessLink1149_ActiveSignal_name* shall be active. The specified *accessLink1149_ActiveSignal_name* shall be inactive when register described in the BSDL file is loaded with any other value that does not contain that *accessLink1149_ActiveSignal_name*.

Explanations

Rule f) allows the implicit connection of multiple scan interfaces to each other in a serial daisy chain. When more than one scan interface appears in the list, they are concatenated in the order shown from ScanIn to ScanOut when the associated *instruction_name* is active.

It is possible for more than one *instruction_name* to activate the same ScanInterface. There is no default ordering or priority of instructions in such cases. However, Rule g) facilitates a technique whereby an *ActiveSignal* uniquely associated with only one instruction may be specified by the user's PDL to trigger the use of that particular instruction. Rule g) also applies where a particular *ActiveSignal* is essential to achieve a configuration of a network and would thus force the retargeting software to utilize only the specific instruction associated with that *ActiveSignal.*

6.5 ICL informational statements

6.5.1 Purpose

The following statements are not used to describe the structural elements of the network or instrument interface. The **`Alias`** and **`Enum`** statements can be used in the instrument interface to make PDL procedures more readable. The Parameter statement supports the use of parameterized ICL modules.

6.5.2 Alias statement

The **`Alias`** statement provides a means to group selected signals to form new buses and to assign more descriptive names to ports or registers to make their usage more intuitive. It is also used to instruct the retargeting tool in certain situations when combined with the AccessTogether and iApplyEndState

elements. The **Alias** command can be used to make PDL more human-readable. Port names, polarities, and bit order, for example, can be encapsulated and referenced by a simple name.

Grammar

```
alias_def : 'Alias' alias_name '=' concat_hier_data_signal
            (';' | ('{' alias_item+ '}' ) ) ;
alias_name : reg_port_signal_id;
alias_item : attribute_def |
             'AccessTogether' ';' |
             alias_iApplyEndState |
             alias_refEnum ;
alias_iApplyEndState : 'iApplyEndState' concat_number ';' ;
alias_refEnum : 'RefEnum' enum_name ';' ;
concat_hier_data_signal : '~'? hier_data_signal (',' '~'? hier_data_signal)* ;
hier_data_signal : (instance_name '.')* reg_port_signal_id ;
```

Generic example

```
Alias alias_name = element_list {
  RefEnum enum_name;
  AccessTogether;
  iApplyEndState value;
}
```

Examples

```
Alias MyName1[2:0] = u1.a[2], u2.u3.b[1:0];
Alias MyName2[1:0] = a[1], C { RefEnum T1;}
Alias RE  = TDR1[63] { iApplyEndState 1'b0;}
// let ADDR[7:0], DATA[31:0], and CMD[2:0] be defined as DataInPorts
Alias CNTL[42:0] = ADDR[7:0], DATA[31:0], CMD[2:0] { AccessTogether; }
```

In the third example, the **iApplyEndState** option is used to instruct the retargeting tool to always return a specific bit of a ScanRegister back to 0 at the end of the iApply time frame. This feature is useful to clear the ReadEn bit when capturing the read value. This could be required in situations in which a register bit is used as a "read trigger" and needs to be cleared following the operation so that its rising edge can be detected as the trigger of the next transaction.

In the fourth example, the AccessTogether option is used to instruct the retargeting to access the ADDR, DATA, and CMD registers all in the same shift operation so that their values are all updated at the same time (which may be necessary to support correct operation of an instrument). More importantly, the presence of this option serves as guidance to the integrator that these three registers are required to be connected so that they update simultaneously, for example by stitching them together into the same scan chain segment. If the retargeting tool cannot achieve the objective of simultaneous access as directed by the AccessTogether option, it produces an error.

Rules

a) The width of the *alias_name* vector shall match the width of the *concat_hier_data_signal* vector.

b) There shall be at most one *alias_refEnum*.

c) The width of the *enum_value* within the *alias_refEnum* shall match the width of the *alias_name*.

d) The width of *alias_name* shall match the width of *concat_number*.

e) The reg_port_signal_id elements shall only refer to *dataInPort_name*, *dataOutPort_name*, *scanRegister_name*, or *dataRegister_name* elements.

f) Alias names referencing hierarchical port identifiers that are not local to the module where the alias is defined shall only be referenced by other alias statements in ICL or by PDL commands.

g) When *alias_def* contains a *hier_data_signal* element with two or more (*instance_name* '.') levels, the *alias_name* shall not be referenced by other ICL constructs.

h) Any *hier_data_signal* element inside an *alias_def* with an *alias_iApplyEndState* statement shall respect the following conditions:

   1) Each bit of the resolved alias shall be a suitable target for a PDL iWrite command.

   2) There shall be no unknown bits within the number supplied for the iApplyEndState value.

   3) Multiple iApplyEndState values shall not specify incompatible values for any target. If there are multiple iApplyEndState values for a single target bit then they shall all be of the same value.

   4) When the resolved alias includes register bits, each such bit shall have a reset value and that reset value shall match the iApplyEndState value.

   5) iApplyEndState objects shall be driven directly from a primary input, a DataRegister with suitable reset value, a ScanRegister with suitable reset value, or a suitable constant value. The connection from the ultimate source to the iApplyEndState object shall not pass a DataOutPort with an EnableSource.

   6) An iApplyEndState may be temporarily overwritten by means of an iWrite. The lifetime of this overwrite is only the next iApply (i.e., overwriting an iApplyEndState does not change the default load value).

i) When the AccessTogether property is present, all elements in the list for this alias name shall be accessible in a single scan chain operation or parallel port operation such that their new values are updated simultaneously.

j) Circular reference to aliases shall be forbidden.

Explanations

The AccessTogether property discussed in semantic rule i) is a directive to the integrator to force the entities specified in the alias group to be connected into the serial access network (or to parallel ports) in such a way that they will all be accessed in a single scan load/unload (or parallel port) operation. This is typically realized by concatenating the associated register bits into a single, uninterrupted segment on the same scan chain (or to a group of simultaneously accessible parallel ports).

Rule h1) refers to the names that can be used in the iWrite command (see 7.9.11). Rule h6) has a special implication if the temporary overwrite happens to be located in the final iApply in a PDL sequence: it obligates the retargeting software to issue another operation to return the target to the specified iApplyEndState.

6.5.3 Enum (enumeration) statement

The **Enum** statement provides the means to use a table of mnemonic values as a methodology to make PDL procedures more readable and portable. Bit values, for example, can be referenced by a simple name. The *enum_symbol* may also be used in the equality check inside the LogicSignal or as a value for the DefaultLoad, ResetValue and CaptureSource statements.

Grammar

```
enum_def : 'Enum' enum_name '{' enum_item+ '}' ;
enum_name : SCALAR_ID ;
enum_item : enum_symbol '=' enum_value ';' ;
```

```
enum_symbol : SCALAR_ID;
enum_value : concat_number;
```

Generic example

```
Enum enum_name {
  name = value; //Repeatable
}
```

Examples

```
Enum result  {
   good = 2'b00;
   bad  = 2'b01;
   ugly = 1'b1,1'b0;        // same as 2'b10
   unk  = 2'b11;
}
```

Rules

a) The width of each *enum_value* shall be the same within an *enum_def*.

b) An *enum_name* shall be unique among all *enum_name* entries within a *module_def*.

c) An *enum_symbol* shall be unique among all *enum_symbol* entries within an *enum_def*.

6.5.4 Parameter and LocalParameter statements

The Parameter statement provides a powerful mechanism to customize instances of a generic module. A default Parameter value can be defined within a module description with associated parameter references. When the module is instantiated using an Instance statement, a Parameter statement included in the Instance statement will override the default value.

The LocalParameter statement is syntactically identical to the Parameter statement but the purpose is different. The LocalParameter statement is intended to be used to define a parameter that cannot be overridden during instantiation.

Grammar

```
parameter_def : 'Parameter' parameter_name '=' parameter_value ';' ;
localParameter_def : 'LocalParameter' parameter_name '=' parameter_value ';' ;
parameter_name : SCALAR_ID;
parameter_value : concat_string | concat_number;
concat_string : (STRING | parameter_ref) (',' (STRING | parameter_ref) )* ;
```

Generic example

```
Parameter param_name = value;
```

A **parameter_ref** type is classified according to the type of the associated **parameter_value** and can be one of the following three types:

— Unsized number such as: **0**, **6**, **9**,**'h7**, **'b1010, $size - 1**

— Sized number such as: **1'b0**, **3'h7**, or **128'd0**

— A string such as: **"myProd(5,7)"**, **"MyID"**, or **"My explanation"**

If defined as a concatenation of strings or fixed numbers (or parameter references of the same type), the **parameter_value** substituted by a parameter reference is the result obtained by performing the concatenation.

Examples

```
Module my_block {
   Parameter myStringParam = "default";
   Parameter nspace = "default";
   Parameter pname  = "myproc";
   Parameter ss   = "default";
   Parameter MSB1 = 1;
   Parameter MSB2 = $width-1;
   Parameter width = 32;
   Parameter CC   = 1;
   Parameter CS   = $MSB2, 1'b0;
      . . .
   Attribute att1 = "$myStringParam";
   ScanRegister A[$MSB1:0]  {
      CaptureSource $CS;
         . . .
   }
   ScanRegister B[$width-1:0];
   DataRegister D[$MSB2:0]  {
      . . .
      ReadCallBack $nspace $pname <R> <D> $ss $CC;
   }
}

Module top {
      . . .
      LocalParameter ABC = 4;
      . . .
   Instance I1 Of my_block  {
      . . .
      Parameter my = "My";
      Parameter string = "String";
      Parameter myStringParam = $my,$string;
      Parameter MSB1 = 31; // A defined as A[31:0] in my_block
      Parameter MSB2 = 11; // D defined as D[11:0]
      Parameter nspace  = "mynspace";
      Parameter pname   = "myproc";
      Parameter ss = "125 5'b0";
     // ReadCallBack mynspace myproc <R><D> 125 5'b0 $CC;
      Parameter CC = 4;
      // ReadCallBack for this instance is now:
      //     ReadCallBack mynspace myproc <R><D> 125 5'b0 4;
      . . .
   }
}
```

A parameter may be used to define the value of another parameter as in the following example:

```
Parameter A = 3'b0;
Parameter B = $A, 6'b110;  // B becomes 9'b000000110
Parameter C = $A, 4;       // Error: unsized number in concatenation
Parameter D = "abc";       // String parameter
Parameter E = $D, "def";   // E becomes "abcdef"
Parameter F = $D, $A;      // Error: dissimilar concatenation types
```

Rules

a) A **parameter_name** shall be unique among all **parameter_name** entries within the same module.

b) A parameter reference shall be used only as an attribute value, as an index value for a vector identifier, as a sized or unsized number, as a concatenation element, or in selected places in the iProc_namespace definition.

c) An unsized number parameter reference shall only appear where an unsized number is expected.

d) A sized number parameter reference shall only appear where a sized number is expected.

e) A string parameter reference shall only appear within a CallBack function or as the value of an attribute.

f) Circular reference to parameters shall be forbidden.

Explanations

Rule a) allows a parameter_name to match a non-parameter name because the dollar prefix used when referencing the parameter satisfies uniqueness. Rule b) prevents the following example, in which a parameter reference is used inappropriately to define an instance name:

```
parameter size = 32;
. . .
Instance inst$size of myModule {}; // Error: invalid parameter_ref usage
```

Note that rules c), d), and e) imply that software tool implementers are required to preserve the parameter type during elaboration. Rule f) forces all parameter references to be resolvable.

6.5.5 Attribute statement

The **Attribute** statement provides an additional opportunity for documentation or utilization by other software.

Grammar

```
attribute_def : 'Attribute' attribute_name ('=' attribute_value )? ';' ;
attribute_name : SCALAR_ID;
attribute_value : (concat_string | concat_number) ;
STRING : '"' (~('"'|'\\')|'\\\\'|'\\"')* '"' ;
// i.e., any character except " or \ unless preceded by a \
```

Generic example

```
Attribute att_name = att_value;
```

Examples

```
Attribute scan_type = "MuxD";
```

Rules

a) An **attribute_name** shall be unique among all **attribute_name** entries within the element in which it is declared.

b) parameter_ref shall refer only to a parameter having a sized or unsized number or a string value.

6.6 ICL connectivity

ICL has well-defined connectivity rules specifying which types of port functions and components may drive the output port functions of a module. These semantic rules were defined in Clause 5 and in the preceding subclauses of Clause 6; this subclause is merely explanatory; the rules in those subclauses supersede this subclause in case of any conflict.

Figure 52 uses a ScanOutPort and a DataOutPort to illustrate the three possible sources for an output port function of a module: a component within this module, an output port function of a child module, and a (pass-through) input port function of this module.

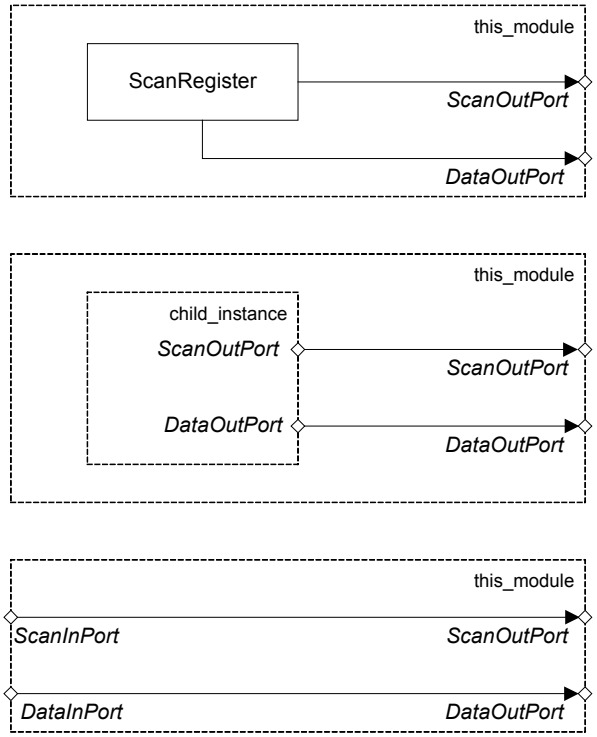


Figure 52—Output port sources

6.7 Inferring information in implicit ICL

For handoff modules, all ports of a module with an associated ICL function are required to be explicitly listed so that the ICL model and the simulation model (RTL or gates) match at their boundaries for those ICL ports. Note that the RTL or gate model may have additional functional ports that are not described in ICL (if they are not part of the access network or an instrument), but all ports described in ICL must be present in the RTL. Enforcing ICL port list consistency between the two descriptions is required to allow retargeted PDL patterns (which utilize ICL) to be applied to the simulation model at corresponding ports. However, for non-handoff modules (i.e., those that are neither the target level for retargeting nor bound for simulation), this requirement may be relaxed. The previous subclauses of this clause have defined the ICL keywords and constructs that are utilized to explicitly describe instrument access networks. However, the explicit description of every detail of these networks is often unnecessary for the process of retargeting instrument patterns to be applied by the designated network controller. ICL thus allows the omission of information in certain contexts in order to simplify the model and reduce its complexity. Such descriptions are referred to as "implicit ICL" since some of the explicit connections are missing. There are two intertwined guiding principles behind this allowance: first, ICL is a behavioral abstraction rather than a strict physical netlist, and second, the key state-change behavior modeled by retargeting software depends

only on the active scan chain. To that end, the details of the physical control signals that operate the scan chain may be omitted; it is the data connectivity of the scan chain that is essential (as traced between the ScanInPort and ScanOutPort). For the special case of broadcast scan, the SelectPort associated with each ScanInPort-to-ScanOutPort segment is also required to be explicitly described.

Rules

a) Other than for broadcast scan configurations, if any of CaptureEnPort, ShiftEnPort, UpdateEnPort, SelectPort, ResetPort, or TCKPort are missing from a module with a ScanInPort and/or ScanOutPort, then the behavior of those signals shall be assumed such that when the ScanInterface is part of the active scan chain, the associated scan registers capture, shift, update, and reset at the appropriate times.

b) In a scan broadcast configuration, if any of CaptureEnPort, ShiftEnPort, UpdateEnPort, ResetPort, or TCKPort are missing from a module with a ScanInPort and/or ScanOutPort, then the behavior of those signals is assumed such that when all of the SelectPorts of the ScanInterface are active, the associated scan registers capture, shift, update, and reset at the appropriate times.

c) Any TMSPort, if it exists on a module, shall be declared.

6.8 Active values for control signals

Most of the control signals (and corresponding ports) in ICL have fixed values that define their active state, while others are left to the user to define. Table 5 lists the ICL control signals that have defined active values; all other signals not in this table do not have a prescribed active value. For consistency with the linguistic convention that signal names are based on transitive verbs (e.g., Enable or Select or Reset), all of the control signals are active high, with the exception of the TRSTPort and ToTRSTPort port functions, which are active low to match IEEE Std 1149.1. The ResetPort and ToResetPort port functions have a user-specifiable property (ActivePolarity), which can override the default active high state (in recognition of the active low value used in IEEE Std 1500).

Rules

a) The active values of signals shall be in accordance with Table 5.

b) The active values of signals not listed in Table 5 shall not be considered prescribed.

Table 5—Defined active values for ICL signal types

Container	ICL signal	Defined active value
Module	ShiftEnPort	1
Module	CaptureEnPort	1
Module	UpdateEnPort	1
Module	ToShiftEnPort	1
Module	ToCaptureEnPort	1
Module	ToUpdateEnPort	1
Module	SelectPort	1
Module	ToSelectPort	1
Module	ResetPort	1, but may be overridden
Module	ToResetPort	1, but may be overridden

Table 5—Defined active values for ICL signal types *(continued)*

Container	ICL signal	Defined active value
Module	TRSTPort	0
Module	ToTRSTPort	0
Module	ToIRSelectPort	1
Module	WriteEnPort	1
Module	ReadEnPort	1
Module ScanOutPort	Enable	1
Module DataOutPort	Enable	1
DataRegister	WriteEnSource	1

7. Procedural Description Language (PDL): level-0

7.1 Purpose

The purpose of PDL is to provide a means to define procedures to operate an instrument. These instrument procedures are documented by PDL in terms of stimuli and expected responses for the ports and/or registers described in the ICL module for the instrument. This is very similar to the PDL, which is part of IEEE Std 1149.1-2013 (where BSDL is used instead of ICL to describe the objects for which the tests are written); the differences between the two versions of the language are documented in Annex D.

To execute PDL written for an instrument at a higher-level module boundary (e.g., a parent module or the device level) into which the instrument has been integrated, it is necessary to translate the instrument-level procedures into operations on the desired level of hierarchy (e.g. the ports and/or scan interfaces of the parent module or the device pins or an IEEE 1149.1 AccessLink) using the network definition described by ICL. The higher-level module that is the destination of this instrument-level procedure translation is called the *target module.* The process of translating a procedure to a target module is called *retargeting.*

This process can require reconfiguration of the serial access networks and can result in a sequence of several operations at the target module. The associated sequential pattern generation can take advantage of complex solving and optimization strategies, including optional merger with procedures for other instruments operating concurrently. To allow a broad set of approaches for retargeting, the procedures are described in a way that leaves sufficient freedom to the algorithm. The semantic rules in this clause carefully deal with the trade-off of correctness, predictability, and freedom of optimization.

Rules

a) When retargeting PDL for an ICL module, that target module shall be of type *handoff* (not of type *internal*).

Explanations

Since internal modules (see 6.4.5) may lack complete specification of their ICL port functions ports, PDL written for them may not be suitable for simulation against a reference netlist with a full specification of all the ports. This would render the retargeting process incomplete.

7.2 PDL levels

There are two levels of PDL. PDL level-0 is a subset of PDL level-1. PDL level-0 contains the basic commands to document the interactions at the instrument interface level. PDL level-1 uses all of the level-0

commands as language extensions to Tool Command Language (Tcl) to deliver the functionality of a programming language, including conditions, loops, variables, and data structures. Tcl was chosen because of its widespread use in design automation tooling and its natural ability to support command-structured procedures. PDL level-1 also defines introspection commands to examine the results when commands are applied in real time to the device under test (DUT).

This subclause will describe the PDL level-0 syntax and semantics while the next subclause will describe PDL level-1 commands.

7.3 Basic PDL concepts

7.3.1 Overview

PDL is a specification of sequences that come in groups of stimuli and expected responses that are applied to the ports and/or registers of an instrument. The specification of stimuli and responses is done through *setup* commands (7.5). The stimuli to the instrument are specified with the command **`iWrite`**. The expected responses are specified with the command **`iRead`**. Alternatively, both stimulus and response can be combined and anonymized with the **`iScan`** command. A group of **`iWrite`**, **`iRead`**, and **`iScan`** setup commands is applied to the instrument with the *action* command **`iApply`**. Additional commands allow the specification of idle loops and the structuring of test programs.

7.3.2 Read-write with capture-shift-update sequence

Each capture-shift-update (CSU) operation to the scan chain initiated by the iApply command is, in general, a read-write operation. The *Capture* operation reads the *current* data, the *Shift* operation makes the captured data observable while shifting in new data, and the *Update* operation writes this *new* data. The **`iRead`** command corresponds to the data in the capture and shift-out operations, while the **`iWrite`** command corresponds to the data in the shift-in and update operations. The `iScan` command contains (overlapped) read and write data. The shift-out of the captured data and shift-in of the new data are overlapped simultaneously during the *Shift* state. The control circuits for some scan registers may include logic that disables either the capture or the update logic (or both), in which case cycling through the CSU states can result in read-only, write-only, or shift-only operations. Which alternative is used can be enforced with the iOverrideScanInterface command and its –captureEn and –updateEn arguments.

An ***`iApply`*** *group* collects one or more `iWrite`, `iRead`, and `iScan` commands for exchange with the DUT. Because of the potential need for reconfiguration of the serial access network, even a single `iWrite` or `iRead` command can result in multiple CSU operations as well as operations on the primary inputs (PIs) and/or primary outputs (POs) of the device. An iApply time frame is illustrated in Figure 53 (which duplicates Figure 37 and is shown again here for convenience).

A tool is allowed to apply the stimuli and expected responses to the instrument with a large degree of freedom regarding sequence and order. A group of **`iWrite`** and **`iRead`** commands can result in any number of these sequences. A small set of rules enforces predictability for anticipated applications.

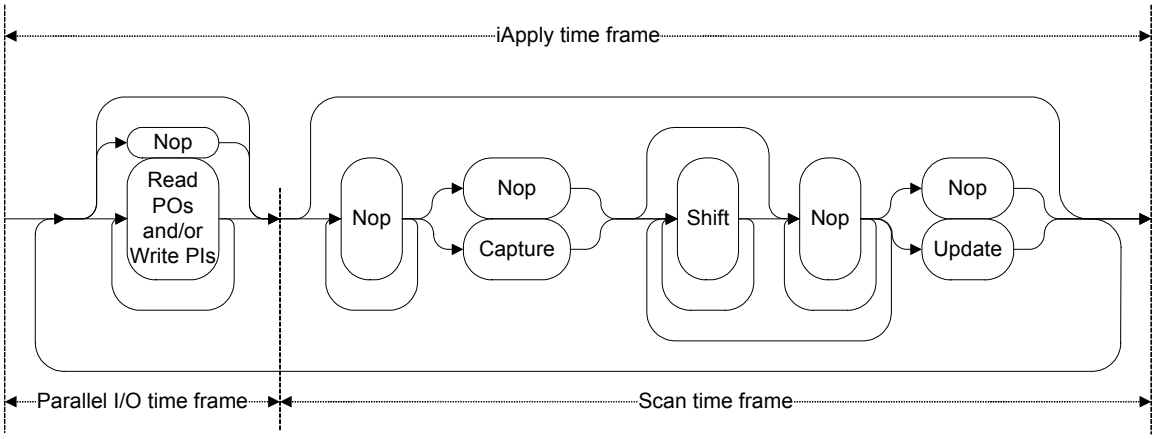


Figure 53—Sequence of operations during an iApply

Rules

a) The data to be observed by an iRead operation shall be acquired on the first possible opportunity and ignored on any subsequent opportunities, even if available.

b) The data to be controlled by a group of iWrite operations shall be valid after the final Update event associated with the iApply that ends that grouping.

Explanations

The important requirement for expected responses specified by **`iRead`** is that the data recording (or the comparison between the expected response and the observed value) is required to be done on the first opportunity to observe the value and ignored afterwards. This makes sure that volatile data that can only be read once is correctly compared. The important requirement for the stimuli specified with **`iWrite`** is that the values specified are all required to be valid after the final Update event in the `iApply` group delivering them. This means that nodes (Registers) can have arbitrary values during the iApply-timeframe (either because the state of the hardware is unknown until the first access or to give the retargeting tool a degree of freedom to reconfigure the network). However, when the iApply timeframe has finished all stimuli will have been applied.

Since the preceding definitions allow for just one value to be written to or read from a node in an **`iApply`** group, only the last value that is specified for a node will be considered.

7.3.3 Controlling and expected state models

The **`iRead`**, **`iWrite`**, and **`iScan`** commands modify only the state of a network and instrument model that is derived from the ICL descriptions, but they do not affect the device until an **`iApply`** command is encountered. For each port or register of the ICL, the model keeps a *controlling state* (modified by iWrite or by retargeting tool actions necessary to perform writes and reads) and an *expected state* (modified by iRead). The `iScan` command modifies both the controlling state and expected state models. The controlling state has two possible values '0' and '1'. The expected state has three possible values: '0' that means “expect 0,” '1' that means “expect 1,” 'X' that means “do not compare.”

The **`iWrite`** and **`iScan`** commands modify the controlling state in the model and the **`iRead`** and **`iScan`** commands define the expected state of the circuit. These commands thus add to a set of changes to the model (referred to as an iApply group) that are collected together but not immediately executed (until an **`iApply`** statement is encountered). Upon that **`iApply`**, a sequence of Capture-Shift-Update (CSU) operations is derived from the modeled states that alter the state of the target device modeled by the ICL

such that by the end of the last CSU associated with the commands in the **iApply** group, the hardware state matches the modeled state. The set of changes in the model is empty after the **iApply**. For the **iApply**, the content of the shift-in data is derived from the controlling state and the shift-out data is derived from the expected state through the retargeting process outlined in the next subclause. Note that a single iApply group may result in many different CSU operations depending on how the network is organized and on the differences between its current and next states. In order to keep track of how these operations modify the state of the circuit and result in intermittent deviations from the controlling state, the retargeting tool will need to model a third state variable called the *logic state*. At the end of the iApply timeframe, the logic state will match the controlling state in all nodes specified by iWrite calls. The logic state will have to model (but not limited to) the states '0', '1' and 'X', where 'X' is unknown. In the beginning (after power-up of the device), the logic state will be all-X and after the Reset, it will assume all the ResetValues of the ICL model, where such values are supplied.

As a consequence of the preceding definitions, the order of **iWrite**, **iRead,** and **iScan** commands within an **iApply** group does not define the order of the CSU sequences that ultimately alter the state of the circuit. The following subclause discusses the implications of this approach in more detail.

7.3.4 Order of actions

There are two very important, and perhaps superficially contradictory, policies that address the order of actions within an iApply group, as follows:

— The order and time separation in which the registers are written or read within an iApply group is not deterministic.

— In cases where a register bit is referenced multiple times within an iApply group, the last value specified shall be used.

The superficial contradiction is that the first rule indicates that order is not important, while the second rule indicates that it is. The clarification is that order cannot be important during the application of the stored model state to the device, since register accesses may require reconfiguration of the network. Order is only important in the creation of the stored model state. In other words, the top-to-bottom order of the statements in the PDL matters to the process of specifying the state, but once that state has been specified, the order in which that state is delivered within an iApply group need not be deterministic.

When processing an iRead or iWrite command, the stored model state is modified as defined by the command operand. Any portion not modified retains the previous value. Each register and port that can be referenced in an iRead or iWrite command can be translated into bits in a scan chain that is defined by the ICL description. As mentioned previously, for multiple iWrite commands within the same iApply group that refer to the same register or port, the later command dominates. This convention can be used to initialize all the bits in a register to a single value (e.g., all zeroes) and then to set an individual bit to the opposite value (e.g., one):

```
# desired result:
iWrite my_reg[1023:0] 0;  # the 0 auto-expands to fill entire width of register
iWrite my_reg[234] 1;     # set bit 234 to 1
iApply;                   # result: whole 1024-bit register is 0 except bit 234
```

The usefulness of this feature is accompanied by its ability to allow perhaps unintended overwrite operations; it is advisable for tool implementers to conditionally warn users when bits are overwritten.

```
# the importance of order in this context:
iWrite my_reg[234] 1;     # set bit 234 to 1
iWrite my_reg[1023:0] 0;  # set all bits to 0 (including bit 234)
iApply;                   # result: entire register is 0
```

7.3.5 Black-box instruments and the iScan command

Where privacy is required, a PDL procedure using iScan commands can be written for a black-box ICL module that includes only the port declarations. A black-box instrument may contain a "black-boxed" serial access network. The operation of the "black-boxed" serial access network is described by the iScan command, which is possibly created by a pattern generation tool with access to the non-black-box description of the ICL module.

Black-box modules are not simple placeholders with unknown contents; on the contrary, they have precisely defined contents known to the module creator but hidden from the integrator. There are very specific rules for handling black boxes with hidden serial access networks that depend on knowing only the precise length of the constituent scan chains while otherwise preserving the privacy desired by the black-box module creator. This is accomplished with the **`iScan`** command.

7.4 Retargeting of PDL

From a device perspective, the **`iWrite`** and **`iRead`** commands specify stimuli and expected responses for a model of the registers and ports of the device that is defined in the ICL description. With the **`iApply`** command, these internal values have to be translated into stimulus for the primary inputs and primary outputs of the device, including AccessLinks such as the IEEE 1149.1 TAP. The following process of translating the internal requirements into operations on the device interface is called *retargeting*. The process can be applied at intermediate levels of hierarchy as well, as is shown in the following example. Tools can implement a wide variety of approaches for retargeting such as techniques from automated test pattern generation. Hence only the basic principles will be shown here.

Consider the example in Figure 54:

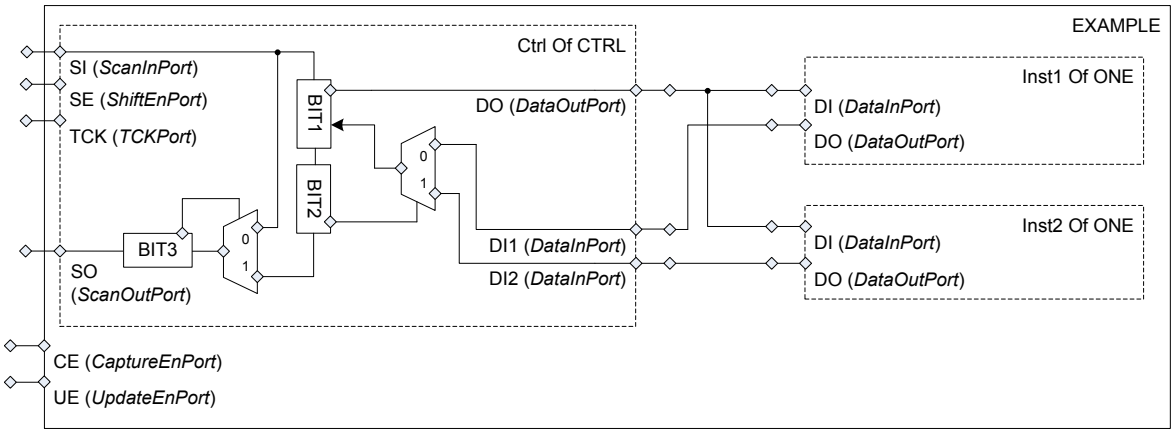


Figure 54—Example for retargeting

In the example illustrated in Figure 54, the top-level entity of the design has a scan interface. The retargeting tool will translate any internal stimuli and expected values to scan-operations on this interface. Assume that the registers reset to BIT1=1, BIT2=0, BIT3=0.

Now, consider the following sequence of four PDL commands. A valid retargeting is shown in the right-hand column (more efficient retargeting is possible).

… iReset iWrite Inst1.DI 0 iRead Inst2.DO 1 iApply …	1) Scan in 1'b1, Expect scan out 1'bX 2) Scan in 3'b111, Expect scan out 3'bXXX 3) Scan in 3'b111, Expect scan out 3'b1XX 4) Scan in 3'b000, Expect scan out 3'bXXX

The **iRead** requirement is satisfied by scan 3), the **iWrite** requirement is satisfied by scan 4). Scans 1) and 2) are just preconditioning of the network to be able to execute 3) and 4). In 1), BIT3 is scanned to change the scan mux such that it puts BIT1 and BIT2 onto the active scan chain. In 2), BIT2 is updated such that the data mux selects Inst2.DO for it to be observed with the next Capture Enable pulse.

A particular consequence of the PDL rules for stimuli is that a retargeting tool is allowed to shift BIT1 with 1'b1 (instead of the user-specified 1'b0) in the preconditioning scan 2). The retargeting tool is only required to satisfy the user requirement in the last Update Enable cycle to a register. Moreover, it is not determined which state a retargeting tool asserts for unspecified network control at the end of the iApply sequence. In the example, it is valid to set the scan mux to either 0 or 1. A retargeting tool may use this freedom for example for test time optimization.

Users can specify conflicting stimuli:

… iReset iWrite Inst1.DI 0 iWrite Inst2.DI 1 iApply …	Retargeting tool shall report conflict.

In the preceding case, it is impossible to satisfy both statements in the last Update Enable cycle. Both Inst1.DI and Inst2.DI are controlled by BIT1. The retargeting tool shall detect this type of conflict, report it, and abort the iApply execution.

On the other hand, if two stimuli or expected values are specified for the same ICL element, the last specified value will be used:

… iReset iWrite Inst2.DI 0 iWrite Inst2.DI 1 iRead Inst2.DO 1 iRead Inst2.DO 0 iApply …	1) Scan in 1'b1, Expect scan out 1'bX 2) Scan in 3'b111, Expect scan out 3'bXXX 3) Scan in 3'b101, Expect scan out 3'b0XX

It is important to be aware of the degree of freedom left to the tool. The retargeting software is allowed to freely adjust bits as needed as long as the final Update operation results in the specified value. Whenever a specific sequential behavior is required, the order of actions may be enforced by inserting **iApply** statements to force actions immediately rather than be queued and dispatched in a less predictable order.

7.5 PDL level-0 overview

7.5.1 Purpose

PDL level-0 is a pattern specification format that allows specification of procedures that can be called programmatically to operate instruments. PDL level-0 has two classes of commands: *setup* commands and *action* commands. A setup command is one that affects only the model of an instrument, typically in preparation for an upcoming action command. An action command has direct impact on the instrument or device such as applying modeled values or executing idle cycles.

7.5.2 PDL level-0 commands

Table 6 lists the PDL level-0 commands along with the type, parameters, and purpose.

Table 6—PDL Commands

Command	Type	Parameters	Purpose
iPDLLevel	Setup	**('0' | '1') '-version STD_1687_2014'**	Identify PDL flavor
iPrefix	Setup	**instance_path**	Specify hierarchical prefix
iReset	Action	**'-sync'?**	Reset the network
iWrite	Setup	**(register | port | alias) value**	Queue data to be written
iRead	Setup	**(register | port | alias) value**	Queue data to be read
iScan	Setup	**scanInterface_name '-ir'? length '-si' siData '-so' soData**	Queue data to be scanned
iOverrideScanInterface	Setup	**<scanInterfaceList> -captureEn (on) | off -updateEn (on) | off -broadcast on | (off)**	Indicate the capture, update, and broadcast behavior to be imposed on a list of scan interfaces
iApply	Action	**[-together]**	Execute queued operations
iClock	Setup	**ClockPort**	Specify a system clock, which is required to be running
iClockOverride	Setup	**ToClockPort '-source' clockPort -freqMultiplier mult –freqDivider div -period int [unit]**	Override definition of system clock when it is generated on-chip
iRunLoop	Action	**(cycleCount ('-tck' | '-sck' port)? | '-time' tvalue)**	Issue a number of clocks
iProc	Setup	**procName '{' arguments* '}' '{'commands+'}'**	Wrapper for a PDL procedure
iProcsForModule	Setup	**[namespace::]moduleName – iProcNameSpace nameSpace_name**	Identify the module in the ICL with which subsequent iProcs are associated

Table 6—PDL Commands *(continued)*

Command	Type	Parameters	Purpose
iUseProcNameSpace	Setup	**nameSpace_name**	Use namespace for subsequent iCalls
iCall	Setup	**hierProcName (arguments)***	Invoke a PDL procedure
iNote	Action	**-comment | -status text**	Send text to runtime
iMerge	Setup	**-begin | -end**	Allow merging (concurrent evaluation) of iCalls
iTake	Setup	**instance | register | port**	Disallow other merge threads from modifying a model resource
iRelease	Setup	**instance | register | port**	Re-allow other merge threads to modify a model resource
iState	Action	**reg|port value -LastWrittenValue | -LastReadValue | -LastMiscompareValue**	Document the current state of the network

7.5.3 Syntax compatibility with Tcl

PDL is syntactically compatible with Tcl. That is, any PDL program could be executed in a Tcl interpreter that implements the PDL commands through an extension. Because of this, white-space characters cannot be ignored as they are in ICL. In the following grammar fragments, white space is denoted by the grammar token **WS** and the command delimiters `';'` and **Newline** are denoted by the grammar token **eoc** (for "end of command").

PDL allows port index ranges to be written with either parentheses "()" or ICL bracket notation "[]". However, special care is required (by the IEEE 1687 runtime implementer) to support the bracketed vector index and range (as are used in ICL) in a Tcl environment. Natively in Tcl, round parentheses () are used to denote vector index and range, while square brackets `[]` have an entirely different meaning: they are intercepted in the interpreter and result in command substitution. Fortunately, ICL compatibility can be maintained by a tool implementer by taking advantage of Tcl's unknown command handler mechanism so that square-bracket vector indices and ranges can be supported in PDL:

a) A procedure named "::unknown" is required to be provided that detects valid ICL index and range formats. It returns the index or range unaltered with the surrounding square brackets for parsing by the PDL extension.

b) When the PDL extension is loaded, any Tcl proc-defined commands that would be a valid prefix to ICL index or range formats are required to be removed.

c) A wrapper for the Tcl proc definition function `proc` is required that disallows users from defining procs that are valid prefix to ICL index or range format.

It is expected that EDA tools that process PDL will provide an unknown handler that meets the preceding criteria. These tools are also required to implement the PDL level-0 command extensions.

Example of Tcl unknown handler

```
proc ::unknown {args} {
    ... # Check that $args actually has a valid ICL index or range
    return "\[$args\]"
}
```

7.5.4 PDL level-0 structure

PDL procedures are executed in the context of an instrument's ICL description. The association of PDL procedures and an ICL module is done through the command **iProcsForModule**. Procedures are declared through the command **iProc**. All PDL statements that interact with the instrument are only allowed inside iProcs to enable portable and reusable sequence descriptions. Specification of the top-level entry point to interpreting iProcs (the MAIN iProc) is left to the tool implementation.

The procedure definition (similar to the format used by Tcl) is as follows:

```
iproc_def      : 'iProc' WS procName WS '{' WS? argument ( WS argument )* WS ?
                 '}' WS '{' commands+ '}' eoc ;
argument        : scalar_id | ( '{' argWithDefault '}' ) ;
argWithDefault : scalar_id WS args ;
```

The keyword **iProc** is followed by a unique procedure name. Next comes a mandatory pair of curly braces, which may optionally contain a white-space separated list of arguments. If included, the optional arguments are used to substitute data (either register/port names or values) into the body of the procedure. Finally, another mandatory pair of curly braces contains the body of the procedure, which consists of PDL level-0 commands.

For the purposes of organization, compactness, and clarity, PDL procedures may be organized hierarchically where a PDL procedure may be called by another PDL procedure (using the `iCall` command).

A key feature of PDL level-0 is that it can be "flattened" into a transcript of operations that occur at the instrument interface. The process of flattening is further discussed when dealing with the **iMerge** command.

7.5.5 PDL level-0 example

For illustrative purposes, a simple example of the usage of PDL level-0 is shown as follows:

```
iPDLLevel 0 –version STD_1687_2014; # PDL level-0 only
iProcsForModule mbist;              # identifies the target module

iProc start_bist {bist_mode} {
   iWrite BIST_engage[42] 0b1;      # BIST_engage[42] is a register bit
   iWrite BIST_mode $bist_mode;     # $bist_mode: argument substitution
   iApply;                          # Perform the actions
   iRunLoop 10000 -tck;             # Apply 10,000 TCK periods
}

iProc read_signature {} {
   iRead bist_sig 0xaaa5555;        # get the result and compare
   iApply
}
iProc run_bist {} {
    iCall start_bist 0b110;          # value "0b110" passed to start_bist
    iCall read_signature;            # get the result
}
```

This example contains three procedures: **start_bist**, **read_signature**, and **run_bist**.

— The **start_bist** procedure has a single argument (**bist_mode**), which is the value corresponding to what should be written into the **BIST_mode** register. The two **iWrite** commands set up the values in the registers required to configure the test, and after these registers are loaded by the **iApply** statement, the instrument is clocked for 10 000 TCK cycles.

— The **read_signature** procedure has no arguments and simply reads the value in the **bist_sig** register and compares the result with the expected value.

— The **run_bist procedure** also has no arguments. It calls the two other procedures, passing the value **0b110** into the **start_bist** procedure.

Instead of a very specific **start_bist** procedure, a more general procedure could have been written using either of two different argument substitution schemes:

```
iProc start_bist {bistreg bist_mode}{
   iWrite $bistreg 0b1;           # $bistreg chooses register
   iWrite BIST_mode $bist_mode;   # $bist_mode: argument substitution
   iApply;
   iRunLoop 10000 -tck;
}

iProc run_bist {} {
   iCall start_bist BIST_engage[42] 0b110
   iCall read_signature
}
```

Or, just using the register bit as the argument to be substituted, and providing parameter to read_signature:

```
iProc start_bist {bistID bist_mode} {
   iWrite BIST_engage[$bistID] 0b1;   # select BIST_engage bit
   iWrite BIST_mode $bist_mode;        # $bist_mode: argument substitution
   iApply;
   iRunLoop 10000 -tck;
}

iProc read_signature { signature } {
   iRead bist_sig $signature;        # get the result and compare
   iApply
}


iProc run_bist {} {
   iCall start_bist 42 0b110
   iCall read_signature 0xaaaa5555
}
```

In all three variants, the flattened transcript for the **run_bist** procedure is the same:

```
iWrite BIST_engage[42] 0b1
iWrite BIST_mode 0b110
iApply
iRunLoop 10000 -tck
iRead bist_sig 0xaaaa5555
iApply
```

7.6 PDL general rules

The following rules apply to PDL commands.

Rules

a) Level-0-compliant PDL shall be comprised of the commands listed in Table 6.

b) If an argument starts with a “$” character, argument substitution shall occur before the command is executed.

c) Any element bit of *DataInPort, SelectPort, DataRegister, ScanRegister* of ICL instances shall have a modeled controlling state.

d) Any element bit of *DataOutPort*, *DataRegister*, *ScanRegister* of ICL instances shall have a modeled controlling state as well as an expected state.

e) The controlling state shall be either '0' or '1'. The initial controlling state shall be the non-X DefaultLoadValue specified in the ICL, or the non-X ResetLoadValue specified in the ICL if no DefaultLoadValue is specified, or '0' if neither is specified.

f) The expected state shall be one of the three values '0', '1', 'X', where 'X' is a don't-care that is ignored during the iApply. The reset (initial) expected state shall be 'X'.

g) After the **`iApply`** command, the changes to the controlling state shall be assumed to have been applied to the circuit and the expected state shall be have been compared with the circuit state. The expected state shall revert to 'X'.

h) PDL port operations shall be assumed to be asynchronous to any clocks, except where the IEEE 1687 hardware rules in Clause 5 mandate otherwise (e.g., TCK on ScanRegisters).

i) The **`iApply`** command shall cause stimulation of Primary Inputs, measures of Primary Outputs and CSUs to all registers and ports that were referenced by **`iWrite`** or **`iRead`** commands (see Figure 53).

j) An **`iReset`** command shall be considered valid *only* in iProcs that are associated with the top-level target module.

k) A PDL command shall comply with following lexical conventions that are consistent with Tcl:

   1) PDL is case sensitive.

   2) PDL keywords are *not* reserved.

   3) White space consists of the space and tab characters.

   4) Commands are terminated with either a newline (and optional carriage return) or a semi-colon, whichever occurs first.

   5) A comment is recognized if the first non-blank character of a *command* is a pound sign (#) and is terminated by a newline character. A comment can appear on a line by itself or on a line following a semi-colon.

   6) Line continuation is supported by entering a backslash-newline character pair. The backslash and newline, plus any leading white space on the next line will be replaced by a single space. Therefore, commands can only be continued where white space is valid.

   7) The language is procedural: commands are processed and acted upon in the order received. Acted upon could mean either queued for execution or executed depending upon what is required for implementation of the semantic rules specified for the command

Recommendations

l) It is recommended that the retargeting software does not implement a system reset at either the beginning or end of the test procedure that is not in response to a PDL command, since concatenation of tests often relies on the absence of a Reset.

Explanations

Rule c) identifies the elements that determine the configuration of the active interfaces and scan chains. These must be kept track of by the retargeting software, hence their presence in the controlling state model.

7.7 Generic PDL tokens

The grammar used to describe PDL is written using the following syntax convention, which is in ANTLR4 [B1] format:

- The colon (`:`) delimits the token name and should be read as "*is defined as*"
- A semicolon (`;`) terminates a definition
- Single quotes (`'`) surround one or more literal characters that appear as shown
- Parenthesis surround a group that can be an alternative choice or repeated
- A question mark (`?`) signifies that the group can be repeated "zero or one times"
- An asterisk (`*`) signifies that the group can be repeated "zero or more times"
- A plus (+) signifies that the group shall be repeated "one or more times"
- The vertical bar (`|`) separates alternatives
- The ellipsis (`..`) between two characters means "these two alternatives and all alternatives in the set bounded by them"
- A backspace (\) means that the next character is a literal character
- A tilde (~) means except-symbol
- Comments in the grammar start with // and continue to the end of the line

Grammar

```
// ***************
// Generic Tokens
// ***************

SCALAR_ID : ('a'..'z' | 'A'..'Z')('a'..'z' | 'A'..'Z' | '0'..'9' | '_')*;
ARGUMENT_REF : '$' SCALAR_ID;LBRACKET : '[';
RBRACKET : ']';
LPAREN : '(';
RPAREN : ')';
COLON : ':';
DOT : '.';

// =======
// Numbers
// =======
pdl_number : POS_INT | TCL_HEX_NUMBER | TCL_BIN_NUMBER |ARGUMENT_REF;
POS_INT      : ('0'..'9')+;
TCL_HEX_NUMBER : '0x' ( '0'..'9' | 'A'..'F' | 'a'..'f' | 'x' | 'X' )+;
TCL_BIN_NUMBER : '0b' ( '0'..'1' | 'x' | 'X' )+;
ISCAN_HEX_NUMBER: '{' WS? '0x' ( '0'..'9' | 'A'..'F' | 'a'..'f'
                               | 'x' | 'X' | WS | NL )+ '}'
                | '"' WS? '0x' ( '0'..'9' | 'A'..'F' | 'a'..'f'
                               | 'x' | 'X' | WS | NL )+ '"' ;
ISCAN_BIN_NUMBER: '{' WS? '0b' ( '0'..'1' | 'x' | 'X' | WS | NL )+ '}'
                | '"' WS? '0b' ( '0'..'1' | 'x' | 'X' | WS | NL )+ '"' ;
tvalue : (EVALUE | DVALUE)TSUFFIX ;

EVALUE : POS_INT'.'POS_INT'e-'POS_INT ;
DVALUE : POS_INT'.'POS_INT ;
TSUFFIX : 's' | 'ms' | 'us' | 'ns' | 'ps' | 'fs' | 'as' ;
```

```
// =====================
// Comments and Whitespace
// =====================
SL_COMMENT : '#' (~('\r'|'\n'))* ;
WS : ( ' ' | '\t' | '\\' '\r'? '\n' )+ ;
QUOTED : '"' (~('\"'))* '"' ; // Collapse into one token (mainly for iNote)
eoc : SEMICOLON | NL;
SEMICOLON : ';';
NL : '\r'? '\n' ;

// A PDL command stream consists of the following:
pdl_source : (WS | eoc | pdl_level_def | iprocsformodule_def
            | iuseprocnamespace_def | iproc_def | SL_COMMENT)+ ;
// ===========
// Identifiers
// ===========
dot_id : scalar_id ( '.' scalar_id )*;
scalar_id : SCALAR_ID;

// ******************
// Generic Identifiers
// ******************
instancePath  : dot_id;
scanInterface_name : (instancePath DOT)? scalar_id | ARGUMENT_REF ;
port   : hier_signal ;
reg_or_port   : hier_signal ;
reg_port_**or_instance : hier_signal ;
hier_signal : (instancePath DOT)? reg_port_signal_id | ARGUMENT_REF ;
reg_port_signal_id : scalar_id | vector_id ;
vector_id : scalar_id LBRACKET ( index | range ) RBRACKET |
           scalar_id LPAREN   ( index | range ) RPAREN;
index : pdl_number;
range : index COLON index  ;
enum_name : scalar_id ;
instance_name : scalar_id ;

// ==============
// Command tokens
// ==============
cycleCount : pdl_number;
sysClock : hier_signal ;
sourceClock : hier_signal ;
chain_id: scalar_id | ARGUMENT_REF;
length : pdl_number ;
procName : scalar_id ;
```

7.8 PDL numbers

Numbers in PDL level-0 are specified as unsigned integer constants in decimal, hexadecimal, or binary format. These forms are compliant with Tcl integer numbers: simple decimal numbers or a number consisting of a two-character prefix (**`0x`** or **`0X`** for hexadecimal, 0b or 0B for binary) followed by the numerical value using the corresponding character sets {0–9, a–f, A–F, x, X} for hexadecimal, {0, 1, x, X} for binary.

Example 1—Unsized constant numbers

```
123    # a valid decimal number
0x123  # a valid hexadecimal number
0xAa   # a valid hexadecimal number
0xx    # a valid hexadecimal number; same as 0bxxxx
0b011  # a valid binary number
```

```
0bx11  # a valid binary number
0B1    # a valid binary number
```

Rules

a) Omission of the bit range for a register or port or alias in PDL shall imply that the full width and index order of that element shall be used as defined in the ICL description.

b) Undersizing a value shall result in the assignment of the specified bit values to the LSBs of the associated register, with the unspecified MSBs being assigned either a `0` or `x` depending on the most significant bit of the assigned value: if the MSB of the assigned value is `x`, then the remaining unspecified bits will be assigned `x`; otherwise, they will be assigned `0`.

c) Truncating of extra leading 0s or Xs shall be allowed, but otherwise over-specifying a value shall result in an error.

Explanations

Rule a) allows PDL commands to use only the scalar_id portion of a vector_id name (i.e., without the square brackets and bit range) when the entire width and order of the named element (as defined in the ICL) are being referenced. Rule b) dictates the padding procedure to the applied when a PDL number is undersized relative to the element being referenced. Rule c) covers the case when a PDL number has more bits specified than the element being referenced: truncation is allowed only when it does not change the value of the number, otherwise it is an error.

[Note that Tcl compatibility of (unquoted) square brackets requires certain preconditions in the Tcl environment as outlined in 7.5.3.]

Example 2—iWrite to 8-bit registers with data to be written

```
iWrite reg_a 0xff;            # implicit register width, full value width
iWrite reg_a[3] 1;            # overwrite a single bit of a multi-bit reg
iWrite reg_b(7:0) 0xAA;       # explicit reg width; full hex value width
iWrite reg_c[7:0] 0b00110101; # explicit reg width; full binary value width
iWrite reg_e 0;               # implicit reg width; auto-expanded value
iWrite reg_f 2;               # implicit reg width; type-converted value
iWrite reg_g[4:2] 0b010;      # subrange of reg width; binary value
```

Example —iRead from 8-bit registers with expected values

```
iRead reg_p 0xff;            # implicit reg width, full expected value width
iRead reg_q 0;               # implicit reg width; auto-expanded exp value
iRead reg_r 2;               # implicit reg width; auto-expanded exp value
iRead reg_s 0x00;            # implicit reg width; hex expected value
iRead reg_t[6:4] 0b110;      # implicit reg width; bin expected value
iRead reg_u(6:4) 0xe;        # explicit reg width; hex expected value
iRead reg_w[7:0] 0bxxxx1xxxx;# explicit reg width; extra-long binary value
iRead reg_x 0bx;             # implicit reg width; same as 0xXX
iRead reg_y[4:2] 0b010;      # subrange of reg width; binary value
iRead reg_z[4:2] 0x0;        # subrange of reg width; same as 0b000
```

7.9 PDL level-0 commands

7.9.1 Namespaces

All identifiers shall be unique within an **iProc** procedure. **iProc** names are unique within a combination of ICL and PDL namespaces. For detailed semantic rules see definition of the **iProc** command.

7.9.2 Prefixes

In PDL, all references to ICL entities (instances, registers, ports) is done through relative path names. For example, when providing PDL for a particular ICL module, its registers can be referenced relative to the ICL module. However, for a tool implementing IEEE Std 1687, all ICL references are ultimately resolved to absolute hierarchical names. For example: Register *r* within ICL instance *a.b* has the absolute hierarchical name *a.b.r*

Since there is more than one command that influences the hierarchical scope (iCall, iPrefix), a formal mechanism for describing the semantic rules between these two commands is introduced. In the following, the scope inferred by iCall is called the iCallPrefix. The iCallPrefix is only modified by the iCall command and specifies the scope/prefix when entering the present iProc through an iCall command. The iPrefix command can add to the iCallPrefix, resulting in the effective prefix *effPrefix*. The effective prefix *effPrefix* is then the hierarchical prefix/scope that is used by all commands, including iCall.

Example ICL

```
NameSpace SimpleModule;

Module ModuleOne {
       Enum enumDerived {
                    TRUE=1; FALSE=0;
       }
       DataInPort diDerived
                    { RefEnum enumDerived; }
       DataOutPort doDerived
                    { RefEnum enumDerived;}
       Instance inBase Of Base {
                    InputPort diBase = diDerived;
       }
}

Module Base {
       Enum enumBase { TRUE=1; FALSE=0; }
       DataInPort diBase{ RefEnum enumBase; }
       DataOutPort doBase{ RefEnum enumBase; }
}

NameSpace Integrator;

Module Top {
       Instance one Of SimpleModule::ModuleOne
                    { InputPort diDerived = di; }
       Instance two Of SimpleModule::ModuleOne
                    { InputPort diDerived = di; } }
```

The following example PDL is annotated with 14 execution steps, assuming that the retargeting tool is instructed to retarget Top.TestA. Each execution step is denoted by a comment with a number ***N).*** The

execution begins at ***1)*** and proceeds to ***14***). It is instructive to follow the numbers as they jump up and down through the example as the iCall statements are processed in order, serially. Also note that there are two (different) iProcs with the same name ("Init"), one associated with module SimpleModule::ModuleOne and the other with SimpleModule::Base. The two "Init" procedures are unique because of this association (performed by the iProcsForModule command). The "iCall Init" at number 4 is calling the Init proc registered with the ICL module associated with the ICL instance "one." The "iCall Init" at number 7 is calling the Init proc registered with the ICL module associated with the ICL instance "one.inBase."

```
iPDLLevel 0

iProcsForModule Integrator::Top
iProc TestA {} {
   # 1) iCall_prefix is empty
   iCall one.LinkingOne # 2) will result in iCallPrefix "one"
   iCall two.LinkingOne # 8) will result in iCallPrefix "two"
}

iProcsForModule SimpleModule::ModuleOne
iProc LinkingOne {} {
   # 3) iCall_prefix is "one"
   # 9) iCall_prefix is "two"
   iCall Init # 4) and 10) Nothing added to iCallPrefix
}

iProc Init {} {
   # 5) iCall_prefix is "one"
   # 11) iCall_prefix is "two"
   iPrefix inBase # 6) effPrefix is now "one.inBase"
                  # 12) "two.inBase"
   iCall Init     # 7) iCallPrefix will be "one.inBase"
                  # 13) "two.inBase"
}

iProcsForModule SimpleModule::Base
iProc Init {} {
   # 8) iCall_prefix is "one.inBase"
   # 14) iCall_prefix is "two.inBase"
   ...
}
```

Rules

a) The *effective* prefix of any ICL register, port, or instance reference (called *effPrefix* in the following) shall be the prefix created by **`iCall`** commands concatenated with the instancePath supplied by **`iPrefix`**. *(*i.e., *effPrefix* = *iCallPrefix* + '.' + instancePath + '.').

7.9.3 iPDLLevel command

The **`iPDLLevel`** command indicates that subsequent commands consist solely of PDL of the indicated level and version and is required to be the first PDL command encountered. The level identifies the PDL as just the basic commands (level-0) or the full set of both basic and extended Tcl commands (level-1). The version is used to distinguish IEEE 1687 PDL from IEEE 1149.1-2013 PDL. The indicated level and version remain in effect until another **`iPDLLevel`** command is encountered.

Grammar

```
pdl_level_def  : 'iPDLLevel' WS pdl_level WS '-version' WS versionString
                 eoc ;
pdl_level      :  '0' | '1';
versionString  : 'STD_1687_2014' ;
```

Examples

```
iPDLLevel 0 –version STD_1687_2014; # PDL level-0 only
```

Rules

a) If **pdl_level** is **0**, only the rules in Clause 7 shall apply. If **pdl_level** is **1**, the commands in Clause 8 as well as Tcl commands are allowed; please refer to Clause 8.

b) The **iPDLLevel** command shall be the first PDL command in a PDL stream and shall indicate that subsequent commands consist solely of PDL of the indicated level and version. This indication shall apply until another **iPDLLevel** command is encountered.

7.9.4 iProcsForModule command

The **iProcsForModule** command is used to specify which ICL module the subsequent **iProc** statements are associated with. Note that there could be multiple instances of the module and, therefore, multiple instances for which the pattern could be retargeted.

The standard does not provide a procedure for specifying which Module should be the target Module (i.e., the top-level) and how the initial iProc entry-point is specified. This detail is left to the tool provider.

iProcsForModule can appear multiple times in a PDL-file to define iProcs for more than one ICL Module. After a valid module has been specified with iProcsForModule, iProcs can be defined with the iProc command.

Grammar

```
iprocsformodule_def  : 'iProcsForModule' WS module_name
                       (WS '-iProcNameSpace' WS pdl_namespace_name)? eoc;
module_name : (icl_namespace_name '::')? scalar_id ;
icl_namespace_name : scalar_id;
pdl_namespace_name : scalar_id;
```

Examples

```
iProcsForModule MbistModule;    # PDL was written for this module

iProcsForModule vendorA::HSS;   # PDL was written
                                # for module in a specific ICL namespace

iProcsForModule vendorA::HSS –iProcNameSpace temp_measure; # PDL was written
                                # for module in ICL namespace vendorA,
                                # but the to-be-added iProcs will be placed
                                # in PDL namespace temp_measure to
                                # make sure other operation's/test's iProcs
                                # do not clash
```

Rules

a) *scalar_id* in the *module_name* shall be a Module defined in the ICL default namespace or in the namespace specified by *icl_namespace_name* when present.

7.9.5 iProc command

The **iProc** command identifies the name of the procedure and, optionally, lists any arguments included as variables in the procedure. **iProc** names unambiguously reference a procedure defined for the targeted module/instrument; i.e., within a name space. iProcs with the same name in the namespaces will assume the definition that was provided last. The syntactical and semantic rules follow those of a Tcl **proc**. However, a PDL level-0 **iProc** is required to adhere to the grammar and the semantic rules upon definition of the **iProc**, whereas a Tcl **proc** is not semantically evaluated until it is called.

The arguments are a space separated ordered list. A pair of arguments enclosed within braces—an argument and the associated default—can also be included. Arguments without a default value must be listed before those with a default value.

For **iCall** invocations, arguments are listed in the order they appear in the **iProc** command (positional operands). If the arguments include a default, they may be omitted from an **iCall** statement; otherwise, they shall be included. Once an **iCall** command omits an argument, all of the remaining arguments are omitted as well.

Grammar

```
iproc_def      : 'iProc' WS procName WS '{' ( WS? argument ( WS? argument )*
                  WS? | WS? ) '}' WS '{' commands '}' eoc ;
argument       : scalar_id | ( '{' WS ? argWithDefault WS? '}' ) ;
argWithDefault : scalar_id WS args ;
args           : pdl_number | enum_name | reg_or_port;
commands       : command+;
command: WS? (icall_def              |
                iuseprocnamespace_def      |
                iprefix_def                |
                ireset_def                 |
                iread_def                  |
                iwrite_def                 |
                iscan_def                  |
                ioverridescaninterface_def |
                iapply_def                 |
                inote_def                  |
                imerge_def                 |
                itake_def                  |
                irelease_def               |
                iclock_def                 |
                iclock_override_def        |
                istate_def                 |
                irunloop_def               |
                SL_COMMENT) WS? eoc | WS? eoc ; // empty line acceptable
```

Examples

```
iProc myproc {arg1 arg2 { arg3 24 } { arg4 0x32 } { arg5 1024 } { }
  . . .
}

iProc myproc2 { } {
   . . .
}
```

The **myproc** procedure has five (5) arguments: **arg1**, **arg2**, **arg3**, **arg4,** and **arg5**. The last three arguments have defaults of 24, 0x32, and 1024**.**

Rules

a) Once the **argWithDefault** rule has been entered, it shall be entered for the remainder of the arguments.

b) An iprocsformodule_def shall always appear somewhere in the PDL command stream prior to an iproc_def.

c) An **argument** name shall not be the same as a name of a parameter in the ICL module specified by the iProcForModule command and to which the iProc is registered.

d) A procName that is defined more than once shall keep the definition that is defined last.

Explanations

Rule a) refers to the second and third items in the grammar specification. Each argument defined for an iProc may have an optional default value, but the order in which arguments are defined is constrained by the presence or absence of those optional default values. This rule implies that after the first argument with a default, there may not be any arguments without a default. Rule c) prevents a name collision between ICL parameters and PDL arguments.

7.9.6 iUseProcNameSpace command

The **iUseProcNameSpace** command is used to specify a namespace that is used for all the subsequent iCalls made within the iProc in which the iUseProcNameSpace command appears. The purpose is analogous to iPrefix, in that it saves the user from typing in the NameSpace on each iCall command. The scope of this command is limited to within the iProc in which it appears. A second appearance of this command within the same iProc changes the NameSpace to the argument given in that second appearance, and so forth with further appearances. An iUseProcNameSpace command with '-' argument resets the namespace to that of the root module.

Grammar

```
iuseprocnamespace_def  : 'iUseProcNameSpace' (pdl_namespace_name|'-') ;
```

Examples

```
iUseProcNameSpace MyNameSpace; # set namespace to MyNameSpace

iUseProcNameSpace -;  # return to root name space
```

Rules

a) **pdl_namespace_name** shall be defined for the module associated with the instance_name of the iProc being iCalled**.**

7.9.7 iPrefix command

The **iPrefix** command is a convenience command intended to simplify PDL by supporting design hierarchy within the ICL Modules. The **instancePath** parameter is the instance path to the object in ICL relative to the module to which the current **iProc** is associated. The parameter provides an instance path name prefix for all subsequent instance, register, and port names. If defined, *all* subsequent register and

port and instance names (until the end of the iProc or until a new iPrefix is specified) will be prefixed with the **instancePath** parameter plus a trailing dot (.).

The path name specified with **iPrefix** always overwrites the previously specified instance path name. If the **iPrefix** command specifies instance path parameter '-', the previous path name is deleted. The default prefix at the beginning of an iProc is the empty string.

Grammar

```
iprefix_def          : 'iPrefix' WS (instancePath|'-') ;
```

Examples

```
iPrefix myinst1.myinst2;   # identifies an instance path
iRead myport;              # same as iRead myinst1.myinst2.myport
iRead myinst3.myport;      # same as iRead myinst1.myinst2.myinst3.myport
iPrefix -;                 # removes the instance path
iRead myinst3.portA;       # same as iRead myinst3.portA
```

Rules

a) The instancePath shall reference an ICL instance, relative to the module specified by **iProcsForModule**.

b) instancePath shall be used for any ICL register, port, or instance reference in subsequent commands (excluding iPrefix) until a new prefix is specified or until the end of the iProc.

c) **'iPrefix -'** shall mean that the effective Prefix is reverted to the *iCallPrefix* with which the **iProc** was called.

7.9.8 iCall command

The **iCall** command provides a mechanism to invoke an **iProc** from within another **iProc**. Note that while an argument can be passed to the **iProc**, there is no means provided to return argument value. The **iCall** name is either an **iProc** name or an instance path prefixed to an **iProc** name.

In order to enable the scopes of argument names and hierarchical prefixes, a mechanism similar to execution stack frame is described here [Rule g)]. In particular, after an **iCall** all prefixes and argument values are required to be equivalent to the values before the **iCall**.

The semantic rules also define an indirect evaluation mechanism called **iProc** flattening [Rule h)], which is important only during parallel execution with **iMerge**. **iProc** flattening resolves all prefixing and argument substitution but defers execution of the PDL commands inside the **iProc** until after the result of the **iProc** flattening has been merged. Introduction of this mechanism for **iCall** results in a simplified set of **iMerge** rules, however its benefit will not become clear to the reader until after the introduction of **iMerge** later in this clause.

Even though listed procedurally, both rules h) and i) are meant to explain the concept and restrict the semantics, but they do not impose an implementation.

Grammar

```
icall_def : 'iCall' WS (instancePath DOT)? (pdl_namespace_name '::')? procName
            ( WS args )* ;
```

Examples

```
iProc myproc1 {arg1 arg2 { arg3 24 } { arg4 50 } { arg5 1024 }} {
   . . .
}
iProc myproc2 { } {
   . . .
}

iCall myproc1 10 20;             # use default for last three arguments
iCall I1.myproc1 10 20 30 40 50; # all args specified
iCall I1.I2.myproc1  10 20 30;   # arg4 is missing so can't specify arg5
iCall I1.I2.myproc1  10;         # Error: needs two arguments
iCall I4.I5.myproc2    ;         # no args
iCall I4.I5.myproc2  10;         # Error: no arg allowed
iCall I4.I5.vendor2::myproc2  10; # call vendor2's iProc myproc2
```

Rules

a) **procName** shall reference an **iProc** definition in the PDL namespace **pdl_namespace_name** targeting the ICL module that *effPrefix*+**instancePath** is an instance of (or *effPrefix* if **instancePath** is omitted).

b) **procName** shall be defined by an **iProc** command prior to being called by **iCall.**

c) If **pdl_namespace_name** is omitted, the namespace defined by **iUseProcNameSpace** shall be used instead. If **iUseProcNameSpace** has not defined a **pdl_namespace_name**, the global (i.e., no) PDL namespace is assumed.

d) The number of arguments in the **iCall** command shall equal or exceed the number of arguments with no default in the **iProc** definition of **procName**.

e) The number of arguments supplied by the iCall shall not exceed the total number of arguments specified by the iProc definition.

f) Once an argument with a default is omitted, no other arguments shall be included.

g) The order of arguments in the **iCall** command shall match the order of arguments in the associated **iProc** definition.

h) Evaluation of the **iCall** command shall consist of the following:

   1) If **instancePath** is specified, update *iCallPrefix* to the current *effPrefix* concatenated with **instancePath**+'.'. (i.e., *iCallPrefix* = *effPrefix* + **instancePath** + **'.'**, *effPrefix* = *iCallPrefix*).

   2) Replacing all argument definitions by the arguments defined by **procName**, replace the non-defaulted arguments with the arguments specified by the **iCall**. (like adding a new cmd stack frame in Tcl).

   3) Evaluating all statements in **procName**.

   4) Restoring the previous parameter and argument definitions.

   5) Restoring *iCallPrefix* and *effPrefix* to their previous values.

i) Flattening of the **iCall** command shall consist of the following:

   1) If **instancePath** is specified, update *effPrefix* by concatenating the current *effPrefix* with **instancePath**+'.'. (i.e., *iCallPrefix* = *effPrefix* + **instancePath** + **'.'**, *effPrefix* = *iCallPrefix*).

   2) Replace all argument definitions by the arguments defined by **procName**, replace the non-defaulted arguments with the arguments specified by the **iCall**.

3) Substitute all parameter and argument references in **`procName`**. Evaluate all iPrefix commands and replace all ICL port, register or instance references with relative paths to the previous *effPrefix*.

4) Substitute any **`iMerge -begin`** … **`iMerge -end`** block by the merged result.

5) Flatten any **`iCall`** in the result of step 2.

6) Substitute the **`iCall`** command by the result of step 3.

7) Restore the previous parameter and argument definitions.

8) Restore *iCallPrefix* and *effPrefix* to their previous values.

j) For the top-most iProc, the *iCallPrefix* and *effPrefix* shall be initialized by the retargeting tool to match the top-most instance (that the iProc is retargeted to).

7.9.9 iReset command

The **`iReset`** command forces the ResetPort to be asserted on the target module. If the target module is the top-level module and has an AccessLink, its reset sequence is invoked (for example, by TLR or TRST* when the AccessLink is IEEE 1149.1). The controlling state and expected state in the model return to their mandated initial states. iReset implies that the retargeting tool returns its internal circuit and network model (the logic state) to the ResetValue state specified in the ICL. If there is no ResetValue specified in the ICL for a state element, then its value is unaffected by the iReset command.

Grammar

```
ireset_def : 'iReset' WS (-sync)? ;
```

Examples

```
# reset the network and instruments defined to be in reset
# when the instrument access network is in reset
iReset
```

Rules

a) The controlling state as well as the expected state shall return to their initial values when an iReset command is encountered.

b) The iReset command shall be present only in an iProc associated with the targeted ICL module.

NOTE—Attempting to retarget an iReset command shall result in an error.

c) The iReset command shall reset the circuit by toggling the ICL ResetPort functions on the target ICL module. It shall also invoke the reset sequence of the associated AccessLink. In the case of IEEE Std 1149.1, this means to use asynchronous TRST* if available, and synchronous TMS if not.

d) If the **`-sync`** option is supplied, the reset shall be done by issuing the synchronous reset sequence of the AccessLink even if there are TRSTPorts on the top-level ICL module. In the case of IEEE Std 1149.1 this means to use a sequence of five TMS=1 patterns for reset.

e) Supplying the –sync option with the iReset command targeted for a module without a TMSPort shall have the same effect as not supplying the –sync option.

Explanations

The iReset command instructs the device to perform a reset operation, but the specific action (i.e., the waveform on a pin or a port) is a function of the hardware available on the device (as described in the ICL). The device may have a top-level TAP, which may or may not have a TRST* pin, or the device may be a module with a ResetPort. For each of those three contexts, the waveforms on the relevant port functions of a module in response to an iReset command are illustrated in Table 7, Table 8, and Table 9.

Table 7—iReset action when top-level TAP has no TRST* pin

Port function	Waveform
`TMSPort`	111110
`ResetPort –ActivePolarity 1 (if any)`	010000
`ResetPort –ActivePolarity 0 (if any)`	101111

Table 8—iReset action when top-level TAP has a TRST* pin

Port function	Waveform
`TRSTPort`	101
`TMSPort`	000
`ResetPort –ActivePolarity 1 (if any)`	010
`ResetPort –ActivePolarity 0 (if any)`	101

Table 9—iReset –sync action

Port function	Waveform
`TRSTPort`	11111
`TMSPort`	11111
`ResetPort –ActivePolarity 1 (if any)`	01000
`ResetPort –ActivePolarity 0 (if any)`	10111

NOTE—The –sync option is irrelevant when there is no TMSPort on the target module.

7.9.10 iRead command

The purpose of the **`iRead`** command is to define data to be observed and shifted out of the instrument during a subsequent **`iApply`** command. As an option, expected data can be defined for comparison with the observed data. Otherwise, the scan data will not be compared, just read (and logged, depending on the tool implementation). When an **`iApply`** command is executed, CSU operations are generated that make the queued iRead registers observable, and the shifted-out scan data is compared to the expected observe model.

Multiple **`iRead`** commands can be entered prior to an **`iApply`** command. Each **`iRead`** command modifies the current expected observe model and schedules the register or port for observation. Therefore, multiple references to the same register or port can overwrite the expected values of previous commands.

Grammar

```
iread_def           : 'iRead' WS reg_or_port (WS ( pdl_number | enum_name ))?;
```

The **`reg_or_port`** name refers either to a scan or data register defined in ICL or an ICL **`DataOutPort`**. The **`reg_or_port`** names are subject to the effective prefix (see *effPrefix* in 7.9.2) that is modified by the **`iCall`** and **`iPrefix`** commands.

The action taken by this command is to modify the expected state in the model of the register or port. The **reg_or_port** is read during the next iApply, even if the expected state that was provided is 'X' (or not supplied at all).

An unsized pdl_number will be extended to match the width of the register or port that is being read. It is considered an error if the unsized number contains more significant bits than are available in **reg_or_port**.

Examples

All registers are defined as length 10, enum BIT4_HIGH is 10'bXX_XXX1_XXXX.

```
   iRead myreg1 0b010X0x01;   # zero-filled to: 0b00010x0x01
   iRead myinst.myreg2 0b11;  # zero-filled to: 0b0000000011
   iRead myreg3 0xFxd;        # Error: Does not fit myreg3[9:0]
   iRead myreg4               # read myreg4 but do not compare
   iRead myreg5 BIT4_HIGH     # # Reads all 8 bits, but will compare just
myreg5[4]
```

Rules

a) The **reg_or_port** name when prefixed by *effPrefix* shall be a valid ICL register name, **DataOutPort** name or an alias that ultimately points to these types in the ICL data model with respect to the target module. Under the condition that the **reg_or_port** is not in the fan-in of a select signal associated with a ScanInterface, it may also be a DataInPort, a SelectPort, or a ToSelectPort.

b) The size of **pdl_number** shall match the size of **reg_or_port**: an unsized **pdl_number** shall be extended to match the width of the register or port that is being read. It shall be considered an error if the unsized number contains more significant bits than are available in reg_or_port.

c) If **enum_name** is specified, the **ENUM_ID** referenced by **enum_name** shall be an enumerated value of the **RefEnum** attached to **reg_or_port.**

d) Care-bits (*0*/*1*) of **pdl_number** or **enum_name** overwrite the expected state and shall be compared with the hardware state of **reg_or_port** during an **iApply**. Don't-care *X* bits of **pdl_number** or **enum_name** clear the expected observe model state and reset it to *X*, no comparison will take place.

e) The hardware state of **reg_or_port** shall be observed (i.e., read) during a subsequent **iApply**. This implies that reg_or_port shall be read even if the expected state is all-X. The observed value of **reg_or_port** shall be compared with the specified bits of the expected state of **reg_or_port**.

f) If **reg_or_port** is observed multiple times during a sequence of CSU operations, only the first observation shall be used for comparison with the expected state.

Explanations

Rule a) mirrors rule a) in the next subclause (see 7.9.11); see the explanation there, keeping in mind that iRead actions have many of the same sensitization conditions as iWrite actions. Rules c) and d) refer to **enum_name**, which is defined in ICL (see 6.5.3) and is treated exactly like a plain number.

7.9.11 iWrite command

The purpose of the **iWrite** command is to define new data for registers or ports that will be controlled through the scan chain or primary inputs during a subsequent **iApply** command. The register and port

name parameters refer to registers and ports each having a defined length and current value. The references to ICL register and port names are subject to the effective prefix (*effPrefix*) as mentioned before.

When a port or register is referenced by this command, the bits in the controlling state defined by the register or port name parameter are overwritten with user data and the register or port is scheduled for an update. When an **iApply** command is executed, the controlling state is the target objective for a sequence of CSU operations that apply the model bits to the hardware.

Multiple **iWrite** commands can be entered prior to an **iApply** command. Each **iWrite** command modifies the controlling state. Therefore, multiple references to the same register bits or ports within an **iApply** group result in overwriting the controlling value specified by previous **iWrite** commands of the same **iApply** group.

Grammar

```
iwrite_def : 'iWrite' WS reg_or_port WS ( pdl_number | enum_name );
```

The **reg_or_port** (prefixed by *effPrefix*) refers to either a scan or data register defined in ICL or an ICL **DataInPort**.

The action taken by this command is to modify the controlling state of the register or port. The model bits defined by the **reg_or_port** parameter are overwritten with the data specified by **pdl_number** or **enum_name**. When an **iApply** command is found, a sequence of CSU operations is created that applies the new state of the controlling state to the hardware.

Examples

All registers are defined as length 10.

```
iWrite inst1.myreg3 0xffd; # Error, would discard bits
iWrite myreg4 0x373;       # same as 0b1101110011
iWrite myreg4 0xXX0;       # Results in 0b1101110000 (X->previous)
iWrite myreg4 0xF;         # Results in 0b0000001111 (pad with 0s)
```

Rules

a) The **reg_or_port** name when prefixed by *effPrefix* shall be a valid ICL data_signal, **WriteEnPort**, **ReadEnPort**, **AddressPort,** or an alias that ultimately points to these types. However, SelectPorts (or data signals that fan into such SelectPorts) that are part of a ScanInterface shall not be written.

b) The size of pdl_number shall match the size of **reg_or_port**: an undersized pdl_number shall be extended according to rule b) in 7.8 to match the width of the register or port that is being written. It shall be considered an error if the oversized number contains more significant bits than are available in **reg_or_port**.

c) If **enum_name** is specified, the **ENUM_ID** referenced by **enum_name** shall be an enumerated value of the **RefEnum** attached to **reg_or_port.**

d) The bits of **pdl_number** or **enum_name** overwrite the controlling state of **reg_or_port**. 'X' bits leave the controlling state in its previous state. During the next **iApply**, **reg_or_port** shall be updated through a primary input or the scan chain with the controlling state at the time of the **iApply**.

e) If **reg_or_port** can be controlled multiple times during the sequence of primary input / primary output (PI/PO) + CSU operations of an **iApply**, the controlling state shall be in place at least by the end of the iApply time frame.

f) When a group of scan registers is configured for broadcast, the iWrite commands shall be issued only to those scan registers that are part of the active scan chain and not to those shadowing the scan registers on the active scan chain.

Explanations

Rule a) requires that the retargeting software be fully responsible for justifying the SelectPort of the ScanInterface to a high value when the ScanInterface is part of the active scan chain or to a low value when the ScanInterface is not part of the active scan chain (as described by the semantic rules in the SelectPort or ToSelectPort subclauses). This guidance applies to the integrator connecting the scan chains and running the retargeting tool. Note that this rule does not restrict the user from writing to a ScanRegister bit in a SIB; that is a physically separate and much safer object to write (since it leaves the retargeting software with other options to activate or deactivate the associated ScanInterface). It is, however, possible to model a non-plug-and-play scan interface where there is no select port but there is instead a logical combination of DataInPort values that activate the scan interface. In the example below, the retargeting software can deduce what to do to activate the scan interface (i.e., drive DI[2:0] to 3'b010 or 3'b011), but the PDL and surrounding logic must be designed with care to make sure the scan interface is not activated when not part of the active scan chain. For example, the PDL may issue an iWrite command on the DI port, but it must enforce that DI[2:1] will not be written to 2'b01, otherwise the scan segment that is enabled by that condition may get accidently disturbed.

```
Module ex1 {
  ScanOutPort so { Source m1; }
  DataInPort DI[2:0];
  ScanMux m1 SelectedBy DI {
    3'b01x : tdr[0];
  }
  …
}
```

Rule c) refers to enum_name, which is defined in ICL (see 6.5.3) and is treated exactly like a plain number.

Rule f) refers to a configuration of instances with scan interfaces in which a common scan data bit is transmitted to multiple scan register destinations concurrently (i.e., “broadcast”). A broadcast group of ScanInterfaces is defined in the ICL Instance statement. A ScanInterface that is on the active scan chain is said to be “shadowed” by the other ScanInterfaces in the broadcast group that are not on the active scan chain. This rule requires that in such situations the object of the iWrite command be associated with the ScanInterface that is on the active scan chain.

7.9.12 iScan command

The purpose of the **iScan** command is to define scan data for a "black-box" instrument. A black-box module is a concealed module in which a serial access network is present but concealed from the ICL description. A test procedure using an **iScan** command takes full control of the network inside the black box, including any possible scan chain reconfiguration. The scanInterface_name parameter defines the name of the ScanInterface (which is itself comprised of a ScanIn/ScanOut pair and associated control signals as defined in the ICL) of the black-box module. Since the **iScan** command is a “setup” command, the scan data just overwrites a controlling and expected state in the model (as with **iWrite**/**iRead)**. However, in contrast to **iRead** and **iWrite** this model state has variable length/size that is defined by an

option of the iScan command. While intended for black-box modules, iScan can be used in non-black-box module situations as well.

The iScan command specifies the precise length of the scan chain between the ScanInPort and ScanOutPort of the ScanInterface it is interacting with, along with the data payloads being delivered and expected. A length parameter defines the length of both the input and output scan data. All four (4) parameters (scan interface, length, ScanIn value, and ScanOut value) shall be defined for every **iScan** command. This enables the retargeting software to assume the correctness of the specified length and the integrity of the registers inside of the blackbox. Since **iScan** is a setup command, it only modifies the model state (in the same manner as **iWrite** and **iRead**) and requires a following **iApply** command to initiate the series of CSU operations on the device.

The user assumes complete responsibility for correctly supplying the appropriate scan data to each shift path, paying particular attention to the length of the shift path not only during but also after the CSU sequence. This is because the length of a given scan chain inside a black-box module could be variable (i.e., that scan chain may contain scan muxes that reconfigure the chain based on the value of bits shifted into it and latched during the Update event). The tools managing an IEEE 1687 network are required to always know the precise length of every scan chain, including those within a black-box module. Each **iScan** command specifies the length of the chain for the payload in that command, but after application the length of the scan chain may change. If followed by another **iScan** command, this new length will be reflected in that new command.

Whenever a black-box's ScanInterface is present on the active scan chain, its length is required to be precisely known. There are three methods for the IEEE 1687 tool to determine this length: (1) if there is an iScan of the black-box module associated with the current iApply, the length parameter of that iScan defines the length of the black-box's ScanInterface; (2) if the black–box's ScanInterface had been previously accessed with the –stable option, the length of the black-box module is assumed to be stable and the previously iScanned value should be re-applied; (3) after an iReset, the length of the black box is as defined in the ICL and is also stable when the associated default value for the black-box module is reapplied.

There are two important implications about black-box design and usage with respect to maintaining a precise length. First, the black-box module is required to be designed with at least one stable state that can be repeated an arbitrary number of times without changing the length of the black-box module. The default length and value for a black-box module in the ICL are required to have this property. Second, whenever an iScan access to a black-box module results in a reconfiguration that changes the length of the black-box module, it is required to be followed by another iScan access with the new length. If the data in that new iScan access corresponds to a stable state (i.e., it has the property that it may be repeated without changing the length of the black-box module), then that iScan is to be issued with the –stable option if no further iScans need to be issued in the sequence.

Shift data can be passed to **iScan** either as a simple PDL number, or as a multi-line string in curly braces, which concatenates all of the numbers contained within the braces.

Grammar

```
iscan_def        : 'iScan' WS ('-ir' WS)? scanInterface_name
                   (WS '-chain' WS chain_id)? WS length
                   (WS '-si' WS iscan_data)? (WS '-so' WS iscan_data)?
                   ('-stable')? ;
iscan_data    : pdl_number | ISCAN_HEX_NUMBER | ISCAN_BIN_NUMBER ;
```

Examples

```
   iScan myif 8 -si 0b01010101 -so 0b11xx00xx;
   iScan myif 8 -si 0b11 -so 0b11;      # zero-filled to 0b00000011
```

```
iScan myif 6 -si 0xfd -so 0xXX;      # error: truncation disallowed

# examples of white space:
iScan myif 12 –si 0b000000000000
# … is equivalent to:
iScan myif 12 –si "0b0000 0000 000 0"
# … is equivalent to:
iScan myif 12 –si "0b0000 \
                   0000 \
                   000 \
                   0"
# … is equivalent to:
iScan myif 12 –si { 0b0000
                    0000
                    000
                    0 }
```

Rules

a) **scanInterface_name** prefixed by *effPrefix* shall reference a client **scanInterface_def** in the target module.

b) If the **'-chain'** option is provided, **chain_id** shall reference a valid sub-chain inside the parallel ScanInterface **scanInterface_name**. The scan data provided will be used for the specified chain. **iScan –chain** statements that reference the same ScanInterface may have different lengths.

c) The length of scan data described by both the si and so **iscan_data** shall not exceed the length specified by **length**. **iscan_data** shall be subject to the PDL/ICL padding rules if its length is smaller than length. The **length** argument shall match the length of the scan chain referenced by the **scanInterface_name** at the time the iScan is executed. If the **-si** argument is not provided its default value is 0b0, if the **-so** argument is not provided its default value is 0bX, with padding extending the value up to **length**.

d) Scan data is entered with the LSB as the rightmost bit that is physically closest to the output scan port. Therefore, scan data always shifts visually from left-to-right.

e) Scan data shall be maintained by controlling state and expected state models (such as **iWrite/iRead**). The data provided with the option **–si** shall be used for the controlling state, the data provided after the option **–so** shall be the expected data (for the expected state) to be compared to the scan-out data.

f) Any **iScan** operation not supplying the **'-stable'** option shall result in the black-box module being shifted exactly once (and no more). During the next iApply, another **iScan** operation must follow in order to make sure that shifting is possible again when accessing the black-box module.

g) An **iScan** operation with the **'-stable'** option shall be capable of being repeated without changing the length of the black-boxed serial access network.

h) If the option **'-ir'** is specified, the scan operation shall be done in the SHIFT_IR state of an IEEE 1149.1 AccessLink. An error shall be generated if **scanInterface_name** is not a client TAP interface.

i) After **iReset**, a stable state of known length shall be assumed from the DefaultLoadValue ICL statement.

Explanations

Rule f) is essential to force black-box iScan accesses that are not repeatable (perhaps because they alter the length of the scan chain in the black box) to be ultimately followed by an iScan access that leaves the black box in a stable state (i.e., one in which that last iScan may be repeated). This could result in a series of user-specified iScan operations that must be performed on successive iApply events until stability is restored.

Rule h) covers a special case for iScan access at the device or client TAP level. Although the typical IEEE 1687 serial access network is associated with the "DR" side of the TAP state machine, it is possible that an iScan command may need to be directed to the "IR" side of a TAP state machine (to select a different network via a different TAP instruction, for example). The "-ir" option for the iScan command is used to indicate such situations. The same rules about stable states apply for this case.

7.9.13 iOverrideScanInterface command

The purpose of the iOverrideScanInterface command is to impose user-specified behavior on three aspects of the operation of a ScanInterface: whether or not it allows the capture of new data into the scan registers connected to that interface, whether or not it allows the update of new data shifted into the scan registers connected to that interface, and whether or not it participates in scan broadcasting. The retargeting software is responsible for making these impositions by justifying the associated control signals through the network. Those control signals obey specific ICL description requirements detailed in the rules in this subclause, and are asserted or deasserted based on the options specified in this command. If those control signals are not actually present in the ICL description, an error condition results.

Grammar

```
ioverridescaninterface_def : 'iOverrideScanInterface' WS scanInterfaceRef_list
                             (WS '-capture' WS ('on'|'off'))?
                             (WS '-update' WS ('on'|'off'))?
                             (WS '-broadcast' WS ('on'|'off'))? ;
scanInterfaceRef_list : scanInterfaceRef (WS scanInterfaceRef)*;
scanInterfaceRef : (instancePath DOT)?  scanInterface_name;
```

Examples

```
# disable capture on scan interface client1 of instance blocka_i1:
  iOverrideScanInterface blocka_i1.client1 -capture off ;
# disable update on two scan interfaces:
  iOverrideScanInterface block2_i1.c2 block2_I1.c3 -update off ;
# broadcast u1.target to u2.shadow1 and u3.shadow2:
  iOverrideScanInterface u1.target u2.shadow1 u3.shadow2 -broadcast on ;
# broadcast data written to "target" (the ScanInterface on the active chain)
# to the scan registers behind u2.shadow1 and u3.shadow2:
  iWrite u1.reg1 0xABCD; iApply ;
# broadcast data scanned into u1.target to the scan registers behind
# u2.shadow1 and u3.shadow2:
  iScan  u1.target 8 -si 0b00001000 -so 0b001xxxxx; iApply ;
```

Rules

a) If the "-capture" option is used, the **captureEnPort_name** associated with each **scanInterface_name** specified shall have a **data_signal** as its source in the ICL.

b) When "-capture off" is specified, the **data_signal** which sources the **captureEnPort_name** associated with each **scanInterface_name** specified shall be justified to 0; otherwise, that **data_signal** signal is justified to 1.

c) The elements uniquely observable though the **scanRegister_captureSource** of a **scanRegister_def name** that has been disabled from capturing shall not be the target of an iRead command.

d) If the "-update" option is used, the **updateEnPort_name** associated with each **scanInterface_name** specified shall have a data_signal as its source in the ICL.

e) When "-update off" is specified, the **data_signal** which sources the **updateEnPort_name** associated with each **scanInterface_name** specified shall be justified to 0; otherwise, that **data_signal** signal is justified to 1.

f) The elements uniquely in the fan-out cone of a **scanRegister_def name** that has been disabled from updating shall not be the target of an iWrite command.

g) If the"-broadcast" option is used, each **scanInterfaceRef** shall be identified in the **AllowBroadcastOnScanInterface** list in the ICL for the **instance_def** with which it is associated.

h) When "-broadcast on" is specified and any **scanRegister_name** behind the first **scanInterfaceRef** (i.e., the target) is to be placed on the active scan chain, the scan registers behind other elements of the **scanInterfaceRef** shall receive a copy of the data scanned in to the first **scanInterfaceRef and** scanInterfaceRef shall be activated (i.e., its associated **selectPort_name** shall be justified to 1).

i) If the"-broadcast" option is used, and the first **scanInterfaceRef** is on the active scan chain, the length of the scan chain behind the first scanInterfaceRef (i.e., the target) shall be greater than or equal to the length of the scan chains behind the other **scanInterfaceRef_list**.

j) Each of the –capture, –update, and –broadcast options shall remain in effect for each scanInterfaceRef until another –iOverrideScanInterface command for a scanInterfaceRef is executed with that option.

k) The iOverrideScanInterface statement, when used for broadcasting, shall contain only a list of ScanInterfaces having ScanInPort sharing a common source. The path between the common source and the ScanInPort may contain scan multiplexers properly selected to point to a common fan-out point in the scan chain.

l) The number of bits in the value argument (**pdl_number | enum_name**) of an iWrite command whose object argument (**reg_or_port**) is a scan register in a broadcast group shall match the number of bits in the longest scan register in that group.

m) The retargeting software shall not be required to enable the broadcast mode, but the user may issue the "iOverrideScanInterface –broadcast on" command to explicitly instruct the retargeting software to utilize broadcast scan mode.

n) Once set by the user, the broadcast scan mode shall be respected by the retargeting software in order to properly assign the values of the broadcasted bits during each scan load.

o) Once set by the user, the broadcast mode shall remain in effect until the user explicitly disables it via the "iOverrideScanInterface –broadcast off" PDL command. An error shall result if the associated configuration bits need to be set to different values to satisfy other iWrite/iRead commands but the broadcast scan mode has not been disabled by the user.

p) If the specified **scanInterfaceRef** has multiple scan chains associated with it, the action of the **iOverrideScanInterface** command shall apply to all of the chains.

Explanations

Rule a) requires that the ScanInterface on which the user would like to disable capture does indeed include a structure described by the ICL that is capable of having its CaptureEnPort deactivated; i.e., that port is driven by a data_signal that effectively gates the associated captureEn when justified to a 0, as opposed to, say, a captureEn_signal, which cannot. Rule b) instructs the retargeting software to control the CaptureEnPort appropriately to enable or disable the capture. Rule c) imposes the restriction that the capture source of a scan register, or anything else that is solely observable via that capture source, cannot

be the object of an iRead action if the capture capability has been disabled on the scan interface associated with that scan register.

Rules d), e), and f) are symmetrical with rules a), b), and c), but apply to the UpdateEnPort and the data being driven from the scan register instead of the data being captured into it.

Rule g) requires that the ScanInterfaces for which the user would like to perform broadcasting are indeed allowed by the ICL to do so. Rule h) indicates that all the ScanInterfaces participating in a broadcast write operation will have their ScanInterfaces selected when the first ScanInterface listed (known as the "target") is on the active scan chain. Rule i) is necessary to prevent unknown data from being shifted downstream from the broadcast group.

Rule j) enforces stickiness of iOverrideScanInterface options: they remain in effect for the named ScanInterfaces until explicitly changed by the PDL stream. This keeps the behavior of the scan interface under the user's control. Rule k) enforces that the broadcast originates from a single source to all the interfaces involved.

7.9.14 iApply command

The purpose of the **`iApply`** command is to apply the model state previously defined by **`iWrite`**, **`iRead`** and/or **`iScan`** commands to the hardware.

The **`iApply`** command generates a sequence of PI/PO and CSU operations (to IR and DR scan chains) to update the hardware state (i.e., logic state) such that the writes reflect the controlling state of the model by the end of the **`iApply`** time frame and to observe the hardware state of the queued **`iReads`** once before the end of the **`iApply`** time frame, in order to compare it to the expected state of the model .

Grammar

```
iapply_def : 'iApply' (WS '-together')?;
```

Examples

```
iWrite myport 0xf0;     # same as 0b0011110000, assuming 10-bit myport
iWrite myport(9:8) 3;   # myport = 0b1111110000
iApply;                 # apply queued updates
iWrite myport(3) 1;     # Changes myport to 0x3f8
iApply;                 # Writes 0x3f8 to myport(9:0)
                        # if myport can only be accessed as a whole
```

Rules

a) All scan registers that were referenced by an **`iWrite`** since the last **`iApply`**, as well as scan registers that control a **`DataRegister`** or a **`DataInPort`** referenced by such an **`iWrite,`** shall be subject to at least one **`UpdateEn`** cycle during the **`iApply`**.

b) A callback register access may not be combined with any other register access in the same iApply group.

c) If the '-together' option is specified, all iWrites specified in the iApply group shall be updated on the same TCK cycle. It is an error if it is impossible for the retargeting tool to update all iWrites on the same TCK cycle.

d) If an iApply with the '-together' option is subject to merging with iMerge, the retargeting tool shall respect that the '–together' property is still enforced after the merging.

Explanations

Rule c) is often associated with the **ApplyTogether** property of an ICL **Alias** statement. The existence and satisfaction of that ICL property generally allows the –together option to iApply to also be satisfied, but there may also be situations where the ICL property is not present but the PDL writer wishes to convey intent to the retargeting tool; that is the purpose of this option.

7.9.15 iClock command

The purpose of the **iClock** command is to specify that a system clock is to be running. The command verifies that the clock port has a valid controlled source and computes the cumulative multiplication factor and division ratio along the clock path. Only clocks that successfully passed this **iClock** check can be referenced by the -sck option in the **iRunLoop** command.

Grammar

```
iclock_def : 'iClock' sysClock;
```

sysClock is checked for a sensitized clock path that allows **sysClock** to be operated from the target module boundary.

Examples

ICL:

```
ToClockPort MySclk {
   Source ClockIn;
   FreqMultiplier 10;
}
```

PDL:

```
iClock MySclk
```

Rules

a) The **sysClock** shall be a valid ICL **ToClockPort** or **ClockPort** declaration.

b) An error shall be issued if no sensitized path exists from a valid clock source to the specified clock port. A valid clock source is a ClockPort of the target module (reference source) or a ToClockPort with a period property (embedded oscillator).

7.9.16 iClockOverride command

The purpose of the **iClockOverride** command is to override the definition of system clock when it is generated on-chip. The freqMultiplier and freqDivider options are used to scale the frequency of the clock with respect to the sourceClock pin or port, which is required to be a clock_signal. Any iClock/iRunLoop command used after the iClockOverride command uses the overridden values when tracing the clock.

Grammar

```
iclock_override_def : 'iClockOverride' WS sysClock
                      (WS '-source' WS sourceClock)?
                      (WS '-freqMultiplier' WS pdl_number)?
                      (WS '-freqDivider' WS pdl_number)? ;
```

Rules

a) **sysClock** shall be a valid ICL **ToClockPort** declaration.

b) If specified, **sourceClock** shall be a valid ICL **clockPort** or **ToClockPort** declaration.

c) If specified, **'-freqMultiplier' pdl_number** shall override the **freqMultiplier** attribute of **sysClock.** If not specified, the multiplier shall be as defined in the ICL.

d) If specified, **'-freqDivider' pdl_number** shall override the **freqDivider** attribute of **sysClock.** If not specified, the divider shall be as defined in the ICL.

e) If specified, **sourceClock shall overr**ide the **Source** att**ribute of sysClock**. If not **specified**, the source clock shall be as defined in the ICL.

Examples

```
iClock MySclk;                          # Uses ICL default
iRunLoop 1000 –sck MySclk;              # 1000 clocks based on default scaling
iCall PLLChange 5;                      # Calls an iProc that updates mult
iClockOverride MyToSclk -freqMultiplier 5; # Override ICL ...
                                        # ... should happen inside iProc above
iRunLoop 1000 –sck MySclk;              # still based on default scaling
iClock MySclk;                          # Override taken into account
iRunLoop 1000 –sck MySclk;              # 1000 clocks based on updated mult
```

7.9.17 iRunLoop command

The purpose of the **iRunLoop** command is to generate a specified minimum number of idle clock cycles. While all clocks are always assumed to be free running, an optional parameter allows to specify which reference clock is used for counting idle cycles. If the clock is not defined by the optional parameter, it is assumed to be the test clock (TCK). If the –time option is used, the number of equivalent clock cycles is calculated that will consume at least that amount of time. The port parameter name is subject to the *effPrefix*.

A typical application for the **iRunLoop** command is to generate clock cycles for an instrument built-in self-test (BIST) procedure. If multiple commands are specified, additional clock cycles will be generated.

Grammar

```
irunloop_def : 'iRunLoop' ( cycleCount ( '-tck' | '-sck' port )? |
               '-time' tvalue);
```

The **cycleCount** parameter defines the minimum number of clock cycles needed. The **cycleCount** refers either to the test clock that is indicated by the (default) **-tck** parameter or to a system clock defined by the **port** parameter if the **-sck** option is selected. In place of a **cycleCount**, **iRunLoop** also allows specification of a time delay that will be elapsed with all clocks free-running.

During pattern retargeting, the **cycleCount** parameter will be adjusted if the referenced clock port definition includes a **FreqMultiplier** or **FreqDivider** parameter in its transitive fan-in. If included, the period parameter should be divided by the **FreqMultiplier** parameters and multiplied by the **FreqDivider** parameters along the active clock path when retargeting the PDL **iRunLoop** command for the device.

For example, if the original **cycleCount** parameter is 2 000 and the ICL Clock **FreqDivider** parameter is 10, then the **iRunLoop** retargeted **cycleCount** is 20 000. During the device-level test, the number of

clock cycles that should be applied to the device **sysClock** port is 20 000, which will result in 2 000 clock cycles at the instrument boundary.

Examples

ICL:

```
ToClockPort MySclk2 {
   Source ClockIn;
   FreqDivider 10;
}
```

PDL:

```
iRunLoop 2000 -sck MySclk2; # generate 2,000 sck clocks
```

Retargeted PDL:

```
iRunLoop 20000 -sck ClockIn; # generate 20,000 sck clocks
```

Rules

a) If included, **sysClock** prefixed by *effPrefix* shall be a valid reference to an ICL ClockPort.

b) If **sysClock** is specified, the **sysClock** port shall be cycled at least **cycleCount** times during the **iRunLoop**. TCK may or may not be toggling.

c) If **cycleCount** is specified with **–tck** or without any option, TCK shall be toggling at least **cycleCount** times.

d) If **tvalue** is specified with the –time option, iRunLoop shall elapse at least the specified time. During this time all clocks shall be free-running.

e) It shall be an error if iRead or iWrite commands have not been applied when the iRunLoop command is encountered.

f) Until such time as the iRunLoop duration has elapsed, the subsequent PDL command shall not be executed.

g) When **-sck port** is used, the specified **port** shall have been previously specified with an iClock command so that its source has been checked and its cumulative multiplication and division value computed.

Explanations

Rule d) allows the specification of a delay in terms of an absolute time value rather than a number of clock cycles. The duration of the delay is simply converted into a minimum number of clock cycles based on the defined period of the named clock (either **sysClock port** or TCK).

Rule e) forces there to be an iApply statement prior to the initiation of an iRunLoop so that the controlling and expected states of the device are known. Although it is an action command, the iRunLoop command does not perform an implicit iApply to execute any iWrite or iRead commands that have been queued since the previous iApply.

7.9.18 iMerge command

The purpose of the **iMerge** command is to expose potential parallelism in iProc execution. The concurrency model for **iMerge** is pessimistic, meaning that it allows a trivial conflict resolution. Whenever access to two requested resources conflicts (using the iTake command), conflict resolution is allowed to use

a trivial serialization of the complete iProcs. This results in predictability and correctness for the user and at the same time allows a high degree of freedom for the tool vendor (ranging from solutions that implement iMerge as a no-op (meaning everything is serialized) to highly sophisticated scheduling algorithms).

iMerge indicates that a set of **iCall** commands may be first flattened and then merged (only **iCall** and **iNote** are allowed in an **iMerge** block of an **iProc**). A flattened **iCall** has all its Prefixes, Argument substitutions, and iCall statements recursively substituted for the literal equivalent (see iCall semantic rules). Hence it consists of a flat, linear sequence of just the basic setup/action commands with literal names and numbers. Note that the flattening approach is just a model to describe the semantic rules for merging. Actual implementation of merging may use different approaches, especially when considering merging for PDL-1.

The choice of which **iCalls** are indeed executed in parallel (and in which order) is completely up to the tool. During merging, no assumption on the persistence of certain **iWrite**/**iRead** assignments can be made (an **iWrite** value could be altered immediately by a merged-in **iProc**). This freedom can be restricted by putting a reservation on the **iWrite**/**iRead** target using the command **iTake**. If a reservation exists, a tool shall enforce that the controlling and expected states of resources that have been reserved are not altered at any point by **iCalls** other than the one owning the Register in between **iTake** and **iRelease**. If the user wishes to release controlling and expected state bits reserved with **iTake** (such that they can be used by another parallel **iCall),** the user shall explicitly **iRelease** the resource. Idle loops specified with **iRunLoop** can be merged but are required to have at least the specified number of idle cycles.

Note that the number of CSU and functional clock cycles to execute the commands may change compared to serial execution, due to the other **iProcs** executed in parallel.

Recursive merging is allowed (e.g., **iCall** in an **iMerge** calls another **iProc** that itself contains **iMerge**) since merging works on flattened **iProc**s. All semantic rules hold even when merging an already merged **iProc**.

Grammar

```
imerge_def          : 'iMerge' ( '-begin' | '-end' );
```

Examples

```
iMerge -begin
iCall proc1
iCall proc2
iMerge -end
```

Rules

a) Only **iCall** and **iNote** commands are allowed after **iMerge -begin** until **iMerge -end**.

b) All **iNote** commands will be executed first.

c) Each **iCall** shall be flattened independently from the other **iCalls**.

d) The **iApply** queue shall be empty upon **iMerge -begin** and all outstanding updates and observations shall have been applied to the hardware.

e) All merged **iCall**s share one common controlling state Model and one common expected state Model. The state of the model depends on the (resulting) merged sequence.

f) Inside **iMerge -begin/-end**, an **iCall** that leaves entries on the **iApply** queue (i.e., has **iWrite** /**iRead**/**iScan** before return that were not applied with **iApply**) shall not be allowed.

g) After **iMerge -end**, the controlling state model and expected state model shall correspond to the state implied by the merged sequence of CSU operations (iApply-timeframes).

h) When interleaving **iApply** groups during **iMerge** (the merging process), no **iCall** may interfere with the controlling state, expected state, **iScan** sequence, or clock definitions of registers or ports that were taken by **iTake** in another **iCall**. Serialization is a valid resolution of these conflicts.

i) The order of the **iApply** groups that is defined by each **iCall** shall be preserved by the merging.

j) Several **iRunLoop** operations may be merged but shall satisfy the specified minimum cycle count of each merged **iRunLoop**.

k) There shall be no iReset commands within any of the procedures being called within an iMerge block; the presence of an iReset command constitutes an error.

7.9.19 iTake command

The purpose of the **iTake** command is to take ownership of an ICL resource. There are two meanings for taking ownership; the distinction is based on the context of the PDL.

In the context of a single PDL command stream (i.e., one not between **iMerge –begin** and **iMerge –end** commands), an **iTake** command followed by an iWrite command to the associated taken resource is the mechanism by which the PDL author constrains the retargeting software. A **data_signal** that is taken by an **iTake** command becomes unchangeable by the retargeting software after the **iApply** that executes an **iWrite** command whose object is that taken **data_signal**. Prior to being taken and written, the retargeting software is free to change the value. In fact, even after the **iTake** (but prior to the application of the **iWrite**), the retargeting software is free to change the value. Once written by an **iWrite** command in the PDL, however, the constraint can be lifted only by an **iRelease** command for that resource.

In the context of a PDL **iCall** command that is to be merged with other **iCall commands**, an **iTake** command is the mechanism that reserves the resource for exclusive use by the **iProc** containing the **iTake**. Once an **iProc** has taken a resource, values assigned to the resource with **iWrite** or **iRead** are not allowed to be altered by other **iCall**s until the resource is explicitly released with **iRelease**. As in the case of the single PDL command stream, the retargeting software is also prohibited from modifying the taken resource until it is released.

Grammar

```
itake_def           : 'iTake' WS reg_port_or_instance ;
```

Examples

```
iTake RegA
iTake InstA
```

Rules

a) **reg_port_or_instance** shall reference a register, a port, or an instance definition.

b) If **reg_port_or_instance** references an instance, the control of all of the registers, ports, and instances in **reg_port_or_instance** shall be taken.

c) Starting with the **iTake** of **reg_port_or_instance**, any value assigned to **reg_port_or_instance** using **iRead** or **iWrite** or **iScan** shall not be altered by merging another **iProc**. If **reg_port_or_instance** is a **ClockPort** or **ToClockPort**, the frequency of the port shall not be altered by merged PDL.

d) Starting with the **iTake** of **reg_port_or_instance**, any value assigned to **reg_port_or_instance** using **iWrite** shall not be altered by the retargeting software. The retargeting software shall be permitted to change the value of a taken resource before it is the object of an applied **iWrite** command or after that resource is released by an **iRelease** command.

e) **iProcs** invoked with **iCall** shall inherit ownership of all resources taken by the calling **iProc**. Resources taken by the callee shall remain taken until returning to the caller.

f) The reservation on **reg_port_or_instance** shall persist in the flattened result of the **iMerge -begin** … **iMerge -end** block (to enable recursive merging).

Explanations

Rule d) permits the retargeting software to alter a taken resource as long as that resource has not been written to in the PDL command stream. Once the **iApply** that executes an **iWrite** of that resource occurs, the retargeter is prohibited from modifying the resource until it is released. In general, **iTake** commands must precede **iWrite** commands that reserve the resource, unless both the **iTake** and **iWrite** commands are in the same **iApply** group.

7.9.20 iRelease command

iRelease allows the release of ownership of a resource that has been reserved with **iTake**. After **iRelease** of the resource, the resource may be freely altered upon merging with other **iProc**s.

Grammar

```
irelease_def           : 'iRelease' WS reg_port_or_instance ;
```

Examples

```
iRelease RegA
iRelease InstA
```

Rules

a) **reg_port_or_instance** shall reference a register, a port, or an instance definition.

b) If **reg_port_or_instance** references an instance, all of the registers, ports, and instances in **reg_port_or_instance** shall be released.

c) Starting with the **iRelease** of **reg_port_or_instance**, any value assigned to **reg_port_or_instance** using **iRead** or **iWrite** shall be permitted to be altered upon merging with another **iProc.** If **reg_port_or_instance** is a **ClockPort** or **ToClockPort**, the frequency of the port shall be permitted to be altered by merged PDL.

d) Starting with the iRelease of reg_port_or_instance, any value assigned to reg_port_or_instance using iWrite shall be permitted to be altered by the retargeting software.

e) Resources released in **iProcs** invoked with **iCall** shall release the resource for the calling **iProc** as well.

f) An **iRelease** command will be respected only if it is issued by the same procedure that issued the **iTake** of the named resource.

7.9.21 iNote command

Similar to attributes in ICL, the **iNote** command is a mechanism to pass free-form information to the runtime environment. The quoted information is prefaced by one of two tags to indicate that it is either a comment or a status message. Though no strict checking of these tags is mandated, the general guideline is to use a comment tag for general information that will be passed along with the formatted patterns, and a status tag for milestones or events that the system can use to report progress. No text passed to **iNote** may result in an error during processing of the **iProc** (since it may contain information for software downstream of the retargeting tool), only warnings are allowed.

For Tcl compatibility, all double quotes **""**, braces **{}**, or brackets **[]** are required to be matched pairs.

Grammar

```
inote_def          : 'iNote' WS ( '-comment' | '-status' ) WS QUOTED;
```

Examples

```
iNote –status "INFO: PLLs started"
iNote –comment "test conditions should be stable now"
```

Rules

a) **iNote** shall not result in an error condition. Malformed parameters shall be ignored or may result in warning messages.

b) **QUOTED text** shall be carried over to any textual representations of patterns. **QUOTED text** shall immediately succeed the textual representation of the preceding action command (e.g., iApply/iRunLoop/iNote).

Recommendations

c) The iNote -comment command should be used to provide detailed comments in any PDL that may be used to generate test vectors.

d) The iNote -status command should be used to denote important milestones in the test flow.

7.9.22 iState command

The purpose of the **iState** command is to document the current state of a network. This command causes no transactions with the device; rather, it serves as a method to communicate the state of the network that resulted from all of the previous transactions with the device. The purpose of this documentation and communication is to facilitate the transfer of control of the on-chip network from one external controller to another. For example, a device connected to an Ethernet-to-JTAG converter on the Internet could be used by multiple users, each of whom has controlling software that tracks the controlling state and expected state of the network. When one user relinquishes ownership of the converter to another, user, the iState command serves to communicate the state of the device. Without this command, the incoming controller would not know the state of the network (and thus the lengths of the TDRs) and would therefore be unable to operate the network without resetting it first.

The arguments to the iState command consist of a register name or the name of a SelectPort or DataInPort on the top-level module and an associated value whose lengths must match. Unlike the iWrite command, internal ports cannot be used as arguments to iState (since they are not state points), but top-level pins are valid state points. The references to ICL register and pin names are subject to the effective prefix (*effPrefix*) as mentioned before.

Optionally, flags may be set to describe the nature of the values associated with the register or pin names; these flags are mutually exclusive. LastWrittenValue is the default and indicates that the value is what was last written to the register or pin. LastReadValue indicates that the value is what was last read from the register or pin (by an iRead command). LastMiscompareValue indicates that the value is what was last expected from the register or pin when it was read (by an iRead command with an expected data field).

One possible implementation of the primary use model for this command consists of the following two steps:

1) The current controller writes a text file of iState commands for essential state elements in the network. The set of elements included in this file certainly include all registers that determine the length of the TDRs, and may include all registers and top-level DataInPorts and SelectPorts. There is no PDL command that causes this text file of iState commands to be written; both the contents of the file and the decision of when to write it are left to the software controlling the network.

2) The incoming controller reads this text file of iState commands and adjusts its map of the network to be consistent with the values specified for the elements listed. At the conclusion of reading all the iState commands, the controller will have up-to-date controlling state and expected state models and may then proceed to process PDL commands.

Note again that the incoming controller is not issuing any transactions with the device as it reads the iState commands; it is merely setting values in internal database models of the state of the network so that it can subsequently operate it. There are thus no iApply commands associated with an iState file, and it is considered a setup command rather than an action command.

The use model for the other variations (enabled by LastReadValue and LastMiscompareValue) is to communicate the history/result of previous transactions between tools. This is of importance with respect to the PDL-1 commands iGetReadData and iGetMiscompares.

Grammar

```
istate_def : 'iState' WS reg_or_port WS pdl_number (WS '-LastWrittenValue' |
             WS '-LastReadValue' | WS '-LastMiscompareValue')? ;
```

The **reg_or_port** (prefixed by *effPrefix*) refers to either a scan or data register or a top-level DataInPort/SelectPort defined in ICL.

The action taken by this command is to modify the controlling and expected state models for the register or pin (but not the actual device). The model bits defined by the **reg_or_port** parameter are updated with the **pdl_number** data according to the flag that is active (LastWrittenValue by default).

It is essential that **pdl_number** have a well-defined size (subject to padding rules), and that its size match the size of **reg_or_port**.

Examples

All registers (or aliases of registers) are defined as length 10.

```
iState myreg4 0b1101110011;
iState SIBS 0b0010011011;
iState inst1.myreg3 0xffd;           # Error, would discard bits
iState a.b.regz 0b1001010000; # defaults to LastWrittenValue;
iState a.b.regz 0b1001010000 –LastWrittenValue; # same as previous
iState a.b.regz 0b1001010000 –LastReadValue;
iState a.b.regz 0b0001000000 –LastMiscompareValue;
```

Rules

a) The **reg_or_port** parameter prefixed by *effPrefix* shall be a valid register or top-level DataInPort or SelectPort name.

b) The size of **pdl_number** shall be made to match the size of **reg_or_port**: an undersized **pdl_number** shall be extended according to rule b) in 7.8 to match the width of the register or port that is being specified. It shall be considered an error if the oversized number contains more significant bits than are available in reg_or_port.

c) **pdl_number** may contain any unknown (**x**) values, which serve merely as placeholders and indicate that the network state for the corresponding location has never been written (for **LastWrittenValue**), read (for **LastReadValue**), or compared (for **LastMiscompareValue**).

d) The bits of **pdl_number** overwrite the controlling and expected state models of **reg_or_port** when the (default) LastWrittenValue flag is active.

e) If **reg_or_port** appears multiple times in the file of iState commands, only the last value is used.

f) If no option is set or the **-LastWrittenValue** is supplied, the tool shall update its internal logic state for **reg_or_port** with **pdl_number**.

g) If **-LastReadValue** is supplied, the tool shall update its internal representation of previously read register values for **reg_or_port** with **pdl_number**.

h) If **-LastMiscompareValue** is supplied, the tool shall update its internal representation of previously miscompared read register values for **reg_or_port** with **pdl_number**.

i) No actual device operations shall be executed or queued as a result of an iState command.

8. Procedural Description Language: level-1 (Tcl)

8.1 Purpose

While PDL level-0 (described in Clause 7) is sufficient to represent all activity on the instrument access network, it is a *static* language (i.e., it lacks elements common to most programming languages like looping, conditional branching, and many other coding features that are characteristic of an *interactive* language). For instruments whose functionality requires a more complex representation or in environments where interaction with the device determines the flow of the test, a high-level programming language is essential. To those ends, Tcl (Tool Command Language) (see Osterhout and Jones [B6]) has been selected as the language because it is already widely used in electronic design automation (EDA) tools and is therefore familiar to the design community. Additionally, Tcl is not only a full-featured programming language, but also supports the concept of command extensions, which fits quite nicely with the use model for PDL.

8.2 Tcl command extensions

Tcl can be thought of as a language of command keywords with arguments. Many commands (e.g., for, if, while, set) are prespecified in the language, but Tcl allows the user to create new commands (referred to as *command extensions*) that perform custom functions. In the context of PDL, the custom Tcl commands required to interact with a device-under-test include the following:

— Reading data from registers or ports

— Flagging when miscompares have occurred

— Flagging when an unexpected condition has occurred during the execution of a procedure

There are four command extensions described in this clause to perform those functions.

It is also important to note that the entirety of PDL level-0 (from Clause 7) has been carefully architected to be Tcl-compatible. In other words, every PDL command (both level-0 and level-1) is a syntactically correct Tcl command extension, meaning that any PDL may be executed by a Tcl interpreter incorporating the PDL Tcl extensions.

8.3 PDL level-1 overview

In order to enable PDL level-1 (Tcl) procedures to be portable across vendors and tools, it is minimally necessary to standardize mechanisms to transfer data and status from the device into Tcl variables. The four PDL level-1 commands to do this are shown in Table 10.

Table 10—PDL level-1 commands

Command	Arguments and options	Purpose
`iGetReadData`	`register | port | alias | ScanInterface [-bin | -hex | -dec]`	Return (as a string in the specified unsized number format) the value from the most recently applied iRead operation on register or output port (or an alias consisting of either or both). May contain x-values.
`iGetMiscompares`	`register | port | alias | ScanInterface [-bin | -hex | -dec]`	Return (as a string in the specified unsized number format) the XOR of the value from the most recently applied iRead operation on a register or output port (or an alias consisting of either or both) and the value expected for that iRead operation. May contain x-values.
`iGetStatus`	`[-clear]`	Return the decimal number of iApply miscompares that have occurred since the last time that iGetStatus –clear was issued. Clear the count afterwards if directed.
`iSetFail`	`message [-quit]`	Return the message string to the controlling program to indicate an unexpected condition, with the optional directive to abort execution.

8.4 PDL level-1 commands

The details of the four PDL level-1 commands are presented in the following subclauses.

8.4.1 iGetReadData command

The purpose of the `iGetReadData` command is to return the value from the most recently applied iRead operation. The permissible target objects of the iGetReadData are the same as for the iRead command, namely registers, output ports, and aliases composed of registers or output ports. Range modifiers are allowed. Note that this iRead is not necessarily in the most recent iApply group; it is simply the last time that an iRead was performed on the named register, output port, or alias; the tracking of the most recent iRead is separate for each index of the register or port.

The returned value represents the data captured in the register or observed at the output port due to the execution of the iRead command. Any subsequent operation performed on the register or port not initiated

by iRead commands, for example a shift initiated by the application tool, does not constitute a read operation that is tracked for the purpose of reporting the value through the iGetReadData command.

The return value is a string. The string represents an unsized number and may be in binary, hexadecimal, or decimal format as specified by the user; binary is the default. Specifying binary format will prepend the characters “0b” to the read data value string, hexadecimal format will prepend “0x” to the value string, and decimal format will simply return the value string. This treatment will allow a consuming Tcl program to correctly evaluate these strings as numbers in expressions.

The string returned by the iGetReadData command may contain x-characters. The 'x' is used to denote the unknown value in the case where there were no iRead commands on the given index of the register or output port. For aliases, the referenced register or port indices will be considered, not the iRead of the alias itself. In the binary format, 'x' is used on a by-bit basis. In the hexadecimal format, an upper case 'X' is used if all 4 bits of the binary representation of the respective hexadecimal character in the hexadecimal number string are unknown; a lower case 'x' is used to denote that some, but not all bits of the binary representation of the hexadecimal character are unknown. If there are any unknown bits and the decimal format is requested, the return string defaults to 'X'.

The return value is required to match the size of the element (i.e., register or output port) named in the command. Note that the iGetReadData command returns the data seen by the iRead command independent of whether or not an expected value was provided to the iRead command.

Grammar

```
iget_read_data_def : ‘iGetReadData’ WS (reg_or_port
                          | scanInterface_name (WS '-chain' WS chain_id)? )
                     ( WS format )? ;
format  : '-dec' | ‘-bin’ | ‘-hex’ ;
```

Generic example

```
 iGetReadData Reg_or_port_or_Alias_or_ScanInterface_name [-chain
chain_id] [-dec | -bin | -hex ]
```

Examples

Example 1:

```
ICL:
  ScanRegister RegA [5:0] { … };
  ScanRegister RegB [5:0] { … };
  Alias lbits = RegA[1:0], RegB[1:0] ;

PDL:
   iRead RegA
   iRead RegB(3:0) 0xf
   iApply

Tcl:
  iGetReadData RegA –hex ;       # returns a string representing
                                 # a hexadecimal number
  iGetReadData RegB(3:1) ;       # returns a string of 3 bits in binary
  iGetReadData lbits –bin ;      # returns a string of 4 bits in binary,
                                 # none of which are ‘x’, since all bits
                                 # this alias refers to were iRead
```

Example 2:

```
ICL:
  ScanRegister RegA [5:0] { … };
  ScanRegister RegB [5:0] { … };
  Alias lbits = RegA[1:0], RegB[1:0] ;

PDL:
   iRead RegA
   iRead RegB(2:1) 0xf ;        # the only difference to Example 1
   iApply

Tcl:
  iGetReadData RegA –hex ;      # returns a string representing
                                # a hexadecimal number
  iGetReadData RegB(3:1) ;      # returns a string of 3 bits in binary
                                # with the MSB being 'x'.
                                # example: "0bx11""
  iGetReadData lbits –bin ;     # returns a string of 4 bits in binary,
                                # with the bit position representing
                                # RegB[0] being 'x'
                                # example: "0b101x"
```

Example 3:

```
ICL:
  ScanRegister RegA [5:0] ;
  ScanRegister RegB [5:0] ;
  Alias lbits = RegA[1:0], RegB[1:0] ;

PDL:
   iRead RegA
   iRead lbits 0b110x ;         # the only difference to Example 1
   iApply

Tcl:
  iGetReadData RegA –hex ;      # returns for example "0xXx"
  iGetReadData RegB(3:1) ;      # returns a string of 3 bits in binary
                                # example: "0bxx1"
  iGetReadData lbits –bin ;     # returns a string of 4 bits in binary,
                                # the bit position representing
                                # RegB(0) has valid data,
                                # since it was read (but not compared)
                                # example: "0b0011"
```

Rules

a) Specifying "-bin" for binary format shall cause the return value to be in binary with "0b" prefix.

b) Specifying "-hex" for hexadecimal format shall cause the return value to be in hexadecimal base with 0x prefix.

c) Specifying "-dec" for decimal format shall cause the return value to be in decimal without prefix.

d) If no format is specified in the iGetReadData command, the format shall default to binary.

e) iGetReadData shall return a string whose length matches the length of the named object of **`reg_or_port`** in the respective format, plus 2 for the radix prefix if binary or hex. If a

scanInterface_name with optional **chain** was referenced rather than **reg_or_port**, iGetReadData shall return a string whose length matches the **length** supplied to the preceding iScan command for the ScanInterface and chain (plus the prefix).

f) Permissible objects for **reg_or_port** shall be the same as in the **iRead** command.

g) **scanInterface_name** prefixed by *effPrefix* shall reference a client **scanInterface_def** in the target module.

h) If the **'-chain'** option is provided, **chain_id** shall reference a valid sub-chain inside the parallel ScanInterface **scanInterface_name**.

i) If **reg_or_port** is observed multiple times during a sequence of CSU operations, only the first observation shall be returned.

j) If **reg_or_port** is observed one or multiple times during one or multiple **iReads** in one or multiple **iApply** groups, only the value of the last read for each index of the register or output port shall be returned.

k) In case the return value format is binary, and if an index in **reg_or_port** has never been read, the return value for this index shall be 'x'.

l) In case the return value format is hexadecimal, and if all four indices represented in one hexadecimal character in reg_or_port have never been read, the return value for this hexadecimal character shall be 'X'.

m) In case the return value format is hexadecimal, and if some, but not all four indices represented in one hexadecimal character in reg_or_port have never been read, the return value for this hexadecimal character shall be 'x'.

n) In case the return value format is decimal, and if any index in **reg_or_port** has never been read, the return value of **iGetReadData** shall be 'X'.

8.4.2 iGetMiscompares command

The purpose of the **iGetMiscompares** command is to return the bitwise exclusive-or of the value from the most recently applied iRead operation on the register or port named in the command and the value expected for that iRead operation. The return value is a string and may be in binary, hexadecimal, or decimal format as specified by the user; binary is the default. Specifying binary format will pre-pend the characters “0b” to the read data value string, hexadecimal format will prepend “0x” to the value string, and decimal format will simply return the value string. This treatment will allow a Tcl program that is parsing the return value from this command to correctly evaluate these strings as numbers in expressions, provided that there are no 'X' values.

The iGetMiscompares command returns 'X' string when there were no iRead commands on the given register or port element.

If there were miscompares, the comparison vector will have ones in the bit positions corresponding to the mismatches and zeroes in the bit positions that have not miscompared. The resulting vector will then be formatted into strings according to the option specified, with binary format used by default.

The return value is required to match the size of the specified register or port elements and reflect the exclusive-or of the value that was seen in response to the iRead command most recently issued on the register or port and the expected value specified for that same iRead. Note that this iRead is not necessarily in the most recent iApply group; it is simply the last time that an iRead was performed on the named register or port.

The use of alias names with the iGetMiscompares command operates as if the constituent elements of the alias had been first substituted and expanded into a series of iGetMiscompares commands.

Grammar

```
iget_miscompares_def : 'iGetMiscompares' (reg_or_port
                       | scanInterface_name (WS '-chain' WS chain_id)? )
                    ( WS format )? ;
format  : '-dec' | '-bin' | '-hex' ;
```

Generic example

```
 iGetMiscompares  Reg_or_port_or_Alias_or_ScanInterface  name  [-chain
chain_id] [-dec | -bin | -hex ]
```

Examples

```
iGetMiscompares RegA –hex
iGetMiscompares PinB;  # returns a string in binary format by default
# assume ICL contains "alias AliasRegA_PinB { RegA, PinB }"
iGetMiscompares AliasRegA_PinB –bin;
# same as: iGetMiscompares RegA –bin
#          iGetMiscompares PinB –bin
```

Rules

a) iGetMiscompares shall return a string whose length matches the length of the named object of **reg_or_port** in the respective format, plus 2 for the radix prefix. If a **scanInterface_name** with optional **chain** was referenced rather than **reg_or_port**, iGetMiscompares shall return a string whose length matches the **length** supplied to the preceding iScan command for the ScanInterface and chain (plus the prefix).

b) Permissible objects for reg_or_port shall be the same as in the iRead command.

c) **scanInterface_name** prefixed by *effPrefix* shall reference a client **scanInterface_def** in the target module.

d) If the **'-chain'** option is provided, **chain_id** shall reference a valid sub-chain inside the parallel ScanInterface **scanInterface_name**.

e) Each bit in the value shall be set according to the most recent observation and compare done on the bit: '1' if the bit miscompared to the expected value specified by the iRead, '0' if the bit did not miscompare and X if the bit was never observed.

f) Specifying "-bin" for binary format shall cause the return value to be prepended with "0b."

g) Specifying "-hex" for hexadecimal format shall cause the return value to be prepended with "0x."

h) If no format is specified in the **iGetMiscompares** command, the format of the string returned value shall default to binary and shall cause the return value to be prepended with "0b."

8.4.3 iGetStatus command

The **iGetStatus** command is used to obtain the number of failing comparisons (defined hereafter) since the last time that count was reset (by **iGetStatus –clear**) or from the beginning of the test session. A "failing comparison" refers to each binary digit of an **iRead** that has an expected value and that fails the comparison to that value. If there are multiple failing bits in a given **iRead**, or multiple failing **iReads** within an **iApply** group, each failing bit increments the number of failing comparisons.

The status is automatically reset to 0 at the start of the test session and increments whenever there is a miscompare on any bit. The current count is returned as a decimal number by the **iGetStatus** command. If the **-clear** option is specified, the current count is returned and then the status is reset to 0.

Grammar

```
iget_status_def : 'iGetStatus' ( '-clear' )? ;
```

Generic example

```
iGetStatus  [-clear]
```

Examples

```
iGetStatus -clear
iCall myProc
set num_fails [iGetStatus]
```

Rules

a) **iGetStatus** shall return a decimal number corresponding to the number of failing per-bit comparisons from **iRead** statements with expected values since the last time that an **iGetStatus -clear** command was issued or since the time the test session began if no **iGetStatus -clear** command has been issued.

b) **iGetStatus -clear** shall reset the number of failures to zero after returning the current value of the failure count.

c) The current value of the failure count shall be incremented by 1 when any bit in any iRead statement fails to match the expected value.

8.4.4 iSetFail command

The **iSetFail** command is used to indicate that an unexpected condition has occurred and pass a text message back to the controlling program, and optionally indicate that the test execution should cease. This is distinguished from a miscompare event, which is handled with the iGetStatus command.

Grammar

```
iset_fail_def : 'iSetFail' text_message ( '-quit' )? ;
text_message : string ;
```

Generic example

```
iSetFail  [-quit]
```

Examples

iSetFail "Error: unexpected result"
iSetFail "Warning: temperature at upper limit" -quit

Rules

a) **iSetFail -quit** shall cause the execution of a procedure to be aborted.

8.5 PDL level-1 example

```
iGetStatus –clear
iRead myRegister;    # this can be a full hierarchical name
iRead yourRegister 0b010101
iApply
set my_value   [ iGetReadData myRegister ]
set your_value [ iGetReadData yourRegister ]
# iGetReadData returns a binary string value 0b{1,0}*
if {$my_value == 0b000} { iSetFail “device is on fire” –quit; }
elseif { iGetStatus > 0 } { iSetFail “device is defective” }
```

Annex A

(informative)

ICL grammar

This annex describes the grammar of the ICL language. The conventions in the following list apply to the grammar definitions in A.2.

A.1 Conventions

a) The colon (`:`) delimits the token name and should be read as "*is defined as.*"

b) A semicolon (`;`) terminates a definition.

c) Single quotes (`'`) surround one or more literal characters that appear exactly as shown.

d) Parenthesis surround a group that can be an alternative choice or repeated.

e) A question mark (?) signifies that the grouping can be repeated "zero or one times."

f) An asterisk (*) signifies that the grouping can be repeated "zero or more times."

g) A plus (+) signifies that the group is to be repeated "one or more times."

h) A period (.) signifies that any character is accepted.

i) The vertical bar (|) separates alternatives.

j) The ellipsis (`..`) between two characters means "these two alternatives and all alternatives in the set bounded by them."

k) A backspace (\) means that the next character is a literal.

l) An expression specified with no spaces between them indicates that no space is allowed; if a space is specified then any number of white space character are allowed including tabs, carriage returns and comments.

m) Tokens are case sensitive unless otherwise indicated.

n) Comments in the grammar start with // and continue to the end of the line.

A.2 ICL language definition

The grammar format used in this annex is extracted from Clause 6, but augmented with some additional fragments to make it compatible with the ANTLR4 parser generator (available in antlrworks 2.0 and onward).

```
// ICL grammar v20130821
grammar ICL;
hier_port
// ICL grammar v20140303
SCALAR_ID : ('a'..'z' | 'A'..'Z')('a'..'z' | 'A'..'Z' | DEC_DIGIT | '_')* ;
// ANTLR4 has all keywords reserved.  Since ICL keywords are not reserved, an
additional rule will be required to properly handle.
pos_int : '0' | '1' | POS_INT ;

POS_INT : DEC_DIGIT('_'|DEC_DIGIT)* ;
```

```
size : pos_int | '$' SCALAR_ID ;
UNKNOWN_DIGIT : 'X' | 'x';

DEC_DIGIT : '0'..'9' ;
BIN_DIGIT : '0'..'1' | UNKNOWN_DIGIT ;
HEX_DIGIT : '0'..'9' | 'A'..'F' | 'a'..'f' | UNKNOWN_DIGIT ;
DEC_BASE : '\'' ('d' | 'D') (' ' | '\t')*;
BIN_BASE : '\'' ('b' | 'B') (' ' | '\t')*;
HEX_BASE : '\'' ('h' | 'H') (' ' | '\t')*;
UNSIZED_DEC_NUM : DEC_BASE POS_INT ;
UNSIZED_BIN_NUM : BIN_BASE BIN_DIGIT('_'|BIN_DIGIT)* ;
UNSIZED_HEX_NUM : HEX_BASE HEX_DIGIT('_'|HEX_DIGIT)* ;
sized_dec_num : size UNSIZED_DEC_NUM ;
sized_bin_num : size UNSIZED_BIN_NUM ;
sized_hex_num : size UNSIZED_HEX_NUM ;

vector_id : SCALAR_ID '[' (index | range) ']' ;
index : integer_expr ;
range : index ':' index ;
number : unsized_number | sized_number | integer_expr ;
integer_expr : integer_expr_lvl1 ;
integer_expr_lvl1 : integer_expr_lvl2 ( ('+' | '-') integer_expr_lvl1 )? ;
integer_expr_lvl2 : integer_expr_arg (('*' | '/' | '%') integer_expr_lvl2 )? ;
integer_expr_paren : '(' integer_expr ')'; // Parentheses
integer_expr_arg : integer_expr_paren | pos_int | parameter_ref ;
parameter_ref : '$'(SCALAR_ID) ;
unsized_number : pos_int | UNSIZED_DEC_NUM | UNSIZED_BIN_NUM |
                 UNSIZED_HEX_NUM ;
sized_number : sized_dec_num | sized_bin_num | sized_hex_num;
concat_number : '~'? number (',' '~'? number)* ;
//semantic rules prevents inverting unsized numbers and having more than one
//unsized number within a concat_number. See section 6.4.10.
concat_number_list : concat_number ( '|' concat_number )* ;

hier_port : (instance_name '.')+ port_name ;
port_name : SCALAR_ID | vector_id ;
register_name : SCALAR_ID | vector_id ;
instance_name : SCALAR_ID;
namespace_name : SCALAR_ID;
module_name : SCALAR_ID;
reg_port_signal_id : SCALAR_ID | vector_id;
signal : (number | reg_port_signal_id | hier_port ) ;
reset_signal : '~'? signal ;
scan_signal : '~'? signal ;
data_signal : '~'? signal ;
clock_signal : '~'? signal ;
tck_signal : signal ;
tms_signal : signal ;
trst_signal : signal ;
shiftEn_signal : signal ;
captureEn_signal : signal ;
updateEn_signal : signal ;

concat_reset_signal : (reset_signal | data_signal)
                      ( ',' reset_signal | data_signal )*;
concat_scan_signal : (scan_signal | data_signal)
                     ( ',' scan_signal | data_signal )*;
concat_data_signal : data_signal ( ',' data_signal)*;
concat_clock_signal : (clock_signal | data_signal)
                      ( ',' clock_signal | data_signal)*;
concat_tck_signal : (tck_signal | data_signal)
                    ( ',' tck_signal | data_signal)*;
concat_shiftEn_signal : (shiftEn_signal | data_signal)
```

```
                        ( ',' shiftEn_signal | data_signal)* ;
concat_captureEn_signal : (captureEn_signal | data_signal)
                          ( ',' captureEn_signal | data_signal )*;
concat_updateEn_signal : (updateEn_signal | data_signal)
                         ( ',' updateEn_signal | data_signal )*;
concat_tms_signal : (tms_signal | data_signal)
                    ( ',' tms_signal | data_signal)*;
concat_trst_signal : (trst_signal | data_signal)
                     ( ',' trst_signal | data_signal )*;
icl_source : iclSource_items+ ;
iclSource_items : nameSpace_def | useNameSpace_def | module_def;
nameSpace_def : 'NameSpace' namespace_name? ';' ;
useNameSpace_def : 'UseNameSpace' namespace_name? ';' ;
module_def : 'Module' module_name '{' module_item* '}' ;

module_item : useNameSpace_def |
              port_def |
              instance_def |
              scanRegister_def |
              dataRegister_def |
              logicSignal_def |
              scanMux_def |
              dataMux_def |
              clockMux_def |
              oneHotDataGroup_def |
              oneHotScanGroup_def |
              scanInterface_def |
              accessLink_def |
              alias_def |
              enum_def |
              parameter_def |
              localParameter_def |
              attribute_def ;
port_def : scanInPort_def |
           scanOutPort_def |
           shiftEnPort_def |
           captureEnPort_def |
           updateEnPort_def |
           dataInPort_def |
           dataOutPort_def |
           toShiftEnPort_def |
           toUpdateEnPort_def |
           toCaptureEnPort_def |
           selectPort_def |
           toSelectPort_def |
           resetPort_def |
           toResetPort_def |
           tmsPort_def |
           toTmsPort_def |
           tckPort_def |
           toTckPort_def |
           clockPort_def |
           toClockPort_def |
           trstPort_def |
           toTrstPort_def |
           toIRSelectPort_def |
           addressPort_def |
           writeEnPort_def |
           readEnPort_def ;
scanInPort_def : 'ScanInPort' scanInPort_name (';' |
                 ( '{' attribute_def* '}' ) ) ;
scanInPort_name : port_name ;
scanOutPort_def : 'ScanOutPort' scanOutPort_name ( ';' |
```

```
                  ( '{' scanOutPort_item* '}' ) ) ;
scanOutPort_name : port_name;
scanOutPort_item : attribute_def |
                   scanOutPort_source |
                   scanOutPort_enable ;
scanOutPort_source : 'Source' concat_scan_signal ';';
scanOutPort_enable : 'Enable' data_signal ';';
shiftEnPort_def : 'ShiftEnPort' shiftEnPort_name (';' |
                  ( '{' attribute_def* '}' ) ) ;
shiftEnPort_name : port_name ;
captureEnPort_def : 'CaptureEnPort' captureEnPort_name (';' |
                    ( '{' attribute_def* '}' ) ) ;
captureEnPort_name : port_name ;
updateEnPort_def : 'UpdateEnPort' updateEnPort_name (';' |
                   ( '{' attribute_def* '}' ) ) ;
updateEnPort_name : port_name ;
dataInPort_def : 'DataInPort' dataInPort_name ( ';' |
                 ( '{' dataInPort_item* '}' ) ) ;
dataInPort_name : port_name ;
dataInPort_item : attribute_def |
                  dataInPort_refEnum |
                  dataInPort_defaultLoadValue ;
dataInPort_refEnum : 'RefEnum' enum_name ';' ;
dataInPort_defaultLoadValue : 'DefaultLoadValue' (concat_number |
                              enum_symbol) ';' ;
dataOutPort_def : 'DataOutPort' dataOutPort_name (';' |
                  ( '{' dataOutPort_item* '}' ) ) ;
dataOutPort_name : port_name ;
dataOutPort_item : attribute_def |
                   dataOutPort_source |
                   dataOutPort_enable |
                   dataOutPort_refEnum ;
dataOutPort_source : 'Source' concat_data_signal ';' ;
dataOutPort_enable : 'Enable' data_signal ';' ;
dataOutPort_refEnum : 'RefEnum' enum_name ';' ;
toShiftEnPort_def : 'ToShiftEnPort' toShiftEnPort_name (';' |
                    ( '{' toShiftEnPort_items* '}' ) ) ;
toShiftEnPort_name : port_name ;
toShiftEnPort_items : attribute_def |
                      toShiftEnPort_source ;
toShiftEnPort_source : 'Source' concat_shiftEn_signal ';' ;
toCaptureEnPort_def : 'ToCaptureEnPort' toCaptureEnPort_name (';' |
                      ( '{' toCaptureEnPort_items* '}' ) ) ;
toCaptureEnPort_name : port_name ;
toCaptureEnPort_items : attribute_def |
                        toCaptureEnPort_source ;
toCaptureEnPort_source : 'Source' captureEn_signal ';' ;
toUpdateEnPort_def : 'ToUpdateEnPort' toUpdateEnPort_name (';' |
                     ( '{' toUpdateEnPort_items* '}' ) ) ;
toUpdateEnPort_name : port_name ;
toUpdateEnPort_items : attribute_def | toUpdateEnPort_source ;
toUpdateEnPort_source : 'Source' updateEn_signal ';' ;
selectPort_def : 'SelectPort' selectPort_name (';' |
                 ( '{' attribute_def* '}' ) ) ;
selectPort_name : port_name ;
toSelectPort_def : 'ToSelectPort' toSelectPort_name (';' |
                   ( '{' toSelectPort_item+ '}' ) ) ;
toSelectPort_name : port_name ;
toSelectPort_item : attribute_def | toSelectPort_source ;
toSelectPort_source : 'Source' concat_data_signal ';' ;
resetPort_def : 'ResetPort' resetPort_name (';' |
                ( '{' resetPort_item* '}' ) ) ;
resetPort_name : port_name ;
```

```
resetPort_item : attribute_def |
                 resetPort_polarity ;
resetPort_polarity : 'ActivePolarity' ('0'| '1') ';' ;
toResetPort_def : 'ToResetPort' toResetPort_name (';' |
                  ( '{' toResetPort_item+ '}' ) ) ;
toResetPort_name : port_name ;
toResetPort_item : attribute_def |
                   toResetPort_source |
                   toResetPort_polarity;
toResetPort_source : 'Source' concat_reset_signal ';' ;
toResetPort_polarity : 'ActivePolarity' ('0'| '1') ';' ;
tmsPort_def : 'TMSPort' tmsPort_name (';' |
              ( '{' attribute_def*'}' ) ) ;
tmsPort_name : port_name ;
toTmsPort_def : 'ToTMSPort' toTmsPort_name (';' |
                ( '{' toTmsPort_item* '}' ) ) ;
toTmsPort_name : port_name ;
toTmsPort_item : attribute_def |
                 toTmsPort_source ;
toTmsPort_source : 'Source' concat_tms_signal ';' ;
toIRSelectPort_def : 'ToIRSelectPort' toIRSelectPort_name (';' |
                     ( '{' attribute_def* '}' )) ;
toIRSelectPort_name : port_name ;
tckPort_def : 'TCKPort' tckPort_name (';' |
              ( '{' attribute_def* '}' ) ) ;
tckPort_name : port_name ;
toTckPort_def : 'ToTCKPort' toTckPort_name (';' |
                ( '{' attribute_def* '}' ) ) ;
toTckPort_name : port_name ;
clockPort_def : 'ClockPort' clockPort_name
                (';' | ( '{' clockPort_item* '}' ));
clockPort_name : port_name ;
clockPort_item : attribute_def |
                 clockPort_diffPort ;
clockPort_diffPort : 'DifferentialInvOf' concat_clock_signal ';' ;
toClockPort_def : 'ToClockPort' toClockPort_name (';' |
                  ( '{' toClockPort_item+ '}' ) ) ;
toClockPort_name : port_name ;
toClockPort_item : attribute_def |
                   toClockPort_source |
                   freqMultiplier_def |
                   freqDivider_def |
                   differentialInvOf_def |
                   period_def ;
toClockPort_source : 'Source' concat_clock_signal ';' ;
freqMultiplier_def : 'FreqMultiplier' pos_int ';' ;
freqDivider_def : 'FreqDivider' pos_int ';' ;
differentialInvOf_def : 'DifferentialInvOf' concat_clock_signal ';' ;
period_def : 'Period' pos_int
             ('s' | 'ms' | 'us' | 'ns' | 'ps' | 'fs' | 'as')? ';' ;
trstPort_def : 'TRSTPort' trstPort_name (';' |
               ( '{' attribute_def* '}' ) ) ;
trstPort_name : port_name ;
toTrstPort_def : 'ToTRSTPort' toTrstPort_name (';' |
                 ( '{' toTrstPort_item+ '}' ) ) ;
toTrstPort_name : port_name ;
toTrstPort_item : attribute_def |
                  toTrstPort_source ;
toTrstPort_source : 'Source' concat_trst_signal ';' ;
addressPort_def : 'AddressPort' addressPort_name (';' |
                  ( '{' attribute_def*'}' ) ) ;
addressPort_name : port_name ;
writeEnPort_def : 'WriteEnPort' writeEnPort_name (';' |
```

```
                  ( '{' attribute_def* '}' ) ) ;
writeEnPort_name : port_name ;
readEnPort_def : 'ReadEnPort' readEnPort_name (';' |
                 ( '{' attribute_def* '}' ) ) ;
readEnPort_name : port_name ;
instance_def : 'Instance' instance_name 'Of' (namespace_name? '::')?
               module_name (';' | ( '{' instance_item* '}' ) ) ;
instance_item : inputPort_connection |
                allowBroadcast_def |
                attribute_def |
                parameter_override |
                instance_addressValue ;
inputPort_connection : 'InputPort' inputPort_name '=' inputPort_source ';' ;
allowBroadcast_def : 'AllowBroadcastOnScanInterface' scanInterface_name ';' ;
inputPort_name : port_name ;
inputPort_source : concat_reset_signal |
                   concat_scan_signal |
                   concat_data_signal |
                   concat_clock_signal |
                   concat_tck_signal |
                   concat_shiftEn_signal |
                   concat_captureEn_signal |
                   concat_updateEn_signal |
                   concat_tms_signal |
                   concat_trst_signal ;
parameter_override : parameter_def;
instance_addressValue : 'AddressValue' number ';' ;
scanRegister_def : 'ScanRegister' scanRegister_name (';' |
                   '{' scanRegister_item* '}') ;
scanRegister_name : register_name ;
scanRegister_item : attribute_def |
                    scanRegister_scanInSource |
                    scanRegister_defaultLoadValue |
                    scanRegister_captureSource |
                    scanRegister_resetValue |
                    scanRegister_refEnum ;
scanRegister_scanInSource : 'ScanInSource' scan_signal ';' ;
scanRegister_defaultLoadValue : 'DefaultLoadValue' (concat_number |
                                enum_symbol) ';' ;
scanRegister_captureSource : 'CaptureSource' (concat_data_signal |
                             enum_symbol) ';';
scanRegister_resetValue : 'ResetValue' (concat_number |
                          enum_symbol)';' ;
scanRegister_refEnum : 'RefEnum' enum_name ';' ;
dataRegister_def : 'DataRegister' dataRegister_name (';' |
                   ( '{' dataRegister_item+ '}' ) ) ;
dataRegister_name : register_name ;
dataRegister_item : dataRegister_type |
                    dataRegister_common ;
dataRegister_type : dataRegister_selectable |
                    dataRegister_addressable |
                    dataRegister_readCallBack |
                    dataRegister_writeCallBack ;
// Common to all types:
dataRegister_common : dataRegister_resetValue |
                      dataRegister_defaultLoadValue |
                      dataRegister_refEnum |
                      attribute_def ;
dataRegister_resetValue : 'ResetValue' (concat_number |
                          enum_symbol) ';';
dataRegister_defaultLoadValue : 'DefaultLoadValue' (concat_number |
                                enum_symbol)';' ;
dataRegister_refEnum : 'RefEnum' enum_name ';' ;
```

```
//For Selectable Data Register:
dataRegister_selectable : dataRegister_writeEnSource |
                          dataRegister_writeDataSource;
dataRegister_writeEnSource : 'WriteEnSource' '~'? data_signal ';' ;
dataRegister_writeDataSource : 'WriteDataSource' concat_data_signal ';' ;
// Addressable Data Register:
dataRegister_addressable : dataRegister_addressValue;
dataRegister_addressValue : 'AddressValue' number ';' ;
// CallBack Data Register:
dataRegister_readCallBack : dataRegister_readCallBack_proc |
                            dataRegister_readDataSource ;
dataRegister_readCallBack_proc : 'ReadCallBack' iProc_namespace
                                   iProc_name iProc_args* ';';
dataRegister_readDataSource : 'ReadDataSource' concat_data_signal ';' ;
dataRegister_writeCallBack : 'WriteCallBack' iProc_namespace
                             iProc_name iProc_args* ';' ;
iProc_namespace : (namespace_name? '::')? ref_module_name
                  ( '::' sub_namespace )? ;
iProc_name : SCALAR_ID | parameter_ref ;
iProc_args : '<D>' |
             '<R>' |
             number |
             STRING |
             parameter_ref ;
sub_namespace : SCALAR_ID |
                parameter_ref ;
ref_module_name : SCALAR_ID |
                  parameter_ref ;
logicSignal_def : 'LogicSignal' logicSignal_name
                  '{' logic_expr ';' '}' ;
logicSignal_name : reg_port_signal_id;
logic_expr : logic_expr_lvl1 ;
logic_expr_lvl1 : logic_expr_lvl2 ( ('&&' | '||') logic_expr_lvl1 )? ;
logic_expr_lvl2 : logic_expr_lvl3 ( ('&' | '|' | '^') logic_expr_lvl2 )? |
                  ( ('&' | '|' | '^') logic_expr_lvl2 );
logic_expr_lvl3 : logic_expr_lvl4 ( ('==' | '!=') logic_expr_num_arg )?;
logic_expr_lvl4 : logic_expr_arg (',' logic_expr_lvl4 )?;
logic_unary_expr : ('~'|'!') logic_expr_arg;
logic_expr_paren : '(' logic_expr ')';
logic_expr_arg : logic_expr_paren |
                 logic_unary_expr |
                 concat_data_signal ;
logic_expr_num_arg : concat_number |
                     enum_name |
                     '(' logic_expr_num_arg ')' ;
scanMux_def : 'ScanMux' scanMux_name 'SelectedBy' scanMux_select
              '{' scanMux_selection+ '}' ;
scanMux_name : reg_port_signal_id ;
scanMux_select : concat_data_signal ;
scanMux_selection : concat_number_list':' concat_scan_signal ';' ;
dataMux_def : 'DataMux' dataMux_name 'SelectedBy' dataMux_select
              '{' dataMux_selection+ '}' ;
dataMux_name : reg_port_signal_id ;
dataMux_select : concat_data_signal ;
dataMux_selection : concat_number_list':' concat_data_signal ';' ;
clockMux_def : 'ClockMux' clockMux_name 'SelectedBy' clockMux_select
               '{' clockMux_selection+ '}' ;
clockMux_name : reg_port_signal_id ;
clockMux_select : concat_data_signal ;
clockMux_selection : concat_number_list':' concat_clock_signal ';' ;
oneHotScanGroup_def : 'OneHotScanGroup' oneHotScanGroup_name
                      '{' oneHotScanGroup_item+ '}' ;
oneHotScanGroup_name : reg_port_signal_id ;
```

```
oneHotScanGroup_item : 'Port' concat_scan_signal ';' ;
oneHotDataGroup_def : 'OneHotDataGroup' oneHotDataGroup_name
                      '{' oneHotDataGroup_item+ '}' ;
oneHotDataGroup_name : reg_port_signal_id ;
oneHotDataGroup_item : instance_def |
                       dataRegister_def |
oneHotDataGroup_portSource ;
oneHotDataGroup_portSource : 'Port' concat_data_signal ';' ;
scanInterface_def : 'ScanInterface' scanInterface_name '{'
                    scanInterface_item+ '}' ;
scanInterface_name : SCALAR_ID;
scanInterface_item : attribute_def | scanInterfacePort_def |
                     defaultLoad_def | scanInterfaceChain_def ;
scanInterfacePort_def : 'Port' reg_port_signal_id ';';
scanInterfaceChain_def : 'Chain' scanInterfaceChain_name
                         '{' scanInterfaceChain_item+ '}' ;
scanInterfaceChain_name : SCALAR_ID;
scanInterfaceChain_item : attribute_def | scanInterfacePort_def |
                          defaultLoad_def ;
defaultLoad_def : 'DefaultLoadValue' concat_number ';' ;
accessLink_def : accessLink1149_def | AccessLinkGeneric_def ;
// Actual parser will need to add gated semantic predicate here or rule will
// match 1149 definition as well
AccessLinkGeneric_def : 'AccessLink' SPACE SCALAR_ID SPACE 'Of' SPACE
                        SCALAR_ID SPACE? ;
fragment AccessLinkGeneric_block :'{' ( AccessLinkGeneric_block |
        AccessLinkGeneric_text | SPACE | STRING )* '}' ;
fragment AccessLinkGeneric_text : [^{}"\t\n\r ]+; // Not used in ANTLR parser,
accessLink_genericID : SCALAR_ID;accessLink1149_def : 'AccessLink'
                       accessLink_name 'Of'
                       ('STD_1149_1_2001' | 'STD_1149_1_2013')
                       '{' 'BSDLEntity' bsdlEntity_name ';'
                       bsdl_instr_ref+ '}' ;
accessLink_name : SCALAR_ID;
bsdlEntity_name : SCALAR_ID ;
bsdl_instr_ref : bsdl_instr_name '{' bsdl_instr_selected_item+ '}' ;
bsdl_instr_name : SCALAR_ID ;
bsdl_instr_selected_item : 'ScanInterface'
                           '{' (accessLink1149_ScanInterface_name ';')+ '}' |
                           ('ActiveSignals'
                           '{'(accessLink1149_ActiveSignal_name ';')+ '}' ) ;
accessLink1149_ActiveSignal_name : reg_port_signal_id ;
accessLink1149_ScanInterface_name : instance_name('.' scanInterface_name)? ;
alias_def : 'Alias' alias_name '=' concat_hier_data_signal
            (';' | ('{' alias_item+ '}' ) ) ;
alias_name : reg_port_signal_id;
alias_item : attribute_def |
             'AccessTogether' ';' |
             alias_iApplyEndState |
             alias_refEnum ;
alias_iApplyEndState : 'iApplyEndState' concat_number ';' ;
alias_refEnum : 'RefEnum' enum_name ';' ;
concat_hier_data_signal : '~'? hier_data_signal (',' '~'? hier_data_signal)* ;
hier_data_signal : (instance_name '.')* reg_port_signal_id ;
enum_def : 'Enum' enum_name '{' enum_item+ '}' ;
enum_name : SCALAR_ID ;
enum_item : enum_symbol '=' enum_value ';' ;
enum_symbol : SCALAR_ID;
enum_value : concat_number;
parameter_def : 'Parameter' parameter_name '=' parameter_value ';' ;
localParameter_def : 'LocalParameter' parameter_name '=' parameter_value ';' ;
parameter_name : SCALAR_ID;
parameter_value : concat_string | concat_number;
```

```
concat_string : (STRING | parameter_ref) (',' (STRING | parameter_ref) )* ;
attribute_def : 'Attribute' attribute_name ('=' attribute_value )? ';' ;
attribute_name : SCALAR_ID;
attribute_value : (concat_string | concat_number) ;
STRING : '"' (~('"'|'\\')|'\\\\'|'\\"')* '"' ;

SPACE : ( ' ' | '\t' | ('\r'? '\n') )+ -> skip ;
// Multi-line Comments
ML_COMMENT : '/*' .*? '*/' -> skip ;
// Single-line Comments
SL_COMMENT : '//' (~('\r'|'\n')*) '\r'? '\n' -> skip ;
```

Annex B

(informative)

PDL level-0 grammar

B.1 Conventions

The grammar format used in this annex is extracted from Clause 7, but augmented with some additional fragments to make it compatible with the ANTLR4 parser generator (available in antlrworks2.0 and onward).

This annex describes the grammar of the PDL language, in the following grammar:

a) The colon (`:`) delimits the token name and should be read as "*is defined as.*"

b) A semicolon (`;`) terminates a definition.

c) Single quotes (`'`) surround one or more literal characters that appear exactly as shown.

d) Parenthesis surround a group that can be an alternative choice or repeated.

e) A question mark (?) signifies that the grouping can be repeated "zero or one times."

f) An asterisk (*) signifies that the grouping can be repeated "zero or more times."

g) A plus (+) signifies that the group are to be repeated "one or more times."

h) A period (.) signifies that any character is accepted.

i) The vertical bar (|) separates alternatives.

j) The ellipsis (`..`) between two characters means "these two alternatives and all alternatives in the set bounded by them."

k) A backspace (\) means that the next character is a literal.

l) Comments in the grammar start with // and continue to the end of the line.

B.2 Grammar

```
// PDL0 grammar v20130806
grammar PDL;
pdl_source : (WS | eoc | pdl_level_def | iprocsformodule_def
             | iuseprocnamespace_def | iproc_def | SL_COMMENT)+ ;
// ===========
// Identifiers
// ===========
dot_id : scalar_id ( '.' scalar_id )*;
scalar_id : SCALAR_ID | keyword; // keywords are not reserved
keyword : 'iPDLLevel' |
          'iProcsForModule' |
          'iProc' |
          'iPrefix' |
          'iCall' |
          'iReset' |
          'iUseProcNameSpace' |
```

```
            'iRead' |
            'iWrite' |
            'iScan' |
            'iApply' |
            'iNote' |
            'iTake' |
            'iRelease' |
            'iMerge' |
            'iClock' |
            'iClockOverride' |
            'iRunLoop'|
            'iOverrideScanInterface' |
            'iState' |
            'iGetReadData' |
            'iGetMiscompares' |
            'iGetStatus' |
            'iSetFail' |
            'on' |
            'off'; // Keywords in ANTLR4 are reserved,
                   // non-reserved keywords must be listed explicitly.
                   // The list here may be incomplete for ANTLR4.

// ******************
// Generic Identifiers
// ******************
instancePath  : dot_id;
scanInterface_name : (instancePath DOT)? scalar_id  ;
port   : hier_signal ;
reg_or_port   : hier_signal ;
reg_port_or_instance : hier_signal ;
hier_signal : (instancePath DOT)?
              reg_port_signal_id | ARGUMENT_REF ;

reg_port_signal_id
    : scalar_id | vector_id ;
vector_id
    : scalar_id LBRACKET ( index | range ) RBRACKET
    | scalar_id LPAREN   ( index | range ) RPAREN;
index : pdl_number;
range : index COLON index  ;

enum_name : scalar_id ;
instance_name : scalar_id ;

// **************
// Generic Tokens
// **************

SCALAR_ID : ('a'..'z' | 'A'..'Z')('a'..'z' | 'A'..'Z' | '0'..'9' | '_')*;
ARGUMENT_REF : '$' SCALAR_ID;
LBRACKET : '[';
RBRACKET : ']';
LPAREN : '(';
RPAREN : ')';
COLON : ':';
DOT : '.';

// ===============
// Command tokens
// ===============
cycleCount : pdl_number ;
sysClock : hier_signal ;
sourceClock : hier_signal ;
```

```
chain_id: scalar_id;
length : POS_INT ;
procName : scalar_id ;

// =======
// Numbers
// =======
pdl_number : POS_INT | TCL_HEX_NUMBER | TCL_BIN_NUMBER | ARGUMENT_REF;
POS_INT      : ('0'..'9')+;
TCL_HEX_NUMBER : '0x' ( '0'..'9' | 'A'..'F' | 'a'..'f' | 'x' | 'X' )+;
TCL_BIN_NUMBER : '0b' ( '0'..'1' | 'x' | 'X' )+;
ISCAN_HEX_NUMBER : '{' WS? '0x' ( '0'..'9' | 'A'..'F' | 'a'..'f' | 'x' | 'X' |
WS | NL )+ '}'

                 | '"' WS? '0x' ( '0'..'9' | 'A'..'F' | 'a'..'f' | 'x' | 'X' |
WS | NL )+ '"' ;

ISCAN_BIN_NUMBER : '{' WS? '0b' ( '0'..'1' | 'x' | 'X' | WS | NL )+ '}'
                 | '"' WS? '0b' ( '0'..'1' | 'x' | 'X' | WS | NL )+ '"' ;
tvalue : (EVALUE | DVALUE)TSUFFIX ;
EVALUE : POS_INT'.'POS_INT'e-'POS_INT ;
DVALUE : POS_INT'.'POS_INT ;
TSUFFIX : 's' | 'ms' | 'us' | 'ns' | 'ps' | 'fs' | 'as' ;

// ====================
// Comments and Whitespace
// ====================
SL_COMMENT : '#' (~('\r'|'\n'))* ;
WS : ( ' ' | '\t' | '\\' '\r'? '\n' )+ ;
QUOTED : '"' (~('\"'))* '"' ; // Collapse into one token (mainly for iNote)
eoc : SEMICOLON | NL;
SEMICOLON : ';';
NL : '\r'? '\n' ;

// =======
// Commands
// =======
pdl_level_def : 'iPDLLevel' WS pdl_level WS '-version' WS versionString eoc ;
pdl_level : '0' | '1';
versionString : 'STD_1687_2014' ;
// -------
iprocsformodule_def : 'iProcsForModule' WS module_name (WS '-iProcNameSpace' WS
pdl_namespace_name)? eoc ;
module_name : (icl_namespace_name '::')? scalar_id ;
icl_namespace_name : scalar_id;
pdl_namespace_name : scalar_id;
// -------
iuseprocnamespace_def : 'iUseProcNameSpace' (pdl_namespace_name|'-') ;
// -------
iproc_def : 'iProc' WS procName WS '{' (WS? argument ( WS argument )* WS? |
WS?)
                '}' WS '{' commands '}' eoc ;
commands : command* ;
argument : scalar_id | ( '{' WS? argWithDefault WS? '}' ) ;
argWithDefault : scalar_id WS args ;
args : pdl_number | enum_name | reg_or_port;
command  : WS? (icall_def |
                iuseprocnamespace_def |
                iprefix_def |
                ireset_def |
                iread_def |
                iwrite_def |
                iscan_def |
```

```
                iapply_def |
                inote_def |
                ioverridescaninterface_def |
                imerge_def |
                itake_def |
                irelease_def |
                iclock_def |
                iclock_override_def |
                istate_def |              irunloop_def |
                SL_COMMENT) WS? eoc | WS? eoc ; //empty line acceptable
// -------
iprefix_def : 'iPrefix' WS (instancePath|'-') ;
// -------
icall_def : 'iCall' WS (instancePath DOT)? (pdl_namespace_name '::')? procName
( WS args )* ;
// -------
ireset_def : 'iReset' (WS '-sync')? ;
// -------
iread_def : 'iRead' WS reg_or_port WS (( pdl_number | enum_name ))?;
// -------
iwrite_def : 'iWrite' WS reg_or_port WS ( pdl_number | enum_name );
// -------
iscan_def : 'iScan' WS ('-ir' WS)? scanInterface_name
                (WS '-chain' chain_id)? WS length
            (WS '-si' WS iscan_data)? (WS '-so' WS iscan_data)? ('-stable')? ;
iscan_data : pdl_number | ISCAN_HEX_NUMBER | ISCAN_BIN_NUMBER
           ;
// -------
ioverridescaninterface_def : 'iOverrideScanInterface' WS scanInterfaceRef_list
                             (WS '-capture' WS ('on'|'off'))?
                             (WS '-update' WS ('on'|'off'))?
                             (WS '-broadcast' WS ('on'|'off'))? ;
scanInterfaceRef_list : scanInterfaceRef (WS scanInterfaceRef)*;
scanInterfaceRef : (instancePath DOT)? scanInterface_name;
// -------
iapply_def : 'iApply' (WS '-together')?;
// -------
iclock_def : 'iClock' WS sysClock;
// -------
iclock_override_def : 'iClockOverride' WS sysClock
                      (WS '-source' WS sourceClock)?
                      (WS '-freqMultiplier' WS POS_INT)?
                      (WS '-freqDivider' WS POS_INT)? ;
// -------
irunloop_def : 'iRunLoop' WS ( cycleCount ( WS '-tck' | WS '-sck' port )?
                             | '-time' WS tvalue);
// -------
imerge_def : 'iMerge' WS ( '-begin' | '-end' );
// -------
itake_def : 'iTake' WS reg_port_or_instance ;
// -------
irelease_def : 'iRelease' WS reg_port_or_instance ;
// -------
inote_def : 'iNote' WS ( '-comment' | '-status' ) WS QUOTED;
// -------
istate_def : 'iState' WS reg_or_port WS pdl_number (WS '-LastWrittenValue' | WS
             '-LastReadValue' | WS '-LastMiscompareValue')? ;
```

Annex C

(informative)

PDL level-1 grammar

There are four Tcl command extensions that are defined by IEEE Std 1687 as PDL level-1.

```
//PDL1 grammar 20120328

iget_read_data_def : 'iGetReadData' WS (reg_or_port
                        | scanInterface_name (WS '-chain' WS chain_id)? )
                    ( WS format )? ;
format  : '-dec' | '-bin' | '-hex' ;
iget_miscompares_def : 'iGetMiscompares' (reg_or_port
                        | scanInterface_name (WS '-chain' WS chain_id)? )
                    ( WS format )? ;

iget_status_def : 'iGetStatus' ( '-clear' )? ;

iset_fail_def : 'iSetFail' text_message ( '-quit' )? ;
text_message : string ;
```

Annex D

(informative)

PDL differences between IEEE Std 1687-2014 and IEEE Std 1149.1-2013

D.1 Introduction

Both IEEE Std 1687-2014 and IEEE Std 1149.1-2013 use PDL as the language to express test content, but the two working groups developed different dialects of PDL. While superficially similar, the two versions contain important differences, both in syntax and in meaning. This annex provides an analysis of the two dialects of PDL by comparing the level-0 commands, the level-1 commands, and the general operation of the two versions. It is important to note that the iPDLLevel command (which is required in every PDL command stream) unambiguously identifies the dialect contained in the file so that the tools reading the file will know which parsing and semantic rules to apply.

D.2 PDL level-0 command differences

D.2.1 Overview

Comparing the PDL level-0 command tables of the two standards reveals specific differences in basic command names, options, and general functionality. The differences fall into three categories: extra IEEE 1149.1 commands, extra IEEE 1687 commands, and different syntax/meaning of shared commands.

D.2.2 Extra IEEE 1149.1-2013 PDL level-0 commands

The PDL level-0 commands that are unique to IEEE Std 1149.1-2013 are shown in Table D.1.

Table D.1—Extra IEEE 1149.1 PDL0 commands

#	Command(s)	Comment
1	iSource	Functions like a C #include to allow nested PDL.
2	iSetInstruction	Does not apply to IEEE Std 1687, since it is related to BSDL.
3	iLoop, iUntil, ifTrue, ifFalse, ifEnd, iSetFail	Conditional control for PDL execution. iSetFail is an IEEE 1687 level-1 command.
4	iTRST, iTMSreset, iTMSidle	Performs direct control of TAP reset and state; a related IEEE 1687 command is iReset.
5	iProcGroup	Performs a similar function to the IEEE 1687 iProcsForModule, but against BSDL constructs.

D.2.3 Extra IEEE 1687 PDL level-0 commands

The PDL level-0 commands that are unique to IEEE Std 1687 are shown in Table D.2.

Table D.2—Extra IEEE 1687 PDL level-0 commands

#	Commands(s)	Comment
1	iUseProcNameSpace	Allows iProcs from different vendors to be attached to the same module.
2	iReset	Broad based ICL network reset. Similar IEEE 1149.1 commands are iTRST, iTMSreset.
3	iProcsForModule	Performs a similar function to the IEEE 1149.1 iProcGroup, but against ICL constructs.
4	iState	Restores the controlling state and expected state models in the controlling software, but perform no device interactions.

D.2.4 Differences in shared PDL level-0 commands

The differences in the PDL level-0 commands used in common between the two languages are shown in Table D.3.

Table D.3—Differences in PDL level-0 commands

#	Command(s)	Difference
1	iPDLLevel	IEEE Std 1687 requires it as the first line of a file, while IEEE Std 1149.1 allows iSource commands first.
2	iPrefix	IPrefix does not apply to iCall commands in IEEE Std 1149.1, but it does apply to iCall commands in IEEE Std 1687. It applies to the objects in iWrite, iRead, and iScan commands in both languages.
3	iProc	IEEE Std 1149.1 reserves the following names of iProcs (main, init_setup, init_run, and ecid).
4		IEEE Std 1149.1 supports various options after the procedure name to indicate usage restrictions or other information about the procedure. IEEE Std 1687 has none of this. Option “-TMSreset” or “-TRSTreset” indicate that reset commands are contained in procedure. Usage restrictions options are “-export”, “-mission”, “-noninvasive”, and “-noninteractive”. Generic information is passed with the “-info <text>” option.
5		IEEE Std 1149.1 indicates restrictions on the use of the argument name “args”.
6	iRunLoop	IEEE Std 1149.1 in cyclecount mode does not have an option to explicitly indicate use of the TCK clock, it is assumed if the “-sck <clock>” option is NOT used.
7		IEEE Std 1149.1 adds the option “-tck_off” for both the cyclecount with –sck or time delay modes to indicate that the TCK clock should NOT be free running during the iRunLoop operation.
8		IEEE Std 1687 supports a suffix (ms, us, etc.) with an integer time value time, while IEEE Std 1149.1 uses real numbers and units of only seconds.
9	iTake/iRelease	IEEE Std 1149.1 indicates that a resource reserved inside an iProc must be released before leaving the iProc. IEEE Std 1687 does not have this requirement in order to support nested iMerge statements.

Table D.3—Differences in PDL level-0 commands *(continued)*

#	Command(s)	Difference
10	iMerge	Between the iMerge –begin and –end, IEEE Std 1149.1 allows the additional statements iTake and iRelease. IEEE Std 1687 does not allow iTake and iRelease outside the iCall because they must be associated to one PDL stream. Both standards allow iCall and iNote.
11		IEEE 1687 flags any queued but unapplied actions when the iMerge –end is encountered as errors, while IEEE Std 1149.1 does not.
12	iCall	IEEE Std 1149.1 has the "-direct" option that bypasses the step to lookup the associated object via the instance.
13	iScan	IEEE Std 1687 has the option "-ir" that restricts when the scan operation is done.
14	iRead	IEEE Std 1149.1 allows the use of a mnemonic for the expected value; IEEE Std 1687 allows enumerations associated with the object being read to be used.
15	iWrite	IEEE Std 1149.1 allows a mnemonic and options for alternate ways to indicate the values to be scanned in. IEEE Std 1687 allows typed enumerations.
16	iApply	IEEE Std 1687 has the option "-together" to force related objects to be loaded in the same operation.
17		IEEE Std 1149.1 has options (-skipRTI, -shiftPause) that are used to control the states the TAP controller passes through during the scan operation. These options are left to tool providers in IEEE Std 1687.
18		IEEE Std 1149.1 has the option –nofail to control how failed compares are flagged. This is related to its extra conditional control commands, so is not of concern to IEEE Std 1687.
19	iClock	IEEE Std 1687 references an ICL clock port to trigger a check that the specified clock port has a controlling path to a valid clock source, while IEEE Std 1149.1 has options (-period <seconds>) for minimum period and whether other clocks should be free running (-on, -off).
20		IEEE Std 1149.1 will set the local status flag if the indicated clock has some sort of error.
21	iProcGroup & iProcsForModule	IEEE Std 1687 has the option "-iProcNameSpace" for iProcsForModule; IEEE Std 1149.1 does not.
22		IEEE Std 1149.1 indicates that procedures not preceded by an iProcGroup command are considered to be associated with the top of the unit under test, while IEEE Std 1687 applies the iProcsForModule command to every subsequent procedure until the next iProcsForModule command.
23		IEEE Std 1149.1 indicates that multiple iProcGroup commands can be used in a PDL file and their affect stops at the end of the file. IEEE Std 1687 does not have this restriction.

D.3 PDL level-1 commands

D.3.1 Overview

Comparing the PDL level-1 command tables of the two standards reveals specific differences in basic command names, options, and general functionality. The differences fall into three categories: extra IEEE 1149.1 commands, extra IEEE 1687 commands, and different syntax/meaning of shared commands.

D.3.2 Extra IEEE 1149.1-2013 PDL level-1 commands

The PDL level-1 command unique to IEEE 1149.1-2013 is shown in Table D.4.

Table D.4—Extra IEEE 1149.1 PDL level-1 commands

#	Command	Comment
1	iGet	Similar to the IEEE 1687 iGetReadData command, but has operational differences, see below.

D.3.3 Extra IEEE 1687 PDL level-1 commands

The PDL level-1 commands unique to IEEE Std 1687 are shown in Table D.5.

Table D.5—Extra IEEE 1687 PDL level-1 commands

#	Command	Comment
1	iGetReadData	Similar to the IEEE 1149.1 iGet command, but has operational differences, see below.
2	iGetMiscompares	This may be handled by the IEEE 1149.1 iGet "–fail" option, though with differences in the returned values.
3	iSetFail	For IEEE Std 1149.1 this is a level-0 command.

D.3.4 Differences in shared PDL level-1 commands

The differences in the PDL level-1 commands used in common between the two languages are shown in Table D.6.

Table D.6—Differences in PDL level-1 commands

#	Command	Differences
1	iGetReadData vs. iGet	iGetReadData returns only the read data; other commands return other data.. The IEEE 1149.1 iGet has options to select several other choices for the type of data returned. iGetReadData (and iGetMiscompares) may also take a ScanInterface as an argument that is not present in iGet.
2		IEEE Std 1149.1 provides different means to indicate data is not available or an error condition.
3	iGetStatus	IEEE Std 1149.1 returns the string "PASS" or "FAIL". The fail condition can also be set by the iSetFail command.
4		IEEE Std 1687 returns the NUMBER of failing bits from expected data comparisons by iRead statements.
5	iSetFail	IEEE Std 1149.1 additionally sets the expected data comparison accumulated failure flag. While IEEE Std 1687 keeps data compare failures separate from this commands failure indication.

D.4 General operation differences

In addition to the language differences, there are also differences in the basic concepts and implementation details between IEEE Std 1687 and IEEE Std 1149.1-2013.

a) In IEEE Std 1149.1-2013, a single iApply group can only reference registers in a single TDR activated by a single instruction. This restriction does not apply to IEEE Std 1687.

b) IEEE Std 1149.1 reserves predefined names for a set of specific iProcs (main, init_setup, init_run, and ecid) that could be used as the entry point for processing under defined conditions.

c) IEEE Std 1149.1 provides the iSource command to allow nesting of PDL files with a prescribed structure. IEEE Std 1687 does not mandate a file structure for PDL, but instead uses the principle of a PDL stream in which only a few command ordering rules are required.

d) Undersized numbers are padded in different ways: IEEE Std 1687: 7.8—depends on the value of the MSB, IEEE Std 1149.1: C.2.7—depends on whether the operation is an iRead or iWrite.

e) Oversized numbers are an error in IEEE Std 1687, while in IEEE Std 1149.1 they are only an error if the truncated bits contain values other than '0'. Same sections as for undersized numbers.

f) Both standards provide the basics about how PDL is compatible with Tcl syntax; IEEE Std 1149.1 C.3.3.2 goes into detail on the syntax supported as well as how the syntax is processed.

g) IEEE Std 1149.1 ends the effect of iProcGroup at the end of a file. IEEE Std 1687 continues to apply the iProcsForModule command to all subsequent procedures until another such command is encountered in the PDL stream. IEEE Std 1687 does not enforce a file structure, but instead defines the concept of a PDL stream.

D.5 Creating interoperable PDL

D.5.1 Principle

Despite the inherent differences between IEEE 1149.1 and IEEE 1687 PDL dialects, it is possible to write PDL that can operate in both environments. The principle is simple: create a core PDL that utilizes only commands (and command options) that are shared between the two dialects, and wrap the core PDL with standard-specific PDL commands.

The PDL commands are split into two sets of commands: Standard-specific and Core. The Standard-specific commands have different syntax and are dependent primarily on the design specification language (i.e., ICL or BSDL). The Core commands are a subset of other commands with restrictions on syntax options such that they are usable in IEEE Std 1149.1 or IEEE Std 1687. Other commands are excluded entirely. There are some other general restrictions on syntax to allow use in either environment. Finally a set of PDL files is structured so that the C preprocessor, cpp, can be used to "wrap" the Standard-specific part around the Core part for the standard to be targeted.

D.5.2 Example PDL files

CoreCmds.pdl:

```
iProc MyTest {} {
  iWrite ...
  iRead ...
  iApply
  ...
}
```

IEEE1687or1149Cmds.pdl:

```
#ifndef do1149
              iPDLLevel 0 -version STD_1687_2014
              iProcsForModule Company::MyInstrument
#include "CoreCmds.pdl"
#else
              iPDLLevel 0 -version STD_1149_1_2013
              iProcGroup MyInstrument
```

```
#include "CoreCmds.pdl"
#endif
```

To create an IEEE 1687 compatible PDL file "P1687Cmds.pdl" run:

```
cpp –P P1687or1149Cmds.pdl P1687Cmds.pdl
```

To create an IEEE 1149 compatible PDL file "1149Cmds.pdl" run:

```
cpp -Ddo1149 -P P1687or1149Cmds.pdl 1149Cmds.pdl
```

Then each tool can read its corresponding PDL file (1149Cmds.pdl or 1687Cmds.pdl, respectively).

D.5.3 General restrictions

These restrictions apply to the PDL command file and may affect the associated ICL and BSDL files.

a) The serial access network of the design to be driven by the Core PDL file must be representable in both IEEE 1687 ICL and IEEE 1149.1 BSDL constructs and be manageable by the restricted PDL command set.

b) The length of values must precisely match the length of the corresponding registers so there is no needed extension or truncation of the value.

c) The names of elements in the design specification language (ICL or BSDL) must start with a letter followed by letters and numbers and avoid PDL keywords.

d) Names with indexing or ranges in register names must use only parenthesis '(...)', not brackets '[...]'.

e) The use of ICL aliases/enums and BSDL mnemonics for register values must be coordinated such that the names and values are the same. This allows a single name to be used in the Core part of the PDL files to reference either an IEEE 1149.1 mnemonic or an IEEE 1687 alias/enumeration.

f) Object references must be restricted to only registers for commands affecting scan data, since only registers are common to both design specification languages.

g) To support retargeting in IEEE Std 1687, PDL must be restricted to level-0.

h) To avoid confusing cpp, PDL comments should only follow a PDL command (even if it is a null command (“: # comment”) and should never start as the first character on a line.

D.5.4 Standard-specific PDL commands

The commands in Table D.7 are used in the standard-specific part of the PDL file.

Table D.7—Standard-specific wrapper PDL commands

Command	Restrictions	Environment
iPDLLevel <level> -version <version>	Each standard uses its corresponding version string. Use only level-0.	Both
iProcsForModule <modulename>	This is an ICL-specific reference that can include ICL namespaces. The option -iProcNameSpace is not allowed since IEEE Std 1149.1 does not have PDL namespaces.	IEEE Std 1687
iProcGroup <entity, package, instance>	This is a BSDL-specific reference.	IEEE Std 1149.1

D.5.5 Core PDL commands

The commands in Table D.8 are usable in the Core part of the PDL file. They attempt to be environment-neutral with the indicated set of restrictions. Optional arguments enclosed in '[...]'.

Table D.8—Core PDL commands

Command	Restrictions
iProc <procname> { [<arguments>] } { <body> }	No options are allowed. The argument name 'args' is not allowed. IEEE 1687 PDL namespace references are not allowed in the body. IEEE Std 1149 reserves the procnames: main, init_setup, init_run and ecid.
iPrefix <instance path> or '-'	Requires ICL and BSDL description of design hierarchy to match.
iCall <proc instance path> [{ <arguments }]	No options are allowed. IEEE 1687 PDL namespace not allowed in instance path.
iWrite <register> <value>	The <value> can be bin, hex, dec or enum/mnemonic (see general restrictions subclause comments on enum/mnemonic). IEEE 1149 options -reset, -default and -safe not allowed.
iRead <register> [<value>]	The <value> can be bin, hex, dec or enum/mnemonic (see general restrictions subclause comments on enum/mnemonic).
iApply	No options from either dialect are allowed. The users iRead/iWrite commands in an iApply action can only operate on a single TDR(BSDL instruction).
iClock <clockname>	No options allowed. There could be a problem since IEEE Std 1149.1 requires the -period option to be used somewhere in defining the clock, but IEEE Std 1687 does not have this option. Instead this is handled in the ICL ClockPort and ToClockPort statements.
iClockOverride <clock> -source <src_clock> -freqMultiplier <number>	IEEE Std 1149 requires the -source and -freqMultiplier options, while IEEE Std 1687 does not. Options -freqDivider, -on and -off not allowed. There could be a problem with how the -freqMultiplier <number> is used between the standards and issues with associated iClock command. This command may have to be EXCLUDED.
iRunLoop <count> [-sck <clock>] | -time <seconds>	Options -tck and -tck_off not allowed. There could be issues with <clock> since in IEEE Std 1687 it is a port specified by iClock, while in IEEE Std 1149 it is defined by iClock or iClockOverride. If the iClock and iClockOverride commands are excluded, the -sck option may have to be dropped.
iMerge –begin | -end	Must surround only iCall commands.
iTake <resource>	Name of resource must be consistent between the two structural descriptions (BSDL, ICL).
iRelease <resource>	Name of resource must be consistent between the two structural descriptions (BSDL, ICL).
iNote –comment | -status <text>	Compatible.

D.5.6 Excluded PDL commands

Table D.9 of commands cannot be used due to compatibility issues.

Table D.9—Non-interoperable PDL commands

Command	Reason
iUseProcNameSpace	IEEE 1149.1 does not have PDL namespaces and the associated '::' syntax.
iReset	No IEEE 1149 equivalent command.
iOverrideScanInterface	ICL-specific structure reference (ScanInterface) for which IEEE 1149 has no equivalent.
iState	No IEEE 1149 equivalent command.
iTRST	No IEEE 1687 equivalent command; JTAG TAP specific.
iTMSreset	No IEEE 1687 equivalent command; JTAG TAP specific.
iTMSidle	No IEEE 1687 equivalent command; JTAG TAP specific.
iSetInstruction	No IEEE 1687 equivalent command; BSDL specific.
iLoop	No IEEE 1687 equivalent command.
iUntil	No IEEE 1687 equivalent command.
ifTrue	No IEEE 1687 equivalent command.
ifFalse	No IEEE 1687 equivalent command.
ifEnd	No IEEE 1687 equivalent command.
iScan	Mismatch in target of command; IEEE 1149.1 is a BSDL register, while IEEE 1687 is an ICL ScanInterface.
iSetFail	Syntax same, but perform slightly different functions; IEEE 1149 works on fail/miscompare flag with conditional commands, IEEE 1687 indicates other failures. Also supported in different levels of PDL: IEEE 1149 = 0, IEEE 1687 = 1.
iGetStatus	Same syntax, but functional differences. IEEE 1149 returns a PASS/FAIL string. IEEE 1687 returns number of fails.
iGetReadData	Returns iRead data. A subset of the functionality of IEEE 1149 iGet command. PDL level-1 command.
iGetMiscompares	Returns XOR of iRead and expected data. A subset of the functionality of IEEE 1149 iGet command. PDL level-1 command.
iGet	Returns either iWrite, iRead, expected or XOR(?) data. Covers IEEE 1687 iGetReadData and iGetMiscompares commands plus extra functionality and mnemonic format.

Annex E

(informative)

Examples

One of the goals of this standard is to be sufficiently flexible to allow support of a wide variety of instruments and access mechanisms, which includes various legacy implementations. This goal may be seen as being in conflict with another desired outcome from a typical standard: forcing a single, consistent implementation. This annex is an attempt to reconcile that conflict by providing a set of examples of commonly used access mechanisms. Note that these implementations are not normative – they are merely examples of architectures and the Instrument Connectivity Language (ICL) and Procedure Description Language (PDL) used to implement them.

The conventions used in the figures in this annex are as follows:

- A module will be outlined by a dashed rectangle.
- The module name will appear in the bottom center of the rectangle.
- If a module is an instance, the instance name will follow the module name.
- A port will be denoted by a diamond-shaped symbol on the periphery of the rectangle.
- The port's width, when more than a single bit, will be indicated by a number next to a slash through the external wire connected to the port.
- The port's name will appear nearest the diamond-shaped symbol.
- The port's function is shown in the ICL, not in the figure.

E.1 Example context

The next group of examples will utilize a common structure, shown in Figure E.1, with some of the diagrammatic constructs identified with italic text. The module ("Module1") contains five instances: two of Module Instrument (instances I1 and I2), two of Module SReg1 (instances reg1 and reg2), and one of Module SIB (instance SIB1). The module also has port functions that comprise an IEEE 1687 client interface. Connections between ports are shown either with wires (as where the SO port of instance reg2 drives the SO port of Module1) or by naming (as where the SI port of Module1 is connected to the SI port of instance SIB1 because the name "SI" is used at both locations). Instance ports whose names are in light grey text may be explicitly connected (as is the case between the scan host port of instance SIB1 and the scan client port of instance reg1), or may be omitted, in which case their connections will be inferred. When ICL primitives are used (like the ScanRegister and ScanMux inside instance SIB1), they appear with grey fill.

Subsequent examples will build various network configurations using the drawing conventions shown in Figure E.1. For compactness, the ICL for the examples is cumulative; i.e., an example may instantiate modules that were defined in earlier examples. Also for compactness (and to capture the essence of ICL as an abstract language rather than a netlist language), the examples use the implicit connection scheme for the control signals.

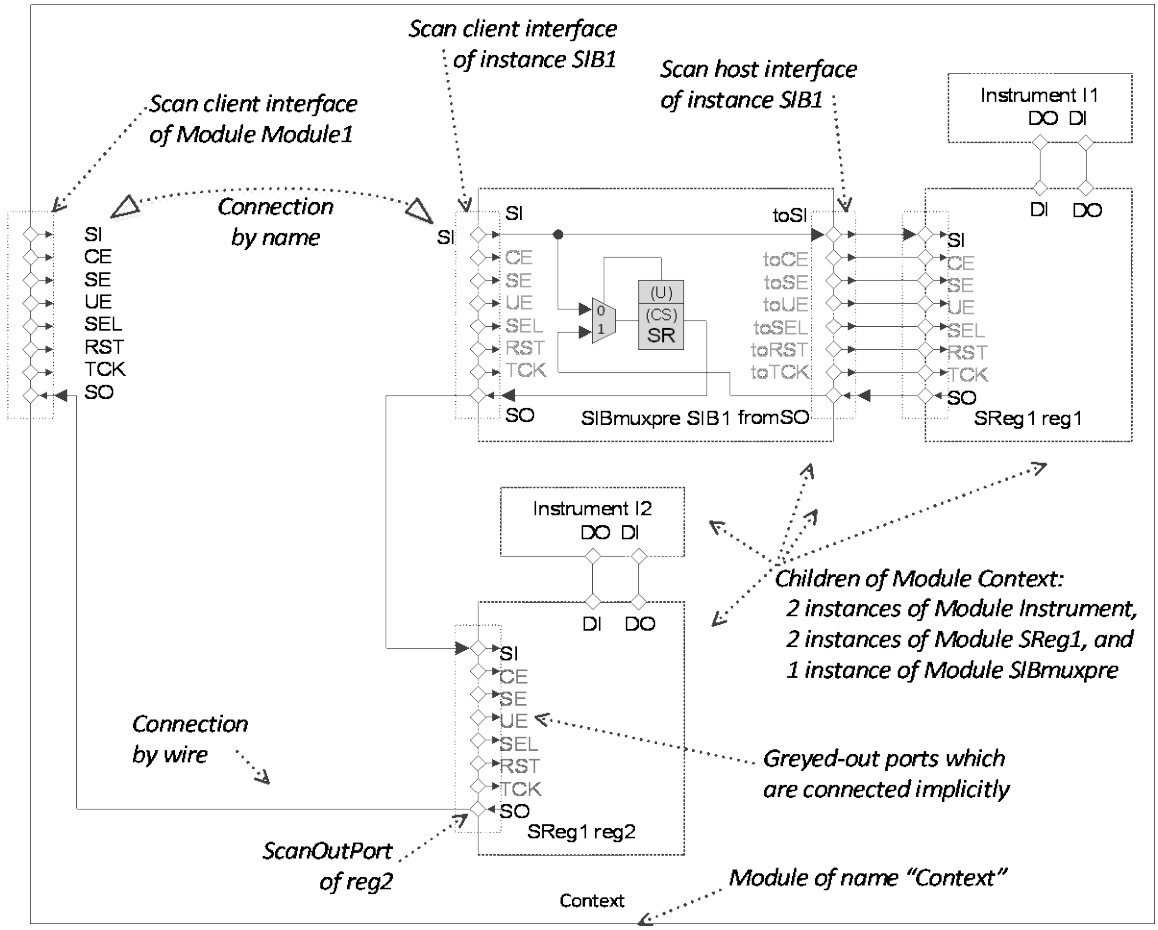


Figure E.1—Context for subsequent examples

Though the following examples will build up concepts one at a time, the ICL that corresponds to Figure E.1 is shown here for completeness:

```
Module Context {
  ScanInPort    SI;
  ScanOutPort   SO { Source  reg2.SO;}
  ShiftEnPort   SE;
  CaptureEnPort CE;
  UpdateEnPort  UE;
  SelectPort    SEL;
  ResetPort     RST;
  TCKPort       TCK;

  Instance SIB1 Of SIB_mux_pre {
    InputPort SI = SI;
    InputPort fromSO = reg1.SO;
  }
  Instance I1 Of Instrument { InputPort DI = reg1.DO; }
  Instance reg1 Of SReg {
    InputPort SI = SIB1.toSI;
    InputPort DI = I1.DO;
  }
```

```
  Instance I2 Of Instrument { InputPort DI = reg2.DO; }
  Instance reg2 Of SReg {
    InputPort SI = SIB1.SO;
    InputPort DI = I2.DO;
  }
}
```

E.2 Instrument example

A generic instrument is shown in Figure E.2. This one has an 8-bit wide DataInPort and an 8-bit wide DataOutPort, but other widths and numbers of uniquely named ports are perfectly valid.

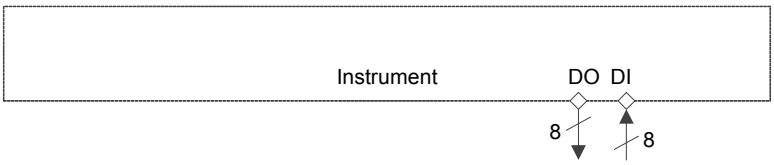


Figure E.2—Example generic instrument (8 bits wide)

The ICL for this module is as follows:

```
Module Instrument {
  DataInPort  DI[7:0];
  DataOutPort DO[7:0];
}
```

The preceding simple ICL description contains only the DataInPort and DataOutPort defintions. PDL written for this ICL would utilize only those defined ports, as in this snippet:

```
iWrite DI 0b01010000
iApply
iWrite DI[7] 0b1
iApply
iWrite DI[7] 0b0
iApply
iRead DO[1] 0b1
iApply
iRead DO[0] 0b1
iApply
iRead DO[7:2]
```

The ICL description for the instrument may have included further embellishments to improve PDL readability, as shown here:

```
  Module Instrument {

  DataInPort  DI[7:0];
  DataOutPort DO[7:0];

  Alias enable = DI[7] { RefEnum YesNo; }
  Alias mode[3:0] = DI[6:5],DI[3:2] { RefEnum Modes; }
  Alias data[2:0] = DI[4],DI[1:0];
  Alias okay = DO[0]{ RefEnum PassFail; }
  Alias done = DO[1] { RefEnum YesNo; }
```

```
  Alias count[5:0] = DO[7:2];

  Enum PassFail { Pass = 1'b1;
                  Fail = 1'b0; }
  Enum YesNo    { Yes  = 1'b1;
                  No   = 1'b0; }
  Enum Modes    { red   = 4'b0011;
                  blue  = 4'b1000;
                  green = 4'b0100; }
}
```

Note that if both versions of the module were included in the ICL, the last occurring (i.e., the embellished version) would be used. With those additions to the ICL, that same snippet of PDL could have been written as follows:

```
iWrite mode blue
iWrite enable No
iWrite data 0b100
iApply
iWrite enable Yes
iApply
iWrite enable No
iApply
iRead done Yes
iApply
iRead okay Pass
iApply
iRead count
```

E.3 Scan register example

A simple scan register is shown in Figure E.3. As in the previous example the width is 8 bits. This module includes a scan client interface for plug-and-play compatibility. The figure also shows the capture (C), shift (S), and update (U) portions of the scan register and their connections. These connections are indicated in the ICL by use of the "Source" keyword for inputs.

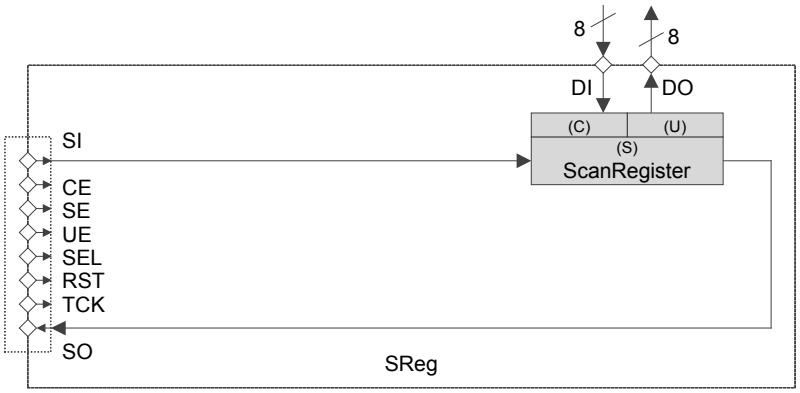


Figure E.3—Scan register example (8 bits)

The ICL for this module containing a ScanRegister is as follows:

```
Module SReg {
  ScanInPort    SI;
  ScanOutPort   SO { Source  SR[0];}
  ShiftEnPort   SE;
  CaptureEnPort CE;
  UpdateEnPort  UE;
  SelectPort    SEL;
  ResetPort     RST;
  TCKPort       TCK;
  DataInPort    DI[7:0];
  DataOutPort   DO[7:0] {Source SR; }
  ScanInterface scan_client { Port SI; Port SO; Port SEL; }

  ScanRegister SR[7:0] { ScanInSource SI;
                         CaptureSource DI;
                         ResetValue 8'b00000000; }
}
```

A more flexible version of the ICL for a parameterized-length ScanRegister (with a default value of 8 bits) is as follows:

```
Module SReg {
  Parameter MSB = 7;
  ScanInPort    SI;
  ScanOutPort   SO { Source  SR[0];}
  ShiftEnPort   SE;
  CaptureEnPort CE;
  UpdateEnPort  UE;
  SelectPort    SEL;
  ResetPort     RST;
  TCKPort       TCK;
  DataInPort    DI[$MSB:0];
  DataOutPort   DO[$MSB:0] {Source SR; }
  ScanInterface scan_client { Port SI; Port SO; Port SEL; }

  ScanRegister SR[$MSB:0] { ScanInSource SI;
                            CaptureSource DI;
                            ResetValue 'b0; }
}
```

The mechanism to call this parameterized module to create a 3-bit register would be to instantiate it as follows:

```
  …
  Instance reg3 of SReg { Parameter MSB = 2; … }
  …
```

Another method to parameterize the scan register definition would be to use the size of the register (along with some simple parameter math) instead of just using the MSB, as follows:

```
Module SReg {
  Parameter Size = 8;
  ScanInPort    SI;
```

```
  ScanOutPort   SO { Source  SR[0];}
  ShiftEnPort   SE;
  CaptureEnPort CE;
  UpdateEnPort  UE;
  SelectPort    SEL;
  ResetPort     RST;
  TCKPort       TCK;
  DataInPort    DI[$Size-1:0];
  DataOutPort   DO[$Size-1:0] {Source SR; }
  ScanInterface scan_client { Port SI; Port SO; Port SEL; }

  ScanRegister SR[$Size-1:0] { ScanInSource SI;
                               CaptureSource DI;
                               ResetValue $Size'b0; }
}
```

In this case, the mechanism to call this parameterized module to create a 3-bit register would be to instantiate it as follows:

```
  …
  Instance reg3 of SReg { Parameter Size = 3; … }
  …
```

E.4 Wrapped instrument example

The two previous examples are combined to produce a wrapped instrument (i.e., an instrument connected to a scan register with a plug-and-play interface) in Figure E.4.

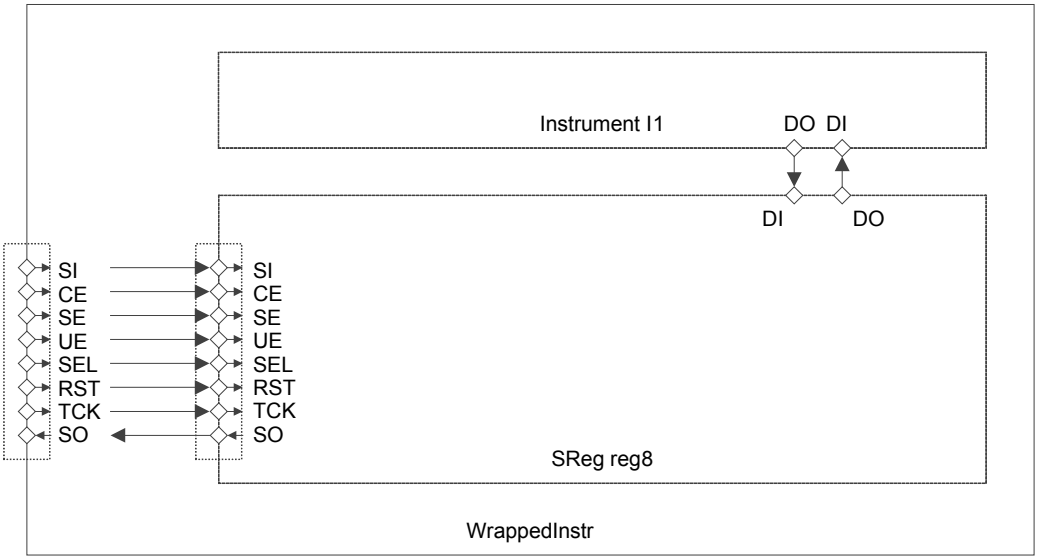


Figure E.4—Wrapped instrument example

The ICL for the wrapped instrument is as follows:

```
Module WrappedInstr {
  ScanInPort    SI;
  ScanOutPort   SO { Source reg8.SO;}
  ShiftEnPort   SE;
  CaptureEnPort CE;
  UpdateEnPort  UE;
```

```
  SelectPort    SEL;
  ResetPort     RST;
  TCKPort       TCK;
  ScanInterface scan_client { Port SI; Port SO; Port SEL; }

  Instance I1 Of Instrument { InputPort DI = reg8.DO; }
  Instance reg8 Of SReg {
    InputPort SI = SI; InputPort DI = I1.DO; Parameter Size = 8;
  }
}
```

The ICL for the WrappedInstr example contains instances (I1 and reg8) of each of the modules (Instrument and SReg) described in the previous examples.

This wrapped instrument will serve as the basis for examples of 13 different configurations, which connect three copies of this module in various methods:

- Daisy chain
- SIB
- Multiple SIBs
- ScanMuxes with local control
- ScanMuxes with remote control
- Nested SIBs (both pre- and post-mux examples)
- Exclusive access (both implicit and explicit ICL examples)
- Exclusive access with broadcast
- Broadcast access
- Branched scan chain
- Branched and merged scan chain
- IEEE Std 1500
- Slave eTAPc

E.5 Daisy-chain example

An example showing three wrapped instruments connected in a daisy chain is shown in Figure E.5. This simple structure is useful when all three instruments are accessed together.

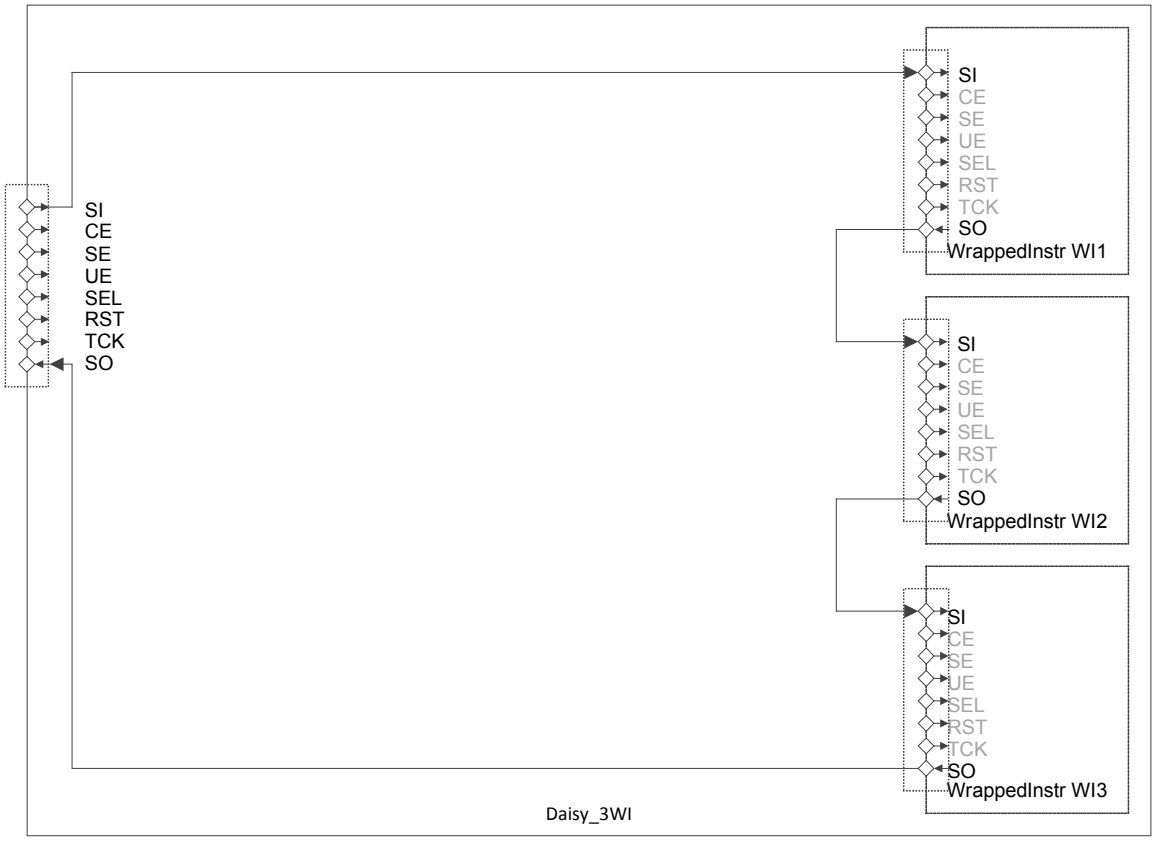


Figure E.5—Daisy Chain example

The ICL corresponding to Figure E.5 is as follows:

```
Module Daisy_3WI {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source WI3.SO; }
  Instance WI1 Of WrappedInstr { InputPort SI = SI; }
  Instance WI2 Of WrappedInstr { InputPort SI = WI1.SO; }
  Instance WI3 Of WrappedInstr { InputPort SI = WI2.SO; }
}
```

E.6 SIB_mux_pre component example

A Segment Insertion Bit (SIB) with the multiplexer preceding the control bit is shown in Figure E.6. This structure is useful for inserting or skipping a scan segment connected to the host port on the right.

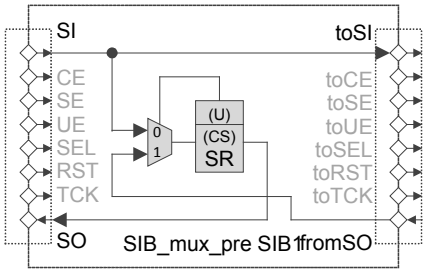


Figure E.6—SIB (mux pre reg) example

The ICL for this type of SIB (with the mux before the scan register bit) is as follows:

```
Module SIB_mux_pre {
  ScanInPort     SI;
  CaptureEnPort  CE;
  ShiftEnPort    SE;
  UpdateEnPort   UE;
  SelectPort     SEL;
  ResetPort      RST;
  TCKPort        TCK;
  ScanOutPort    SO { Source SR; }
  ScanInterface client {
    Port SI; Port CE; Port SE; Port UE;
    Port SEL; Port RST; Port TCK; Port SO;
  }
  ScanInPort       fromSO;
  ToCaptureEnPort  toCE;
  ToShiftEnPort    toSE;
  ToUpdateEnPort   toUE;
  ToSelectPort     toSEL;
  ToResetPort      toRST;
  ToTCKPort        toTCK;
  ScanOutPort      toSI { Source SI; }
  ScanInterface host {
    Port fromSO; Port toCE; Port toSE; Port toUE;
    Port toSEL; Port toRST; Port toTCK; Port toSI;
  }
  ScanRegister SR {
    ScanInSource SIBmux; CaptureSource SR; ResetValue 1'b0;
  }
  ScanMux SIBmux SelectedBy SR {
    1'b0 : SI;
    1'b1 : fromSO;
  }
}
```

E.7 Single SIB example

An example showing three wrapped instruments connected in series behind a SIB is shown in Figure E.7. This structure is useful when the three instruments are generally accessed together, but are occasionally desired to be omitted from the chain.

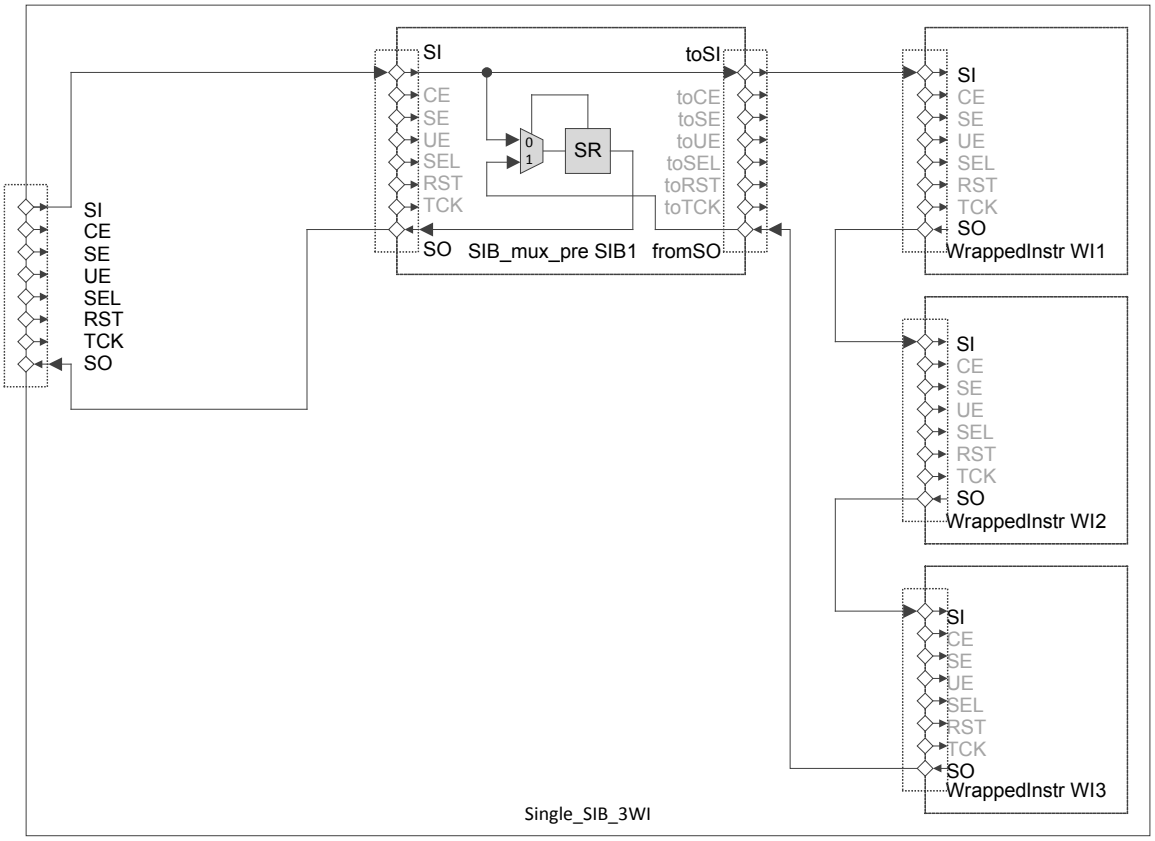


Figure E.7—Serial chain of three wrapped instruments behind a single SIB

The ICL for the single-SIB example is as follows:

```
Module Single_SIB_3WI {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source SIB1.SO; }

  Instance SIB1 Of SIB_mux_pre { InputPort SI = SI;
                                 InputPort fromSO = WI3.SO; }
  Instance WI1 Of WrappedInstr { InputPort SI = SIB1.toSI; }
  Instance WI2 Of WrappedInstr { InputPort SI = WI1.SO; }
  Instance WI3 Of WrappedInstr { InputPort SI = WI2.SO; }
}
```

E.8 Multiple SIB example

An example showing three wrapped instruments connected in series, but with each behind its own SIB is shown in Figure E.8. This structure is useful when the three instruments are generally accessed independently.

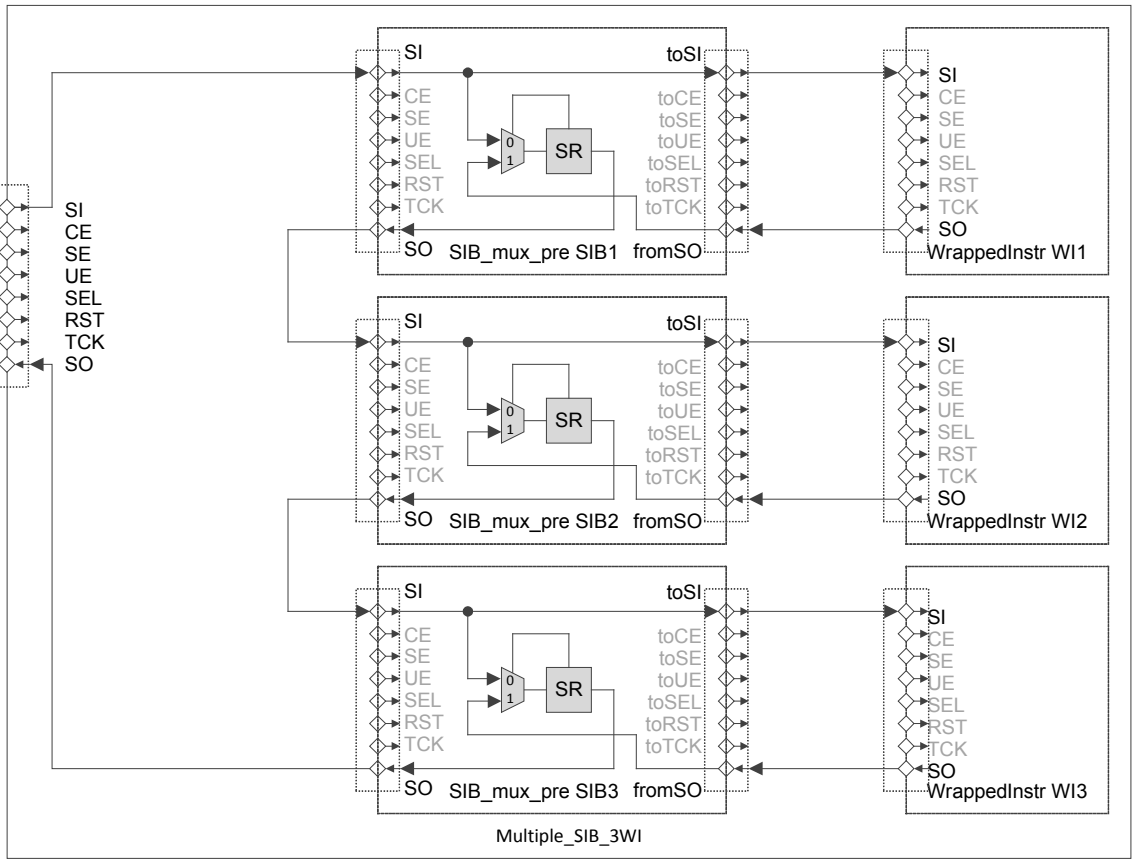


Figure E.8—Three wrapped instruments, each behind its own SIB

The ICL corresponding to Figure E.8 is as follows:

```
Module Multiple_SIB_3WI {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source SIB3.SO; }

  Instance SIB1 Of SIB_mux_pre { InputPort SI = SI;
                                 InputPort fromSO = WI1.SO; }
  Instance SIB2 Of SIB_mux_pre { InputPort SI = SIB1.SO;
                                 InputPort fromSO = WI2.SO; }
  Instance SIB3 Of SIB_mux_pre { InputPort SI = SIB2.SO;
                                 InputPort fromSO = WI3.SO; }
  Instance WI1 Of WrappedInstr { InputPort SI = SIB1.toSI; }
  Instance WI2 Of WrappedInstr { InputPort SI = SIB2.toSI; }
  Instance WI3 Of WrappedInstr { InputPort SI = SIB3.toSI; }
}
```

E.9 Scan muxes with local control example

Figure E.9 shows a similar topology to the 3-SIB example in E.8, but instead of SIBs, this example uses standalone scan multiplexers. The control signals for the muxes come from a 3-bit scan register located at the end of the same scan chain that contains the instruments.

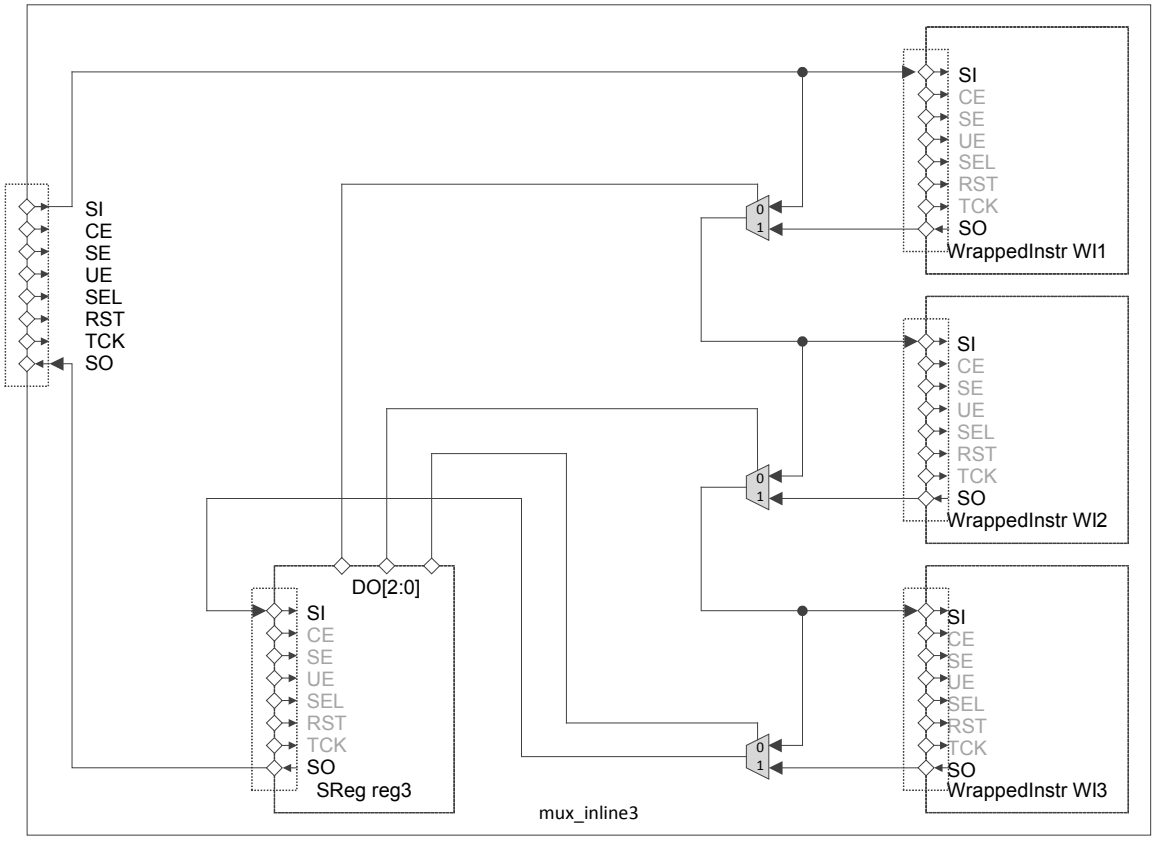


Figure E.9—Scan muxes with local control

The ICL corresponding to Figure E.9 is as follows:

```
Module mux_inline3 {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source reg3.SO; }

  Instance WI1 Of WrappedInstr { InputPort SI = SI; }
  Instance WI2 Of WrappedInstr { InputPort SI = mux1; }
  Instance WI3 Of WrappedInstr { InputPort SI = mux2; }
  Instance reg3 Of SReg { InputPort SI = mux3; Parameter Size = 3;
InputPort DI = 'b0; }
  ScanMux mux1 SelectedBy reg3.DO[2] {
    1'b0 : SI;
    1'b1 : WI1.SO;
  }
  ScanMux mux2 SelectedBy reg3.DO[1] {
    1'b0 : mux1;
    1'b1 : WI2.SO;
  }
  ScanMux mux3 SelectedBy reg3.DO[0] {
    1'b0 : mux2;
    1'b1 : WI3.SO;
  }
}
```

E.10 Scan muxes with remote control example

Figure E.10 shows the identical topology as the 3-scan-mux example in E.9, but instead of the mux control register being part of the same scan chain as the instruments, this example places that register on a separate portion of the serial access network.

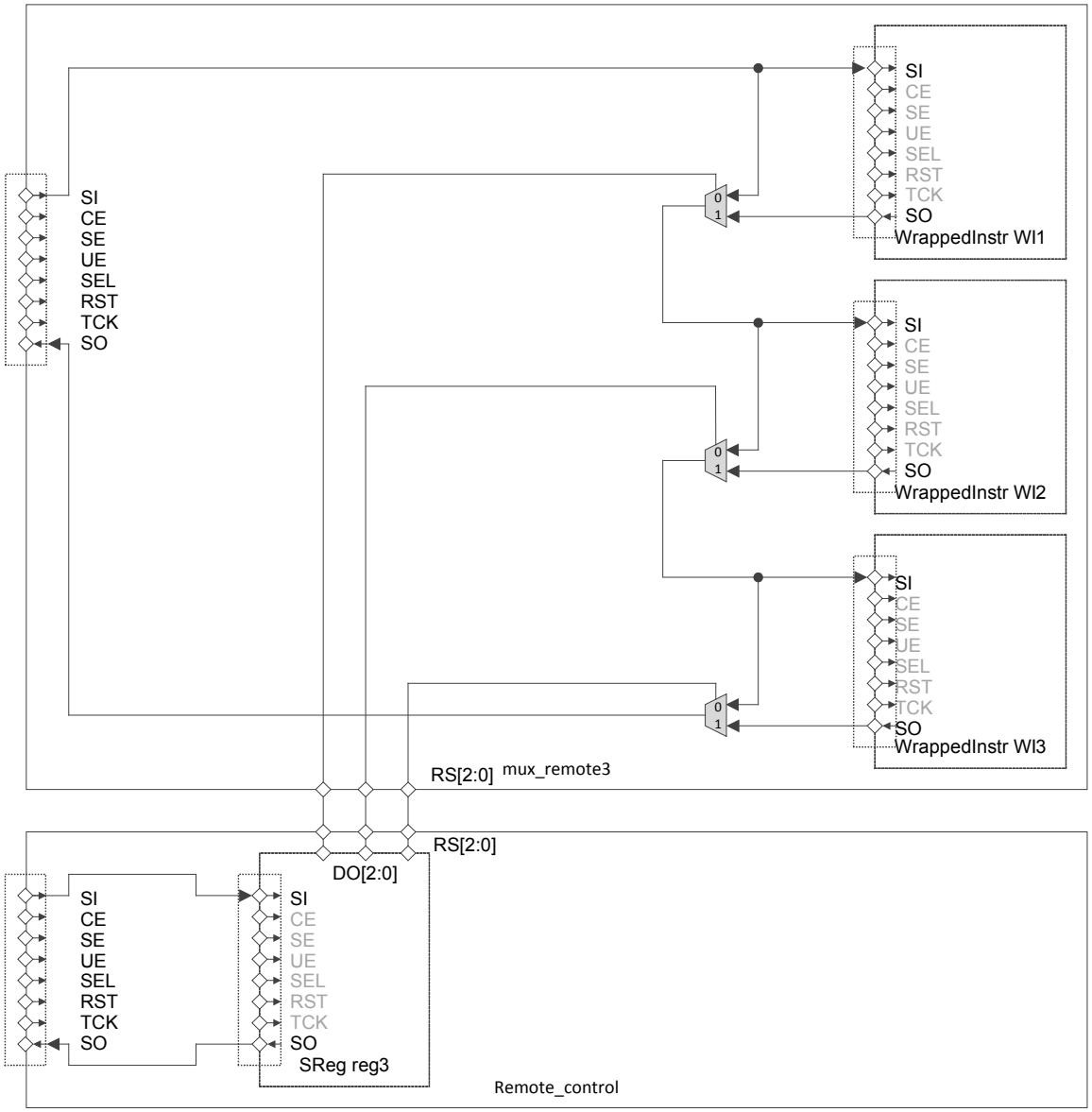


Figure E.10—Scan muxes with remote control

The ICL corresponding to Figure E.10 is as follows:

```
Module mux_remote3 {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source mux3; } DataInPort RS[2:0];

  Instance WI1 Of WrappedInstr { InputPort SI = SI; }
  Instance WI2 Of WrappedInstr { InputPort SI = mux1; }
```

```
  Instance WI3 Of WrappedInstr { InputPort SI = mux2; }
  ScanMux mux1 SelectedBy RS[2] {
    1'b0 : SI;
    1'b1 : WI1.SO;
  }
  ScanMux mux2 SelectedBy RS[1] {
    1'b0 : mux1;
    1'b1 : WI2.SO;
  }
  ScanMux mux3 SelectedBy RS[0] {
    1'b0 : mux2;
    1'b1 : WI3.SO;
  }
}
Module remote_control {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source reg3.SO; }
  DataOutPort RS[2:0] { Source reg3.DO[2:0]; }

  Instance reg3 Of SReg { InputPort SI = SI; Parameter Size = 3;}
}
```

Note that there are two distinct scan interfaces on the left side of the figure: one for the network with the instruments (in Module mux_remote3), and another for the control register (in Module remote_control). This allows the integrator some options when connecting them; namely, a daisy chain of the two interfaces (which would degenerate to example E.9), or the use of two separate networks (which would allow the control bits to be set by always shifting a dedicated (3-bit) scan chain rather than by shifting through the wrapped instruments in a single (variable-length) scan chain in the daisy-chain case. The cost of using this type of a dedicated separate scan chain, however, is that two scan operations (one for the control bits, the other for the wrapped instruments) are required, which will potentially involve extra overhead to change which scan chain is selected. Contrast this with the daisy-chain approach, where the control bits and the wrapped instruments are accessed in a single scan operation.

E.11 Nested SIB example: mux_pre

Figure E.11 shows an example of three wrapped instruments connected by SIBs in a nested fashion; i.e., access to the second instrument involves access to the first, and access to the third instrument involves access to both the first and the second. This structure is useful when the three instruments are related and the downstream instruments are only accessed when needed by the upstream instruments.

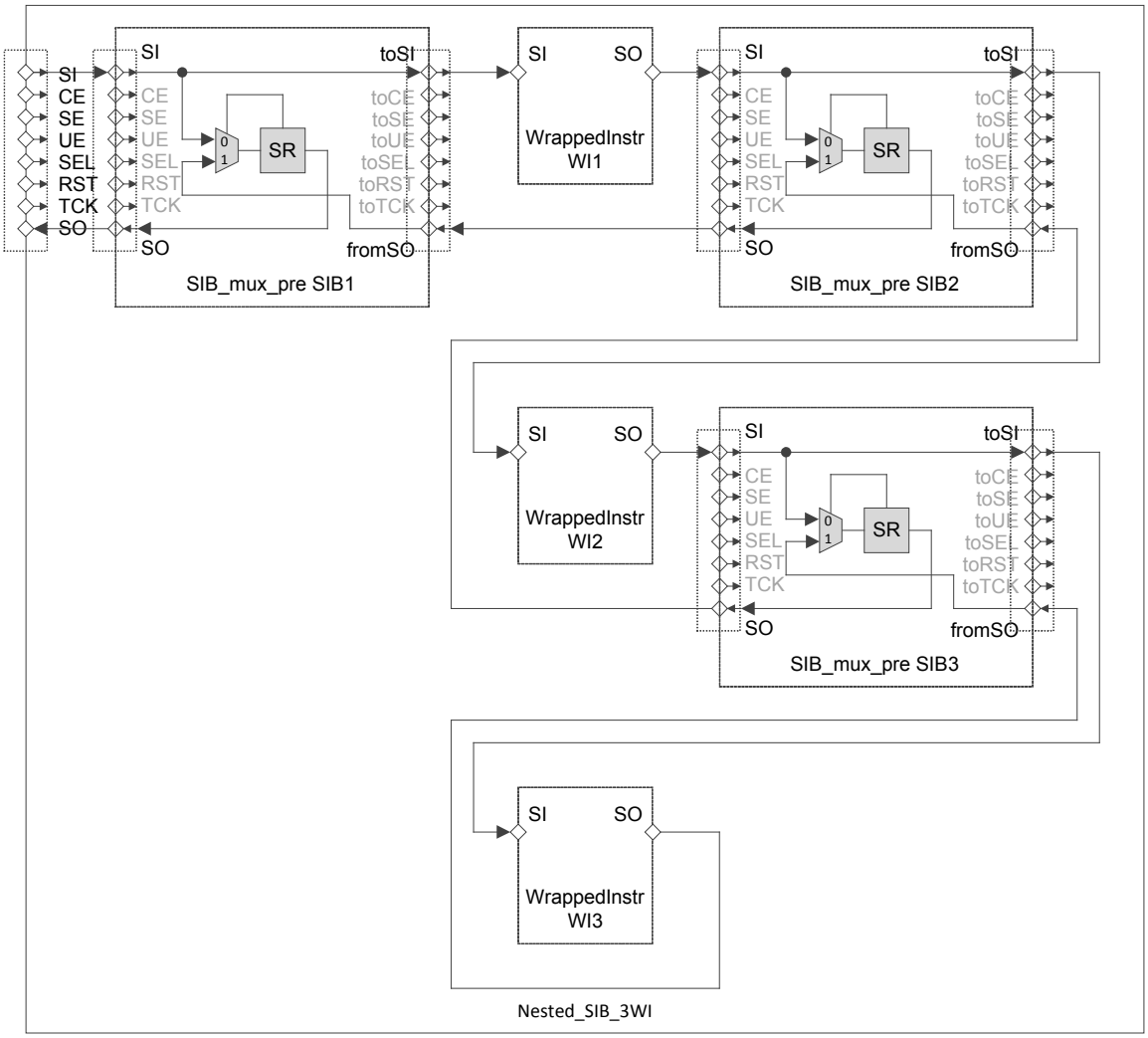


Figure E.11—Nested SIBs: mux_pre

The ICL corresponding to Figure E.11 is as follows:

```
Module Nested_SIB_3WI {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source SIB3.SO; }

  Instance SIB1 Of SIB_mux_pre { InputPort SI = SI;
                                 InputPort fromSO = SIB2.SO; }
  Instance SIB2 Of SIB_mux_pre { InputPort SI = WI1.SO;
                                 InputPort fromSO = SIB3.SO; }
  Instance SIB3 Of SIB_mux_pre { InputPort SI = WI2.SO;
                                 InputPort fromSO = WI3.SO; }
  Instance WI1 Of WrappedInstr { InputPort SI = SIB1.toSI; }
  Instance WI2 Of WrappedInstr { InputPort SI = SIB2.toSI; }
  Instance WI3 Of WrappedInstr { InputPort SI = SIB3.toSI; }
}
```

E.12 BAD Nested SIB example: mux_post

Figure E.12 duplicates the previous example, but uses the SIB_mux_post version of the SIB in order to illustrate the weakness of this SIB version. Specifically, consider the case when all three wrapped instruments are active: tracing the ScanOut path backward from the interface at the left of the BAD_Nested_SIB_3WI module reveals that the three scan muxes are in series (one per SIB). Deeper levels of nesting would result in correspondingly deeper numbers of series mux connections, which will ultimately cause setup timing closure problems at a given TCK frequency. Contrast this with the backward trace of the ScanOut path in the previous example: there is always a scan register between scan muxes, thus making a considerably easier timing path for the ScanOut data.

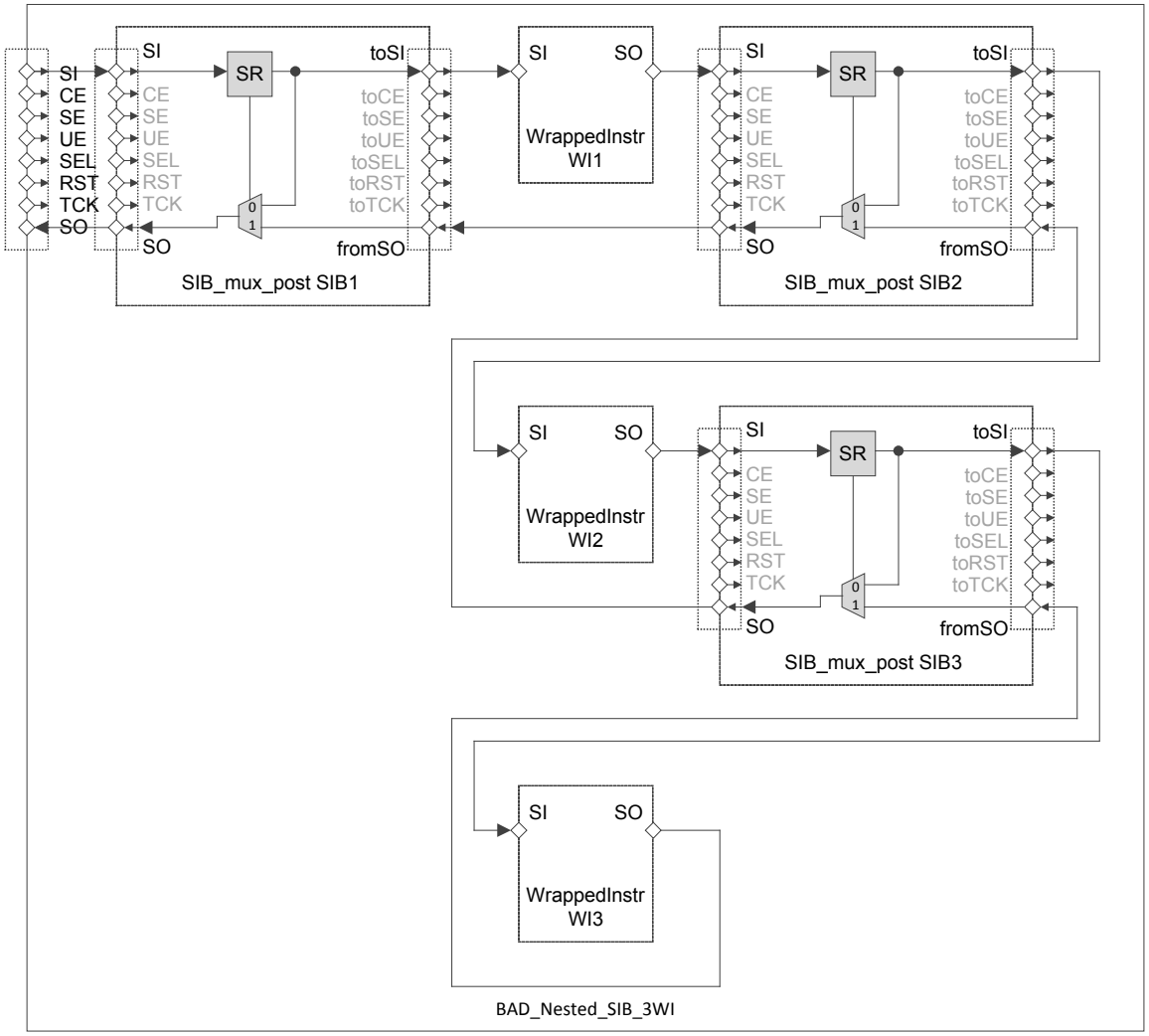


Figure E.12—BAD Nested SIBs: mux_post

The ICL corresponding to Figure E.12 is as follows:

```
Module BAD_Nested_SIB_3WI {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source SIB3.SO; }

  Instance SIB1 Of SIB_mux_post { InputPort SI = SI;
                                  InputPort fromSO = SIB2.SO; }
```

```
  Instance SIB2 Of SIB_mux_post { InputPort SI = WI1.SO;
                                  InputPort fromSO = SIB3.SO; }
  Instance SIB3 Of SIB_mux_post { InputPort SI = WI2.SO;
                                  InputPort fromSO = WI3.SO; }
  Instance WI1 Of WrappedInstr { InputPort SI = SIB1.toSI; }
  Instance WI2 Of WrappedInstr { InputPort SI = SIB2.toSI; }
  Instance WI3 Of WrappedInstr { InputPort SI = SIB3.toSI; }
}
```

E.13 Exclusive access example: implicit ICL

Figure E.13 shows a topology where at most one of the three instruments can be accessed at a time. This structure is useful when only a single instrument is of interest at any given time. Note that the diagram and the associated ICL make use of implicit gating of the SelectPort (as will become apparent when contrasted with the next example). Specifically, the SEL ports of each of the three WI instances are produced implicitly by gating the SEL port of the parent module (Exclusive) with the associated decode of the DO[1:0] signals used to select the active SO signal.

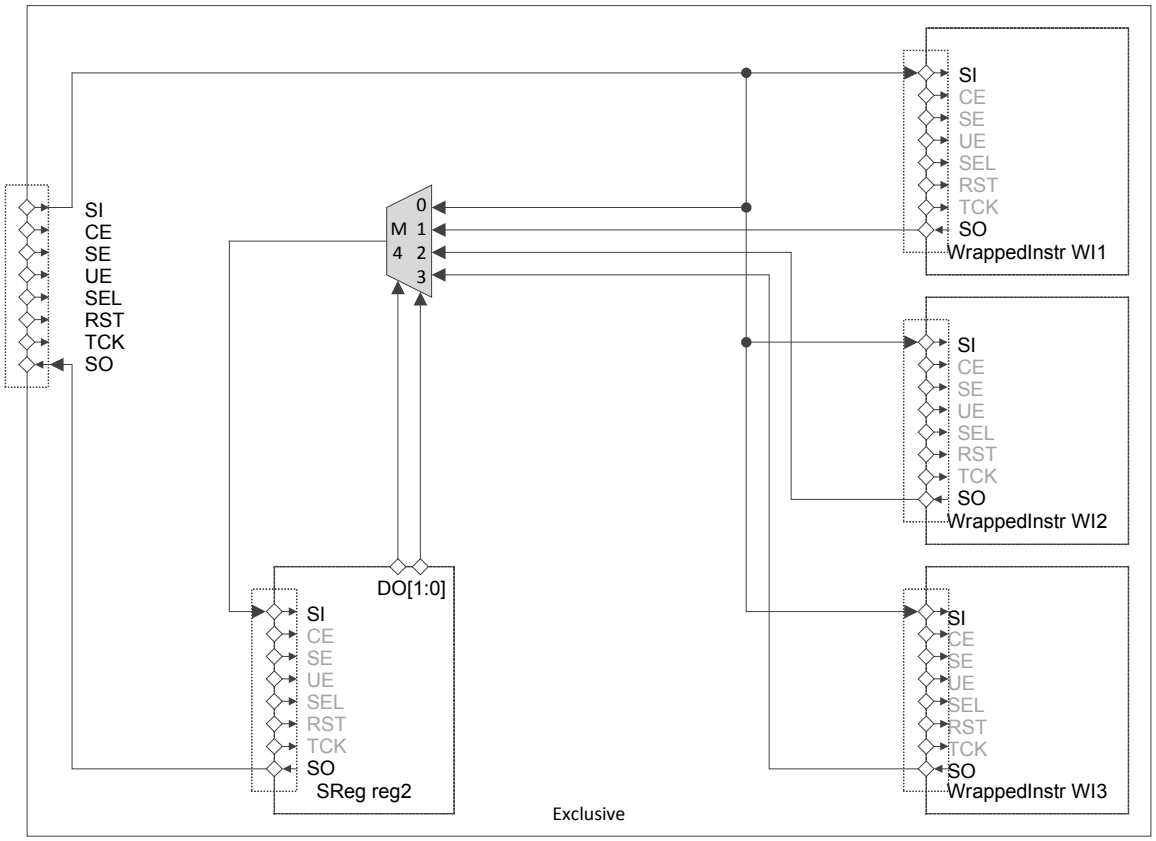


Figure E.13—Exclusive access: implicit ICL

The ICL corresponding to Figure E.13 is shown as follows. Note that the logic associated with the SelectPort of each of the WI instances is implicit and must be deduced by the retargeting tool.

```
Module Exclusive {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source reg2[0]; }
```

```
  Instance WI1 Of WrappedInstr { InputPort SI = SI; }
  Instance WI2 Of WrappedInstr { InputPort SI = SI; }
  Instance WI3 Of WrappedInstr { InputPort SI = SI; }

  ScanMux M4 SelectedBy reg2 {2'b00 : SI;
                              2'b01 : WI1.SO;
                              2'b10 : WI2.SO;
                              2'b11 : WI3.SO;
  }
  ScanRegister reg2[1:0] {ScanInSource M4;
                          CaptureSource 2'b00;
                          ResetValue 2'b00;
  }
}
```

E.14 Exclusive access example: explicit ICL

Figure E.14 duplicates the previous example, but explicitly shows how the SEL ports of each of the three WI instances are produced.

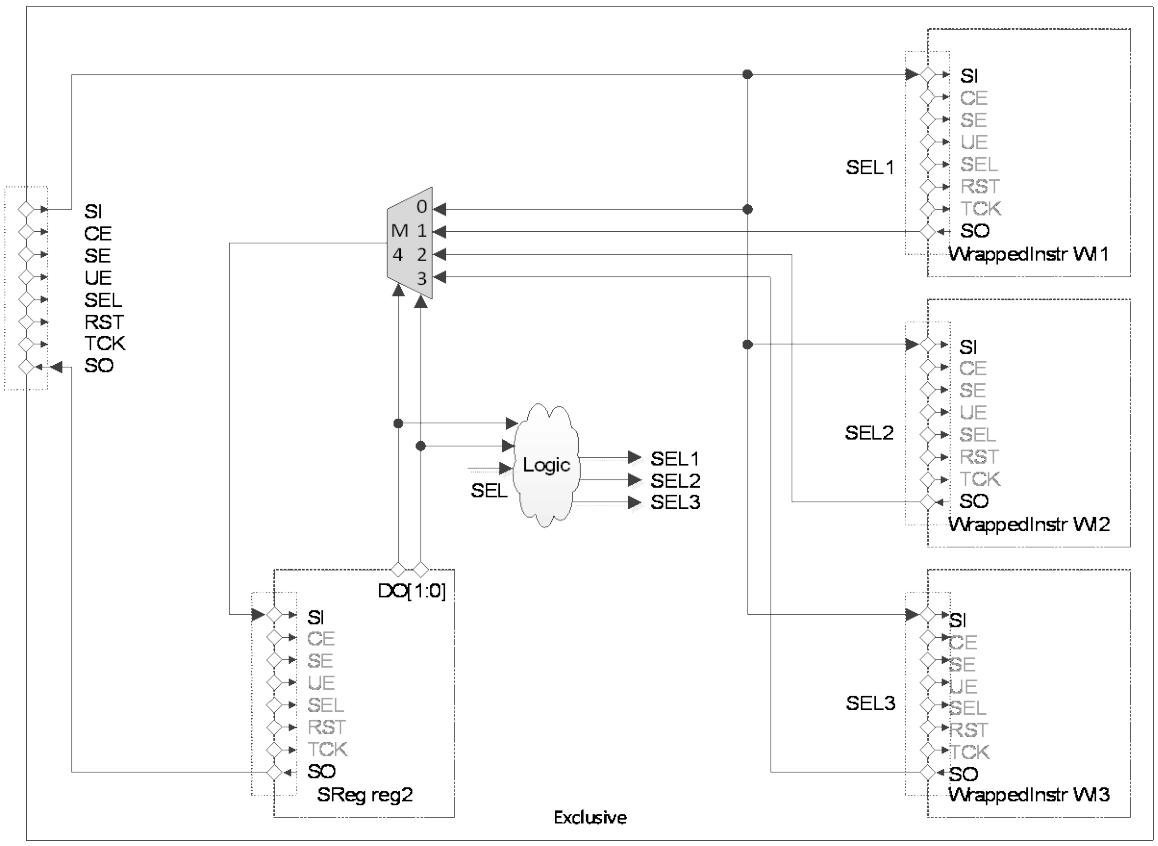


Figure E.14—Exclusive access: explicit ICL

The ICL corresponding to Figure E.14 is as follows. Note that the logic associated with the SelectPort of each of the WI instances is explicit, but could be omitted (as in the preceding example) and deduced to exist.

```
Module Exclusive {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source reg2[0]; }

  Instance WI1 Of WrappedInstr {
                InputPort SI = SI;
                InputPort SEL = SEL1;
  }
  Instance WI2 Of WrappedInstr {
                InputPort SI = SI;
                InputPort SEL = SEL2;
  }
  Instance WI3 Of WrappedInstr {
                InputPort SI = SI;
                InputPort SEL = SEL3;
  }
  // This logic matches the SO selection logic of the mux
  LogicSignal SEL1 {(SEL,reg2[1],reg2[0]) == 3'b101;}
  LogicSignal SEL2 {(SEL,reg2[1],reg2[0]) == 3'b110;}
  LogicSignal SEL3 {(SEL,reg2[1],reg2[0]) == 3'b111;}
  ScanMux M4 SelectedBy reg2 {2'b00 : SI;
                              2'b01 : WI1.SO;
                              2'b10 : WI2.SO;
                              2'b11 : WI3.SO;
  }
  ScanRegister reg2[1:0] {ScanInSource M4;
                          CaptureSource 2'b00;
                          ResetValue 2'b00;
  }
}
```

E.15 Exclusive access with broadcast example

Figure E.15 extends the previous example by adding broadcast functionality so that the shared SI port can be used to simultaneously load the three instances. The output is taken from the bypass path or from one of the instances. Note that the control register has an additional bit that controls the broadcast mode.

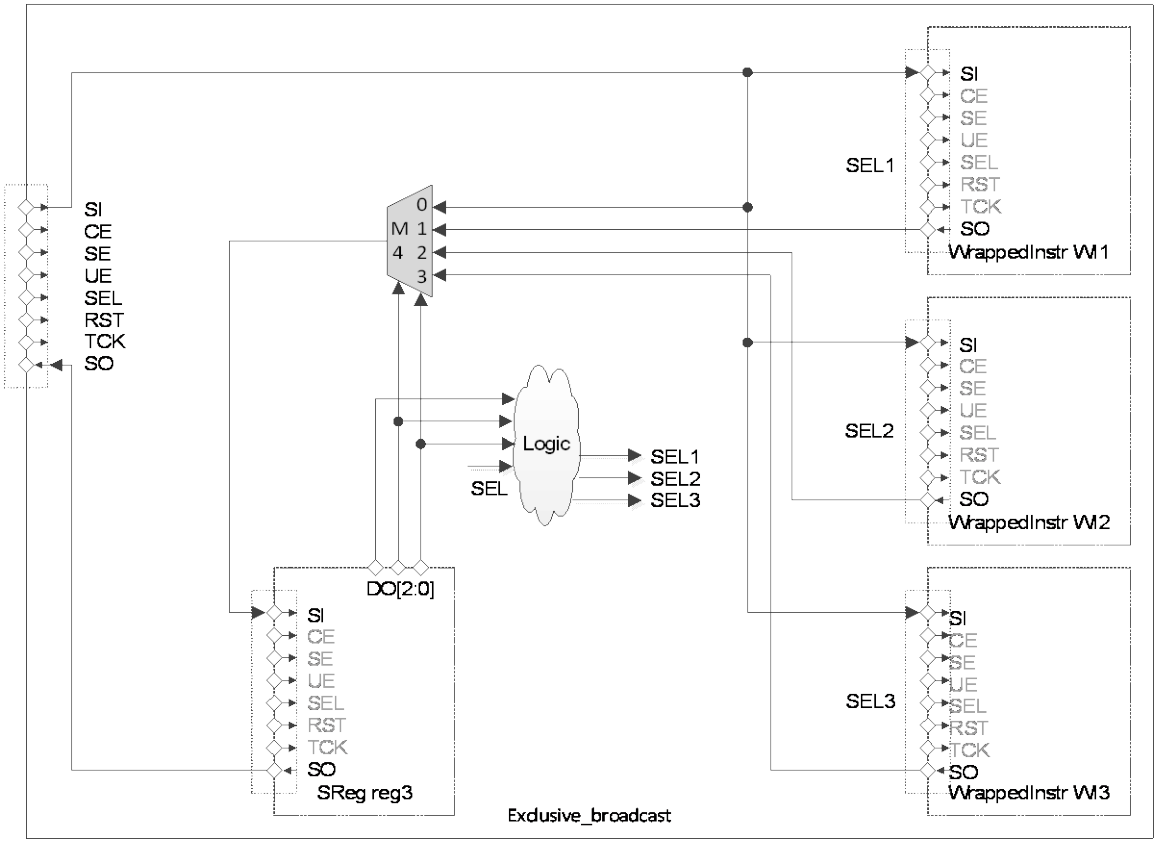


Figure E.15—Exclusive access with broadcast

The ICL corresponding to Figure E.15 is as follows. Because of the presence of the AllowBroadcastOnScanInterface property in each of the WI instances, the logic associated with the SelectPort of each instance must be shown explicitly.

```
Module Exclusive_broadcast {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source reg2[0]; }

  Instance WI1 Of WrappedInstr { InputPort SI = SI;
                 InputPort SEL = SEL1;
                 AllowBroadcastOnScanInterface scan_client;
  }
  Instance WI2 Of WrappedInstr { InputPort SI = SI;
                 InputPort SEL = SEL2;
                 AllowBroadcastOnScanInterface scan_client;
  }
  Instance WI3 Of WrappedInstr { InputPort SI = SI;
                 InputPort SEL = SEL3;
                 AllowBroadcastOnScanInterface scan_client;
  }
  // Select if in broadcast mode or the SO of this instance is selected
  LogicSignal SEL1 {SEL & (reg2[2] | (~reg2[1] & reg2[0]));}
  LogicSignal SEL2 {SEL & (reg2[2] | (reg2[1] & ~reg2[0]));}
  LogicSignal SEL3 {SEL & (reg2[2] | (reg2[1] & reg2[0]));}
  ScanMux M4 SelectedBy reg2 {2'b00 : SI;
                              2'b01 : WI1.SO;
```

```
                              2'b10 : WI2.SO;
                              2'b11 : WI3.SO;
  }
  ScanRegister reg2[1:0] {ScanInSource M4;
                          CaptureSource 2'b00;
                          ResetValue 2'b00;
  }
}
```

E.16 Broadcast or daisy-chain example

Figure E.16 illustrates a broadcast configuration where the three instruments can be loaded simultaneously via broadcast or can be accessed in a daisy chain, depending on the control bit in the 1-bit scan register. This structure is useful to reduce load time when identical copies of instruments are present. Note that during the broadcast scan load, only the last instrument in the chain produces output data that is consumed, and to access either of the first two instances that broadcast mode must be disabled.

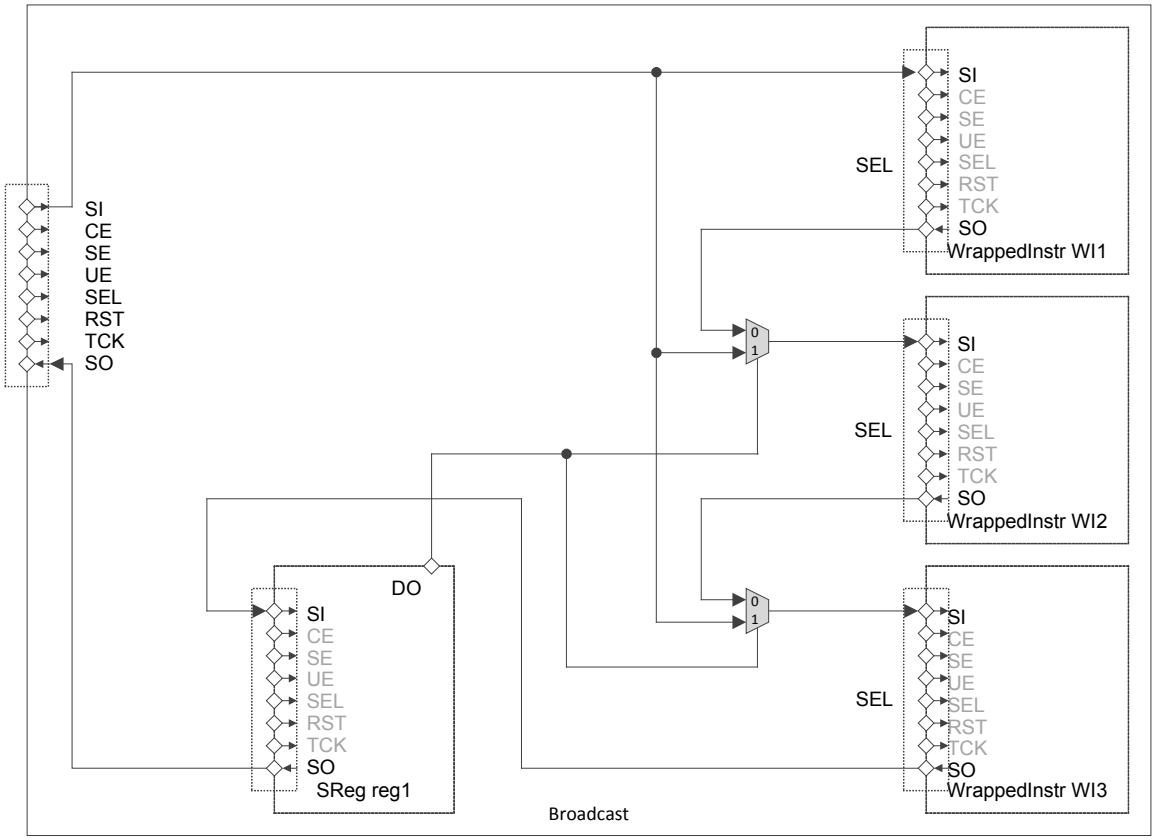


Figure E.16—Broadcast scan example

The ICL corresponding to Figure E.16 is as follows. Note that the instantiations of the wrapped instruments include the directive to allow broadcast on their scan interfaces (which is defined in the WrappedInstr module), as well as explicit definition of their SelectPort connections. Note that in the configuration shown, the SEL ports of the three instances can be shared (i.e., driven directly from the SEL port of the parent module) since they are all active during broadcast and all active during daisy-chain mode.

```
Module Broadcast {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
```

```
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source reg1.SO; }

  Instance WI1 Of WrappedInstr { InputPort SI = SI;
                 InputPort SEL = SEL;
                 AllowBroadcastOnScanInterface scan_client; }
  Instance WI2 Of WrappedInstr { InputPort SI = inmux2;
                 InputPort SEL = SEL;
                 AllowBroadcastOnScanInterface scan_client; }
  Instance WI3 Of WrappedInstr { InputPort SI = inmux3;
                 InputPort SEL = SEL;
                 AllowBroadcastOnScanInterface scan_client; }
  Instance reg1 Of SReg { InputPort SI = WI3.SO; Parameter Size = 1;
                          InputPort DI = 'b0; }
  ScanMux inmux2 SelectedBy reg1.DO {
    1'b0 : WI1.SO;
    1'b1 : SI;
  }
  ScanMux inmux3 SelectedBy reg1.DO {
    1'b0 : WI2.SO;
    1'b1 : SI;
  }
}
```

E.17 Branched scan chain example

Figure E.17 illustrates a scan fan-out configuration where there is a branch in the scan chain (just downstream from the ScanIn port, but the branches could occur at some internal point in the scan chain). Though this superficially resembles the broadcast configuration in the previous example, there is an important difference: this example features scan segments of different lengths, which will require the retargeting software to manage padding the payloads appropriately.

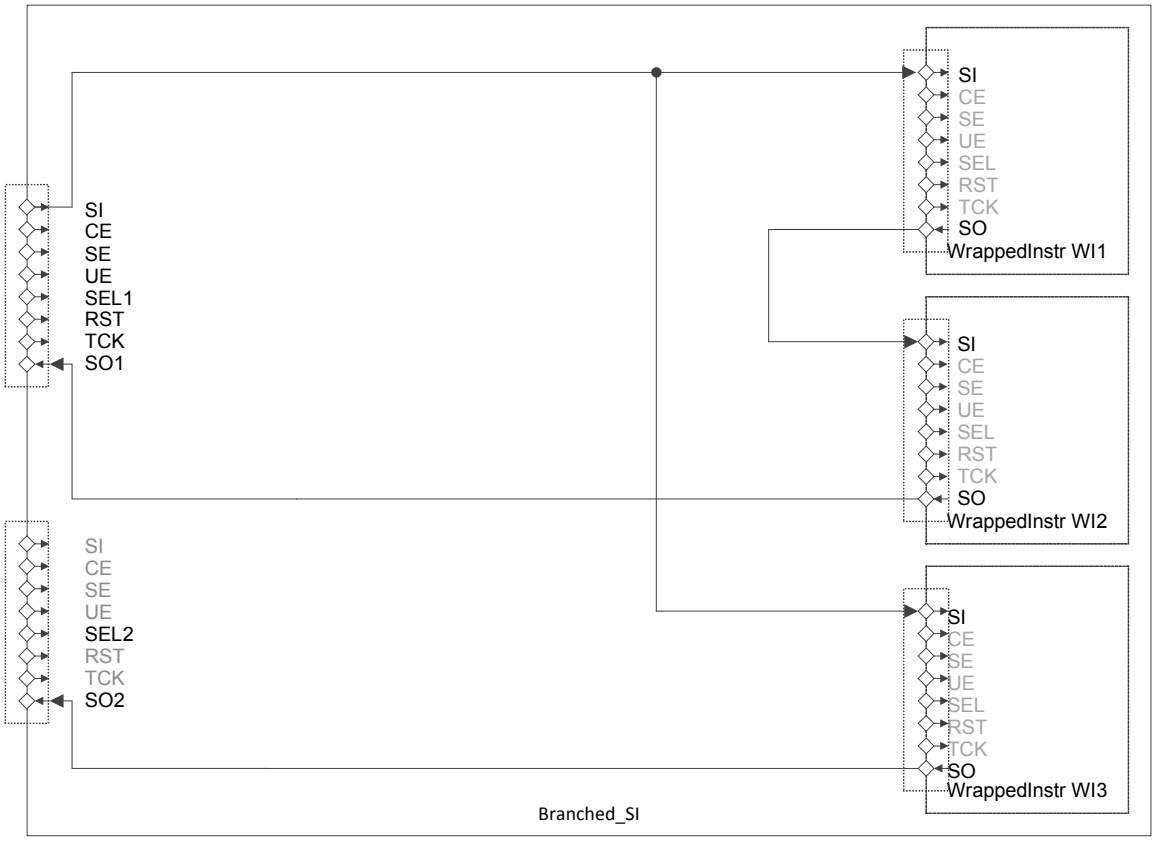


Figure E.17—Branched scan chain with unique ScanOutPorts

The ICL corresponding to Figure E.17 is as follows. The ScanInterface declarations are necessary because there is more than one of them on the parent module. Also note that the SI port need not be listed in the interfaces, since there is only one SI port that is shared by both.

```
Module Branched_SI {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
  UpdateEnPort UE; SelectPort SEL1; ResetPort RST;
  TCKPort TCK; ScanOutPort SO1 { Source WI2.SO; }
  SelectPort SEL2; ScanOutPort SO2 { Source WI3.SO; }
  ScanInterface scan1 { Port SEL1; Port SO1; }
  ScanInterface scan2 { Port SEL2; Port SO2; }

  Instance WI1 Of WrappedInstr { InputPort SI = SI; }
  Instance WI2 Of WrappedInstr { InputPort SI = WI1.SO; }
  Instance WI3 Of WrappedInstr { InputPort SI = SI; }
}
```

E.18 Branched-then-merged scan chain example

Figure E.18 illustrates another scan fan-out configuration that also contains a scan multiplexer to merge the branched segments back into a single chain within the parent module. Since only one of the branches can be active at a time, there is implied gating to produce the SEL signals for the instances: the SEL port for WI1 and WI2 comes from the AND of the parent SEL port with the inverse of the value in reg1; the SEL port for WI3 comes from the AND of the parent SEL port with the value in reg1.

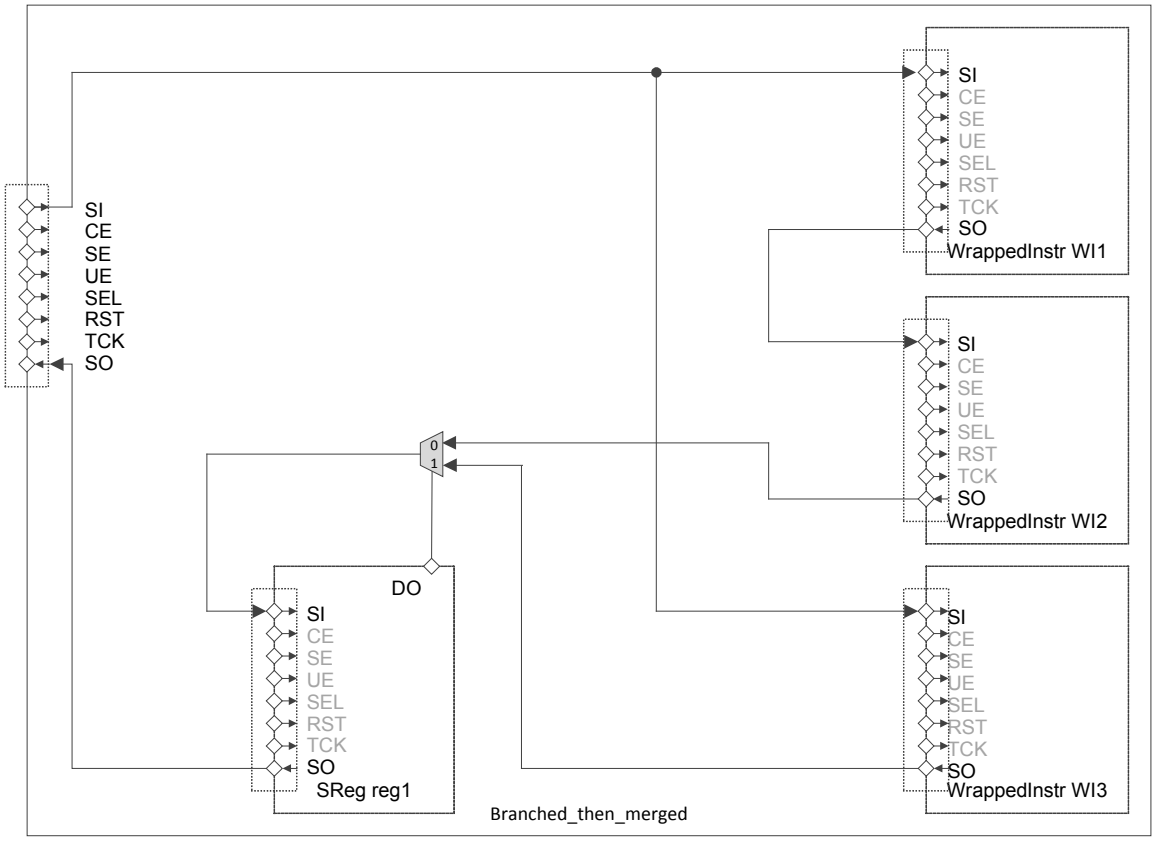


Figure E.18—Branched-then-merged scan chain example

The ICL corresponding to Figure E.18 is as follows. Note that since only WI1 and WI3 share a common source for their SI ports, broadcast could only be used for those two instances. However, performing a broadcast exposes the corner case when reg1 contains a zero: the SO port of the parent module will be driven by the 16-bit segment while doing an 8-bit iWrite to WI3 in broadcast mode. There must be 8 bits downstream from WI2 to hold those extra bits—one of them is provided by reg1, but the other seven need to exist downstream from the parent module.

```
Module Branched_then_merged {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source reg1.SO; }

  Instance WI1 Of WrappedInstr { InputPort SI = SI; }
  Instance WI2 Of WrappedInstr { InputPort SI = WI1.SO; }
  Instance WI3 Of WrappedInstr { InputPort SI = SI; }

  Instance reg1 Of SReg { InputPort SI = mux; Parameter Size = 1;
                          InputPort DI = 'b0; }
  ScanMux mux SelectedBy reg1.DO {
    1'b0 : WI2.SO;
    1'b1 : WI3.SO;
  }
}
```

E.19 IEEE 1500 wrapper serial port

Figure E.19 shows a simple IEEE 1500 configuration, with the three wrapped instruments treated as Wrapper Data Registers (WDRs). Note that the SEL ports of the registers will have logic that matches the SO mux select conditions inferred. Also note that the interface to this instrument is not strictly plug-and-play due to the presence of the SWIR input in the upper left (which is a DataInPort, and thus not part of the scan client interface).

For the purpose of illustrating another style of ICL, the Wrapper Bypass Register (WBY) and the Wrapper Instruction Register (WIR) are built from ScanRegister primitives instead of from instances of SReg modules. With this style, the notion of the implicit Update stage of the ScanRegister becomes apparent in the WIR: the scan chain flows from left to right through the two bits of the WIR ScanRegister, and the data signal from the implicit update stage of each bit (which controls the mux) is shown coming from the bottom of the WIR ScanRegister. With respect to the ICL, this modeling approach requires careful interpretation to understand the distinction between the scan chain and the implicit update stage. For example, the name of the WIR[0] ScanRegister is used in two different contexts: the scan connection flows through the lower data input of the IR_MUX, while it is the implicit update stage that controls the DR_MUX. Had the WIR in this example been coded with an instance of an SReg module rather than a ScanRegister, the use of different ports (SO and DO, respectively) would have made these connections explicit and obvious.

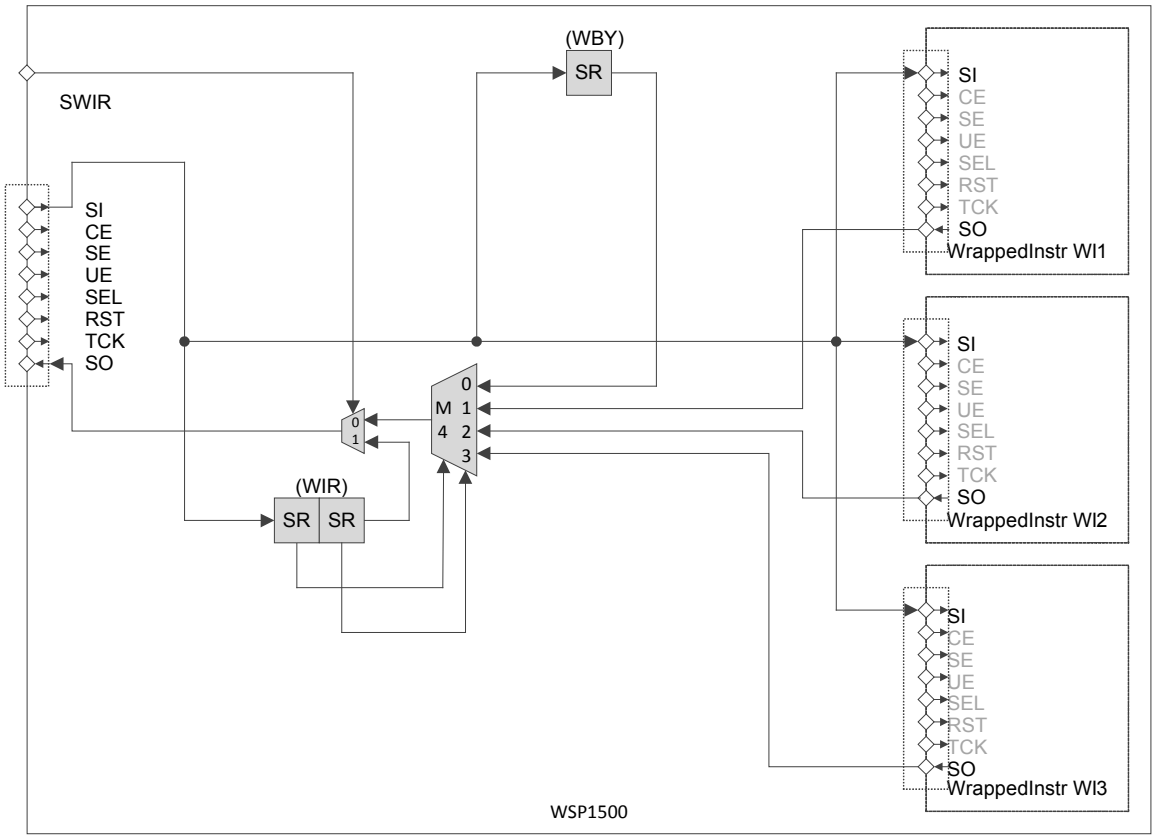


Figure E.19—IEEE 1500 Wrapper Serial Port

The ICL for Figure E.19 is as follows:

```
Module WSP1500 {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
```

```
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source IR_MUX; }
  DataInPort SWIR;

  ScanRegister WIR[1:0] {ScanInSource SI; ResetValue 2'b0;
                         CaptureSource 2'b01;}
  ScanRegister WBY      {ScanInSource SI; CaptureSource 1'b0;}

  Instance WI1 Of WrappedInstr { InputPort SI = SI; }
  Instance WI2 Of WrappedInstr { InputPort SI = SI; }
  Instance WI3 Of WrappedInstr { InputPort SI = SI; }

  ScanMux DR_MUX SelectedBy WIR[1:0] {2'b00 : WBY;
                                      2'b01 : WI1.SO;
                                      2'b10 : WI2.SO;
                                      2'b11 : WI3.SO;
  }
  ScanMux IR_MUX SelectedBy SWIR {1'b0 : DR_MUX;
                                  1'b1 : WIR[0];

  }
}
```

E.20 IEEE 1500 WSP with SWIR bit included

Figure E.20 shows a simple IEEE 1500 configuration similar to the previous example, but with the Select WIR (SWIR) bit included in the parent module. As previously, the SEL ports of the registers (except SWIR) will have implicit logic that matches the SO mux select conditions; the SEL port of the SWIR bit is connected to the SEL port of the parent module, since it is always on the active scan chain. The advantage of this example over the previous example is that the interface is a simple plug-and-play scan client, since the SWIR signal is generated internally. As with the previous example, ScanRegister primitives are used here, requiring careful distinction between the scan chain and the implicit update stage path, as further illustrated here with the SWIR bit.

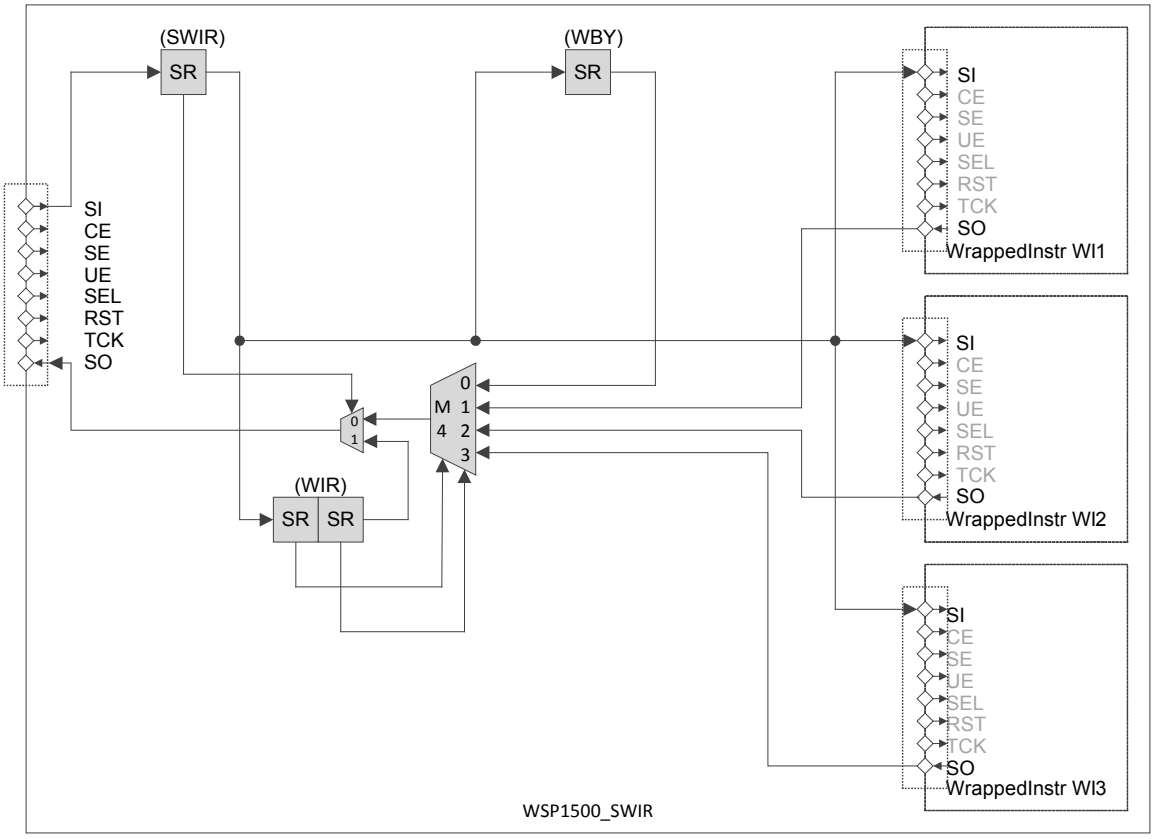


Figure E.20—IEEE 1500 Wrapper Serial Port with Select WIR register

The ICL for Figure E.20 is as follows:

```
Module WSP1500_SWIR {
  ScanInPort SI; CaptureEnPort CE; ShiftEnPort SE;
  UpdateEnPort UE; SelectPort SEL; ResetPort RST;
  TCKPort TCK; ScanOutPort SO { Source IR_MUX; }

  ScanRegister SWIR {ScanInSource SI; ResetValue 1'b1;}
  ScanRegister WBY {ScanInSource SWIR; ResetValue 1'b1;}
  Instance WIR Of SReg { InputPort SI = SWIR; Parameter Size = 2;}

  Instance WI1 Of WrappedInstr { InputPort SI = SWIR; }
  Instance WI2 Of WrappedInstr { InputPort SI = SWIR; }
  Instance WI3 Of WrappedInstr { InputPort SI = SWIR; }

  ScanMux DR_MUX SelectedBy WIR[1:0] {2'b00 : WBY;
                                      2'b01 : WI1.SO;
                                      2'b10 : WI2.SO;
                                      2'b11 : WI3.SO;
  }
  ScanMux IR_MUX SelectedBy SWIR {1'b0 : DR_MUX;
                                  1'b1 : WIR[0];

  }
}
```

E.21 Single embedded TAP controller (eTAPC) example

Figure E.21 illustrates the use of an embedded TAP (eTAP) as the interface to the instruments. Note that the interface on the left side of the parent module is no longer a plug-and-play scan client interface as in the previous examples, but instead it is a plug-and-play TAP client interface. The SEL signal will be driven from a decode of an instruction in the device-level TAP (not shown) to activate the network shown in this figure. Implicitly, the SEL port also gates the TMS of the eTAPC instance. When selected, the eTAPC's eTDO port will drive the device TDO pin. The TDO output of the device-level TAP does not drive the device TDO pin (neither shown); instead, it drives the eTDI port of this module.

The right-hand side of the eTAPC is a plug-and-play scan host interface and may be used to connect to the three WrappedInstr instances in any of the configurations shown in the previous examples; a daisy chain is shown here for simplicity.

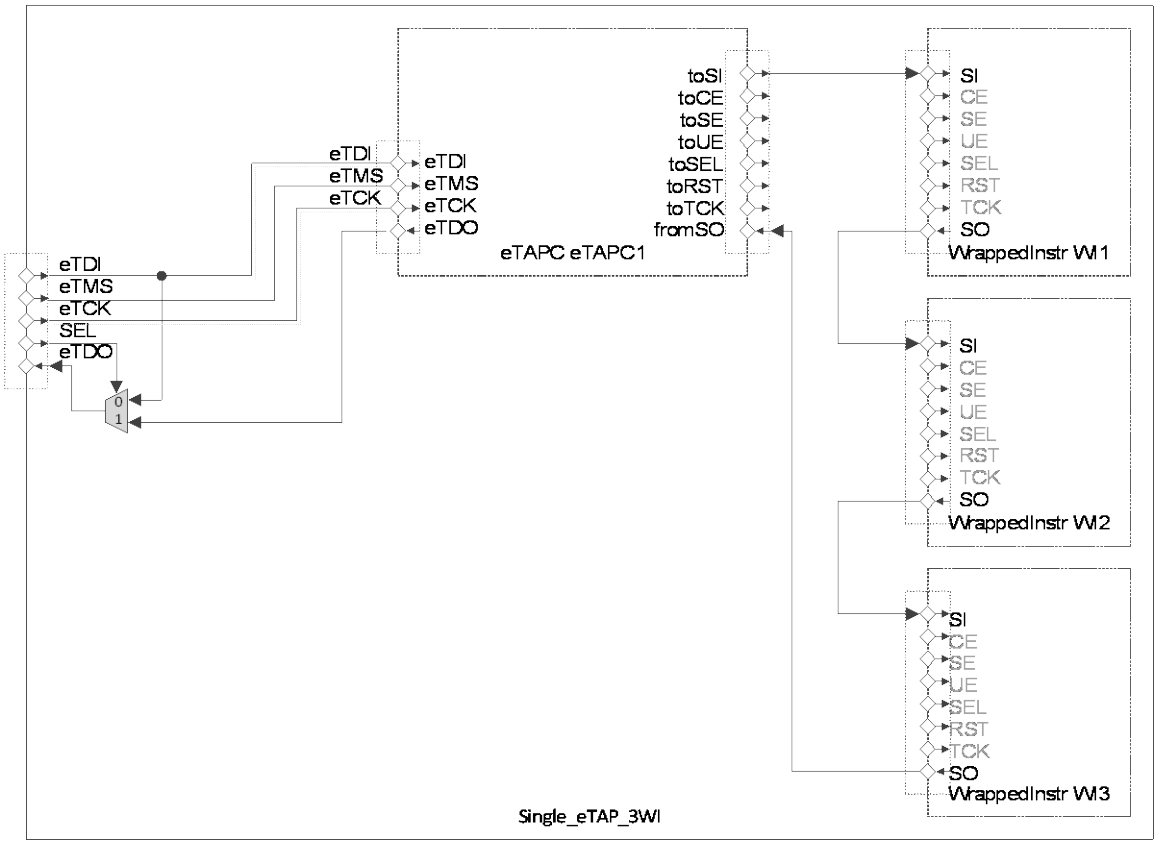


Figure E.21—Embedded TAP controller as an interface to instruments

The ICL for the single-eTAP example is as follows, including the module definition for the eTAPC.

```
Module Single_eTAP_3WI {
  ScanInPort eTDI; TMSPort eTMS; DataInPort SEL;
  TCKPort eTCK; ScanOutPort eTDO { Source M2; }

  Instance eTAPC1 Of eTAPC { InputPort eTDI = eTDI;
                             InputPort eTMS = eTMS;
                             InputPort eTCK = eTCK;
                             InputPort fromSO = WI3.SO; }
  Instance WI1 Of WrappedInstr { InputPort SI = eTAPC1.toSI; }
```

```
  Instance WI2 Of WrappedInstr { InputPort SI = WI1.SO; }
  Instance WI3 Of WrappedInstr { InputPort SI = WI2.SO; }
  ScanMux M2 SelectedBy SEL {1'b0 : eTAPC1.eTDO;
                             1'b1 : eTDI;
  }
}

Module eTAPC {
  ScanInPort   eTDI;
  ScanOutPort  eTDO { Source IRMux; }
  ScanInPort fromSO;
  ScanOutPort toSI {Source eTDI; }
  TMSPort      eTMS;
  TCKPort      eTCK;
  ToSelectPort en  { Source tdrEn0; }
  ToResetPort  tlr { Source FSM.tlr; }
  ToCaptureEnPort CE { Source FSM.CE; }
  ToShiftEnPort SE { Source FSM.CE; }
  ToUpdateEnPort UE { Source FSM.CE; }
  Instance FSM Of TapStates {
    InputPort tms = eTMS;
    InputPort tck = eTCK;    }
  ScanRegister IR[2:0] { ResetValue 3'b000;
                         CaptureSource 3'bx01; }
  ScanRegister Bypass  { CaptureSource 1'b0;   }
  ScanMux IRMux SelectedBy FSM.IRSel { 1'b0 : DRMux;
                                       1'b1 : IR[0]; }
  ScanMux DRMux SelectedBy IR[1:0] { 2'b00 : Bypass;
                                     2'b01 : fromSO;    }
  LogicSignal tdrEn0 { IR[1:0] == 2'b01;  }
  ScanInterface scan_host { Port toSI; Port en; Port fromSO; Port eTCK;
                            Port CE; Port SE; Port UE; Port tlr; }
}

Module TapStates {
  TMSPort tms;
  TCKPort tck;
  ToResetPort tlr;
  ToIRSelectPort IRSel;
  ToCaptureEnPort CE;
  ToShiftEnPort SE;
  ToUpdateEnPort UE;
}
```

E.22 Basic PDL for a simple instrument

The lowest level at which a PDL procedure may be written is for an instrument. The structural interface of an instrument is limited to only DataInPort, DataOutPort, and ClockPort functions, and these ports may connect (through logic or data registers) to device-level pins.

Figure E.22 depicts an instrument. This one happens to have both **`DataInPort`** and **`DataOutPort`** functions with the same width (4 bits).

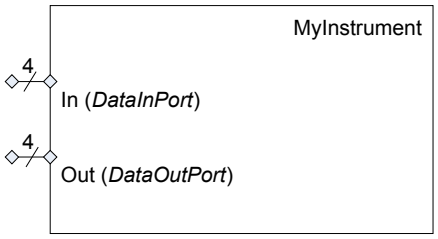


Figure E.22—Simple instrument

The ICL for this simple instrument is as follows:

```
Module MyInstrument {
   DataInPort  In[3:0];
   DataOutPort Out[3:0];
}
```

Some example PDL that may be written at the instrument level is as follows:

```
iProcsForModule MyInstrument

iProc ResetInstr {
  iWrite In[0] 0;
}
iProc CheckInstr {
  iWrite In 0x1;  # note the use of hex ("0x" prefix)
  iApply;
  iRead Out;
  iApply;
}
```

There are two PDL procedures available for MyInstrument: ResetInstr and CheckInstr. The ResetInstr procedure drives the LSB of the input port (In[0]) low, which presumably resets this instrument. The procedure CheckInstr used to operate this instrument consists of a single write and a single read operation. In both cases here, the port name is used without the bit range, which means that the entire width of the port (4 in each case) is used. The data written is a hexadecimal "1", so the bitwise values applied to the "In" port are In[3] = 0, In[2] = 0, In[1] = 0, In[0] = 1. Note that the use of the hexadecimal value differentiates the given statement from these three statements:

```
iWrite In 1;      # defaults to decimal,  sets only the LSB, pads the MSBs with 0s
# or:
iWrite In[0] 1;    # defaults to decimal, explicitly sets only the LSB
# or:
iWrite In[0] 0b1; # explicitly binary, explicitly sets only the LSB
```

The proper alternative method to writing the given statement would be either of these statements:

```
iWrite In 0001;          # defaults to decimal
# or:
iWrite In[3:0] 0001;     # explicit range, defaults to decimal
# or:
iWrite In 0b0001;        # explicitly binary
# or:
iWrite In[3:0] 0b0001; # explicit range, explicitly binary
# or, verbose but correct:
iWrite In[3] 0b0;
```

```
iWrite In[2] 0b0;
iWrite In[1] 0b0;
iWrite In[0] 0b1;       # these 4 must appear together, but in any order
```

After the write data is applied to the instrument (by the first iApply statement), the output data is read from the instrument (by the second iApply statement). Note that no expected value is provided in the iRead command—this is typical of a "measurement" action (as opposed to a "comparison" action).

The notion of a leaf instrument is deliberately very broad in order to cover the widest possible variety of circuits that may be connected to an IEEE 1687 access network. Examples may range from a write-only instrument (e.g., a control bit) to read-only instrument (e.g., an observation bit) to an instrument with both control and observation capabilities (e.g., this example and the next example—a Memory BIST engine) to an instrument with many data and control and observation functions (e.g., a high-speed serial test engine).

E.23 MBIST engine example

A leaf instrument may have multiple DataIn and DataOut ports of different widths, as shown in this example of a Memory BIST engine. Figure E.23 depicts an example of a Memory BIST (MBIST) engine.

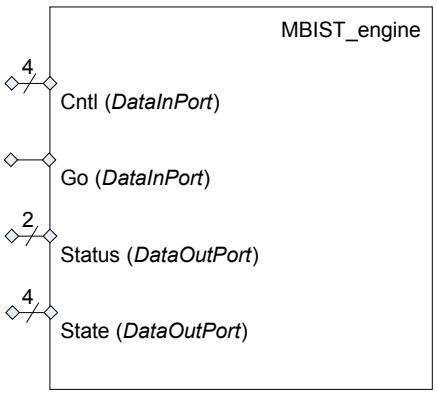


Figure E.23—MBIST engine

The ICL for the MBIST engine is as follows:

```
Module MBIST_engine {
  DataInPort Cntl[3:0];
  DataInPort Go;
  DataOutPort Status[1:0];
  DataOutPort State[3:0];
}
```

The PDL for the MBIST engine is as follows:

```
iProc RunMBIST {Control,LoopCount} {
  iWrite Cntl $Control;  # Control is passed in as a parameter
  iApply;
  iWrite Go 0b1;
  iApply;
  iRunLoop $LoopCount;   # action command, like iApply
  iRead Status 0b11;
  iApply;
}
```

```
iProc RunGalPatMBIST {
  iCall RunMBIST 0b0111 32000;
}
```

This example includes two PDL procedures: one that runs a given MBIST engine, and another that calls the first with desired parameters. The first procedure that runs the MBIST engine uses the parameter passing feature of PDL twice: the control value is passed into the procedure as {Control} and referenced by the first iWrite command as $Control, and the loop count is passed in as {LoopCount} and referenced by the iRunLoop command as $LoopCount. After the instrument is allowed to run for the desired number of clocks, the Status bits are read (and expected to both be logic 1). The second procedure uses iCall to invoke the first procedure, and delivers to it the two parameters it will use (specifically, a control value of 0111 and a loop count of 32000).

This simple but realistic example illustrates the mechanisms for dealing with a leaf instrument. It also hints at the need for both abstraction (e.g., the association between the *GalPat* algorithm and the Cntl value of 0111) and flow control mechanisms in the PDL (e.g., what to do if the "iRead Status" command returns something other than "11"). Both of those concepts will be illustrated in later examples.

E.24 MBIST and associated memory example

Leaf modules for related hardware (such as a memory and the memory BIST engine testing it) may be described independently. Even though there are connections between them, these connections may be omitted from the ICL if they are not pertinent to the delivery of data between the IEEE 1687 access network and the instrument. For example, neither the Read/Write control signal for the memory nor the Write_data are present in the ICL since they are not included as part of the instrumentation data communicated to the network (since a memory BIST failure can be diagnosed by knowing the failing location and the failing data read at that location).

Figure E.24 shows two separate modules containing the MBIST engine and the Memory itself.

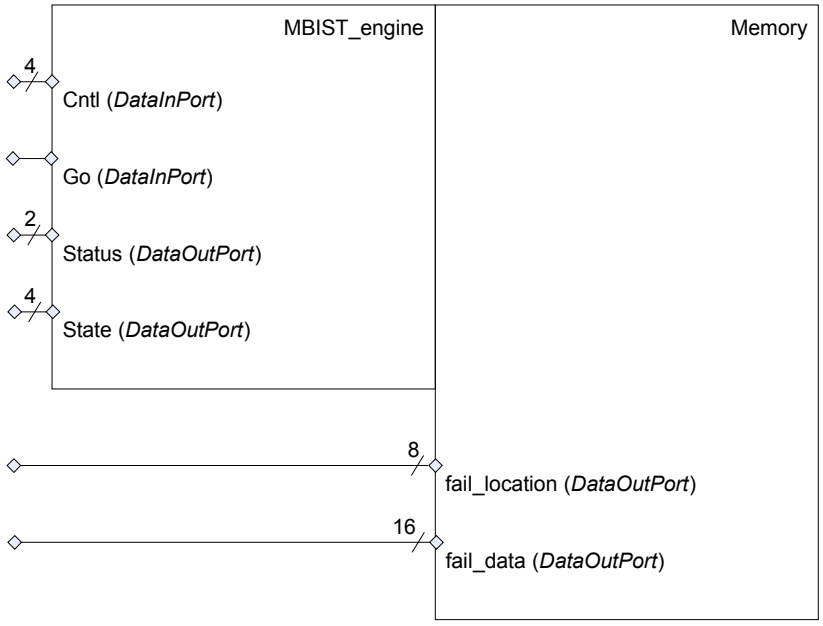


Figure E.24—Separate Memory BIST and Memory modules

The ICL for this example is as follows:

```
Module MBIST_engine {
  DataInPort Cntl[3:0];
  DataInPort Go;
  DataOutPort Status[1:0];
  DataOutPort State[3:0];
}

Module Memory {
  DataOutPort fail_location[7:0];
  DataOutPort fail_data[15:0];
}
```

This ICL builds upon the ICL shown in the previous example by adding a second module; note that there are no connections shown between the two modules because they are irrelevant for IEEE 1687 purposes (though they are certainly present in the RTL for the design).

The PDL for this example is as follows:

```
iProc Collect_MBIST_failure_info {
  iRead Status;
  iRead State;
  iRead fail_location;
  iRead fail_data;
  iApply;
}
```

This simple read-only procedure would be invoked after a Memory BIST failure has been detected. It observes the Status and State bits from the MBIST engine as well as the fail_location and fail_data values from the Memory.

This example, while valid, hints at an improvement that can be made by better organizing the ICL. The single PDL procedure actually addresses two different modules; for bookkeeping and module-level simulation purposes, it is generally desirable to associate PDL with a given (single) module. The next example addresses this issue.

E.25 Combined MBIST and memory example

The use of hierarchy is applied to improve the previous example. Leaf modules for related hardware (such as a memory and the memory BIST engine testing it) may be described together by grouping them as instances within a single parent module. As before, even though there are connections between the child instances, these connections may be omitted from the ICL if they are not pertinent to the delivery of data between the IEEE 1687 access network and the instrument.

Figure E.25 illustrates the use of a parent module (BISTed_Memory) to contain the instances of the two child modules (the MBIST engine, named BIST1, and the Memory, named Mem1).

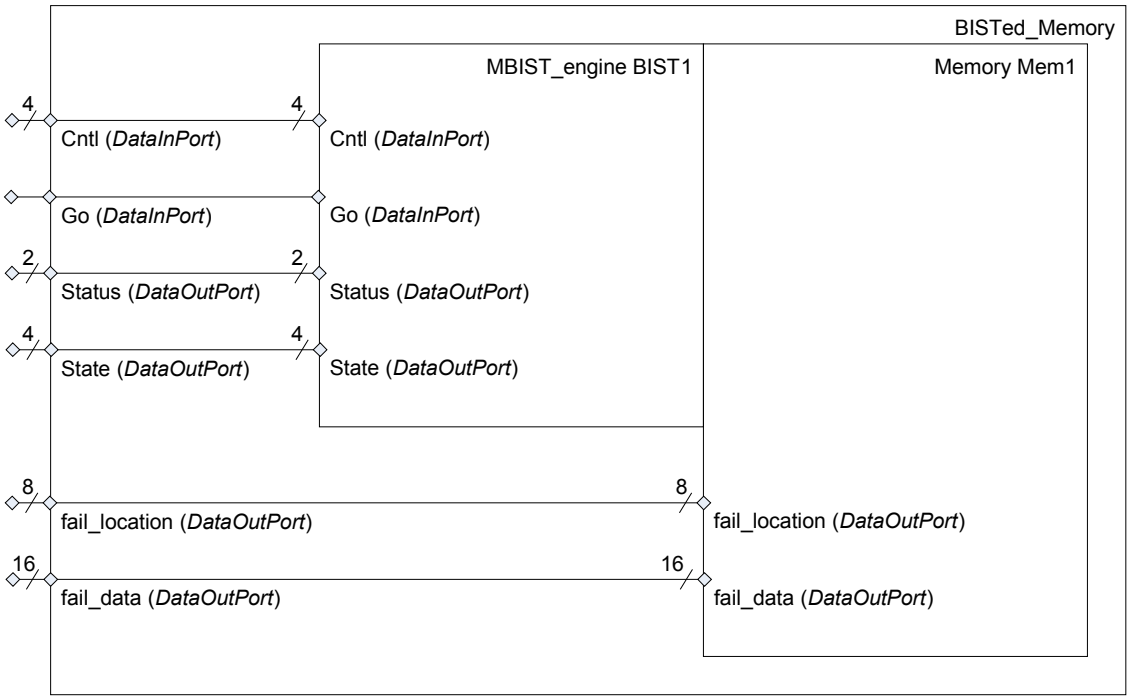


Figure E.25—Combined MBIST and Memory modules

The ICL for this example is as follows:

```
Module BISTed_Memory {
  DataInPort Cntl[3:0];
  DataInPort Go;
  DataOutPort Status[1:0] {
    Source BIST1.Status; }
  DataOutPort State[3:0] {
    Source BIST1.State; }
  DataOutPort fail_location[7:0] {
    Source Mem1.fail_location[7:0]; }
  DataOutPort fail_data[15:0] {
    Source Mem1.fail_data[15:0]; }
  Instance BIST1 Of MBIST_engine {
    InputPort Cntl = Cntl;
    InputPort Go = Go;
  }
  Instance Mem1 Of Memory;
}
```

The top-level module (BISTed_Memory) contains two instances: BIST1 is the MBIST_engine module, Mem1 is the Memory module. Since the source of every signal connection is required to be identified, there are appropriate fields added to the DataOutPort functions of the top-level module as well as for each input port of the child instances. The DataOutPort functions of the child modules BIST1 and Mem1 do not require source identification because they are leaf modules.

Note that in this example the top-level module (BISTed_Memory) includes all of the ports of all of its children. In a more typical scenario, some ports of the children connect only to each other; those captive connections do not become ports of the top-level module.

The same PDL from the previous example is useful here, with the new property that it is now aimed at module "BISTed_Memory."

```
iProc Collect_MBIST_failure_info {
  iRead Status;
  iRead State;
  iRead Address;
  iRead ReadData;
  iApply;
}
```

This simple read-only procedure would be invoked after a Memory BIST failure has been detected. It observes the Status and State bits from the instance of the MBIST_engine module as well as the Address and Read_data values from the instance of the Memory module.

E.26 Addressable instruments

An ICL example of an addressable instrument scheme is as follows. Note that the logical Address in module BlockA is formed by concatenating all the AddressPort functions (AH and AL) top-to-bottom, left-to-right. After that assembly process, the lower 8-bits of the resulting 20-bit address are used.

```
Module BlockA {
  DataInPort DH[7:0];
  DataInPort DL[7:0];
  DataOutPort QH[7:0] { Source Q[15:8] ; }
  DataOutPort QL[7:0] { Source Q[7:0] ; }
  AddressPort AH[19:16];
  AddressPort AL[15:0];
  ReadEnPort RE; //One and only one RE/WE ports on
  WriteEnPort WE; //modules containing addressable regs
  OneHotDataGroup Q[15:0] {
    DataRegister R0[15:0] {
      AddressValue 8'd0;
    }
    DataRegister R1[15:0] {
      AddressValue 8'd1;
    }
    …
    DataRegister R72[15:0] {
      AddressValue 8'd72;
    }
  }
  Alias Ena = R0[15];
  Alias AMode[8:0] = R0[14:6] {
    RefEnum AModes;
  }
  …
  Alias RH = R72[0];
  Enum AModes { M1 = 9'h0; …; M9 = 9'h0A4; }
}

Module BlockB {
  DataInPort D[15:0];
  DataOutPort Q[15:0] { Source QM[15:0]; }
  AddressPort A[19:0];
```

```
  ReadEnPort ReadEn;
  WriteEnPort WriteEn;
  OneHotDataGroup QM[15:0] {
    Instance BlockA_I1 Of BlockA {
      AddressValue 16'h0000;
    }
    Instance BlockA_I2 Of BlockA {
      AddressValue 16'h2300;
    }
    …
  }
}

// Default DataIn mapping is right justified
// Can be explicitly specified with OneHotDataGroup syntax:
OneHotDataGroup Q[15:0] {
  Port BlockA_I1.DOH[7:0];
  Port BlockA_I1.DOL[0:7];
  …
}
```

E.27 Black-box module

When an instrument provider desires to protect his IP, he may deliver it as a black-box module. The length of the scan chain running through the black-box module must be precisely known, so interactions with the iScan command are prescribed.

```
Module GreyBox1 {
  ScanInPort SI; CaptureEnPort Ce; ShiftEnPort Se;
  UpdateEnPort Ue; SelectPort Sel; ResetPort Rst;
  TCKPort Tck; ScanOutPort SO { Source Sib.so; }

  Instance tdr1a Of tdr {
    InputPort SI = SI; InputPort CE = Ce; InputPort SE = Se;
    InputPort UE = Ue; InputPort SEL = Sib1.ten;
    InputPort RST = Rst; InputPort DI = I1.DO; InputPort TCK = Tck;
  }
  Instance I1 Of Instrument { InputPort DI = tdr1a.DO; }

  Instance Sib Of sib {
    InputPort si = SI; InputPort ce = Ce; InputPort se = Se;
    InputPort ue = Ue; InputPort sel = Sel; InputPort rst = Rst;
    InputPort fso = tdr1a.SO; InputPort tck = Tck;
  }
}

Module BlackBox1 {
  ScanInPort SI; CaptureEnPort Ce; ShiftEnPort Se;
  UpdateEnPort Ue; SelectPort Sel; ResetPort Rst;
  TCKPort Tck; ScanOutPort SO;
}
```

Another example is as follows:

```
Module GreyBox2 {
  ScanInPort     SI;     CaptureEnPort     Ce;     ShiftEnPort     Se;
  UpdateEnPort     Ue;     SelectPort     Sel;     ResetPort     Rst;
  TCKPort Tck; ScanOutPort SO { Source Sib2.so; }
  Instance tdr2a Of tdr {
    InputPort SI = SI; InputPort CE = Ce; InputPort SE = Se;
    InputPort UE = Ue; InputPort SEL = Sel;
    InputPort RST = Rst; InputPort DI = I1.DO; InputPort TCK = Tck;
  }
  Instance tdr2b Of tdr {
    InputPort SI = tdr2a.so; InputPort CE = Ce; InputPort SE = Se;
    InputPort UE = Ue; InputPort SEL = Sib1.ten;
    InputPort RST = Rst; InputPort DI = I2.DO; InputPort TCK = Tck;
  }
  Instance tdr2c Of tdr {
    InputPort SI = Sib1.so; InputPort CE = Ce; InputPort SE = Se;
    InputPort UE = Ue; InputPort SEL = Sib2.ten;
    InputPort RST = Rst; InputPort DI = I3.DO; InputPort TCK = Tck;
  }
  Instance I1 Of Instrument { InputPort DI = tdr2a.DO; }
  Instance I2 Of Instrument { InputPort DI = tdr2b.DO; }
  Instance I3 Of Instrument { InputPort DI = tdr2c.DO; }

Instance Sib1 Of sib {
    InputPort si = tdr2a.so; InputPort ce = Ce; InputPort se = Se;
    InputPort ue = Ue; InputPort sel = Sel; InputPort rst = Rst;
    InputPort fso = tdr2b.SO; InputPort tck = Tck;
  }
  Instance Sib2 Of sib {
    InputPort si = tdr2b.so; InputPort ce = Ce; InputPort se = Se;
    InputPort ue = Ue; InputPort sel = Sel; InputPort rst = Rst;
    InputPort fso = tdr2c.SO; InputPort tck = Tck;
  }
}
```

E.28 Wide scan interface example

The ScanInterfaces used in previous examples have featured only a single scan chain. However, there may be cases were a wide ScanInterface is used, as shown in the following ICL and PDL.

```
ScanInterface wide {
  Chain c1 {
    Port si1;
    Port so1;
    DefaultLoadValue 3'b0;
  }
  Chain c2 {
    Port si2;
    Port so2;
    DefaultLoadValue 5'b0;
  }
  Chain c3 {
    Port si3;
    Port so3;
```

```
    DefaultLoadValue 7'b0;
  }
  Port selw;
  Port sel;
}
ScanInterface narrow1 {
  Port si1;
  Port so1;
  Port seln1;
  Port sel;
  DefaultLoadValue 13'b0;
}
ScanInterface narrow2 {
  Port si1;
  Port so3;
  Port seln2;
  Port sel;
  DefaultLoadValue 7'b0;

}
```

Example PDL for this interface is as follows:

```
iScan wide -chain c1 3 -si 0b000
iScan wide -chain c2 5 -si 0b00000
iScan wide -chain c3 7 -si 0b0000000
iApply;
```

E.29 Simple IEEE 1149.1 AccessLink example

Figure E.26 and the accompanying ICL shows how an IEEE 1149.1 TAP at the top-level device is connected to a network using the AccessLink statement. The network here is a simple wrapped instrument (which includes a ScanInterface named scan_client, as shown in E.4), but any network with a defined ScanInterface could be substituted. Note that the TAP, though present in the device, is not explicitly instantiated in the ICL through the traditional "instance" statement, but rather through the AccessLink statement. The reference in the AccessLink statement to the ScanInterface named "scan_client" of instance WI1 provides all the connectivity necessary, which is why the instance statement for WI1 does not include any "InputPort" statements. The AccessLink statement thus provides the attachment point of the device interface to the ICL network.

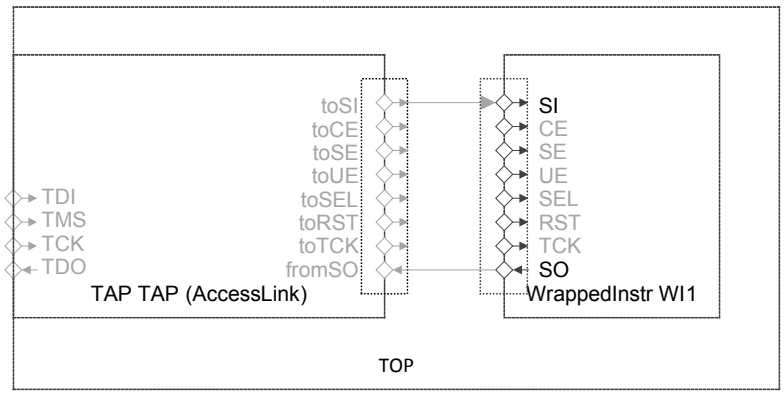


Figure E.26—Simple IEEE 1149.1 AccessLink example

The ICL for the simple IEEE 1149.1AccessLink example is as follows:

```
Module TOP {
  Instance WI1 Of WrappedInstr { }
  AccessLink TAP Of STD_1149_1_2001 {
  BSDLEntity TOP;
    ijtag_en { // Name of instruction
    ScanInterface { WI1.scan_client; }  // ScanInterface
    }
  }
}
```

E.30 Complex IEEE 1149.1 AccessLink example

Figure E.27 and the accompanying ICL shows a more complex example using additional features of the AccessLink statement for an IEEE 1149.1 interface. The features illustrated in this example include the following:

1) The AccessLink statement includes two instructions: one to select the WIR, and the other to select the WDR.

2) As with the previous simple example, most of the InputPort statements in the COREA instances are omitted because they are specified by the ScanInterface reference in the AccessLink statement. The corresponding wires are grayed-out in the figure. However, the other data and clock signals (e.g., P, CLK) that connect to the COREA instances but do not come from the TAP shown in the AccessLink statement are connected as in a typical instance.

3) The SWIR port of the COREA instances is a DataInPort, and it is driven by an ActiveSignal called “toSWIR” from the AccessLink. Since this signal is not part of the ScanInterface, it is shown in the figure. When the “wir_select” instruction is active, the toSWIR signal is also active. When the “wdr_select” instruction is active, the toSWIR signal is inactive (since it is not listed as an ActiveSignal in that section of the AccessLink statement).

4) The AccessLink statement also uses a feature whereby listing two scan interfaces (COREA_I1.SP1 and COREA_I2.SP1) in the same ScanInterface statement implies that they are connected in a daisy chain. For this reason, the WSO to WSI connection between those two COREA instances is grayed-out in the figure and not present anywhere else in the ICL besides the AccessLink statement.

5) Since the COREA module only has one ScanInterface (“SP1”), that portion of the name may be omitted where it would otherwise appear without introducing any ambiguity. This language feature was exploited in the “wdr_select” portion of the AccessLink statement.

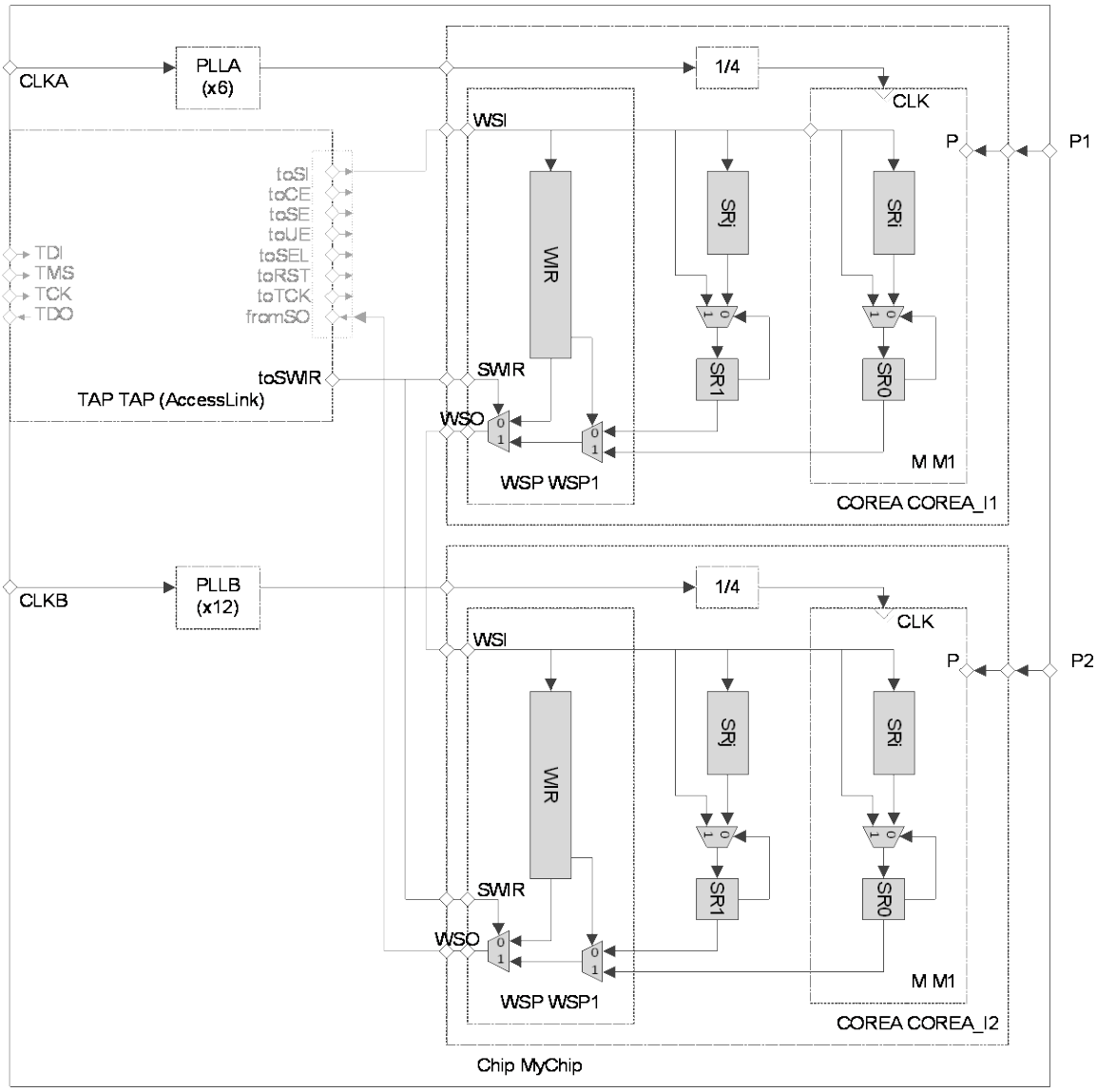


Figure E.27—Complex IEEE 1149.1 AccessLink example

The ICL for the complex IEEE 1149.1 AccessLink example is as follows:

```
Module MyChip {
  ClockPort CLKA;
  ClockPort CLKB;
  DataInPort P1;
  DataInPort P2;
  Instance PLLA_I1 Of PLLA { InputPort ref = CLKA;}
  Instance PLLB_I1 Of PLLB { InputPort ref = CLKB;}
  Instance COREA_I1 Of COREA {
    InputPort P = P1;
    InputPort CLK = PLLA_I1.vco;
    InputPort SWIR = Tap1.toSWIR;
  }
  Instance COREA_I2 Of COREA {
    InputPort P = P2;
    InputPort CLK = PLLB_I1.vco;
```

```
    InputPort SWIR = Tap1.toSWIR;
  }
  AccessLink Tap1  Of STD_1149_1 {
      BSDL_Entity Mychip;
      wir_select { ScanInterface { COREA_I1.SP1;
                                   COREA_I2.SP1;}
                   ActiveSignals { toSWIR ;}
      }
      wdr_select { ScanInterface {COREA_I1;
                                  COREA_I2; }
      // Because COREA only has one scan port, explicitly listing the
ScanInterface (".SP1") is optional
      // Also note that since "toSWIR" is not listed as an ActiveSignal,
so it is INACTIVE here.
      }

  }
}

Module PLLA {
  ClockPort ref;
  ToClockPort vco { FreqMultiplier: 6; }
}
Module PLLB {
  ClockPort ref;
  ToClockPort vco { FreqMultiplier: 12; }
}
Module CoreA {
  ClockPort CLK;
  ScanInPort WSI;
  ScanOutPort WSO {Source : WTAP.WSO; }
  ShiftEnPort shiftWR;
  CaptureEnPort captureWR;
  UpdateEnPort updateWR;
  ResetPort WRST;
  TCKPort WRCK;
  DataInPort SWIR;
  ScanInterface SP1 { Port WSI; Port WSO; Port SEL; Port WSI; Port WSO;
                      Port shiftWR; Port captureWR; Port updateWR;
  }
 . . .
}
From BSDL File:
Entity Mychip
attribute REGISTER_OPCODE = " wir_select (110)," &
                            " wdr_select (111);"
```

Note that the instances of M1 in the two COREA instances receive different clock signals from different PLLs. The following PDL will therefore result in different absolute durations of the iRunLoop command when retargeted for the unique instances of M1 (i.e., CLKA will pulse 4000 times, CLKB will pulse 2000 times).

```
iProc init { } {
  iClock CLK
  iWrite P 0b1;
  iWrite SRi 0b0;
```

```
  iRunLoop 6000 -sck CLK
}
```

E.31 Generic AccessLink example

Figure E.28 and the accompanying ICL shows how a generic AccessLink at the top-level device is connected to a network using the AccessLink statement.

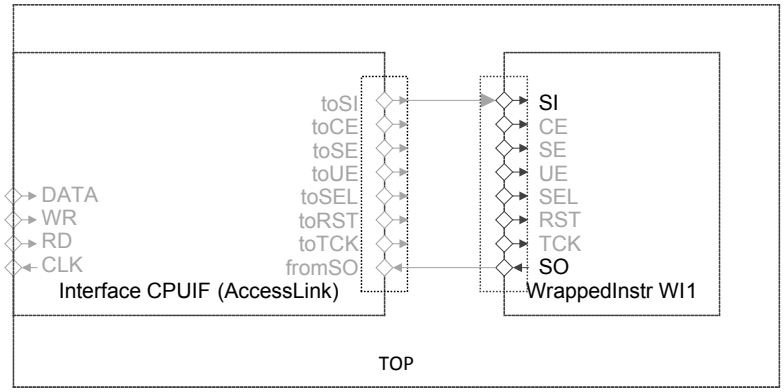


Figure E.28—Generic AccessLink example

The ICL for the generic AccessLink example is as follows:

```
Module chip {
  Instance WI1 Of WrappedInstr {  }
  AccessLink CPUIF  of Interface {
    ScanInterface WI1.scan_client;
    WordSize 8; // 6+2
    CPU_pins {DATA, WR, RD, CLK}
  }
}
```

Note that there is not a prescribed format for the contents of a generic AccessLink statement; it is simply of the form "AccessLink *name* of *type* { *string* }". As long as *type* is not a **`STD_1149_1`**, the string may contain information in any form (subject to the usual guidelines about balancing quotes, parentheses, brackets, and braces). The content of the generic AccessLink is determined by agreement between the provider of the controller that will communicate with the interface and the provider of the ICL (specified by the former and implemented by the latter).

In the preceding contrived example, the interface includes an 8-bit data port (DATA), write and read controls (WR and RD), and a clock (CLK). The DATA word is broken into a payload of 6 bits and a 2-bit command field with this encoding:

00 = shift then go to pause
01 = shift then go to rti
11 = go to reset

Given a PDL iScan command to apply via this CPU interface, the associated controlling software would convert the IEEE 1687 shift operations into CPU read and write operations, taking into account the relative sizes of the scan chain (8 bits, in this case, from the WrappedInstr example of E.4) and the data payload (6 bits). For example, **`iScan 10 -si 1000_0011 -so X1XX_XX11`** would be transformed to this sequence of operations at the CPU interface:

```
# ScanIn:  100000 11PPPP  # P signifies Padding bits to fill the 6-bit payload,
                          # using 0s starting at the beginning of the SI stream

# ScanOut: PPPPX1 XXXX11   # SO padding is with Xs at the end of stream

cpu_read  01_XXXX11

cpu_write 00_110000

cpu_read  01_XXXXX1

cpu_write 01_100000
```

This is just a contrived example of one possible interface to an on-chip instrument access network, but it is useful in illustrating the principle behind the generic AccessLink: it identifies the attachment point to the network (WI1.scan_client in this example) and provides information to the external controlling software (the CPU interface pin names and width in this example).

Annex F

(informative)

Design guidance

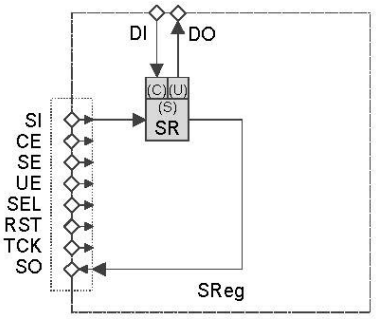
F.1 Introduction

The philosophy behind this standard is to be largely descriptive rather than prescriptive with respect to the structures that can comprise an instrument access network. That said, one downside to providing the user with the power of abstraction is that the implementation details may not be readily obvious. The intent of this annex on design guidance is to illustrate alternative logic-level implementations for two of the foundational building blocks, the scan register and the scan mux (for which there are in-line and remotely controlled variants), along with the tradeoffs associated with each implementation. It is important to note that these implementations are not actual specifications, but merely examples used to explain concepts.

F.2 Scan register implementations

F.2.1 Basic scan register

A diagram and the associated ICL for a single-bit scan register are shown in Figure F.1.




```
Module SReg {
  ScanInPort    SI;
  CaptureEnPort CE;
  ShiftEnPort   SE;
  CaptureEnPort CE;
  UpdateEnPort  UE;
  SelectPort    SEL;
  ResetPort     RST;
  TCKPort       TCK;
  ScanOutPort   SO { Source  SR[0];
                     Attribute LaunchEdge = "Rising"; }
  DataInPort    DI[0:0];
  DataOutPort   DO[0:0] { Source SR; }
  ScanInterface SReg_client { Port SI; Port SO; Port SEL; }
  ScanRegister SR[0:0] {
    ScanInSource SI; CaptureSource DI; ResetValue 1'b0;
  }
}
```

Figure F.1—ICL version of a single-bit scan register

A circuit designer faced with the task of implementing the ICL description in Figure F.1 would need to ask several questions: Where do the middle six ports connect? How is the interface activated? How are the storage elements arranged? How is the reset functionality achieved? An example implementation that resolves these questions and is consistent with the ICL abstraction of a simple scan register with a plug-and-play scan client interface is shown in Figure F.2. Note that Figure F.2 is a logic circuit schematic, not an ICL diagram, and no ICL is written at the detailed level of circuit schematics: ICL is an abstraction.

The scan client interface is the dashed box around the eight ports on the left side of the figure. There are two storage elements, implemented as flip-flops labeled “CS” (for the Capture-Shift stage) and “U” (for the Update stage). There is a DataInPort (DI) and a DataOutPort (DO) on the top of the figure, which provide data to be captured from and consumed by an instrument, respectively.

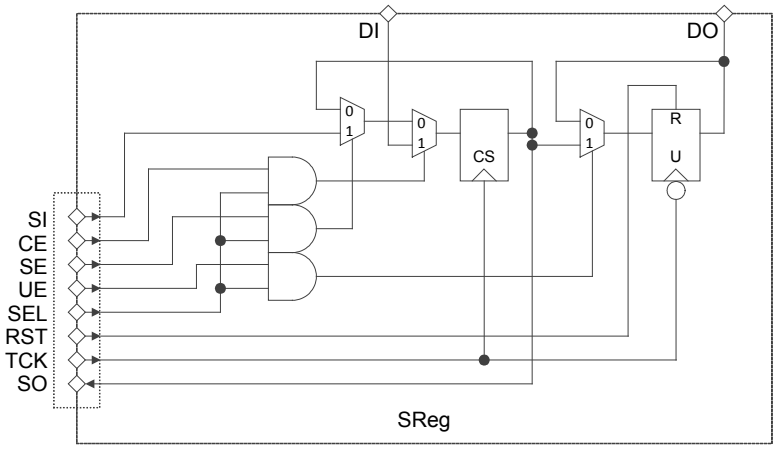


Figure F.2—Basic scan register implementation example

This implementation illustrates the following four important hardware rules:

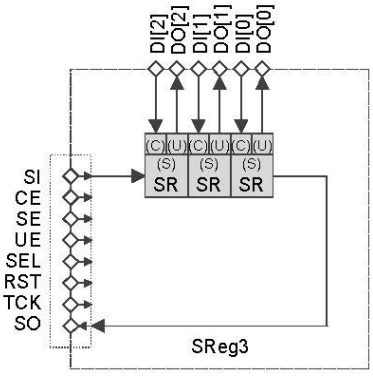
a) The scan register does not respond to the ShiftEn (SE), CaptureEn (CE), or UpdateEn (UE) control signals when the Select signal (SEL) is low. The three AND gates implement this functionality.

b) The DataOutPort does not change during shifting—it is driven by the separate Update flop.

c) The Update flop (hence the DataOutPort) changes on the falling edge of TCK.

d) The interface satisfies the definition of a plug-and-play scan client.

The ICL for the 3-bit scan register is shown in Figure F.3. It is exactly the same as the ICL shown in Figure F.1, but with the MSB of the indices for DI, DO, and SR changed from 0 to 2.




```
Module SReg3 {
  ScanInPort    SI;
  CaptureEnPort CE;
  ShiftEnPort   SE;
  CaptureEnPort CE;
  UpdateEnPort  UE;
  SelectPort    SEL;
  ResetPort     RST;
  TCKPort       TCK;
  ScanOutPort   SO { Source  SR[0];
                     Attribute LaunchEdge = "Rising";}
  DataInPort    DI[2:0];
  DataOutPort   DO[2:0] { Source SR; }
  ScanInterface SReg_client { Port SI; Port SO; Port SEL; }
  ScanRegister SR[2:0] {
    ScanInSource SI; CaptureSource DI; ResetValue 3'b000;
  }
}
```

Figure F.3—ICL version of a 3-bit scan register

When multiple single-bit scan register cells are combined to make a multi-bit register, the following two important observations apply:

1) The AND gating of the SelectPort with the CaptureEnPort, ShiftEnPort, and UpdateEnPort signals need be done only once per multi-bit register.

2) As with any shift register, meeting hold-time requirements from one cell to the next is critical. The relative timing of the SO-to-next-SI path and the TCK signal must be managed accurately.

A 3-bit scan register is shown in Figure F.4. For drawing convenience, the Update stage of each register bit has been relocated to be just above its Capture/Shift stage, but topologically each bit is the same as shown in Figure F.2.

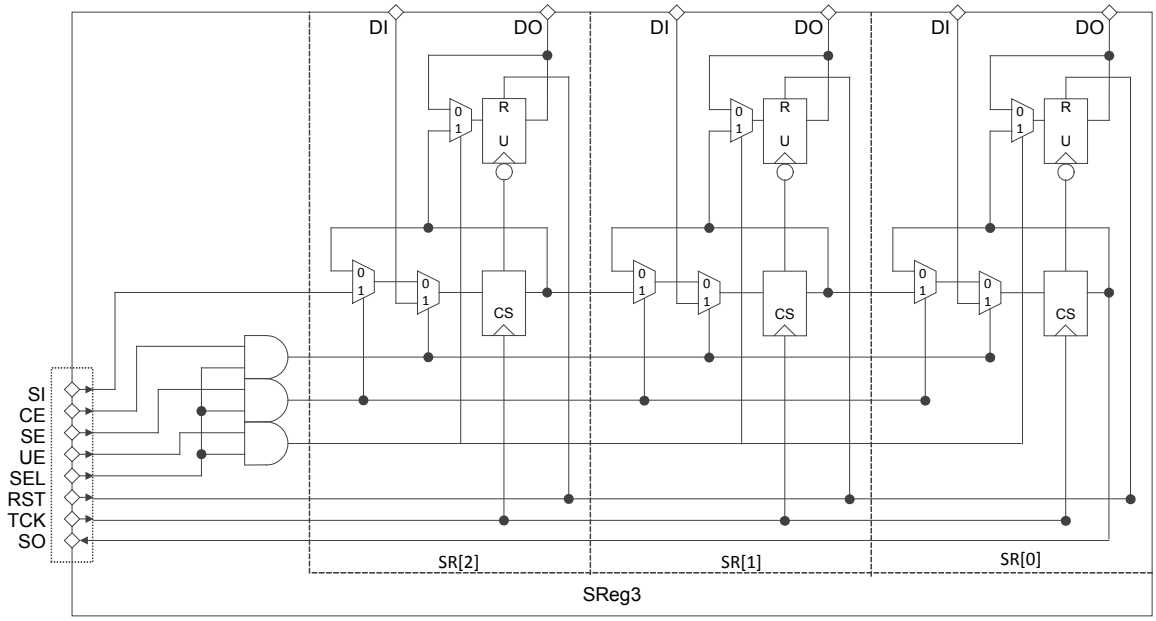


Figure F.4—3-bit basic scan register example implementation

The convention of shifting left-to-right is indicated in Figure F.3. The SelectPort gating is done once on the left and shared for all three cells.

There are two series multiplexer cells (for selecting the shift input and the capture input) on the path between shift storage elements, so with a tightly distributed TCK there should be no hold-time issues between cells. However, in driving from one scan register entity to another (potentially distant) scan register, the TCK skew may not be well controlled, so the introduction of a lockup latch on the cell driving the ScanOutPort may be necessary, as shown in the next subclause.

F.2.2 Scan Register with lockup latch on ScanOutPort

Figure F.5 shows a variant of the basic scan register cell, this time with a lockup latch between the Capture/Shift cell and the ScanOutPort.

In this implementation, the lockup latch is implemented as a negative-edge flip-flop, labeled “LU” in Figure F.5, which delays the valid data window on the SO port by half of a TCK cycle. It could alternatively have been implemented as an active-low latch. In either case, the corresponding ICL has no mandatory syntax to reflect this feature, but a best-practice approach is shown in the example in Figure F.6 where an (optional) attribute (LaunchEdge = "Rising") is attached on the ScanOutPort to document the intent in the ICL. The integrator should always use proper synthesis and timing analysis tools to implement proper flop-to-flop communication.

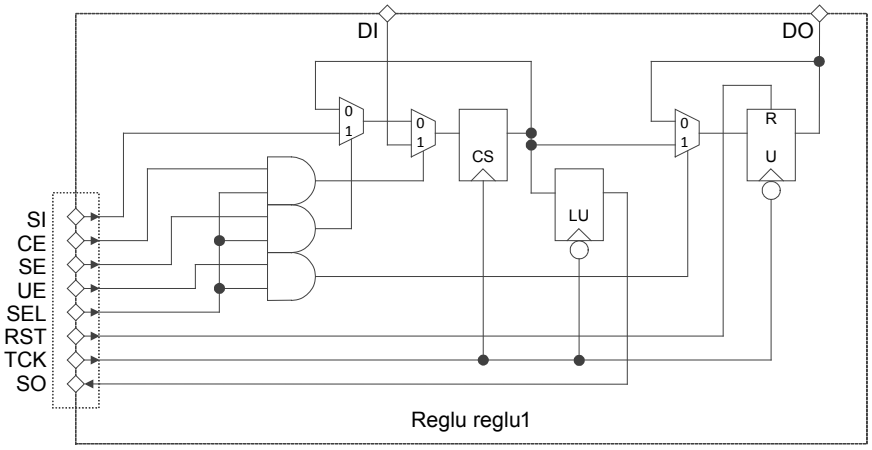
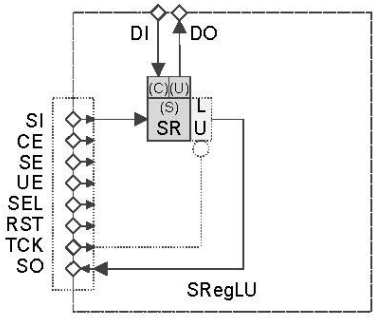


Figure F.5—Scan register with lockup latch on ScanOutPort




```
Module SRegLU {
  ScanInPort    SI;
  CaptureEnPort CE;
  ShiftEnPort   SE;
  CaptureEnPort CE;
  UpdateEnPort  UE;
  SelectPort    SEL;
  ResetPort     RST;
  TCKPort       TCK;
  ScanOutPort   SO { Source  SR[0];
                     Attribute LaunchEdge = "Falling" }
  DataInPort    DI[0:0];
  DataOutPort   DO[0:0] { Source SR; }
  ScanInterface SReg_client { Port SI; Port SO; Port SEL; }
  ScanRegister SR[0:0] {
    ScanInSource SI; CaptureSource DI; ResetValue 1'b0;
  }
}
```

Figure F.6—ICL for scan register with lockup latch driving ScanOutPort

This type of cell is useful for driving the ScanOutPort of a module when the connection to the next module may be subject to clock skew, as illustrated in Figure F.7.

The delta symbol (the grey triangle inside the circle) in Figure F.7 indicates where there could be clock skew between the modules that could create a hold-time violation on the shift path. Both the benefit and the cost of using the cell with the lockup latch are illustrated in the four timing diagrams shown in Figure F.8. Each circuit shown immediately above the corresponding timing diagram represents a particular combination of the Module1 register type (regular “CS” or lockup “LU”) that drives the SO port and the presence or absence of TCK skew on the way to Module2.

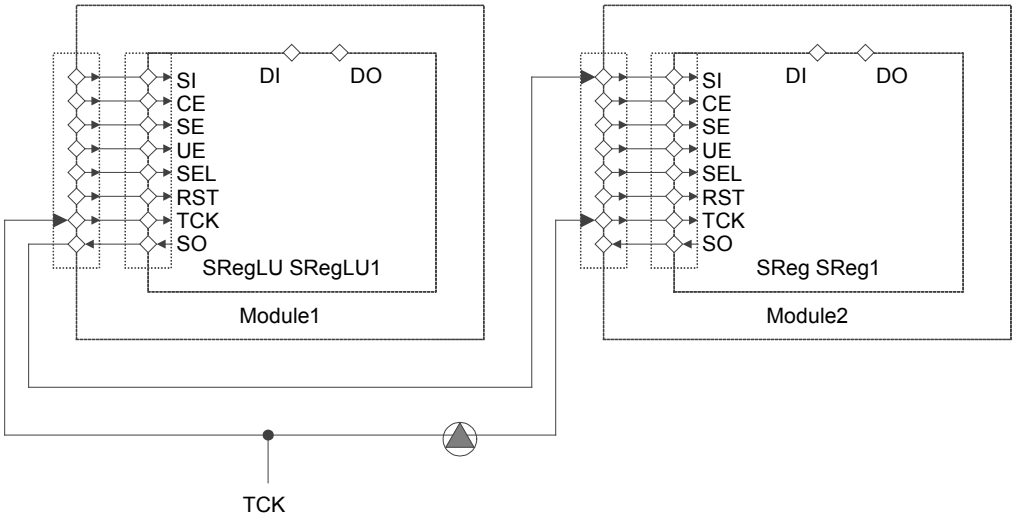


Figure F.7—Location of clock skew between modules

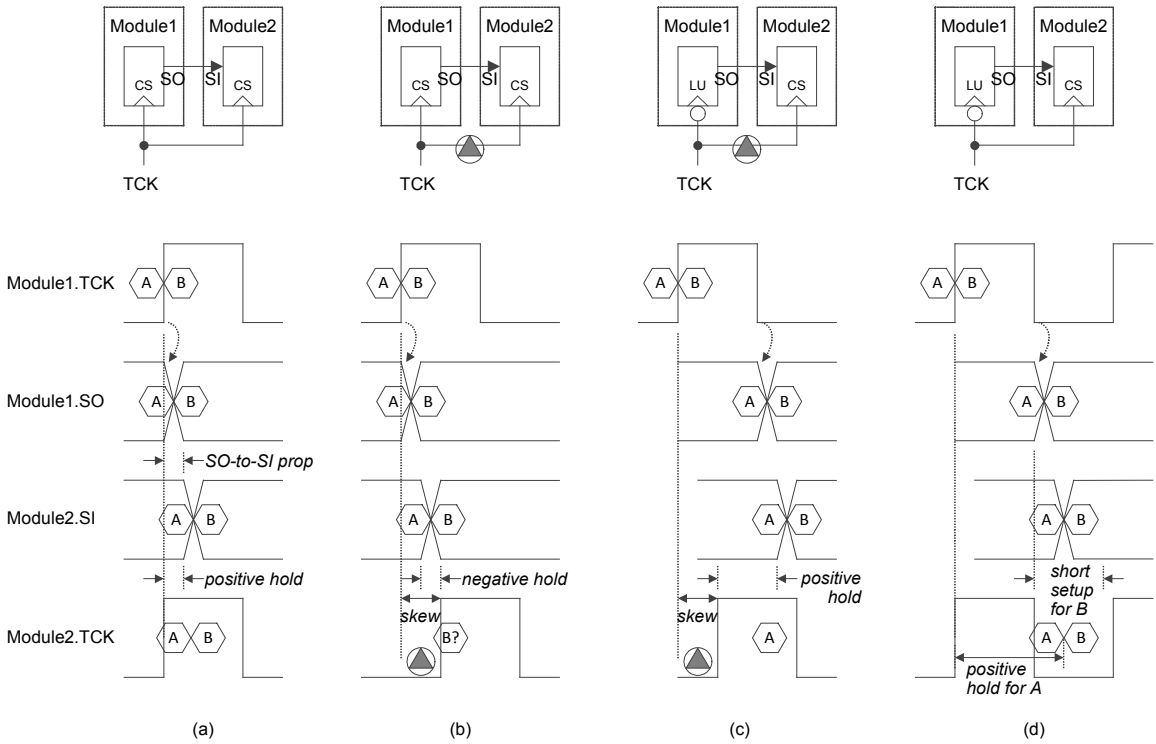


Figure F.8—Four timing scenarios between two modules

The four columns in Figure F.8 represent the following cases:

a) CS register driving another CS register with no skew on TCK.

b) CS register driving another CS register with TCK arriving late at the receiving flop.

c) LU register driving a CS register with skew on TCK.

d) LU register driving a CS register with no skew on TCK.

Each column contains a timing diagram with four rows; the rows show the timing (vertically aligned) for each of four signals:

1) Module1.TCK: the clock port in the module that is driving a ScanOut signal.
2) Module1.SO: the ScanOutPort being driven; launched on the rising edge of TCK.
3) Module2.SI: the corresponding ScanInPort being received by the other module.
4) Module2.TCK: the clock port in the receiving module, which may be skewed.

The letters A and B in the hexagons indicate successive values of the SO-to-SI signal, with the time at which the signal changes from value A to value B being located exactly between the two letters. The current value in the scan register in Module1 is A, and the goal is to shift this value into the scan register in Module2. At the same time, Module1 will shift in its new value (B). The progress of the transition can be followed down through the rows:

— An edge of TCK in the driving module (i.e., Module1) causes it to start the process of changing state from A to B. Columns (a) and (b) use the rising edge of TCK to launch the event; columns (c) and (d) use the falling edge.

— The SO output of Module1 makes its transition from A to B. The timing of this event is governed by the clock-to-Q time of the flip-flop driving SO in Module1.

— The SI input of Module2 makes its transition from A to B after the propagation time of the route between Module1.SO and Module2.SI.

— The rising edge of TCK in the receiving module (i.e., Module2) causes the value at the SI port of Module2 to become its new state. If everything went according to plan, this new state is A, not B (remember, B is the new state for the scan register in Module1, and it should not "shoot-through" to Module2 in the same TCK cycle that it enters Module1).

The interesting event in each column occurs in the fourth row (at the rising edge of TCK in the receiving module):

— When TCK is balanced and there is no skew, the combination of the clock-to-Q time in Module1 and the SO-to-SI prop time provide adequate hold time margin for a successful transfer of state A into Module2 before B arrives at the SI port.

— When there is TCK skew such that Module2 receives its TCK later than the hold-time spec of the flip-flop, then it will not reliably receive state A, and may receive state B (if the skew is sufficiently longer than the sum of the clock-to-Q and prop times) or receive a metastable state.

— The use of a negative-edge lockup ("LU") flop in Module1 will add half of a TCK cycle to the clock-to-Q and prop delays. In the example shown, this is sufficient to overcome the skew problem, but more importantly, it makes the hold time a function of TCK frequency, so even very large skews can be dealt with by lengthening the period of TCK.

— The use of a negative-edge lockup flop in Module1 not only increases the area of that scan register, but it also reduces the available setup time for the SO-to-SI path from a full TCK cycle to half of a TCK cycle. When there is no TCK skew (as is the case in this column), or when the skew is on the other fan-out branch (to Module1), the setup time margin can become a problem for timing closure. Setup time problems can be addressed by decreasing the frequency of TCK, but at the cost of increased test time.

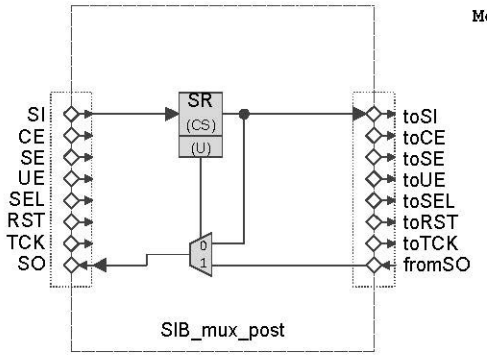
The benefits and costs of balancing the TCK distribution network vs. adding lockup latches must be carefully weighed during scan register implementation.

F.3 Scan multiplexers

F.3.1 In-line segment insertion bit (SIB)

F.3.1.1 SIB with multiplexer after scan register

One intuitive design for a segment insertion bit (SIB) is shown in Figure F.9. This topology places the scan mux after the scan register. The SIB has both a scan client interface on the left and a scan host interface on the right. The segment to be inserted would connect to this host port; for example, imagine the scan client interface of the 3-bit scan register of Figure F.3 being plugged into the right-hand side scan host interface of the SIB in Figure F.9. The select line of the ScanMux is driven by the (implicit) update stage of the ScanRegister.




```
Module SIB_mux_post {
  ScanInPort     SI;
  CaptureEnPort  CE;
  ShiftEnPort    SE;
  UpdateEnPort   UE;
  SelectPort     SEL;
  ResetPort      RST;
  TCKPort        TCK;
  ScanOutPort    SO { Source SIBmux;
                      Attribute LaunchEdge = "Rising";}
  ScanInterface client {
    port SI; port CE; port SE; port UE;
    port SEL; port RST; port TCK; port SO;
  }
  ScanInPort       fromSO;
  ToCaptureEnPort  toCE;
  ToShiftEnPort    toSE;
  ToUpdateEnPort   toUE;
  ToSelectPort     toSEL;
  ToResetPort      toRST;
  ToTCKPort        toTCK;
  ScanOutPort      toSI { Source SR;
                          Attribute LaunchEdge = "Rising";}
  ScanInterface host {
    Port fromSO; Port toCE; Port toSE; Port toUE;
    Port toSEL; Port toRST; Port toTCK; Port toSI;
  }
  ScanRegister SR {
    ScanInSource SI; CaptureSource SR; ResetValue 1'b0;
  }
  ScanMux SIBmux SelectedBy SR {
    1'b0 : SR;
    1'b1 : fromSO;
  }
}
```

Figure F.9—ICL for SIB with mux after scan register

A logic-level implementation of the SIB_mux_post structure is shown in Figure F.10.

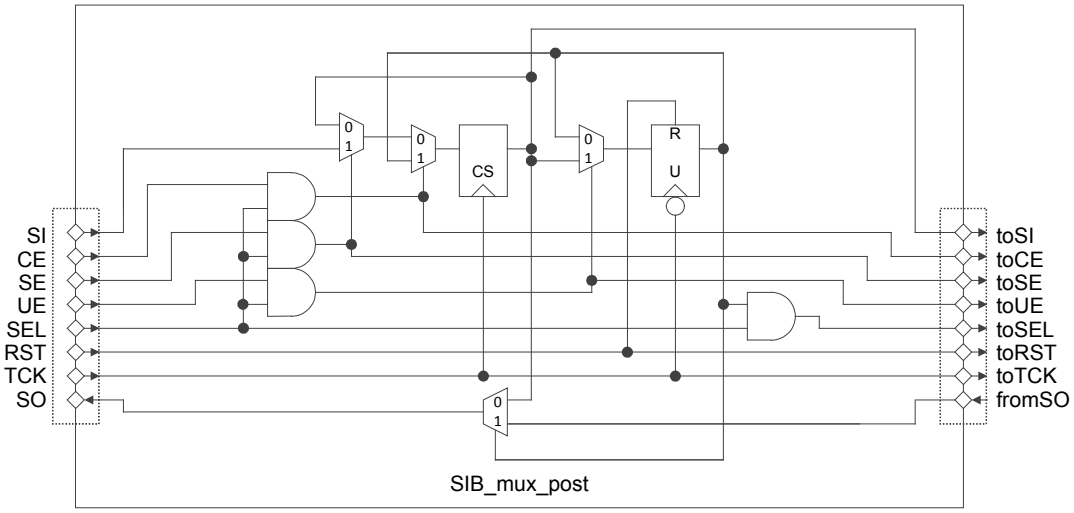


Figure F.10—SIB_mux_post example implementation

A large portion of the SIB_mux_post structure is the same as the basic scan register shown in Figure F.2. The new additions are the scan multiplexer located in the bottom center, the scan host port on the right side, the AND gate that drives the toSEL output. This logic forces the scan client interface that is connected to this scan host interface to be active only when both this module is selected (i.e., the SEL input is active) and there is a 1 in the update flip-flop of this module. The toSI port is driven from the output of the CS flip-flop and will be connected to a ScanInPort of another module. Likewise, the fromSO port will be driven by the ScanOutPort of another module. The implementation in Figure F.10 happens to show the scan host control signals (toCE, toSE, and toUE) being driven by the output of the 3 AND gates, but they could alternatively have been driven by the CE, SE, and UE inputs directly. This is because the scan client that will receive them will also receive the toSEL signal, which contains the required gating terms.

One risk associated with the SIB_post_mux topology is evident when the example of cascading several of these cells together is considered. Specifically, such a configuration will create a string of multiplexers in series on the ScanOut path, which could lead to timing issues. This can be addressed with a different SIB topology, as shown in the next subclause.

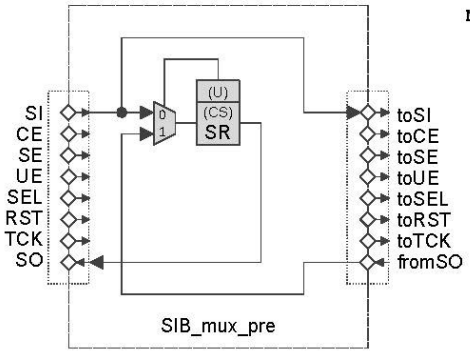
It is also important to note that this SIB_mux_post implementation assumes that the clock skew on TCK is carefully managed so that there are no race conditions between this SIB and the segment it is controlling.

F.3.1.2 SIB with multiplexer before scan register

The ICL description for a SIB with the multiplexer in front of the scan register is shown in Figure F.11.

The structure of the SIB_mux_pre not only places the ScanMux in front of the ScanRegister, it also drives the toSI port directly from the SI port instead of from the ScanOutPort of the ScanRegister (see the explanation for this in F.3.1.5).

An example implementation of the SIB_mux_pre structure is shown in Figure F.12.




```
Module SIB_mux_pre {
  ScanInPort     SI;
  CaptureEnPort  CE;
  ShiftEnPort    SE;
  UpdateEnPort   UE;
  SelectPort     SEL;
  ResetPort      RST;
  TCKPort        TCK;
  ScanOutPort    SO { Source SR;
                      Attribute LaunchEdge = "Rising"; }
  ScanInterface client {
    port SI; port CE; port SE; port UE;
    port SEL; port RST; port TCK; port SO;
  }
  ScanInPort       fromSO;
  ToCaptureEnPort  toCE;
  ToShiftEnPort    toSE;
  ToUpdateEnPort   toUE;
  ToSelectPort     toSEL;
  ToResetPort      toRST;
  ToTCKPort        toTCK;
  ScanOutPort      toSI { Source SI;
                          Attribute LaunchEdge = "Rising"; }
  ScanInterface host {
    Port fromSO; Port toCE; Port toSE; Port toUE;
    Port toSEL; Port toRST; Port toTCK; Port toSI;
  }
  ScanRegister SR {
    ScanInSource SIBmux; CaptureSource SR; ResetValue 1'b0;
  }
  ScanMux SIBmux SelectedBy SR {
    1'b0 : SI;
    1'b1 : fromSO;
  }
}
```

Figure F.11—SIB with multiplexer before scan register

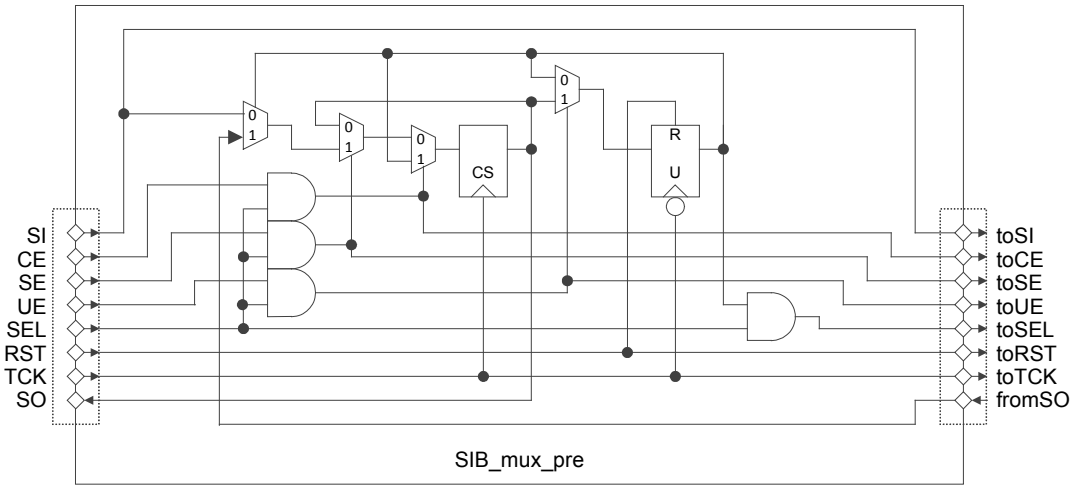


Figure F.12—SIB_mux_pre example implementation

Another trivial variant of this structure that captures a constant zero instead of the value in the update stage is shown in Figure F.13. Note that the ICL for this version would differ from that shown in Figure F.11—the CaptureSource would be “1’b0” instead of “SR.”

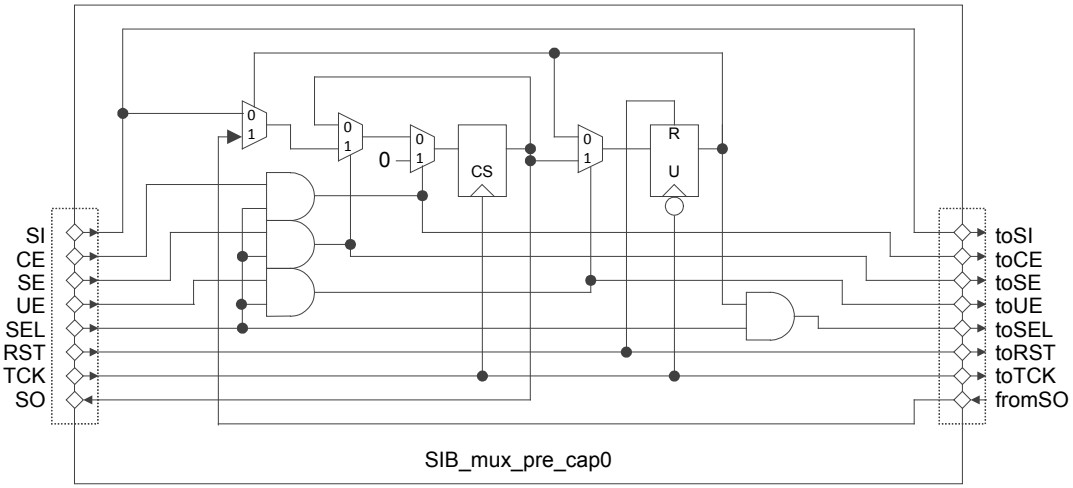


Figure F.13—SIB_mux_pre with 0-capture example implementation

Both Figure F.12 and Figure F.13 place the ScanMux in front of the ScanRegister and thus avoid the risk of long series-connected muxes discussed in the previous subclause. However, like the previous SIB_mux_post design, the SIB_mux_pre topology still has two potential timing issues associated with imperfect TCK distribution [similar to that discussed in Figure F.8(b)]. The next two subclauses present alternative designs that successively solve those problems.

F.3.1.3 SIB_mux_pre with lockup latch

Analogous to F.2.2, which showed how the addition of a lockup latch after the shift stage of a scan register reduces hold-time problems, a SIB can be similarly modified with the same benefit. An example implementation of the SIB_mux_pre with a lockup latch (again implemented as a negative-edge flip-flop) is shown in Figure F.14.

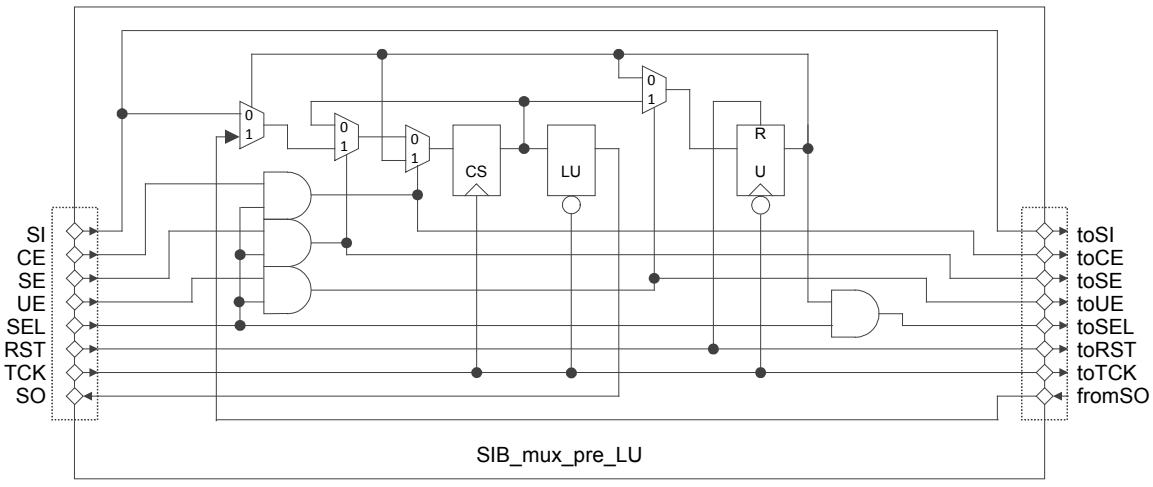


Figure F.14—SIB-mux_pre with lockup latch on ScanOut

Note that the update stage (the "U" flip-flop) is still driven by the shift stage ("CS"), not by the lockup latch. As was the case with the ICL in Figure F.1 and Figure F.6, there is no structural difference between in the ICL in Figure F.14 and Figure F.11, but an (optional) attribute on the SO port notes that the "LaunchEdge" is "Falling" for the design in Figure F.14. This will convey an extra half cycle of hold time margin, at the cost of a half cycle of setup time margin.

F.3.1.4 SIB_mux_pre_LU with safe select timing

All of the SIB cells shown thus far still have one potential timing problem: the toSEL signal from the host port changes on the falling edge of TCK when the UpdateEnPort (UE) is active, which could cause a race condition with the SelectPort of the scan client connected to this scan host port if there is skew on the TCK ports of the two modules. It is important that the downstream scan client respond to the current state of its SEL input, not the next state from the host's toSEL port that may race through. One solution to this problem is to introduce a pipeline delay stage on the toSEL port of the scan host interface, as shown in Figure F.15.

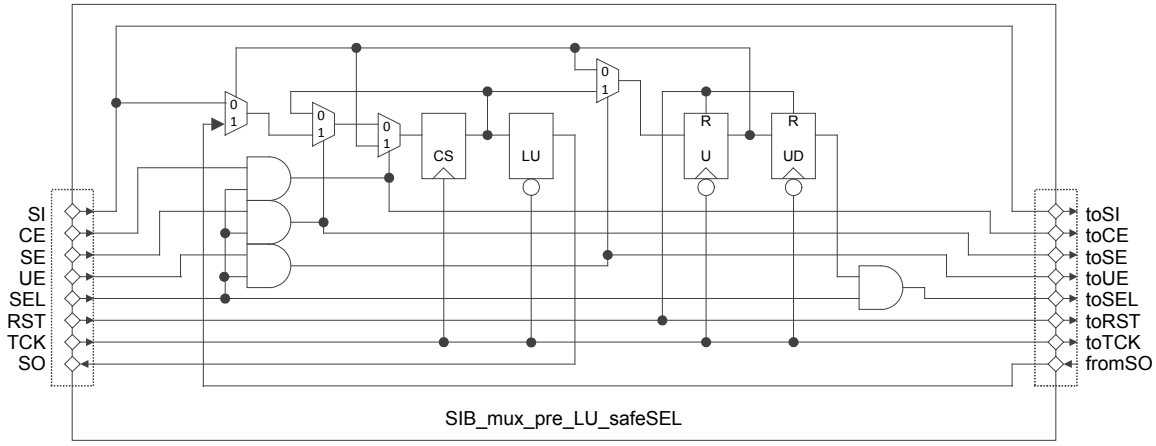


Figure F.15—SIB_mux_pre_LU with pipeline delay for safe toSEL

There is only one addition to the SIB_mux_pre_LU module here: the negative edge flip-flop labeled "UD" (for Update Delay). As the name implies, this flip-flop holds off the new state of the update cell going to the AND gate that drives the toSEL port. Importantly, that is the only destination of this new cell; the undelayed version of the update stage ("U") still controls the ScanMux (and the capture source and the recycled value back into the update stage). Note that the UD flip-flop could alternatively have been a positive-edge flop clocked on the rising edge of TCK.

The choice of a specific implementation of a SIB depends on the physical design parameters that matter most to a designer: if TCK is well balanced and timed, then one of the simpler SIBs (with smaller area) may be used. If low-skew design or timing closure are problematic, then one of the larger but more robust cells would be preferable.

F.3.1.5 Bad SIB topologies

Considering the four possible combinations of mux position (pre- or post-) and "toSI" source (SI or SR), only two have been shown (post- with SR in F.3.1, pre- with SI in F.3.2). The other two were not shown because they both cause irreparable problems, as shown in Figure F.16.

The pre- with SI combination in Figure F.16(a) can cause an inescapable loop if all 1s are shifted into the client. The post- with SR combination in Figure F.16(b) can result in an unresolvable conflict if the first bit of the client is to be set to 0 and the client to is remain on the active scan chain (since that same 0 will also be updated into the ScanRegister SR, which will close the ScanMux and deselect the client). These two topologies are best avoided.

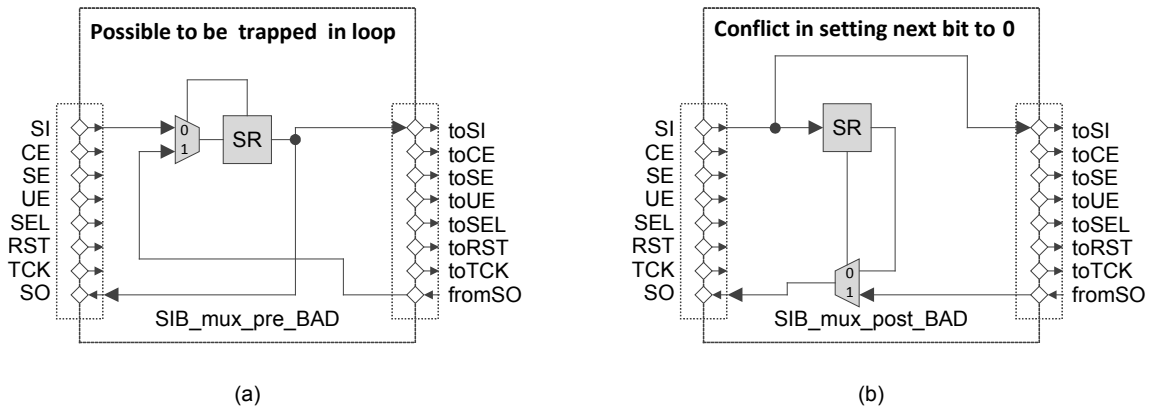


Figure F.16—Bad SIB topologies: to be avoided

F.3.2 Remotely controlled ScanMux (RSM)

F.3.2.1 Definition: in-line (adjacent or distant) vs. remote

Where the in-line SIBs described in F.3.1 placed the mux and the scan register controlling it adjacent to each other in the same shift path, there is no reason why some number of other flip-flops could not be interposed between them, as shown in Figure F.17 for the mux pre- and post- configurations.

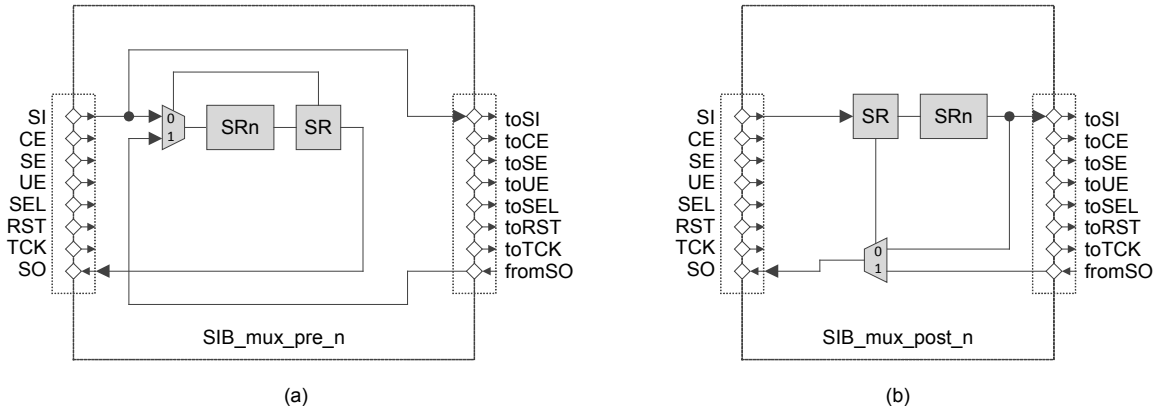


Figure F.17—In-line SIBs with distant (but not remote) control bits

In both Figure F.17(a) and Figure F.17(b), an arbitrary number of ScanRegister bits labeled “SRn” has been placed between ScanMux and its controlling ScanRegister (SR). This is just the simplest case; there could also be arbitrary logical hierarchy and more complex serial access network circuitry placed in the same location as the SRn boxes. The important point is that the scan mux and the scan register controlling it are in-line, meaning that the controlling ScanRegister is on the same active scan chain as the scan elements selected by the ScanMux, no matter if the two are adjacent or separated by some distance. This in-line configuration is contrasted with the remote configuration, where the ScanMux and its controlling ScanRegister are on different networks that cannot be selected simultaneously.

Annex G

(informative)

Bibliography

Bibliographical references are resources that provide additional or helpful material but do not need to be understood or used to implement this standard. Reference to these resources is made for informational use only.

NOTE—There is an unfortunate overloading of the square bracket ("[]") symbols as both the standard notation for bibliography references and the bit range of a vector quantity in both ICL and PDL. All bibliographic references start with a letter "B" followed by a number, which is not legal syntax for vector quantities in ICL or PDL. Though the context should be sufficiently clear to allow the two cases to be easily distinguished, to make it perfectly clear, bibliography references are only present in 1.3, 6.3.4, 7.7, and 8.1. Those subclauses contain no ICL or PDL vector quantities.

[B1] Available at: https://theantlrguy.atlassian.net/wiki/display/ANTLR4/Home.

[B2] IEEE Std 1149.1-1990, IEEE Standard Test Access Port and Boundary-Scan Architecture. [6, 7]

[B3] IEEE Std 1149.4-2010, IEEE Standard for a Mixed-Signal Test Bus.

[B4] IEEE Std 1149.6-2003, IEEE Standard for Boundary-Scan Testing of Advanced Digital Networks.

[B5] IEEE Std 1532-2002, IEEE Standard for In-System Configuration of Programmable Devices.

[B6] Osterhout, John K., and Jones, Ken, *Tcl and the Tk Toolkit.* Addison-Wesley, Boston, MA, 2010, ISBN 978-0-321-33633.

[6] IEEE publications are available from The Institute of Electrical and Electronics Engineers (http://standards.ieee.org/).
[7] The IEEE standards or products referred to in this clause are trademarks of The Institute of Electrical and Electronics Engineers, Inc.

Consensus

WE BUILD IT.

Connect with us on:

Facebook: https://www.facebook.com/ieeesa

Twitter: @ieeesa

LinkedIn: http://www.linkedin.com/groups/IEEESA-Official-IEEE-Standards-Association-1791118

IEEE-SA Standards Insight blog: http://standardsinsight.com

YouTube: IEEE-SA Channel

IEEE
standards.ieee.org
Phone: +1 732 981 0060 Fax: +1 732 562 1571