IEEE STANDARDS ASSOCIATION

IEEE Standard for Test Access Port and Boundary-Scan Architecture

IEEE Computer Society

Sponsored by the
Test Technology Standards Committee

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

13 May 2013

IEEE Std 1149.1™-2013
(Revision of
IEEE Std 1149.1-2001)

IEEE Std 1149.1™-2013
(Revision of
IEEE Std 1149.1-2001)

IEEE Standard for Test Access Port and Boundary-Scan Architecture

Sponsor

Test Technology Standards Committee
of the
IEEE Computer Society

Approved 6 February 2013

IEEE-SA Standards Board

Abstract: Circuitry that may be built into an integrated circuit to assist in the test, maintenance and support of assembled printed circuit boards and the test of internal circuits is defined. The circuitry includes a standard interface through which instructions and test data are communicated. A set of test features is defined, including a boundary-scan register, such that the component is able to respond to a minimum set of instructions designed to assist with testing of assembled printed circuit boards. Also, a language is defined that allows rigorous structural description of the component-specific aspects of such testability features, and a second language is defined that allows rigorous procedural description of how the testability features may be used.
Keywords: boundary scan, boundary-scan architecture, Boundary-Scan Description Language (BSDL), boundary-scan register, circuit boards, circuitry, IEEE 1149.1™, integrated circuit, printed circuit boards, Procedural Description Language (PDL), test, test access port (TAP), very high speed integrated circuit (VHSIC), VHSIC Hardware Description Language (VHDL)

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

PDF: ISBN 978-0-7381-8263-6 STD98160
Print: ISBN 978-0-7381-8264-3 STDPD98160

Notice and Disclaimer of Liability Concerning the Use of IEEE Documents: IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. 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.

Use of an IEEE Standard is wholly voluntary. IEEE disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon any IEEE Standard document.

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained in its standards is free from patent infringement. IEEE Standards documents are supplied "**AS IS**."

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. 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 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.

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 the official position of IEEE or any of its committees and shall not be considered to be, nor 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 to ensure 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. Any person who would like to participate in evaluating comments or revisions to an IEEE standard is welcome to join the relevant IEEE working group at http://standards.ieee.org/develop/wg/.

Comments on standards should be submitted to the following address:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854-4141
USA

Photocopies: Authorization to photocopy portions of any individual standard for internal or personal use is granted by The Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, 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.

Notice to users

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

Updating of IEEE 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. 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://standards.ieee.org/index.html or contact the IEEE at the address listed previously. For more information about the IEEE Standards Association or the IEEE standards development process, visit IEEE-SA Website at http://standards.ieee.org/index.html.

Errata

Errata, if any, for this and all other standards can be accessed 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 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 standard was submitted to the IEEE-SA Standards Board for approval, the P1149.1 Working Group had the following membership:

C. J. Clark, *Chair*
Carol Pyron, *Vice-Chair*
Carl F. Barnhart, *Editor*
Bill Tuthill, *Secretary*

John Braden
Bill Bruce
Richard Cornejo
Adam Cron
Wim Driessen
David Dubberke
Ted Eaton
Heiko Ehrenberg
William Eklow
Peter Elias
Joshua Ferry
Jeff Halnon
Dharma Konda
Roland R. Latvala
Adam W. Ley
Sankaran Menon
Kent Ng
Kenneth P. Parker
Francisco Russi
John Seibold
Roger Sowada
Craig Stephan
Brian Turmelle
Hugh Wallace

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

Gobinathan Athimolom
Carl F. Barnhart
Hugh Barrass
William Borroz
John Braden
Dennis Brophy
Susan Burgess
Gunnar Carlsson
Vivek Chickermane
C. J. Clark
Richard Cornejo
Adam Cron
Alfred Crouch
Frans G De Jong
Jason Doege
Wim Driessen
David Dubberke
Sourav Dutta
Heiko Ehrenberg
William Eklow
Peter Elias
Joshua Ferry
Chris Gorringe
Prashant Goteti
Robert Gottlieb
J. Grealish
Randall Groves
Jeff Halnon
Peter Harrod
Neil Glenn Jacobson
Rohit Kapur
Dharma Konda
Roland R. Latvala
Philippe LeBourg
Adam W. Ley
Teresa Lopes
Greg Luri
Wayne Manges
Colin Maunder
Ian Mcintosh
Harrison Miles Jr.
Jeffrey Moore
Benoit Nadeau-Dostie
Ion Neag
Kenneth P. Parker
Steve Poehlman
Ulrich Pohl
Irith Pomeranz
John Potter
Carol Pyron
Mike Ricchetti
Gordon Robinson
Andrzej Rucinski
Francisco Russi
Bartien Sayogo
John Seibold
Ozgur Sinanoglu
Roger Sowada
Craig Stephan
Cees Stork
Walter Struppler
Stephen Sunter
Bambang Suparjo
Anthony Suto
Efren Taboada
David Thompson
Brian Turmelle
Bill Tuthill
Louis Ungar
Dmitri Varsanofiev
Srinivasa Vemuru
John Vergis
Tom Waayers
Douglas D. Way
Thomas Williams
Henk Wit
Oren Yuen
Janusz Zalewski

When the IEEE-SA Standards Board approved this standard on 6 February 2013, it had the following membership:

Richard H. Hulett, *Chair*
Robert Grow, *Past Chair*
Konstantinos Karachalios, *Secretary*

Satish Aggarwal	Alexander Gelman	Oleg Logvinov
Masayuki Ariyoshi	Paul Houzé	Ted Olsen
Peter Balma	Jim Hughes	Gary Robinson
William Bartley	Young Kyun Kim	Jon Walter Rosdahl
Ted Burse	Joseph L. Koepfinger*	Mike Seavey
Clint Chaplin	David J. Law	Yatin Trivedi
Wael William Diab	Thomas Lee	Phil Winston
Jean-Philippe Faure	Hung Ling	Yu Yuan

*Member Emeritus

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

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

Don Messina
IEEE Standards Program Manager, Document Development

Kathryn Bennett
IEEE Standards Program Manager, Technical Program Development

Introduction

This introduction is not part of IEEE Std 1149.1-2013, IEEE Standard for Test Access Port and Boundary-Scan Architecture.

This standard defines a test access port and boundary-scan architecture for digital integrated circuits and for the digital portions of mixed analog/digital integrated circuits. The facilities defined by the standard seek to provide a solution to the problem of testing assembled printed circuit boards and other products based on highly complex digital integrated circuits and high-density surface-mounting assembly techniques. They also provide a means of accessing and controlling design-for-test features built into the digital integrated circuits themselves. Such features might, for example, include internal scan paths and self-test functions as well as other features intended to support service applications in the assembled product.

In addition, two languages are provided to describe both the structure of the test logic and the procedures needed to use the test logic.

History of the development of this standard

The process of developing this standard began in 1985 when the Joint European Test Action Group (JETAG) was formed in Europe. During 1986, this group expanded to include members from both Europe and North America and, as a result, was renamed the Joint Test Action Group (JTAG). Between 1986 and 1988, the JTAG Technical Subcommittee developed and published a series of proposals for a standardized form of boundary scan. In 1988, the last of these proposals, JTAG Version 2.0, was offered to the IEEE Testability Bus Standards Committee (P1149) for inclusion in the standard then under development. The Testability Bus Standards Committee accepted this approach. It decided that the JTAG proposal should become the basis of a standard within the Testability Bus family, with the result that the P1149.1 project was initiated. Following these decisions, the JTAG Technical Subcommittee became the core of the IEEE Working Group that developed this standard.

After the initial approval of this standard in February 1990 and its subsequent publication, the Working Group immediately began efforts to develop a supplement for the purpose of correction, clarification, and enhancement. This effort, spurred and guided by interaction between developers and users of the original standard, culminated in IEEE Std 1149.1a™-1993, which was approved in June 1993.

The major changes to this standard introduced by IEEE Std 1149.1a-1993 were as follows:

— The addition of two optional instructions, *CLAMP* and *HIGHZ*, which standardized the names and specifications of features often implemented as design-specific features.

— The addition of an optional facility to switch a component from a mode in which it complies to this standard into one in which it supports another design-for-test approach.

Furthermore, starting with a proposal made by Kenneth P. Parker and Stig Oresjo in 1990, an effort was undertaken to develop a language to describe components that conform to this standard. This effort concluded in the approval of IEEE Std 1149.1b™-1994 in September 1994.

The major change introduced to this standard by IEEE Std 1149.1b-1994 was the addition of Annex B, which defines the Boundary-Scan Description Language. All other changes were minor and were strictly for clarification.

The 2001 revision was primarily a housekeeping update, designed to incorporate learning from the first 10 years of the standard's use into the standard document. The principal changes introduced were as follows:

— To reduce the risk of accidental entry into test mode, the requirement that a binary code for the *EXTEST* instruction be {000...0} was removed and use of this binary code for other instructions that result in entry to test mode was deprecated.

— To increase the flexibility with which instructions may be implemented and merged, the implicitly merged *SAMPLE/PRELOAD* instruction was redefined as two separate instructions: *SAMPLE* and *PRELOAD*. *These instructions can continue to share a single binary code*, effectively resulting in a merged *SAMPLE/PRELOAD* instruction, but alternatively, they may now share binary codes with other instructions, provided that no rules applying to any of the merged instructions are violated.

— To enable more efficient implementation of boundary-scan register cells provided at system logic outputs, the source of data to be captured in such cells in response to the *SAMPLE* instruction was allowed to be at the connected system pin. Additionally, three new cell types based on this implementation (**BC_8, BC_9,** and **BC_10**) were added to the standard Boundary-Scan Description Language (BSDL) Package and Package Body.

— To permit more flexible boundary-scan register cell implementations, sharing of circuitry between the boundary-scan register and other elements of the test and/or system logic were allowed in limited cases.

— To support more complete description of IC pin drivers with bus keeper circuits, a new value for <disable result> was defined (**KEEPER**).

— To track the widespread acceptance of BSDL, the language was made a normative part of this standard and its use for documentation was mandated.

Additionally, a number of minor changes were made to correct and clarify the language of this standard.

Changes introduced by this revision

First, this version of the standard affirms what had been required in the previous (2001) version. There are only minor clarifications or relaxations to the rules that are already established. It is expected that components currently compliant with the previous version of this standard will remain compliant with this one. The one exception is that the previously deprecated **BC_6** boundary-scan cell is no longer supported or defined, and the component supplier must provide a user-supplied BSDL package defining the **BC_6** cell for any component using the **STD_1149_1_2013** standard Package and still using that cell.

Second, while this is a major revision, items introduced in this version are optional and intended to provide test improvements for the complex components being created today and in the foreseeable future. There are also significant improvements in documentation capability, including the introduction of a new language to document test procedures unique to the component.

The major changes, listed in the order in which they appear in this standard, are as follows:

In the standard body:

— A new, optional, test mode persistence controller that can maintain the IEEE 1149.1 test logic in test mode even if the active instruction does not force test mode. Clause 6 is now split into 6.1 for the TAP controller and 6.2 for the test mode persistence controller. In support of this new controller, there are three new instructions: *CLAMP_HOLD* and *TMP_STATUS* in 8.20, with the new TMP status test data register in Clause 16; and *CLAMP_RELEASE* in 8.20.

— A new, optional *ECIDCODE* instruction in 8.15 and its electronic chip identification test data register in Clause 13 to supplement the existing *IDCODE* and *USERCODE* instructions and allow for the recovery of an Electronic Chip Identification value used to identify and track individual integrated circuits.

— A new, optional, component initialization mechanism to provide more flexibility in preparing the component for test. The *INIT_SETUP, INIT_SETUP_CLAMP,* and *INIT_RUN* instructions in 8.17, 8.18, and 8.19, and their new initialization data and initialization status test data registers in Clause 14 and Clause 15, respectively. This will allow programmable input/output (I/O) to be set up prior to board or system testing, as well as any tasks required to put the system logic into a safe state for test.

— A new, optional, *IC_RESET* instruction in 8.21 and its reset_select test data register in Clause 17 to provide control of component reset functions through the TAP.

— In 9.2, an optional standard TAP to test data register interface is recommended, and examples of different types of test data register cells using this interface are shown. In addition, the concept of register segments is expanded to allow for segments that may be excluded or included. This is introduced to support power domains that may be powered down, and yet may have a segment of a test data register within that domain. However, the capability was kept general.

— In the new 9.4, the rules for defining and controlling the new excludable and selectable segments are established.

— Boundary-scan register description in Clause 11 has been updated to support:

  i) Optional excludable (but not selectable) boundary-scan register segments
  ii) Optional observe-only boundary-scan register cells to redundantly capture the signal value on all digital pins except the TAP pins
  iii) Optional observe-only boundary-scan register cells to capture a fault condition on all pins, including nondigital pins, except the TAP pins

— Documentation requirements in Clause 18 have been updated for the new capabilities.

Note that where rules were removed or moved in this version of this standard, a placeholder was left behind ("Removed in this version ...." or "Moved to Permision ....") in order to preserve the rule numbering from previous versions. This is intended to simplify the transition for both users and tool vendors in supporting what is a significant change.

In Annex B (Boundary Scan Description Language):

— The entire annex was rewritten for:

  i) Increased clarity of what was normative versus descriptive text
  ii) Increased consistency in presentation

— BSDL is no longer a "proper subset" of VHDL, but it is now "based on" VHDL. See B.4. In particular, new pin type keywords were introduced in B.8.3 that are not needed in VHDL but give a more accurate description of each port in BSDL.

— Formal definitions of language elements are included in B.5 instead of reliance on inheritance from VHDL.

— Some changes to the BNF notation used, including definition of all the special character tokens, are in B.6.

— Pin mapping in B.8.7 now allows for documenting that a port is not connected to any device package pin in a specific mapped device package.

— The boundary-scan register description in B.8.14 introduces new attributes for defining boundary-scan register segments, and introduces a requirement for documenting the behavior of an undriven input.

— New capabilities are introduced for documenting the structural details of test data registers:

  i) Subclause B.8.18 introduces the definition of mnemonics that may be associated with register fields.
  ii) Subclause B.8.19 introduces the ability to name fields within a register or segment.
  iii) Subclause B.8.20 introduces the ability to define the types of cells used in a test data register (TDR) field.
  iv) Subclause B.8.21 introduces the ability to hierarchically assemble segments into larger segments or whole registers.
  v) Subclause B.8.22 introduces the ability to define constraints on the values to be loaded in a register or register field.
  vi) Subclause B.8.23 introduces the ability to associate a register field or bit to specific ports and other information, and to associate a power port to other ports.

— The role of a User Defined Package defined in B.10 has been expanded to support logic IP providers who may need to document test data register segments contained within their IP.

A new annex, Annex C, codifies the Procedural Description Language (PDL), a new language for documenting the procedural and data requirements for some of the new instructions. As mentioned, this version of the standard introduces new instructions for configuring complex I/Os prior to entering the *EXTEST* instruction. As the data required for initialization could vary for each use of the component on each distinct board or system design, this

created the need for a new language for setting internal TDR register fields in order to configure the I/O. It was decided to adopt PDL and tailor it to the BSDL register descriptions and IEEE 1149.1 needs.

A new informative annex, Annex D, shows extended examples of BSDL and PDL used together to describe the structure and the procedures for use of new capabilities.

A new informative annex, Annex E, shows example pseudo-code for the execution of the PDL iApply command, the most complex of the new commands in PDL.

Contents

1. Overview 1
1.1 Scope 1
1.2 Purpose 1
1.3 Document outline 5
1.4 Text conventions 6
1.5 Logic diagram conventions 6

2. Normative references 7

3. Definitions, abbreviations, acronyms, and special terms 8
3.1 Definitions 8
3.2 Abbreviations and acronyms 11
3.3 Special terms 12

4. Test access port (TAP) 13
4.1 Connections that form the TAP 13
4.2 Test clock input (TCK) 13
4.3 Test mode select (TMS) input 14
4.4 Test data input (TDI) 15
4.5 Test data output (TDO) 15
4.6 Test reset input (TRST*) 16
4.7 Interconnection of components compatible with this standard 17
4.8 Subordination of this standard within a higher level test strategy 20

5. Test logic architecture 22
5.1 Test logic design 22
5.2 Test logic realization 23

6. Test logic controllers 24
6.1 TAP controller 24
6.2 Test mode persistence (TMP) controller 39

7. Instruction register 46
7.1 Design and construction of the instruction register 46
7.2 Instruction register operation 47

8. Instructions 50
8.1 Response of the test logic to instructions 50
8.2 Public instructions 51
8.3 Private instructions 53
8.4 *BYPASS* instruction 53
8.5 Boundary-scan register instructions 54
8.6 *SAMPLE* instruction 57
8.7 *PRELOAD* instruction 58
8.8 *EXTEST* instruction 60
8.9 *INTEST* instruction 62
8.10 *RUNBIST* instruction 66
8.11 *CLAMP* instruction 68
8.12 Device identification register instructions 70
8.13 *IDCODE* instruction 70
8.14 *USERCODE* instruction 71
8.15 *ECIDCODE* instruction 73
8.16 *HIGHZ* instruction 74

8.17 Component initialization instructions and procedures 76
8.18 *INIT_SETUP* and *INIT_SETUP_CLAMP* instructions 81
8.19 *INIT_RUN* instruction 82
8.20 *CLAMP_HOLD*, *CLAMP_RELEASE*, and *TMP_STATUS* instructions 84
8.21 *IC_RESET* instruction 88

9. Test data registers 91
9.1 Provision of test data registers 91
9.2 Design and construction of test data registers 94
9.3 Operation of test data registers 106
9.4 Design and control of test data register segments 108

10. Bypass register 116
10.1 Design and operation of the bypass register 116

11. Boundary-scan register 118
11.1 Introduction 118
11.2 Register design 122
11.3 Register operation 124
11.4 General rules regarding cell provision 126
11.5 Provision and operation of cells at system logic inputs 131
11.6 Provision and operation of cells at system logic outputs 138
11.7 Provision and operation of cells at bidirectional system logic pins 154
11.8 Redundant cells 161
11.9 Special cases 163

12. Device identification register 166
12.1 Design and operation of the device identification register 166
12.2 Manufacturer identity code 168
12.3 Part-number code 169
12.4 Version code 170

13. Electronic chip identification (ECID) register 171
13.1 Design and operation of the ECID register 171

14. Initialization data register 172
14.1 Design and operation of the initialization data register 172

15. Initialization status register 175
15.1 Design and operation of the initialization status register 175

16. TMP status register 176
16.1 Design and operation of the TMP status register 176

17. Reset selection register 178
17.1 Design and operation of the reset selection register 178

18. Conformance and documentation requirements 183
18.1 Claiming conformance to this standard 183
18.2 Prime and second source components 184
18.3 Documentation requirements 184

Annex A (informative) Example implementation using level-sensitive design techniques 188

Annex B (normative) Boundary Scan Description Language (BSDL) 189
B.1 General information 189
B.1.1 Document outline 189
B.1.2 Conventions 189
B.1.3 BSDL history 189
B.2 Purpose of BSDL 190
B.3 Scope of BSDL 190
B.4 Relationship of BSDL to VHDL 191
B.4.1 Specifications 191
B.5 Lexical elements of BSDL 192
B.5.1 Character set 192
B.5.2 BSDL reserved words 193
B.5.3 VHDL reserved and predefined words 194
B.5.4 Identifiers 195
B.5.5 Numeric literals 196
B.5.6 Strings 197
B.5.7 Information tag 198
B.5.8 Comments 199
B.6 Syntax definition 199
B.6.1 BNF conventions 199
B.6.2 Commonly used syntactic elements 200
B.7 Components of a BSDL description 202
B.7.1 Specifications 202
B.7.2 Description 203
B.8 Entity description 203
B.8.1 Overall syntax of the entity description 203
B.8.2 Generic parameter statement 204
B.8.3 Logical port description statement 205
B.8.4 Standard use statement 208
B.8.5 Use statement 211
B.8.6 Component conformance statement 212
B.8.7 Device package pin mappings 213
B.8.8 Grouped port identification 216
B.8.9 Scan port identification 219
B.8.10 Compliance-enable description 220
B.8.11 Instruction register description 221
B.8.12 Optional device register description 224
B.8.13 Register access description 227
B.8.14 Boundary-scan register description 229
B.8.15 RUNBIST description 245
B.8.16 INTEST description 247
B.8.17 System clock requirements attribute 249
B.8.18 Register mnemonics description 250
B.8.19 Register fields description 254
B.8.20 Register field assignment description 261
B.8.21 Register assembly description 271
B.8.22 Register constraint description 288
B.8.23 Register and power port association attributes 291
B.8.24 User extensions to BSDL 295
B.8.25 Design warning 297
B.9 Standard BSDL Package STD_1149_1_2013 298
B.10 User-supplied BSDL packages 302
B.10.1 Specifications 302
B.10.2 Description 306
B.10.3 Examples 307

B.11 BSDL example applications 308
B.11.1 Typical application of BSDL 308
B.11.2 Boundary-scan register description 311
B.12 1990 version of BSDL 315
B.12.1 1990 Standard VHDL Package STD_1149_1_1990 316
B.12.2 Typical application of BSDL, 1990 version 319
B.12.3 Obsolete syntax 320
B.12.4 Miscellaneous points on 1990 version 321
B.13 1994 version of BSDL 321
B.13.1 Standard VHDL Package STD_1149_1_1994 321
B.14 2001 version of BSDL 325
B.14.1 Standard VHDL Package STD_1149_1_2001 325

Annex C (normative) Procedural Description Language (PDL) 329
C.1 General information 329
C.1.1 Purpose 329
C.1.2 Dependence on Tool Command Language (Tcl) 330
C.1.3 Dependence on Boundary Scan Description Language (BSDL) 330
C.2 PDL concepts and use model 330
C.2.1 Use model introduction 330
C.2.2 PDL levels 332
C.2.3 PDL procedures 333
C.2.4 Read and write with capture-shift-update sequence 334
C.2.5 Register state definition 334
C.2.6 Level-0 PDL commands 336
C.2.7 Specification of names and values 339
C.2.8 Retargeting 340
C.2.9 Simple PDL Example 341
C.3 PDL Level 0 command reference 343
C.3.1 Understanding a PDL “string” 343
C.3.2 BNF conventions 344
C.3.3 PDL lexical elements and common syntax 345
C.3.4 PDL File 350
C.3.5 Procedure definition commands 351
C.3.6 Test setup commands 356
C.3.7 Test execution commands 360
C.3.8 Flow-control commands 367
C.3.9 Optimization commands 374
C.3.10 Miscellaneous commands 378
C.3.11 Low-level commands 379
C.4 PDL Level 1 command reference 382
C.4.1 Level-1 PDL operation 383
C.4.2 iGet command 383
C.4.3 iGetStatus command 388
C.5 Example BSDL and PDL for the use model 388
C.5.1 BSDL Packages for IP 389
C.5.2 BSDL files for components 390
C.5.3 PDL files supplied by IP supplier 393
C.5.4 PDL files supplied by component supplier 394
C.5.5 PDL files coded by test engineer 395

Annex D (informative) Integrated examples of BSDL and PDL 398
D.1 Initialization example structure and procedures 398
D.1.1 Initialization example using register description attributes 398
D.1.2 Example PDL for INIT example 405

D.2 Multiple wrapper serial port structure and procedures 408
D.2.1 Wrapper serial port structural description.....408
D.2.2 Wrapper serial port example.....417

Annex E (informative) Example iApply execution flow..... 420

Figures

Figure 1-1—Boundary-scan register cell....3
Figure 1-2—Boundary-scannable board design....4
Figure 1-3—Logic symbology used in this standard....6
Figure 4-1—Serial connection using one TMS signal....18
Figure 7-2—Instruction register with decoder between shift and update stages.....49
Figure 8-1—Simplified view of the boundary-scan register....55
Figure 8-2—Example boundary-scan register cell design....56
Figure 8-3—Figure used to illustrate boundary-scan instructions....56
Figure 8-4—Data flow for the *SAMPLE* instruction....58
Figure 8-5—Data flow for the *PRELOAD* instruction....60
Figure 8-6—Data flow for the *EXTEST* instruction....62
Figure 8-7—Data flow for the *INTEST* instruction....64
Figure 8-8—Control of applied system clock during *INTEST*....65
Figure 8-9—Use of TCK as clock for on-chip system logic during *INTEST*....65
Figure 8-10—Use of the *HIGHZ* instruction....75
Figure 8-11—Provision of *HIGHZ* at a two-state pin....76
Figure 8-12—Boundary-scan register control cell with a reset on the update flip-flop R2....87
Figure 9-1—Implementation of the group of test data registers....92
Figure 9-2—Construction of multiple test data registers from shared circuitry....96
Figure 9-3—Gated-clock boundary-scan register gating....97
Figure 9-4—Test data register control gating....97
Figure 9-5—Capture-update TDR cell using gated clocks....98
Figure 9-6—Capture-update TDR cell with nongated clock and optional reset....100
Figure 9-7—Update TDR cell without capture and with nongated clock and optional reset....101
Figure 9-8—Capture TDR cell with nongated clock and without update stage....102
Figure 9-9—Shift-only TDR cell with nongated clock and without update stage.....102
Figure 9-10—Self-monitoring TDR cell with update stage and nongated clocks....104
Figure 9-11—Self-resetting and self-monitoring TDR cell with nongated clocks....105
Figure 9-12—Timing of a self-resetting and self-monitoring TDR cell at *Update-DR*....106
Figure 9-13—Example design containing two optional test data registers....107
Figure 9-14—Scan control of excludable test data register segments....112
Figure 9-15—Scan control of excludable test data register segments with domain control....113
Figure 9-16—Domain POR reset of nested segment-select fields....114
Figure 9-17—Hierarchical reset of nested segment-select fields....114
Figure 9-18—Selectable segments and selection field....115
Figure 9-19—Example segment-select or selection-field cell with ungated clocks....115
Figure 10-1—Bypass register gated-clock implementation....116
Figure 11-1—Component without boundary scan....119
Figure 11-2—Input connections....120
Figure 11-3—Connection of an observe-only boundary-scan register cell....121
Figure 11-4—Insertion of a control-and-observe boundary-scan register cell....121
Figure 11-5—Conceptual view of a control-and-observe boundary-scan register cell....122
Figure 11-6—Boundary-scan shift-register design....124
Figure 11-7—Component that contains analog circuitry....126
Figure 11-8—Placement of boundary-scan register cells....128
Figure 11-9—Component with differential inputs and outputs....129
Figure 11-10—Conceptual schematic of redundant observe-only cells on differential pins....130
Figure 11-11—Provision of a boundary-scan register cell at a system input....131
Figure 11-12—Provision of multiple boundary-scan register cells at one input....132
Figure 11-13—Noninversion of data between pin and TDO....134

Figure 11-14—Noninversion of data between TDI and the system logic....134
Figure 11-15—Input cell with parallel output register [BC_2] 136
Figure 11-16—Input cell without parallel output register [BC_3] 136
Figure 11-17—Cell that forces the system logic input to 1 during *EXTEST* [BC_4] 137
Figure 11-18—Observe-only input cell without control [BC_4]....137
Figure 11-19—Input cell that supports all instructions [BC_1]....138
Figure 11-20—Provision of a boundary-scan register cell at a digital system output pin 139
Figure 11-21—Provision of boundary-scan register cells at system logic outputs....140
Figure 11-22—Provision of cells when one output Is used both as control and data 140
Figure 11-23—Noninversion of data between the system logic and TDO....142
Figure 11-24—Noninversion of data between TDI and a system output pin 143
Figure 11-25—Noninversion of control signal values between the system logic and TDO....144
Figure 11-26—Control of multiple three-state outputs from one signal....145
Figure 11-27—Testing board-level bus lines 146
Figure 11-28—Testing external logic via the boundary-scan register....147
Figure 11-29—Primitive noncompliant output cell design with potential problems....147
Figure 11-30—Circuit illustrating potential boundary-scan test problem 148
Figure 11-31—Output cell that supports all instructions [BC_1]....149
Figure 11-32—Output cell that does not support *INTEST* [BC_2]....149
Figure 11-33—Self-monitoring output cell that supports *INTEST* [BC_9] 151
Figure 11-34—Self-monitoring output cell that does not support *INTEST* [BC_10] 151
Figure 11-35—Boundary-scan register cells at a three-state output—Example 1 [BC_1, control and data] 152
Figure 11-36—Boundary-scan register cells at a three-state output—Example 2 [BC_2, control and data] 153
Figure 11-37—Boundary-scan register cells at a bidirectional pin—Example 1 [BC_1, control] 155
Figure 11-38—Boundary-scan register cells at a bidirectional pin—Example 2 [BC_2 control; BC_7 data] 156
Figure 11-39—Deprecated boundary-scan register cells at a bidirectional pin [BC_2 control; BC_6 data]....158
Figure 11-40—Boundary-scan register cells at an open-collector bidirectional pin [BC_4, input; BC_2, output]...159
Figure 11-41—Boundary-scan register cell at an open-collector bidirectional pin [BC_8] 160
Figure 11-42—Boundary-scan register cells for use at a bidirectional pin where *INTEST* is not provided [BC_2, control; BC_8, data]....161
Figure 11-43—Cell that should not be included in the boundary-scan register....163
Figure 11-44—Input pins used only to control output pins—Case A 164
Figure 11-45—Input pins used only to control output pins—Case B....164
Figure 11-46—Noncompliant use of a single cell for output control and data....165
Figure 11-47—Boundary-scan register cells at a three-state pin where output control is from a system pin [BC_5, control; BC_1, data]....165
Figure 12-1—Structure of the device identification code....166
Figure 12-2—Device identification register gated-clock cell design....167
Figure 16-1—Example TMP status register (nongated clocks)....177
Figure 17-1—Reset selection register overview....178
Figure 17-2—Minimal reset selection register example....181
Figure 18-1—Measuring setup and hold timing....187
Figure 18-2—Measuring propagation delay....187
Figure B-1—Components of a BSDL description....202
Figure B-2—Example use of nonboundary-scan port-types....208
Figure B-3—Example of unconnected pin types....216
Figure B-4—Cell design corresponding to Figure 11-19 and Figure 11-31 230
Figure B-5—Symbolic representation of a boundary-scan register cell....230
Figure B-6—Symbolic representation of a boundary-scan register cell without an update stage 231
Figure B-7—Cell on an input, which pulls to a logic 1 243
Figure B-8—Illustration of use of <input spec> for an IC 243
Figure B-9—Illustration of use of fault detection boundary cells for an IC....244
Figure B-10—Simple local reset structure 270
Figure B-11—Simple selectable register segment structure....280

Figure B-12—Illustrative component power control structure....283
Figure B-13—Simple wrapper serial port....285
Figure B-14—Three wrappers in parallel....287
Figure B-15—Boundary-scan register cell showing possible capture sources....306
Figure B-16—Texas Instruments SN74BCT8374....309
Figure B-17—Component that illustrates several OBSERVE_ONLY and INTERNAL cells....312
Figure B-18—Component that illustrates several merged cells....314
Figure C-1—PDL example board....331
Figure C-2—PDL example detail....332
Figure C-3—PDL scan frame....335
Figure C-4—Data flow during an iApply command....336
Figure C-5—iMerge example....375
Figure C-6—Example circuit board....389
Figure D-1—Hard SerDes IP defined in a package....399
Figure D-2—Simple wrapper serial port....408
Figure D-3—Three wrappers with WSC gating logic....411
Figure D-4—WSP example for interconnect testing....412
Figure D-5—Three wrappers with WSC gating logic and SEGSEL....415

Tables

Table 6-1—Use of controller states for different test types....29
Table 6-2—Test logic operation in each controller state....33
Table 6-3—State assignments for example TAP controller....36
Table 7-1—Instruction register operation in each controller state....47
Table 8-1—Typical initialization sequence, deferred test mode....79
Table 8-2—Including boundary-scan segments in mission mode....80
Table 8-3—Typical initialization sequence, immediate test mode....80
Table 8-4—Including boundary-scan segments in test mode....81
Table 8-5—I/O pin behavior for TMP controller states....86
Table 9-1—Recommended TDR interface for design specific TDRs....95
Table 9-2—Naming of test data registers that share circuitry....96
Table 11-1—Routing of signals in cells at system logic inputs....132
Table 11-2—Mode signal generation for the example cells in Figure 11-15 and Figure 11-16....136
Table 11-3—Mode signal generation for the example cell in Figure 11-19....138
Table 11-4—Routing of signals in cells at system logic outputs....142
Table 11-5—Test for driver B....146
Table 11-6—Mode signal generation for the example cells in Figure 11-31, Figure 11-35, Figure 11-37, and Figure 11-47....149
Table 11-7—Mode signal generation for the example cells in Figure 11-32, Figure 11-34, and Figure 11-40....150
Table 11-8—Mode signal generation for the example cell in Figure 11-33....151
Table 11-9—Mode signal generation for the example cell in Figure 11-36....154
Table 11-10—Mode signal generation for the example cells in Figure 11-38....157
Table 11-11—Mode signal generation for the deprecated example cells in Figure 11-39....159
Table 11-12—Mode signal generation for the example cells in Figure 11-41 and Figure 11-42....160
Table 17-1—Logic hazards of dual transitions of reset-enable and reset-control pairs....182
Table 18-1—Public instructions....183
Table B-1—Scope of BSDL....191
Table B-2—Pin types....206
Table B-3—List of cells defined in the Standard BSDL Package and relevant figure numbers....229
Table B-4—Function element values and meanings....238
Table B-5—Constraint expression operators....290
Table B-6—Unit value definitions....293
Table B-7—Cell context element values and meanings....303
Table B-8—Data source element values and meanings....304
Table B-9—Compliant capture sources for <cell context> of INPUT, CLOCK, and BIDIR_IN....304
Table B-10—Compliant capture sources for <cell context> of OUTPUT2, OUTPUT3, and BIDIR_OUT....304
Table B-11—Compliant capture sources for <cell context> of CONTROL and CONTROLR....304
Table B-12—Compliant capture sources for <cell context> of INTERNAL....305
Table B-13—Compliant capture sources for <cell context> of OBSERVE_ONLY....305
Table C-1—PDL Level-0 commands....337
Table C-2—Handling PDL procedure hierarchy....341
Table C-3—PDL Level-1 commands....383

IEEE Standard for Test Access Port and Boundary-Scan Architecture

1. Overview

1.1 Scope

This standard defines test logic that can be included in an integrated circuit to provide standardized approaches to:

- Testing the interconnections between integrated circuits once they have been assembled onto a printed circuit board or other substrate
- Testing the integrated circuit itself
- Observing or modifying circuit activity during the component's normal operation

The test logic consists of a boundary-scan register and other building blocks and is accessed through a test access port (TAP).

1.2 Purpose

1.2.1 Overview of the operation of this standard

This subclause provides a general overview of the operation of a component compatible with this standard and provides a background to the detailed discussion in later clauses.

The circuitry defined by this standard allows test instructions (which take control of the component outputs and observe the component inputs) and associated test data to be fed into a component and, subsequently, allows the results of execution of such instructions to be read out. All information (instructions, test data, and test results) is communicated in a serial format.

The sequence of operations would be controlled by a bus master, which could be either an automatic test equipment (ATE) or a component that interfaces to a higher level test bus as a part of a complete system maintenance architecture. Control is achieved through signals applied to the test mode select (TMS) and test clock (TCK) inputs of the various components connected to the bus master. Starting from an initial state in which the test circuitry defined by this standard is inactive, a typical sequence of operations would be as follows.

The first steps would be, in general, to load serially into the component the instruction binary code for the particular operation to be performed. The test logic defined by this standard is designed such that the serial movement of instruction information is not apparent to those circuit blocks whose operation is controlled by the instruction. The instruction applied to these blocks changes only on completion of the shifting (instruction load) process.

Once the instruction has been loaded, the selected test circuitry is configured to respond. In some cases, however, it is necessary to load data into the selected test circuitry before a meaningful response can be made. Such data are loaded into the component serially in a manner analogous to the process used previously to load the instruction. Note that the movement of test data has no effect on the instruction present in the test circuitry.

After execution of the test instruction, based where necessary on supplied data, the results of the test can be examined by shifting data out of the component to or through the bus master.

Note that in cases where the same test operation is to be repeated but with different data, new test data can be shifted into the component while the test results are shifted out. There is no need for the instruction to be reloaded.

Operation of the test circuitry may proceed by loading and executing several further instructions in a manner similar to that described and would conclude by returning control from the test circuitry to system circuitry. Note that the state of the system logic may be indeterminate, and care is required in returning to system operation.

1.2.2 Use of this standard to test an assembled product

This subclause outlines the use of the boundary-scan circuitry defined by this standard during the process of testing an assembled product such as a printed circuit board.

The test problem for any product constructed from a collection of components can be decomposed into three goals:

a) To confirm that each component performs its required function
b) To confirm that the components are interconnected in the correct manner
c) To confirm that the components in the product interact correctly and that the product performs its intended function

This approach can be applied to a board constructed from integrated circuits, to a system constructed from printed circuit boards, or to a complex integrated circuit constructed from a set of simpler functional modules. To simplify the discussion, this description henceforth will concentrate on the case of an assembled printed circuit board constructed from a collection of digital integrated circuits.

At the board level, the second goal [goal b)] may be achieved using in-circuit test techniques; the first and third goals [goal a) and goal c)] require a functional test. However in-circuit test techniques have significant limitations when viewed against evolving surface-mount interconnection technology, for example, the difficulty of making reliable contact to miniaturized features of the printed circuit board using a bed-of-nails fixture. How, then, might these three test goals be achieved if test access becomes limited to the normal circuit connections, plus a relatively small number of special-purpose test connections?

Considering goal a), it is clear that the vendor of an integrated circuit used in the board-level design will have an established test methodology for that component. The components could be tested on a proprietary ATE system or by using a self-test procedure embedded in the design. Information on the test methodology adopted is typically not available to the component purchaser. Even where self-test modes of operation are known to exist, they may not be documented and therefore are not available to the component user. Alternative sources of test data for the board test engineer may be the component test libraries supplied with in-circuit test systems or the test programs developed by component users for incoming inspection of delivered devices.

Wherever the test data for a component originates, the next step is to use it once the component has been assembled onto the printed circuit board. If access is limited to the normal connections of the assembled circuit, this task may be far from simple. This is particularly true if the surrounding components are complex or if the board designer has

tied some of the components' connections to fixed logic levels or has left component pins unconnected. Even for slow speed tests, it will not normally be possible to test the component in the same way that it was tested in isolation unless an in-circuit test is achievable.

To help ensure that built-in test facilities can be used or that preexisting test patterns can be applied, a framework is needed that can be used to convey test data to or from the boundaries of individual components so that they can be tested as if they were freestanding. This framework will also allow access to and control of built-in test facilities of components. A boundary scan coupled with a test access bus provides such a framework.

The objective of this standard is to define a boundary-scan architecture that can be adopted as a standard feature of integrated circuit designs, thus, allowing the required test framework to be created on assembled printed circuit boards and other products.

1.2.3 What is a boundary scan?

The boundary-scan technique involves the inclusion of a shift-register stage (contained in a boundary-scan register cell) adjacent to each component pin so that signals at component boundaries can be controlled and observed using scan testing principles.

Figure 1-1 illustrates an example implementation for a boundary-scan register cell that could be used for an input or output connection to an integrated circuit. If it is used for an input, data can either be loaded into the scan register from the input pin through the signal-in port or be driven from the register through the Signal-out port of the cell into the core of the component design, depending on the control signals applied to the multiplexers. Similarly, if it is used for an an output, data can either be loaded into the scan register from the core of the component or be driven from the register to an output pin. As will be discussed in detail in Clause 11, the second flip-flop (controlled by input Clock B) is provided to hold the signal value driven out of the cell while new data are shifted into the cell using input Clock A. This flip-flop is not required in all cases but is included in Figure 1-1 to simplify the discussion.

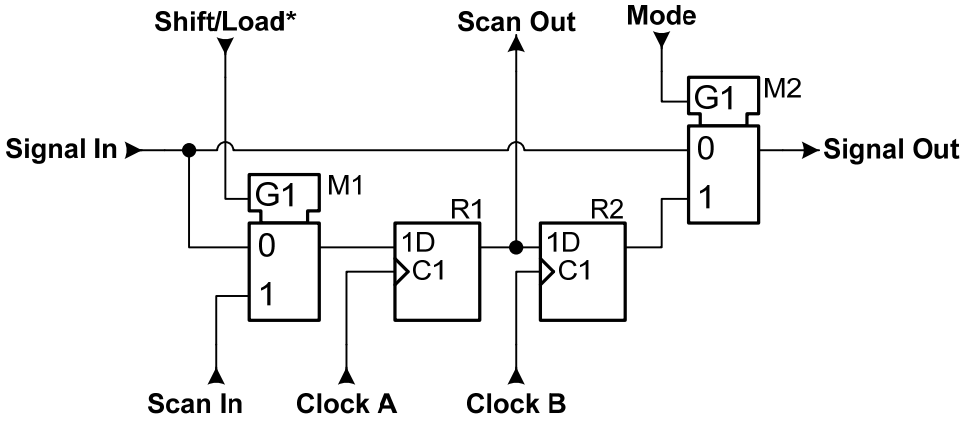


Figure 1-1—Boundary-scan register cell

The boundary-scan register cells for the pins of a component are interconnected to form a shift-register chain around the border of the design, and this path is provided with serial input and output connections and appropriate clock and control signals. Within a product assembled from several integrated circuits, the boundary-scan registers for the individual components could be connected in series to form a single path through the complete design, as illustrated in Figure 1-2. Alternatively, a board design could contain several independent boundary-scan paths.

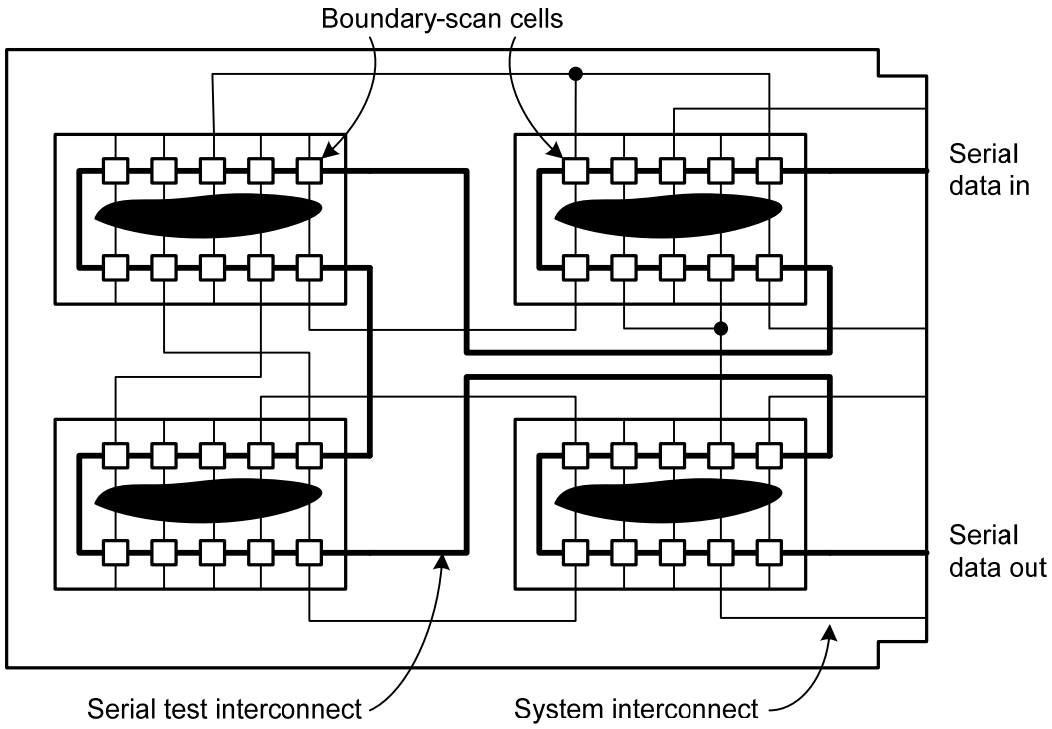


Figure 1-2—Boundary-scannable board design

If all components used to construct a circuit have a boundary-scan register, then the resulting serial path through the complete design can be used in two ways:

a) To allow the interconnections between the various components to be tested, test data can be shifted into all the boundary-scan register cells associated with component output pins and loaded in parallel through the component interconnections into those cells associated with input pins.

b) To allow the components on the board to be tested, the boundary-scan register can be used as a means of isolating system logic from stimuli received from surrounding components while an internal self-test is performed. Alternatively, if the boundary-scan register is suitably designed, it can permit a limited slow-speed static test of the system logic because it allows delivery of test data to the component and examination of the test results.

These tests allow the first two goals discussed earlier to be achieved through the use of the boundary-scan register. In effect, tests applied using the register can detect many of the faults that in-circuit testers currently address, but without the need for extensive bed-of-nails access. The third goal—to test the operation of the complete product functionally—remains and can be achieved either using a functional (through the pins) ATE system or using a system-level self-test, for example.

Note also that by parallel loading the cells at both the inputs and outputs of a component and shifting out the results, the boundary-scan register provides a means of "sampling" the data flowing through a component without interfering with the behavior of the component or the assembled board. This mode of operation is valuable for design debugging and fault diagnosis because it permits examination of connections not normally accessible to the test system.

1.2.4 Use of this standard to achieve other test goals

In addition to its application in testing printed circuit assemblies and other products containing multiple components, the test logic defined by this standard can be used to provide access to a wide range of design-for-test features built

into the components themselves. Such features might include internal scan paths, self-test functions [e.g., memory built-in self test (MBIST) or logic built-in self-test (Logic BIST)], or other support functions.

Design-for-test features such as these can be accessed and controlled using the data path between the serial test data pins of the TAP defined by this standard. Instructions that cause internal reconfiguration of the component's system logic such that the test operation is enabled may be shifted into the component through the TAP.

1.3 Document outline

Circuit designs such as that defined by this standard are more easily understood if their specifications are accompanied by general descriptive material that places the details of the various parts of the design in perspective and provides examples of implementation. Clause 1 therefore contains an overview of the application of this standard to the testing of the digital portions of an electronic product consisting of many integrated circuits.

Subsequent clauses of this document contain the specifications for particular features of this standard. Two classes of material are contained in these clauses.

1.3.1 Specifications

Material titled "Specifications" contain the rules, recommendations, and permissions that define this standard:

a) *Rules* specify the mandatory aspects of this standard. Rules contain the word **shall**.
b) *Recommendations* indicate preferred practice for designs that seek to conform to this standard. Recommendations contain the word **should**.
c) *Permissions* show how optional features may be introduced into a design that seeks to conform to this standard. These features will extend the application of the test circuitry defined by this standard. Permissions contain the word **may**.

1.3.2 Descriptions

CAUTION

The descriptive material contained in this standard is for illustrative purposes only and does not define a preferred implementation. Examples are provided throughout this standard to illustrate possible circuit implementations. Where discrepancies between examples and specifications may occur, the specifications always take precedence. Readers should exercise caution when using these examples in their specific applications. In particular, it is emphasized that the examples are designed to communicate effectively the meaning of this standard. As such, they are logically correct; however, as always, a particular implementation may not operate properly with respect to timing and other parametric characteristics. One example of this concern is that the example TAP Controller implementation depicted in Figure 6-5 reasonably assumes a significantly greater delay in the flip-flop sourcing signal A, for instance, than in the inverter sourcing TCK*. It is possible to design a circuit where this assumption is violated, causing a critical race to occur that would invalidate the behavior of the TAP controller. Therefore, it is highly recommended that implementations be fully verified for compliance under required operating conditions.

Material not contained in "Specifications" is descriptive material that illustrates the need for the features being specified or their application. This material includes schematics that illustrate a possible implementation of the specifications in this standard. The descriptive material also discusses design decisions made during the development of this standard.

1.4 Text conventions

The following conventions are used in this standard:

a) The rules, recommendations, and permissions in "Specifications" are contained in a single alphabetically indexed list. References to each rule, recommendation, or permission are shown in the form:

```
          15.1.1 c) 2)
             |   |  |
Subclause number   |  |
            Index  |
      Option (if any)
```

b) Instruction and state names defined in this standard are shown in *italic* type in the text.

c) Names of states and signals that control the test data registers defined by this standard contain the characters DR, while those that control the instruction register contain the characters IR.

d) Names for signals that are active in their low state have an asterisk as the final character, e.g., TRST*.

e) A positive logic convention is used; i.e., a logic 1 signal is conveyed as the more positive of the two voltages used for logic signals.

1.5 Logic diagram conventions

During the different iterations of this standard, logic diagram figures have been added in various styles with differing logic symbols. Figure 1-3 shows the symbologies currently used for the common combinational logic elements. Symbols for storage elements (generally edge-sensitive flip-flops) are reasonably consistent, although pin names on the element may vary from figure to figure.

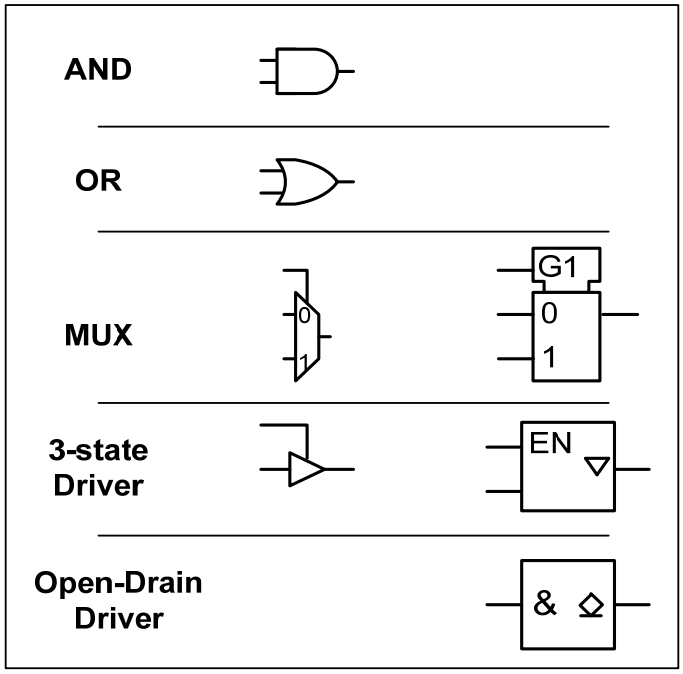


Figure 1-3—Logic symbology used in this standard

2. Normative references

The following referenced documents are indispensable for the application of this standard (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.

EIA/JEP106, JEDEC Publication 106, Standard Manufacturer's Identification Code.[1]

IEEE Std 1076, IEEE Standard VHDL Language Reference Manual.[2, 3]

IEEE Std 1451.0, IEEE Standard for a Smart Transducer Interface for Sensors and Actuators—Common Functions, Communication Protocols, and Transducer Electronic Data Sheet (TEDS) Formats.

The following referenced documents provide extensions to this standard and should be referenced when dealing with analog, capacitively coupled, and differential inputs and outputs.

IEEE Std 1149.4, IEEE Standard for a Mixed-Signal Test Bus.

IEEE Std 1149.6, IEEE Standard for Boundary-Scan Testing of Advanced Digital Networks.

IEEE Std 1149.8.1, IEEE Standard for Boundary-Scan-Based Stimulus of Interconnections to Passive and/or Active Components.

[1] EIA publications are available from Global Engineering Documents (http://global.ihs.com/)

[2] IEEE publications are available from The Institute of Electrical and Electronics Engineers (http://standards.ieee.org/).

[3] The IEEE standards or products referred to in this clause are trademarks of The Institute of Electrical and Electronics Engineers, Inc.

3. Definitions, abbreviations, acronyms, and special terms

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.[4]

3.1 Definitions

active: When associated with a logic level (e.g., active-low), this term identifies the logic level to which a signal shall be set to cause the defined action to occur. When referring to an output driver (e.g., an active drive), this term describes the state in which the driver is capable of determining the voltage of the network to which it is connected.

bidirectional pin: A component pin that can either drive or receive signals from external connections.

blind interrogation: The act of discovering the identity of an unknown component, or discovering the basic construction of a chain of unknown components, by reading out data that are deterministically set by the devices in the chain when reset. For example, devices may have a documented identification (ID) code, which is selected for scan by the default instruction in such devices, so that a user can identify each unknown component in the chain and the user's position within the chain.

capture: Load a value into a data register or the instruction register as a consequence of entry into the *Capture-DR* or *Capture-IR* controller state, respectively.

chip: Typically mounted on a board or some other package with other components. The more general term "component" is normally used in this standard for a compliant object as there may be multiple integrated circuits in a package, even when they behave as a single object. *Syn.:* **integrated circuit (IC)**.

chip-on-board testing: A test of a component after it has been assembled onto a printed circuit board or other substrate. For example, the facilities defined by this standard may be used.

clock: A signal where transitions between the low and high logic levels (or vice versa) are used to indicate when a stored-state device, such as a flip-flop or latch, may perform an operation.

component: An active or passive electronic part. For the sake of this standard, this usually refers to an integrated circuit, although it could include non-integrated-circuit devices mounted on a board. *See also:* **chip**.

deprecate: To discourage the use of configurations or modes of operation that might reduce reliability or create problems in usage. *Syn.:* disapprove, discourage, disparage.

falling edge: A transition from a high to a low logic level. In positive logic, a change from logic 1 to logic 0. Events that are specified to occur on the rising (falling) edge of a signal should be completed within a fixed (frequency-independent) delay specified by the component supplier.

field: In reference to a register or register segment, a logical set of bits within a defined register or segment, which may be addressed or treated as a unit for specific purposes. *Contrast:* **segment**.

hierarchy: Objects may be contained (used) in other objects in a tree-type arrangement with no loops; that is, no circular references. The containing **(**outer, higher**)** object is referred to as the parent, and the contained **(**inner, lower**)** object is referred to as the child, and they are said to be in a parent-to-child relationship. In this standard, only user Packages and Test Data Registers have hierarchy. Packages may "use" (refer to) other packages. Test data registers

[4] The *IEEE Standards Dictionary Online* subscription is available at http://www.ieee.org/portal/innovate/products/standard/standards_dictionary.html.

may be composed of segments and fields, where segments may be composed of other segments and fields**.** Fields may not contain other objects. *See also:* **instance and instantiation.**

high: The higher of the two voltages used to convey a single bit of information. For positive logic, a logic 1.

inactive: When referring to an output driver (e.g., an inactive drive), this term describes the state in which the driver is not capable of determining the voltage of the network to which it is connected.

input pin: A component pin that receives signals from an external connection.

instance and instantiation: These terms refer to the hierarchical inclusion of an object within another object. For the normative purpose of this standard, these terms are used in the documentation of Test Data Registers, which may be composed of segments and fields, and of segments that may be composed of other segments and fields. A single field or segment definition is an object that may be used multiple times, and each use is termed an *instance* of that object. An *instantiation* is the statement that creates and names an instance of a child object within the definition of the parent object. (Similar terminology is used when discussing "instances" or "instantiations" of components on a board.) *See also:* **hierarchy**.

instruction: A binary data word shifted serially into the test logic in order to define its subsequent operation.

integrated circuit (IC): A collection of transistors, resistors, and capacitors and their interconnections constructed to perform specific functions on a single thin slice of a semiconductor crystal.

least significant bit (LSB): The digit in a binary number representing the lowest numerical value. For shift-registers, the bit located nearest to the serial output, or the first bit to be shifted out. The least significant bit of a binary word or shift-register is numbered 0.

level-sensitive scan design (LSSD): A variant of the scan design technique that results in race-free, testable digital electronic circuits.

low: The lower of the two voltages used to convey a single bit of information. For positive logic, a logic 0.

most significant bit (MSB): The digit in a binary number representing the greatest numerical value. For shift-registers, the bit farthest from the serial output, or the last bit to be shifted out. Logic values expressed in binary form are shown with the most significant bit on the left.

nonclock: A signal where the transitions between the low and high logic levels do not themselves cause operation of stored-state devices. The logic level is important only at the time of a transition on a clock signal.

no-connect: A signal of the component, which is brought to an input–output pad of the die but not connected to a package pin due, for instance, to a constrained pin-count for the package.

output pin: A component pin that drives signals onto external connections.

pin: The point at which connection is made between the integrated circuit and the substrate on which it and other components are mounted (e.g., the printed circuit board). For packaged components, this would be the package pin; for components mounted directly on the substrate, this would be the bonding pad. The actual form of connection (bonding wire, landing pad, solder ball, or metal pin inserted into a via) is not material to the definition.

prime source: In the event that several vendors offer pin-for-pin compatible components, the prime source is the vendor that introduced the component type. *Contrast:* **second source**.

private: An optional feature intended solely for use by the component manufacturer. In the context of this standard, an instruction can be defined as private in the Boundary-Scan Description Language (BSDL). *Contrast:* **public**.

power-up reset: A reset of test or system logic that occurs when the power supply goes from OFF to ON. This can be achieved by design of latches that causes them to always power up in a specific state, or by an ON-component or OFF-component circuit that generates an asynchronous reset signal that stays active for some nontrivial time after the power supply has reached its operating voltage. *Syn.:* power-on reset (POR).

public: A design-specific feature, documented in the component data sheet or in the Boundary-Scan Description Language (BSDL), that may be used by purchasers of the component. In the context of this standard, any instruction not defined as private is public. *Contrast:* **private**.

register: Most often a Test Data Register, which is accessed by one or more specific instructions. (See Clause 9.)

reset: The establishment of an initial logic condition that can be either logic 0 or logic 1, as determined by the context.

rising edge: A transition from a low to a high logic level. In positive logic, a change from logic 0 to logic 1. Events that are specified to occur on the rising (falling) edge of a signal should be completed within a fixed (frequency-independent) delay, specified by the component supplier.

sample: To capture the value of a signal at a specific moment in time, such as defined by a clock edge.

scan design: A design technique that introduces shift-register paths into digital electronic circuitry, providing controllability and observability in deeply embedded regions of circuity and thereby improving testability.

scan path: The shift-register path through a circuit designed using the scan design technique.

second source: In the event that several vendors offer pin-for-pin compatible components, second-source suppliers are vendors of the component other than the prime source.

segment: A set of contiguous (in terms of the scan chain) register bits obeying the rules for Test Data Registers and from which a Test Data Register may be assembled. *Contrast:* **field**. (See Clause 9.)

selected test data register: A test data register is selected when it is required to operate by an instruction supplied to the test logic.

signature: A result of a technique for compressing a sequence of logic values output from a circuit under test into a small number of bits of data (the signature) that, when compared with stored known-good data, will indicate the presence or absence of faults in the circuit.

stand-alone testing: A test of a component performed before it is assembled onto a board or other substrate, for example, using automatic test equipment (ATE).

stuck-at fault: A failure in a logic circuit that causes a signal connection to be fixed at 0 or 1 regardless of the operation of the circuitry that drives it.

system: Pertaining to the nontest function of the circuit. *Contrast:* **test logic**.

system logic: Any item of logic that is dedicated to realizing the nontest function of the component or is at the time of interest configured to achieve some aspect of the nontest function.

system pin: A component pin that feeds, or is fed from, the on-chip system logic and carries a digital signal. This term does not include analog signals, voltage references, or power sources.

test logic: Any item of logic that is a dedicated part of the test logic architecture or is at the time of interest configured as part of the test logic architecture.

test mode: The state of a component in which the component's test logic interferes with the flow of signals to and from the system logic. In addition, the system logic may be controlled as needed to prevent an undesired response to system inputs, excessive heating, and so on.

three-state pin: A component output pin where the drive may be either active or inactive (for example, at high impedance).

update: Transfer a logic value from the shift-register stage of a data register cell or an instruction register cell into the latched parallel output stage of the cell as a consequence of the falling edge of the test clock input in the *Update-DR* or *Update-IR* controller state, respectively.

3.2 Abbreviations and acronyms

ASIC	application-specific integrated circuit
ATE	automatic test equipment
ATPG	automatic test pattern generation
BIST	built-in self-test
BNF	Backus-Naur form
BSDL	Boundary-Scan Description Language (see Annex B) or a file containing BSDL statements
CMOS	complementary metal-oxide semiconductor
DC	direct current
ECID	electronic chip identification
ECL	emitter coupled logic
IC	integrated circuit
I/O	input or output
IP	intellectual property, commonly used to refer to a reusable design element
IR	infrared
LSB	least significant bit
LSSD	level-sensitive scan design
LVDS	low-voltage differential signaling
MEMBIST	memory built-in self-test
MSB	most significant bit
PDL	Procedural Description Language (see Annex C) or a file containing PDL statements
PGA	pin-grid array
PLL	phase-locked loop
POR	power-on reset
PRBS	pseudo-random binary sequence
RAM	random access memory
SOC	system-on-a-chip
TAP	test access port (see Clause 4)
TCK	test clock input (see 4.2)
Tcl	Tool Command Language (see Annex C) or a file containing Tcl statements
TDI	test data input (see 4.4)
TDO	test data outputs (see 4.5)
TDR	test data register (see Clause 9)
TEDS	tranducer electronic data sheet
TMP	test mode persistence (see 6.2)
TMS	test mode select (see 4.3)
TRST*	test reset (see 4.6)
TTL	transistor–transistor logic
UUT	unit under test
VHDL	VHSIC Description Language

VHSIC	very high speed integrated circuit
WBR	wrapper boundary register
WBY	wrapper bypass
WDR	wrapper data register
WIR	wrapper instruction register
WSC	wrapper serial control
WSI	wrapper scan in
WSO	wrapper scan out
WSP	wrapper serial port

3.3 Special terms

design specific: For the purpose of this standard, a test-logic element, permitted by and conforming to the rules of this standard, added to a specific component implementation in addition to the elements required or defined by this standard.

redundant: When applied to an element of the test logic, and in particular to boundary-scan register cells, implies that the element could be omitted from the component without jeopardizing compliance with this standard.

4. Test access port (TAP)

The TAP is a general-purpose port (i.e., a group of component inputs and outputs) that can provide access to many test support functions built into a component, including the test logic defined by this standard. It is composed as a minimum of the three input connections and one output connection required by the test logic defined by this standard. An optional fourth input connection provides for asynchronous initialization of the test logic defined by this standard.

4.1 Connections that form the TAP

4.1.1 Specifications

Rules

a) The TAP shall include the following connections (defined in 4.2 through 4.5): TCK, TMS, TDI, and TDO.
b) Where the TAP controller state would be indeterminate after power-up without external control, a TRST* input shall be provided as defined in 4.6 (see also 6.1.3).

NOTE—This requires the TRST* connection unless either the logic is capable of establishing the *Test-Logic-Reset* state at power up or an on-chip POR circuit is provided. See 6.1.3 for use of an on-chip POR circuit.[5]

c) All TAP inputs and outputs shall be dedicated connections to the component (i.e., the pins used shall not be used for any other purpose) as long as the component is in a state compliant with this standard (defined in 4.8).

4.1.2 Description

Dedicated TAP connections are required to allow access to the full range of mandatory features of this standard.

4.2 Test clock input (TCK)

The test clock input (TCK) provides the clock for the test logic defined by this standard.

4.2.1 Specifications

Rules

a) Stored-state devices contained in the test logic shall retain their state indefinitely when the signal applied to TCK is stopped at 0.

Recommendations

b) Since TCK inputs for many components may be controlled from a single driver, the load presented by TCK should be as small as possible.

[5] Notes in text, tables, and figures of a standard are given for information only and do not contain requirements needed to implement this standard.

Permissions

c) Stored-state devices contained in the test logic may retain their state indefinitely when the signal applied to TCK is stopped at 1.

4.2.2 Description

The dedicated TCK input is included so that the serial test data path between components can be used independently of component-specific system clocks, which may vary significantly in frequency from one component to the next. It also permits shifting of test data concurrently with normal system operation of the component. The latter facility is required to support the use of the TAP and test data registers in a design for on-line system monitoring. The provision of an independent clock helps ensure that test data can be moved to or from a component without changing the state of the on-chip system logic. The independent clock is also essential if boundary-scan registers are to be usable for board interconnect testing in all circumstances—including cases where system clock signals are derived in one component for use in others.

While TCK will in many cases be driven by a free-running clock with a nominal 50% duty cycle, there may be situations where the clock needs to stop for a period. One example is when an ATE needs to fetch test data from backup memory (e.g., disc) since some test systems are unable to keep the clock running during such an operation. This standard requires that TCK can be stopped at 0 indefinitely without causing any change to the state of the test logic. While the TCK signal is stopped at 0, stored-state devices are required to retain their state so that the test logic may continue its operation when clock operation restarts. Optionally, a component also may allow TCK to be stopped at 1 for an indefinite period.

Many parts of the test logic perform operations in response to the rising or falling edge of TCK, indicated by the use of the phrase “on the rising (falling) edge of TCK.” These operations have to be completed within a fixed delay after the occurrence of the relevant change at TCK, and this delay has to be specified by the component supplier. Therefore, the phrase “on the rising (falling) edge of TCK” should be interpreted as “within a specified delay after the rising (falling) edge of TCK.”

NOTE—In many applications, the TCK signal applied to components that conform to this standard will have a duty cycle close to 50% (i.e., the periods that the clock spends at 0 and 1 will be equal) at a given frequency. It is expected that all propagation delays will be such that correct operation is achieved under these circumstances (50% duty cycle at a given TCK frequency), particularly when data are being transferred between TDO of one component and TDI of another.

4.3 Test mode select (TMS) input

The value of the signal present at TMS at the time of a rising edge at TCK determines the next state of the TAP controller, the circuit that controls test operations.

4.3.1 Specifications

Rules

a) The signal presented at TMS shall be sampled by the test logic on the rising edge of TCK.
b) The design of the circuitry fed from TMS shall be such that an undriven input produces a logical response identical to the application of a logic 1.

Recommendations

c) Since the TMS inputs for many components may be controlled from a single driver, the load presented by TMS should be as small as possible.

4.3.2 Description

Rule b) of 4.3.1 is included so that the TAP controller moves as quickly as possible into the *Test-Logic-Reset* controller state when the TMS pin is not driven by an external source. For example, an off-board test controller might not have been connected or there might be an open-circuit on a newly assembled printed circuit board. This helps in ensuring that normal operation of the complete design can continue without interference from the test logic (see 6.1.3). For TTL and CMOS technology designs, the rule may be met by including a pull-up resistor in the component's TMS input circuitry.

Signal values presented at TMS are captured by the test logic on the rising edge of TCK. It is expected that the bus master (ATE, bus controller, etc.) will change the signal driven to the TMS inputs of connected components on the falling edge of TCK. The waveforms shown elsewhere in this standard reflect this expectation.

4.4 Test data input (TDI)

Serial test instructions and data are received by the test logic at TDI.

4.4.1 Specifications

Rules

a) The signal presented at TDI shall be sampled into the test logic on the rising edge of TCK.
b) The design of the circuitry fed from TDI shall be such that an undriven input produces a logical response identical to the application of a logic 1.
c) When data are being shifted from TDI toward TDO, test data received at TDI shall appear without inversion at TDO after a number of rising and falling edges of TCK determined by the length of the instruction or test data register selected.

4.4.2 Description

The data pins (TDI and TDO) provide for serial movement of test data through the circuit. The requirement for data to be propagated from TDI to TDO without inversion is included to simplify the operation of components compatible with this standard linked on a printed circuit board.

Values presented at TDI are clocked into the selected register (instruction or test data) on a rising edge of TCK. It is expected that the bus master (ATE, bus controller, etc.) will change the signal driven to the TDI input of the first component on a serial board-level path on the falling edge of TCK. The waveforms shown elsewhere in this standard reflect this expectation.

Rule b) of 4.4.1 is included so that open-circuit faults in the board-level serial test data path cause a defined logic value to be shifted into the test logic. Note that when this constant value is shifted into the instruction register, the bypass register will be selected (as will be discussed further in 8.4). For TTL and CMOS technology designs, this rule may be met by inclusion of a pull-up resistor in the component's TDI input circuitry.

4.5 Test data output (TDO)

TDO is the serial output for test instructions and data from the test logic defined in this standard.

4.5.1 Specifications

Rules

a) Changes in the state of the signal driven through TDO shall occur only on the falling edge of either TCK or the optional TRST*.

b) The TDO driver shall be set to its inactive drive state except when the shifting of data is in progress (see 6.1.2).

NOTE—Valid states for this signal are high (1), low (0), and undriven (Z).

4.5.2 Description

To help ensure a race-free operation, changes on TAP inputs (TMS and TDI) are clocked into the test logic defined by this standard on the rising edge of TCK while changes at the TAP output (TDO) occur on the falling edge of TCK. Similarly, for test logic able to drive or receive signals from system pins (e.g., the boundary-scan register), signals driven out of the component from the test logic change state on the falling edge of TCK, while those entering the test logic are clocked in on the rising edge (as will be discussed in 9.3.2).

The contents of the selected register (instruction or data) are shifted out of TDO on the falling edge of TCK. In the illustrations given in this document, edge-operated circuit designs are generally used. For an edge-operated implementation, note that the TDO output changes shall be delayed until the falling edge of TCK, which can be achieved by including a flip-flop clocked by the falling edge of TCK in the TDO output buffer. Where the registers are constructed from master and slave latches controlled by non-overlapping clocks, the retiming required by Rule a) of 4.5.1 is an inherent feature of the design.

The ability of TDO to switch between active and inactive drive is required to allow parallel, rather than serial, connection of board-level test data paths in cases where this is required. In TTL or CMOS technologies, for example, this requirement may be met through use of a three-state output buffer.

4.6 Test reset input (TRST*)

The optional TRST* input provides for asynchronous initialization, principally at power-up, of the TAP controller (see 6.1.3), the test mode persistence (TMP) controller (see 6.2.3), and possibly other test logic. It has a broader effect than simply moving the TAP controller state machine to the *Test-Logic-Reset* state. TRST* is required when the test logic does not power-up in a known and controlled state, either because of the nature of the underlying technology or because there is no on-chip power-up reset generator.

4.6.1 Specifications

Rules

a) If TRST* is included in the TAP, the TAP controller and TMP controller shall be asynchronously reset to the *Test-Logic-Reset* and *Persistence-Off* controller states, respectively, when a logic 0 is applied to TRST* (see 6.1.3).

NOTE 1—As a result of this event, all other test logic in the component is asynchronously reset to the state required in the *Test-Logic-Reset* controller state.

b) While compliance is enabled and TRST* is included in the TAP, then the design of the circuitry fed from that input shall be such that an undriven TRST* input produces a logical response identical to the application of a logic 1.

NOTE 2—The logical response can only be detected and only needs to be present while the TAP is in IEEE 1149.1-compliant operation. See 4.8 for details.

c) TRST* shall not be used to initialize any system logic within the component.
d) The TRST* input shall be included if on-chip circuitry does not force a reset of the test logic upon power-up.

NOTE 3—See 6.1.3 for further information.

Recommendations

e) To help ensure deterministic operation of the test logic, TMS should be held at 1 while the signal applied at TRST* changes from 0 to 1.

4.6.2 Description

Initialization of the TAP controller in turn causes asynchronous initialization of other test logic included in the design, as discussed in the subsequent clauses of this standard.

Rule b) of 4.6.1 is included to help ensure that, in the case of an unterminated TRST* input, the test logic operation can proceed under control of signals applied at the TMS and TCK inputs. For TTL or CMOS compatible designs, this rule may be met by inclusion of a pull-up resistor in the TRST* input circuitry of the component.

Rule c) of 4.6.1 allows the test logic to be reset independently of the on-chip system logic, and to be independently disabled using methods as shown in Figure 4-5.

Recommendation e) of 4.6.1 is included to allow the test logic to respond predictably when the signal applied to TRST* changes from 0 to 1. If rising edges occur simultaneously at TRST* and TCK when a logic 0 is applied to TMS, a race could occur, and the TAP controller could either remain in the *Test-Logic-Reset* controller state or enter the *Run-Test/Idle* controller state.

4.7 Interconnection of components compatible with this standard

4.7.1 Specifications

Permissions

a) The TAP input and output connections may be interconnected at the board level in a manner appropriate to the assembled product.

4.7.2 Description

Figure 4-1 through Figure 4-4 illustrate four alternative board-level interconnections of components conforming to this standard.

In each example, the test bus may be controlled either by an ATE system or by a component that provides an interface to a test bus at the next level of product assembly (for example, at the board/backplane interface). In this standard, the device that controls the board-level test bus is referred to as the bus master.

Note that the minimum configuration without TRST* (shown in Figure 4-1) contains:

— Two broadcast signals (TMS and TCK) driven by the testability bus master to all slaves in parallel
— A serial path formed by a daisy-chain connection of the serial test data pins (TDI and TDO)

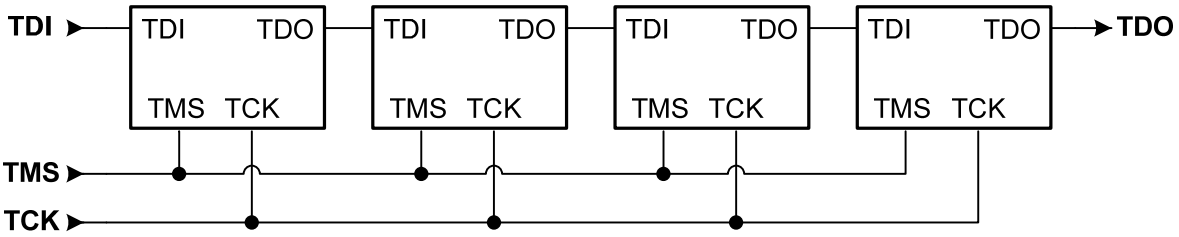


Figure 4-1—Serial connection using one TMS signal

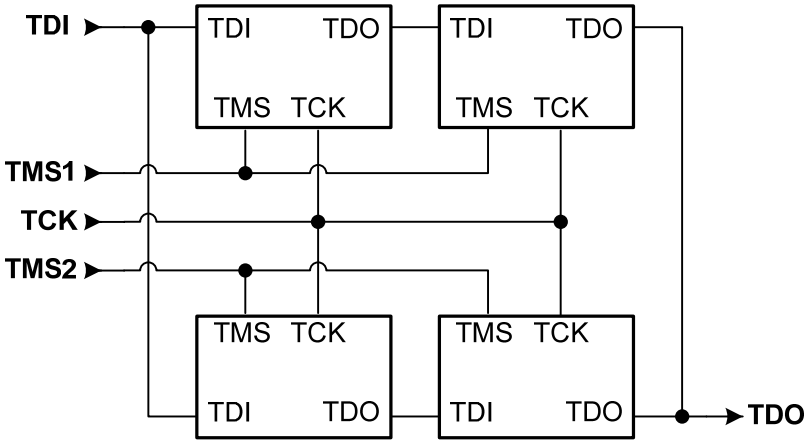


Figure 4-2—Serial/parallel connection using two TMS signals

The hybrid serial/parallel connection shown in Figure 4-2 uses a pair of coordinated TMS signals (TMS1 and TMS2) to scan data through only one serial path at a given time. This configuration makes use of the three-state feature of the TDO output pin where only the components that are scanning data have TDO in the active drive state.

Figure 4-3 shows the four components connected to give four separate serial paths through the complete board design. These paths have separate TDI and TDO signals, but they can be controlled from common TCK and TMS signals.

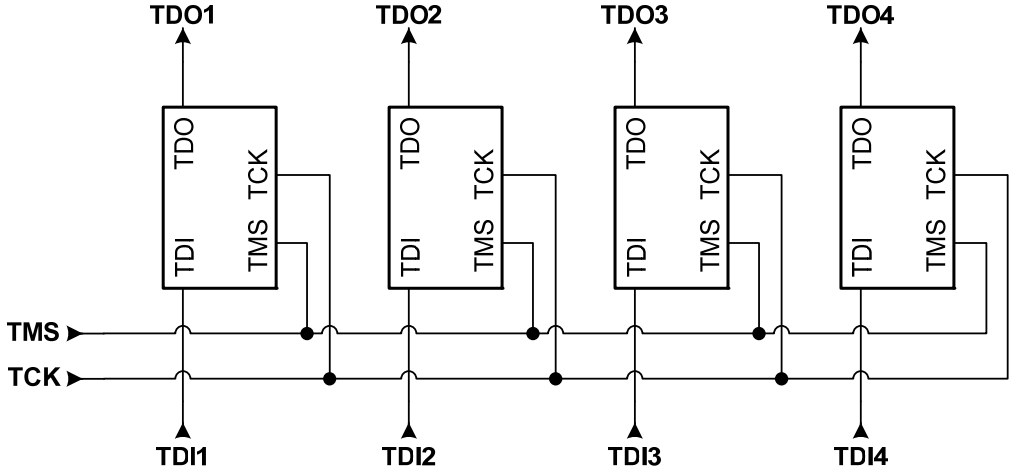


Figure 4-3—Multiple independent paths with common TMS and TCK signals

When choosing a configuration for the board-level interconnection of components conforming to this standard, it is necessary to consider the capability of test equipment and test pattern generators. It is fully expected that any test equipment and/or test pattern generators that intend to support a test methodology based on the boundary-scan architecture defined by this standard would be able to test the board-level configuration of Figure 4-1 since the degenerate form of this configuration is a single conformant component. On the other hand, some test equipment and/or test pattern generators may not be able to test the board-level configurations of Figure 4-2 and Figure 4-3.

Figure 4-4 illustrates the board-level interconnections when the optional TRST* pin is present. TRST* is further connected to one of the three alternatives as shown in Figure 4-5.

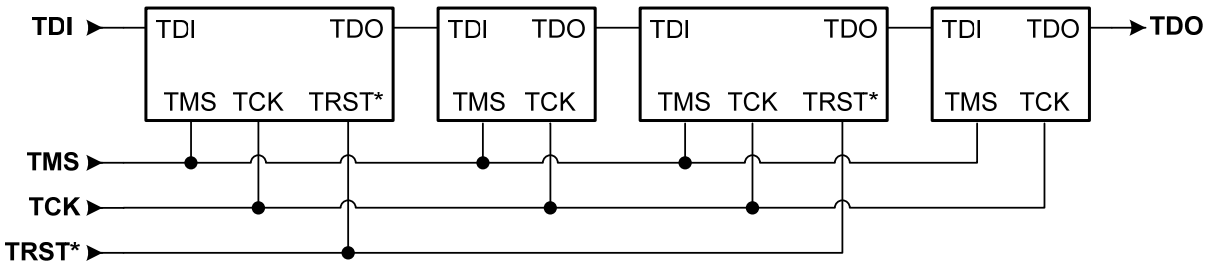


Figure 4-4—Serial connection using one TMS and TRST*

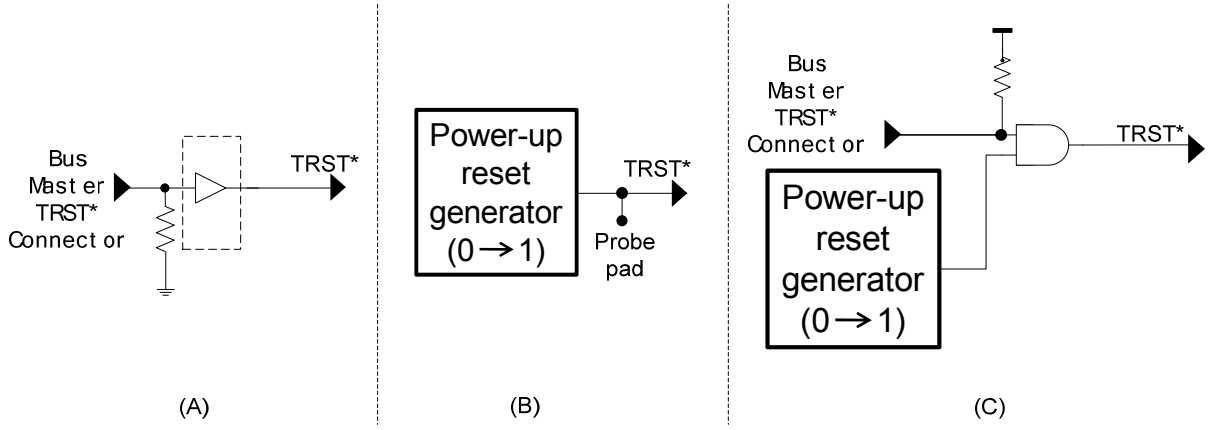


Figure 4-5—Three examples supporting board-level TRST*

Figure 4-5 illustrates three examples of properly connecting TRST* at the board level. Figure 4-5(A) shows a minimal implementation of connecting the TRST* through a resistor to ground. Illustrated inside the dashed box is an optional buffer, which may be used to reliably drive TRST* to logic 0, and to eliminate any current path through the pull-up in the IC receiver and the pull-down resister on the board. Figure 4-5(B) does not support an external bus master connection on TRST*, but it should provide an access point on the board (shown by the stub on the TRST* net) for the ATE. A contact pad is shown. Figure 4-5(C) shows a combination of test bus master and on-board power-up-reset generator. Figure 4-5(B) and Figure 4-5(C) illustrate methods to support TRST* in low-power environments. The "power-up reset generator" in Figure 4-5 is any circuit without a TAP and that can hold its output low until power is stable and then drive its output to a logic high.

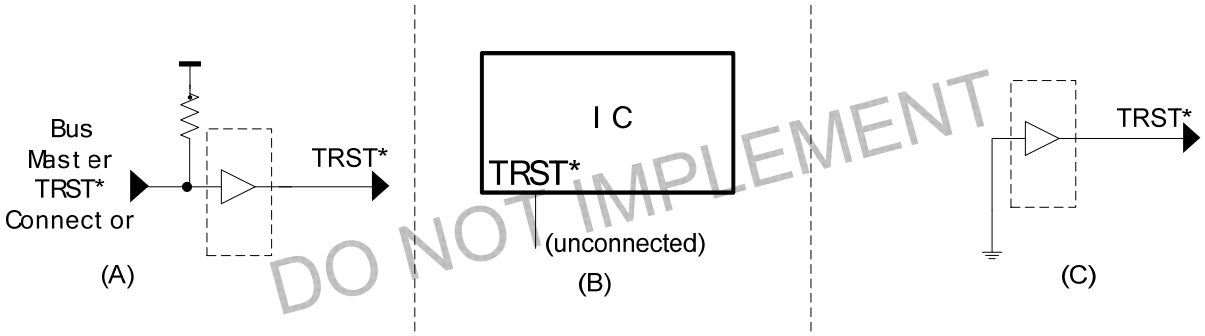


Figure 4-6—Examples of incorrect board-level TRST* connections

Figure 4-6 illustrates three methods, which incorrectly connect TRST* at the board level. Do not implement a TRST* tied to a pull-up or a pull-up on the input of a buffer driving TRST*as shown in Figure 4-6(A). Do not leave TRST* unconnected, as in Figure 4-6(B). Do not tie TRST* to ground or ground the input of a buffer driving TRST*, as in Figure 4-6(C). In Figure 4-6(A) and Figure 4-6(B), there is no guarantee that a logic 0 will be applied to TRST*, so there is a possibility of the test logic interfering with normal system operation. In Figure 4-6(C), the test logic specified by this standard is continuously reset and disabled.

4.8 Subordination of this standard within a higher level test strategy

While the test logic specified by this standard has been designed to be extensible to meet the particular needs of individual designers or companies (for example, by the flexibility of the instruction register), occasions may arise when it will be desirable to terminate compliance with this standard by a component temporarily and enable complementary test functionality. One example would be a component using LSSD for use during "stand-alone" component testing, which cannot be simultaneously operated with the test functionality defined by this standard (which is required to support testing of boards onto which the components implementing the two testing techniques will be assembled).

This subclause defines how compliance with this standard may be "switched on" or "switched off." The rules require the change of test functionality to be under the control of signals applied at one or more component pins. Compliance has to be effected by a single logic pattern applied at these pins, not by a sequence of such patterns.

4.8.1 Specifications

Rules

a) If a component is to be designed having both:
   1) Test functionality compliant with this standard and
   2) Other test functionality that is not to be controlled via the test circuitry and the means of control defined in this standard,

   then compliance with this standard shall be enabled/disabled by one or more steady-state logic patterns (called "compliance-enable patterns") applied at a fixed set of component inputs (called "compliance-enable inputs.")

   NOTE 1—The steady-state combinational logic pattern may be chosen from a set of such "compliance-enable" patterns, all of which have an equivalent effect [see permission h) of 4.8.1].

b) Any one of the compliance-enable patterns, when applied to the compliance-enable inputs without regard to preceding patterns on these inputs, shall cause the component to be fully compliant with this standard.

c) Once compliance with this standard is established by the application of a compliance-enable pattern at the compliance-enable inputs, compliance to this standard shall be maintained continuously until the logic pattern applied at the compliance-enable inputs ceases to be a compliance-enable pattern.

NOTE 2—This rule implies that transition between compliance-enable patterns must produce no untoward effects on compliance. Limiting the number of compliance-enable patterns is one way to prevent problems from arising.

NOTE 3—The rules in other subclauses of this standard apply only when compliance is enabled. Therefore, where compliance-enable inputs are provided, each rule should be considered to be prefaced by "When compliance to this standard is enabled." For example, rule c) of 4.1.1 should be read as stipulating that the TAP pins are dedicated connections and may not be used for any other purpose *while compliance to this standard is enabled*. When compliance is disabled, the TAP connections may be reused—for example, to provide controls for an alternative test mode of component operation.

NOTE 4—The event of enabling compliance with this standard by changing the logic pattern applied at the compliance-enable inputs of a component need not have an effect on the component equivalent to that of power-up of the component.

d) Compliance-enable inputs shall be dedicated inputs to the component and shall not be used for any other purpose.

Recommendations

e) The number of compliance-enable inputs provided on a component should be minimized.

Permissions

f) A component may have zero, one, or more compliance-enable inputs.

g) If a component with compliance-enable input(s) has a TRST* line included in its TAP implementation, the design of the component may require that the TRST* input be driven low at the time of application of a compliance-enable pattern in order to achieve reset of the relevant test logic concurrent with the operation of that test logic.

h) A component may have several compliance-enable patterns, all of which have an equivalent effect.

i) The value applied at the compliance-enable inputs may be changed from one compliance-enable pattern to another while the test logic is active.

4.8.2 Description

If compliance-enable inputs are provided, there shall exist at least one logic pattern that, when applied at the compliance-enable inputs, will result in the component becoming fully compliant with this standard. This pattern must be applied continuously while the test logic is active. Where multiple patterns are provided that will each result in the component becoming fully compliant with this standard, it is permitted to change patterns while the test logic is active as long as such pattern changes cannot cause momentary loss of compliance. Loss of compliance in the middle of a test could make testing impossible.

5. Test logic architecture

This clause defines the top-level design of the test logic accessed through the TAP. Detailed design requirements for the various blocks contained within the test logic design are contained in the subsequent clauses of this standard.

5.1 Test logic design

5.1.1 Specifications

Rules

a) The following elements shall be contained in the test logic architecture:
   1) A TAP controller (see 6.1)
   2) An instruction register (see Clause 7)
   3) A group of test data registers (see Clause 9)

b) The instruction and test data registers shall be separate shift-register based paths that are connected in parallel and have a common serial data input and a common serial data output connected to the TAP TDI and TDO signals, respectively.

c) The selection between the alternative instruction and test data register paths between TDI and TDO shall be made under the control of the TAP controller, as defined in 6.1.2.

Permissions

d) The test logic architecture may also include a test mode persistence controller element as defined in 6.2.

5.1.2 Description

A conceptual view of the top-level design of the test logic architecture defined by this standard is shown in Figure 5-1. This figure, and the others included in the descriptive material contained in this standard, are examples intended only to illustrate a possible embodiment of this standard. *These figures do not indicate a preferred implementation.*

Key features of the design are:

— The TAP controller receives TCK and interprets the signals on TMS. The TAP controller generates clock or control signals or both as required for the instruction and test data registers and for other parts of the architecture. The specification for the TAP controller is contained in 6.1.

— The TAP controller controls the operation (reset, shift, capture, update) while instruction decoding provides the context of the operation (i.e., to which register and associated test logic the action applies).

— The instruction register allows the instruction to be shifted into the design. The instruction is used to select the test to be performed or the test data register to be accessed or both. The specification for the instruction register is contained in Clause 7.

— The instruction register is never undefined and always selects a single test data register to connect between TDI and TDO.

— The group of test data registers. The group of test data registers shall include a bypass and a boundary-scan register. It also may include any of the optional standard test data registers: the device identification, initialization data, initialization status, TMP control, and reset selection registers; and further optional design specific test data registers. A circuit controlled by instruction decoding selects the test data register to drive the output TDO. Further information on the structure of the group of test data registers is contained in Clause 9.

— An optional test mode persistence controller receives TCK and some decodes of the instruction register, and modifies some control signals generated by the TAP controller. It is intended to allow the component to be held in a test mode, and its specification is contained in 6.2.

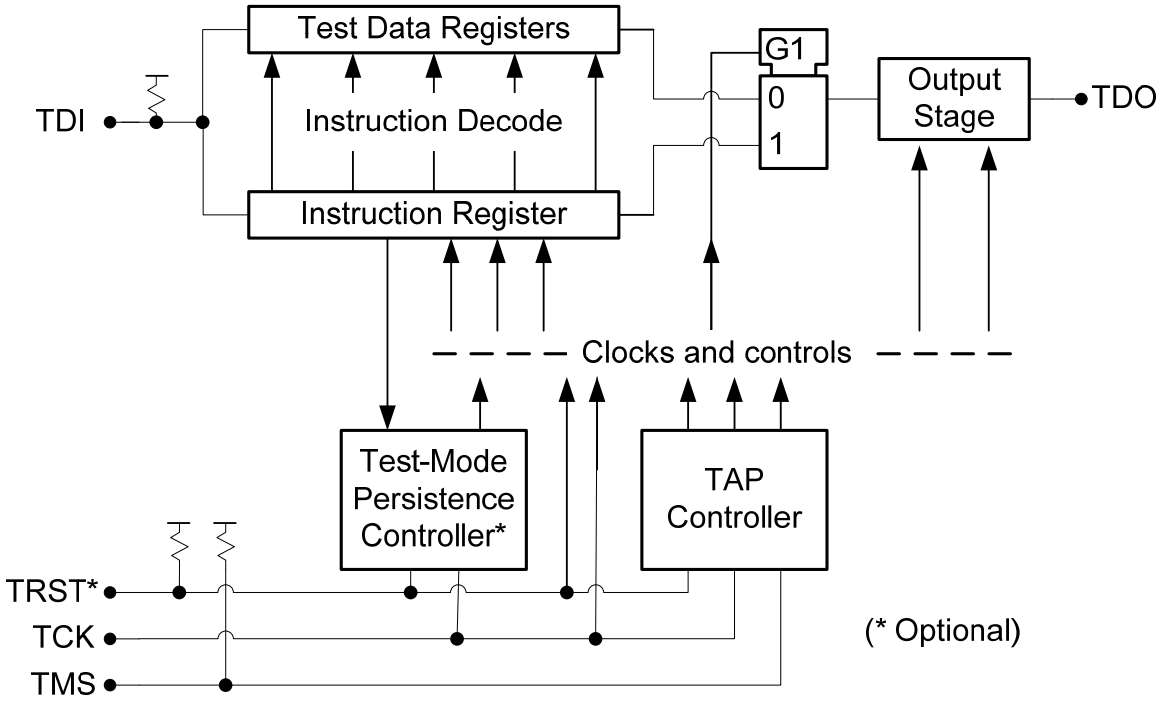


Figure 5-1—Conceptual schematic of the on-chip test logic

Note that, depending on the style of implementation of the test logic defined by this standard, circuitry may be required, in the output stage shown in Figure 5-1, to retime the signal passing through it to occur on the falling edge of TCK.

5.2 Test logic realization

5.2.1 Specifications

Rules

a) The TAP controller, the optional TMP controller, the instruction register, and the associated circuitry necessary for control of the instruction and test data registers shall be dedicated test logic (i.e., these test logic blocks shall not perform any system function).

b) If test access is required to a test data register without causing any interference to the operation of the on-chip system logic, then the circuitry used to construct that test data register shall also be dedicated test logic.

5.2.2 Description

While the example implementations contained in this standard show the various test data registers to be separate physical entities, circuitry may be shared between the test data registers provided that the rules contained in this standard are met. For example, this would allow the device identification register and the boundary-scan register to share shift-register stages; in which case, the requirements of this standard would be met by operating the common circuitry in two different modes—the device identification register mode and the boundary-scan register mode.

6. Test logic controllers

The test logic controllers are finite state machines that control the sequence of operations of the circuitry defined by this standard.

The mandatory TAP controller is a synchronous finite state machine that responds to changes at the TMS and TCK signals of the TAP, controls the behavior of the test logic, and maintains synchronization across all components on a test scan chain to permit shifting, capturing, and updating of data. It is described in 6.1.

The optional TMP controller is a synchronous finite state machine that responds to specific instructions and can force the component to remain in its test mode regardless of the currently active instruction. It is described in 6.2.

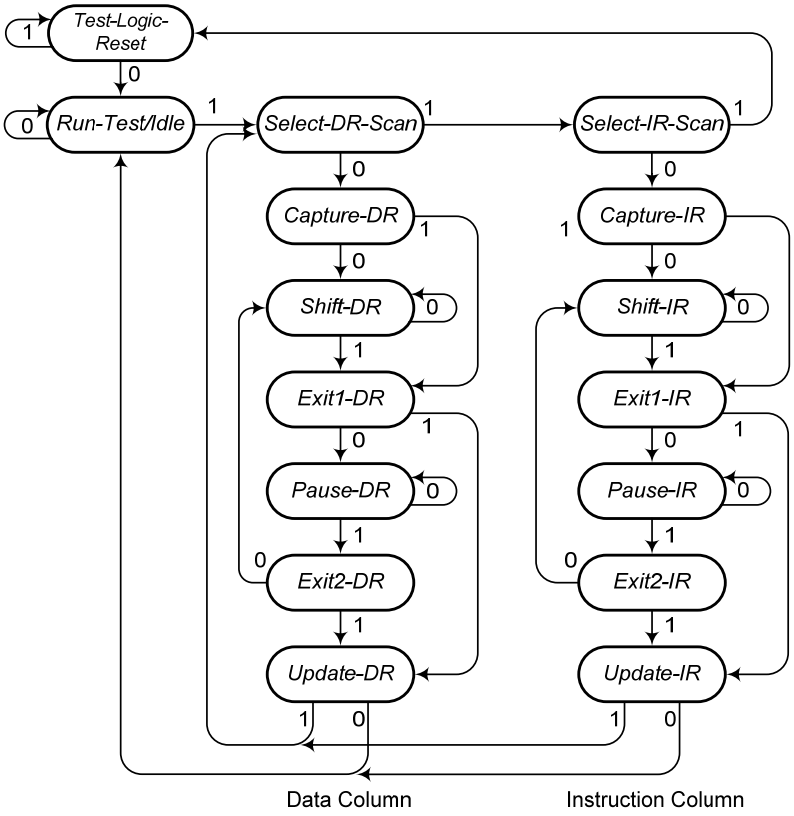
6.1 TAP controller

6.1.1 TAP controller state diagram

6.1.1.1 Specifications

Rules

a) The state diagram for the TAP controller shall be as shown in Figure 6-1.



NOTE—The value shown adjacent to each state transition in this figure represents the signal present at TMS at the time of a rising edge at TCK.

Figure 6-1—TAP controller state diagram

b) All state transitions of the TAP controller shall occur based on the value of TMS at the time of a rising edge of TCK.

c) Actions of the test logic (instruction register, test data registers, etc.) shall occur on either the rising or the falling edge of TCK in each controller state as shown in Figure 6-2.

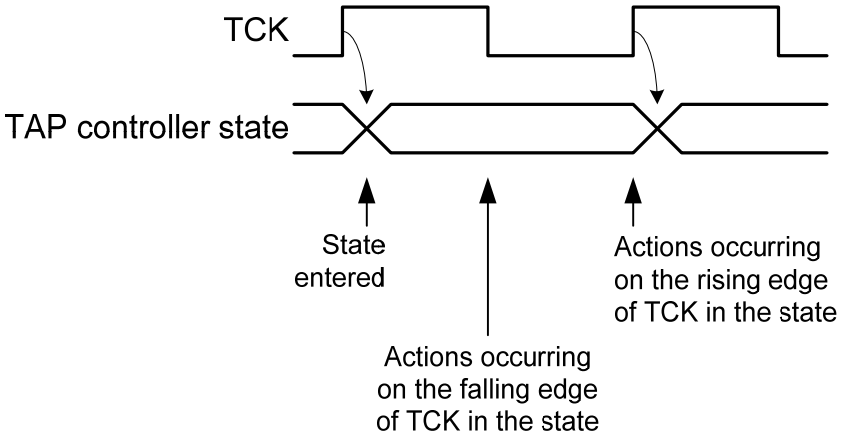


Figure 6-2—Timing of actions in a controller state

6.1.1.2 Description

The behavior of the TAP controller and other test logic in each of the controller states is briefly described as follows. Rules governing the behavior of the test logic defined by this standard in each controller state are contained in later clauses of this standard.

Test-Logic-Reset

If the optional TMP controller is either not provided or in the *Persistence-Off* state, the test logic is placed in a known state so that either normal operation of the on-chip system logic (i.e., in response to stimuli received through the system pins only) can continue unhindered. This is primarily achieved by initializing the instruction register to contain the *IDCODE* instruction or, if the optional device identification register is not provided, the *BYPASS* instruction (see 7.2). If the TMP controller is provided and is in the *Persistence-On* state, the component may be held in test mode in anticipation of further tests.

This state has no direct effect on system logic. During a power-up sequence, where the TAP remains in this state, or no instructions defined as interfering with flow of signals between the system logic and the I/O are made active, the component will perform its designed function. However, if one or more instructions have been executed that interfere with the flow of signals to and from the system logic (such as *EXTEST*), then the system logic may be in an indeterminate state, and reconnecting the pins may result in unpredictable behavior.

No matter what the original state of the controller, it will enter *Test-Logic-Reset* when TMS is held high for at least five rising edges of TCK. The controller remains in this state while TMS is high. If the controller should leave the *Test-Logic-Reset* controller state as a result of erroneous signal values on the TMS or TCK signals (for example, a glitch due to external interference), it will return to the *Test-Logic-Reset* state after three to five rising edges of TCK with the TMS line at the intended high logic level. The intended operation of the test logic is such that no disturbance is caused to on-chip system logic operation as the result of such an error. On leaving the *Test-Logic-Reset* controller state, the controller moves into the *Run-Test/Idle* controller state where no action will occur because the current instruction has been set to select operation of the device identification or bypass register (see 7.2). The test logic is also inactive in the *Select-DR-Scan* and *Select-IR-Scan* controller states.

Note that the TAP controller will also be forced to the *Test-Logic-Reset* controller state by applying a low logic level at TRST*, if such is provided, or at power-up (see 6.1.3).

Run-Test/Idle

A controller state between scan operations. Once entered, the controller will remain in the *Run-Test/Idle* state as long as TMS is held low. When TMS is high and a rising edge is applied at TCK, the controller moves to the *Select-DR-Scan* state.

In the *Run-Test/Idle* controller state, activity in selected test logic occurs only when certain instructions are present. For example, the *RUNBIST* instruction causes a self-test of the on-chip system logic to execute in this state (see 8.10). Self-tests selected by instructions other than *RUNBIST* also may be designed to execute while the controller is in this state.

For instructions that do not cause functions to execute in the *Run-Test/Idle* controller state, time spent in this state constitutes a delay (i.e., Idle) and all test data registers selected by the current instruction shall retain their previous state.

The instruction does not change while the TAP controller is in this state.

Select-DR-Scan

This is a temporary controller state (i.e., the controller exits this state on the next rising edge of TCK) in which all test data registers selected by the current instruction retain their previous state.

If TMS is held low and a rising edge is applied to TCK when the controller is in this state, the controller moves into the *Capture-DR* state and a scan sequence for the selected test data register is initiated. If TMS is held high and a rising edge is applied to TCK, the controller moves on to the *Select-IR-Scan* state.

The instruction does not change while the TAP controller is in this state.

Select-IR-Scan

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state.

If TMS is held low and a rising edge is applied to TCK when the controller is in this state, then the controller moves into the *Capture-IR* state and a scan sequence for the instruction register is initiated. If TMS is held high and a rising edge is applied to TCK, the controller returns to the *Test-Logic-Reset* state.

The instruction does not change while the TAP controller is in this state.

Capture-DR

This is a temporary controller state in which data may be parallel-loaded into the shift-capture path of test data registers selected by the current instruction on the rising edge of TCK that causes the TAP controller to exit this state. If a test data register selected by the current instruction does not have a parallel input, or if capturing is not required for the selected test, the register retains its previous state unchanged.

The instruction does not change while the TAP controller is in this state.

When the TAP controller is in this state and a rising edge is applied to TCK, the controller enters either the *Exit1-DR* state if TMS is held at 1 or the *Shift-DR* state if TMS is held at 0.

Shift-DR

In this controller state, the test data register connected between TDI and TDO as a result of the current instruction shifts data from TDI, one stage toward its serial output, and to TDO on each rising edge of TCK. Test data registers that are selected by the current instruction but are not placed in the serial path retain their previous state unchanged.

The instruction does not change while the TAP controller is in this state.

When the TAP controller is in this state and a rising edge is applied to TCK, the controller either enters the *Exit1-DR* state if TMS is held at 1 or remains in the *Shift-DR* state if TMS is held at 0. The last shift of the test data register occurs on this same rising edge.

Exit1-DR

This is a temporary controller state. If TMS is held high, a rising edge applied to TCK while in this state causes the controller to enter the *Update-DR* state, which terminates the scanning process. If TMS is held low and a rising edge is applied to TCK, the controller enters the *Pause-DR* state.

All test data registers selected by the current instruction retain their previous state unchanged.

The instruction does not change while the TAP controller is in this state.

Pause-DR

This controller state allows shifting of the test data register in the serial path between TDI and TDO to be temporarily halted. All test data registers selected by the current instruction retain their previous state unchanged.

The controller remains in this state while TMS is low. When TMS goes high and a rising edge is applied to TCK, the controller moves on to the *Exit2-DR* state.

The instruction does not change while the TAP controller is in this state.

Exit2-DR

This is a temporary controller state. If TMS is held high and a rising edge is applied to TCK while in this state, the scanning process terminates and the TAP controller enters the *Update-DR* controller state. If TMS is held low and a rising edge is applied to TCK, the controller enters the *Shift-DR* state.

All test data registers selected by the current instruction retain their previous state unchanged.

The instruction does not change while the TAP controller is in this state.

Update-DR

This is a temporary controller state. Some test data registers may be provided with a latched parallel output to prevent changes at the parallel output while data are shifted in the associated shift-capture path in response to certain instructions (e.g., *EXTEST, INTEST*, and *RUNBIST*). Data are latched onto the parallel output of these test data registers from the shift-capture path on the falling edge of TCK in the *Update-DR* controller state. The data held at the latched parallel output should not change other than in this controller state unless operation during the execution of a self-test is required (e.g., during the *Run-Test/Idle* controller state in response to a design-specific public instruction).

All shift-capture paths in test data registers selected by the current instruction retain their previous state unchanged.

The instruction does not change while the TAP controller is in this state.

When the TAP controller is in this state and a rising edge is applied to TCK, the controller enters either the *Select-DR-Scan* state if TMS is held at 1 or the *Run-Test/Idle* state if TMS is held at 0.

Capture-IR

This is a temporary controller state in which a pattern of fixed logic values is parallel-loaded into specific bits of the instruction register shift-capture path on the rising edge of TCK that causes the TAP controller to exit this state. In addition, design-specific data may be parallel-loaded into shift-capture path that are not required to be set to fixed values (see Clause 7).

Test data registers selected by the current instruction retain their previous state. The instruction does not change while the TAP controller is in this state.

When the TAP controller is in this state and a rising edge is applied to TCK, the controller enters either the *Exit1-IR* state if TMS is held at 1 or the *Shift-IR* state if TMS is held at 0.

Shift-IR

In this controller state, the shift-register contained in the instruction register is connected between TDI and TDO and shifts data from TDI, one stage toward its serial output, and to TDO on each rising edge of TCK.

Test data registers selected by the current instruction retain their previous state. The instruction does not change while the TAP controller is in this state.

When the TAP controller is in this state and a rising edge is applied to TCK, the controller either enters the *Exit1-IR* state if TMS is held at 1 or remains in the *Shift-IR* state if TMS is held at 0. The last shift of the instruction register occurs on this same rising edge.

Exit1-IR

This is a temporary controller state. If TMS is held high, a rising edge applied to TCK while in this state causes the controller to enter the *Update-IR* state, which terminates the scanning process. If TMS is held low and a rising edge is applied to TCK, the controller enters the *Pause-IR* state.

Test data registers selected by the current instruction retain their previous state. The instruction does not change while the TAP controller is in this state and the instruction register retains its state.

Pause-IR

This controller state allows shifting of the instruction register to be halted temporarily.

Test data registers selected by the current instruction retain their previous state. The instruction does not change while the TAP controller is in this state and the instruction register retains its state.

The controller remains in this state while TMS is low. When TMS goes high and a rising edge is applied to TCK, the controller moves on to the *Exit2-IR* state.

Exit2-IR

This is a temporary controller state. If TMS is held high and a rising edge is applied to TCK while in this state, termination of the scanning process results, and the TAP controller enters the *Update-IR* controller state. If TMS is held low and a rising edge is applied to TCK, the controller enters the *Shift-IR* state.

Test data registers selected by the current instruction retain their previous state. The instruction does not change while the TAP controller is in this state and the instruction register retains its state.

Update-IR

This is a temporary controller state in which the bits in the instruction register shift-capture path are latched onto the parallel output on the falling edge of TCK in this controller state. Once the new value has been latched, it becomes the current instruction.

Test data registers selected by the current instruction retain their previous state.

When the TAP controller is in this state and a rising edge is applied to TCK, the controller enters the *Select-DR-Scan* state if TMS is held at 1 or the *Run-Test/Idle* state if TMS is held at 0.

General

The *Pause-DR* and *Pause-IR* controller states are included so that shifting of data through the test data or instruction register can be halted temporarily. For example, this might be necessary to allow an ATE system to reload its memory from disc during application of a long test sequence.

The TAP controller states include the three basic actions required for testing: stimulus application (*Update-DR*), execution (*Run-Test/Idle*), and response capture (*Capture-DR*). However, not all these actions are required for every type of test. Table 6-1 lists the actions required for key types of test supported by this standard.

Table 6-1—Use of controller states for different test types

Test type	**Action required in this controller state**		
	Update-DR	***Run-Test/Idle***	***Capture-DR***
Boundary-scan external test (e.g., *EXTEST*)	Yes	No	Yes
Boundary-scan internal test (e.g., *INTEST*)	Maybe	No	Yes
Boundary-scan *PRELOAD*	Yes	No	Maybe
Boundary-scan *SAMPLE*	Maybe	No	Yes
Boundary-scan INIT_SETUP, INIT_SETUP_CLAMP	Yes	No	Maybe
Boundary-scan *INIT_RUN*	No	Maybe	Yes
Internally controlled built-in tests (e.g. *RUNBIST*)	No	Yes	Maybe
Internal scan test (i.e., a design-specific PUBLIC instruction)	Maybe	No	Yes

In Table 6-1, an entry of “Yes” or “No” indicates that the rules for that instruction require or do not require specific actions. An entry of “Maybe” indicates that the rules provide a choice and one choice will require action and another choice will not.

For scan testing, the stimulus is made available for use at the end of shifting or, if a parallel output latch is included, by updating the parallel output in the *Update-DR* controller state. The results of the test are captured into the test data register during the *Capture-DR* controller state.

For internally controlled self-testing circuit designs, the starting values of the registers are available at the end of shifting and no parallel output latch is required. The registers should operate under control of the internal test logic during *Run-Test/Idle*. Assuming the result is already contained in a test data register, no action is required during the *Capture-DR* controller state.

For an internal scan test, the target register consists of a serial concatenation of storage elements that support the normal system operation of the component. The construction of such a register is beyond the scope of this standard, and parallel output latches may or may not be present in a given implementation.

6.1.2 TAP controller operation

6.1.2.1 Specifications

Rules

a) The TAP controller shall change state only in response to the following events:
   1) A rising edge of TCK
   2) A transition to logic 0 at the TRST* input (if provided)
   3) Power-up of the integrated circuit containing the TAP

b) The TAP controller shall generate signals to control the operation of the test data registers, instruction registers, and associated circuitry as defined in this standard (Figure 6-3 and Figure 6-4).

   NOTE 1—In Figure 6-3 and Figure 6-4, the assumption is made that the signals applied to TMS and TDI change state on the falling edge of TCK. The time at which these signals change state is not defined by this standard, but it should be such that the setup and hold requirements of TMS and TDI are met. It is further assumed that the design includes the optional device identification register. Therefore, the figures show the *IDCODE* instruction being set onto the output of the instruction register in the *Test-Logic-Reset* controller state. If the device identification register is not included in the design, the output of the instruction register will be set to the *BYPASS* instruction in the *Test-Logic-Reset* controller state.

c) The TDO output buffer and the circuitry that selects the register output fed to TDO shall be controlled as shown in Table 6-2.

d) Changes at TDO defined in Table 6-2 shall occur on the falling edge of TCK after entry into the state.

   NOTE 2—The TDO driver is actually active from the first falling edge of TCK in the TAP controller state shown to the first falling edge of TCK after leaving the TAP controller state shown. In other words, the TDO is enabled for a period one-half TCK cycle behind the actual TAP controller state.

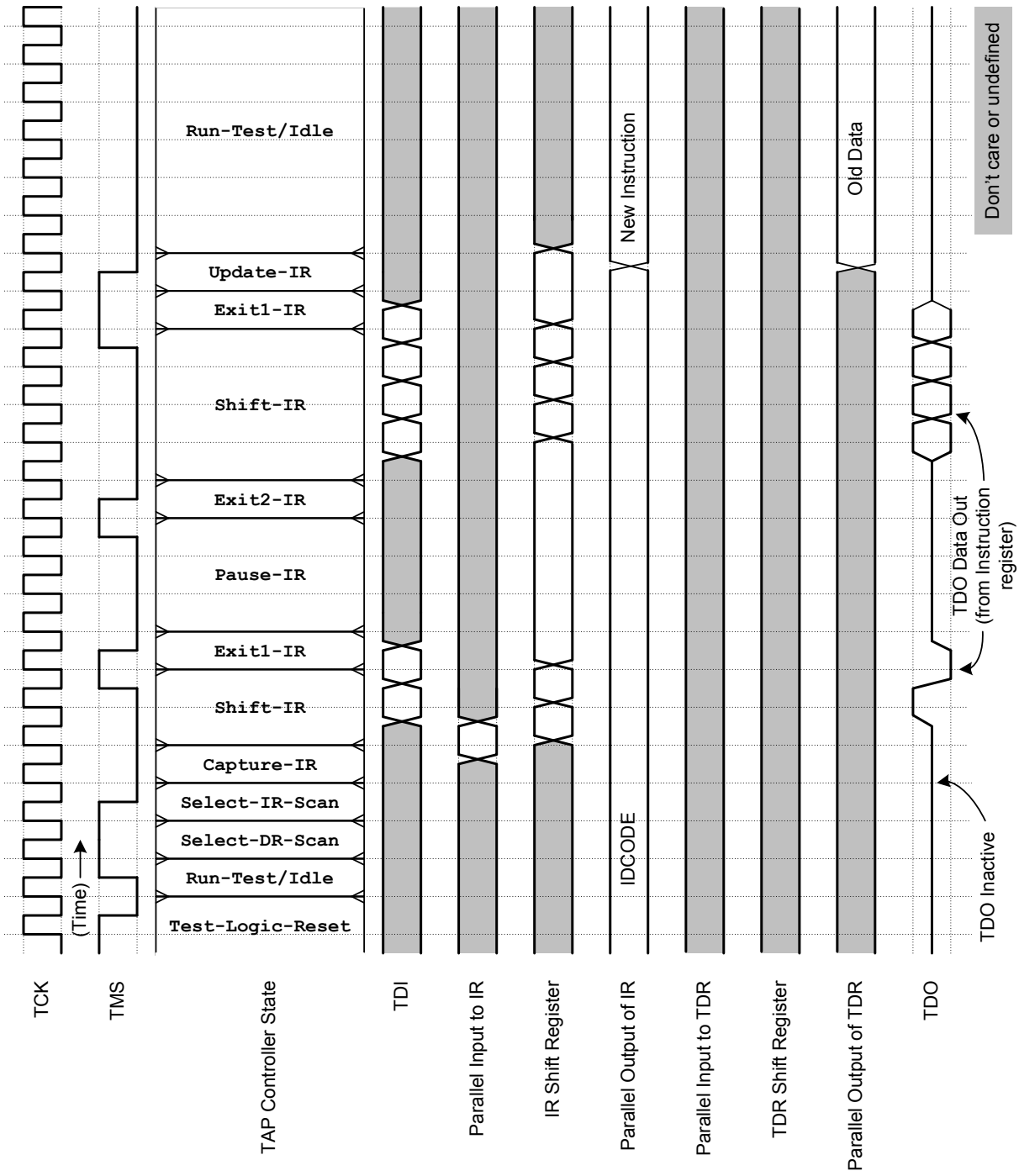


Figure 6-3—Test logic operation: instruction scan

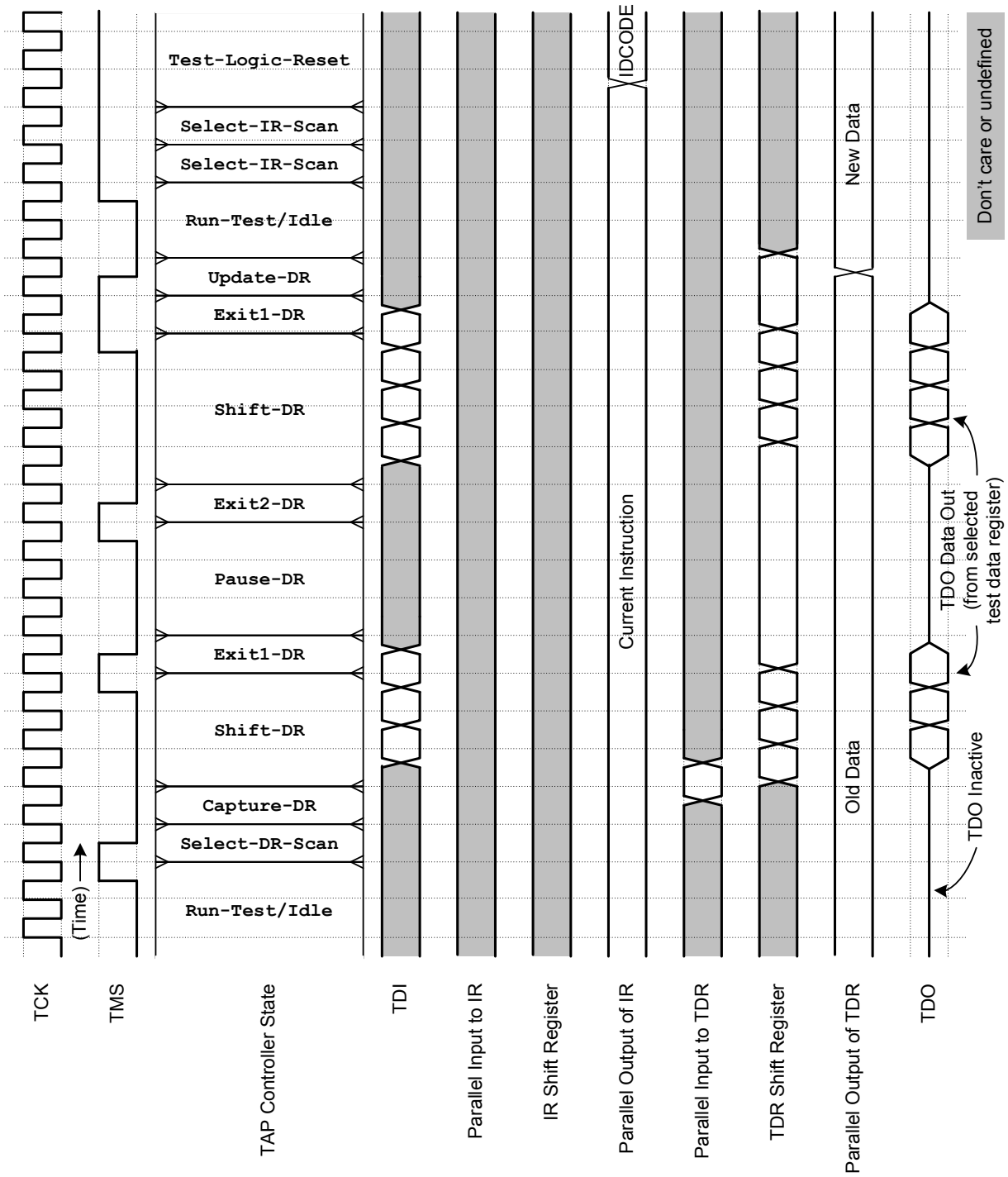


Figure 6-4—Test logic operation: data scan

Table 6-2—Test logic operation in each controller state

Controller State	Register Selected to Drive TDO	TDO Driver
Test-Logic-Reset	Undefined	Inactive
Run-Test/Idle	Undefined	Inactive
Select-DR-Scan	Undefined	Inactive
Select-IR-Scan	Undefined	Inactive
Capture-IR	Undefined	Inactive
Shift-IR	Instruction	Active
Exit1-IR	Undefined	Inactive
Pause-IR	Undefined	Inactive
Exit2-IR	Undefined	Inactive
Update-IR	Undefined	Inactive
Capture-DR	Undefined	Inactive
Shift-DR	Test data	Active
Exit1-DR	Undefined	Inactive
Pause-DR	Undefined	Inactive
Exit2-DR	Undefined	Inactive
Update-DR	Undefined	Inactive

6.1.2.2 Description

An example of a circuit that meets the requirements is shown in Figure 6-5 and Figure 6-6. This circuit generates a range of clock and control signals required not only to control the selection between the alternative instruction and test data register paths and the activity of TDO (as defined in Table 6-2), but also to control the example implementations of other items of test logic that are contained in this standard.

The circuit in Figure 6-5 generates the various control signals used by the example circuits illustrated elsewhere in this standard. Note that the Select signal would be used to control the multiplexer shown in Figure 5-1 and that the Enable signal would be used for three-state control of the TDO output. These control signals support both TDR clocking styles: gated clock and free-running clock with data wrap-back. Gated clocks *Clock-DR* and *Update-DR* and state decode *Shift-DR* support the gated clock style. State decodes *Shift-DR*, *Capture-DR*, and *Update-DR* state support the free-running clock style. The state decodes *Shift-DR* and *Capture-DR* are both delayed to the falling TCK edge so that they are valid when TCK rises.

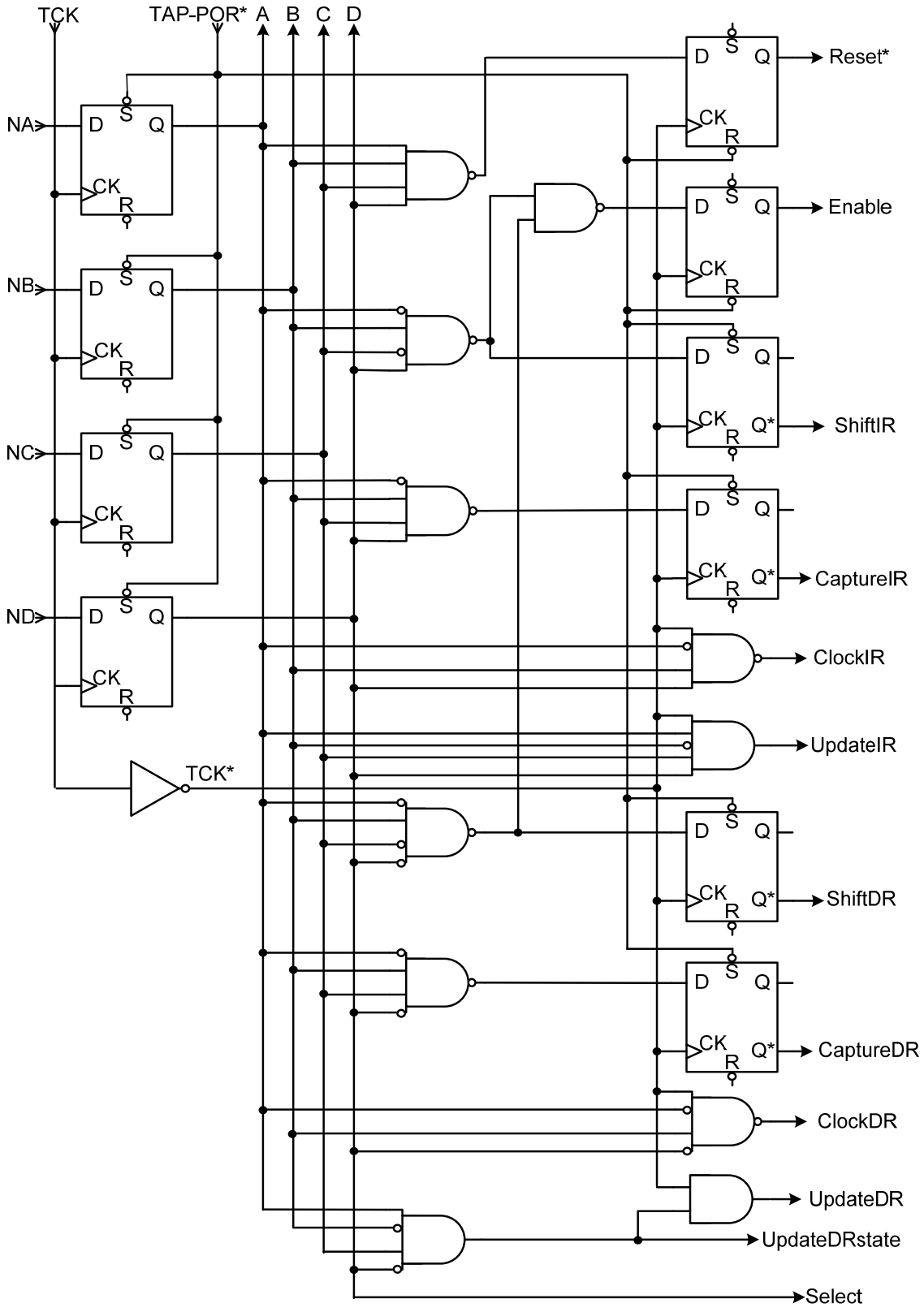


Figure 6-5—TAP controller implementation—state registers and output logic

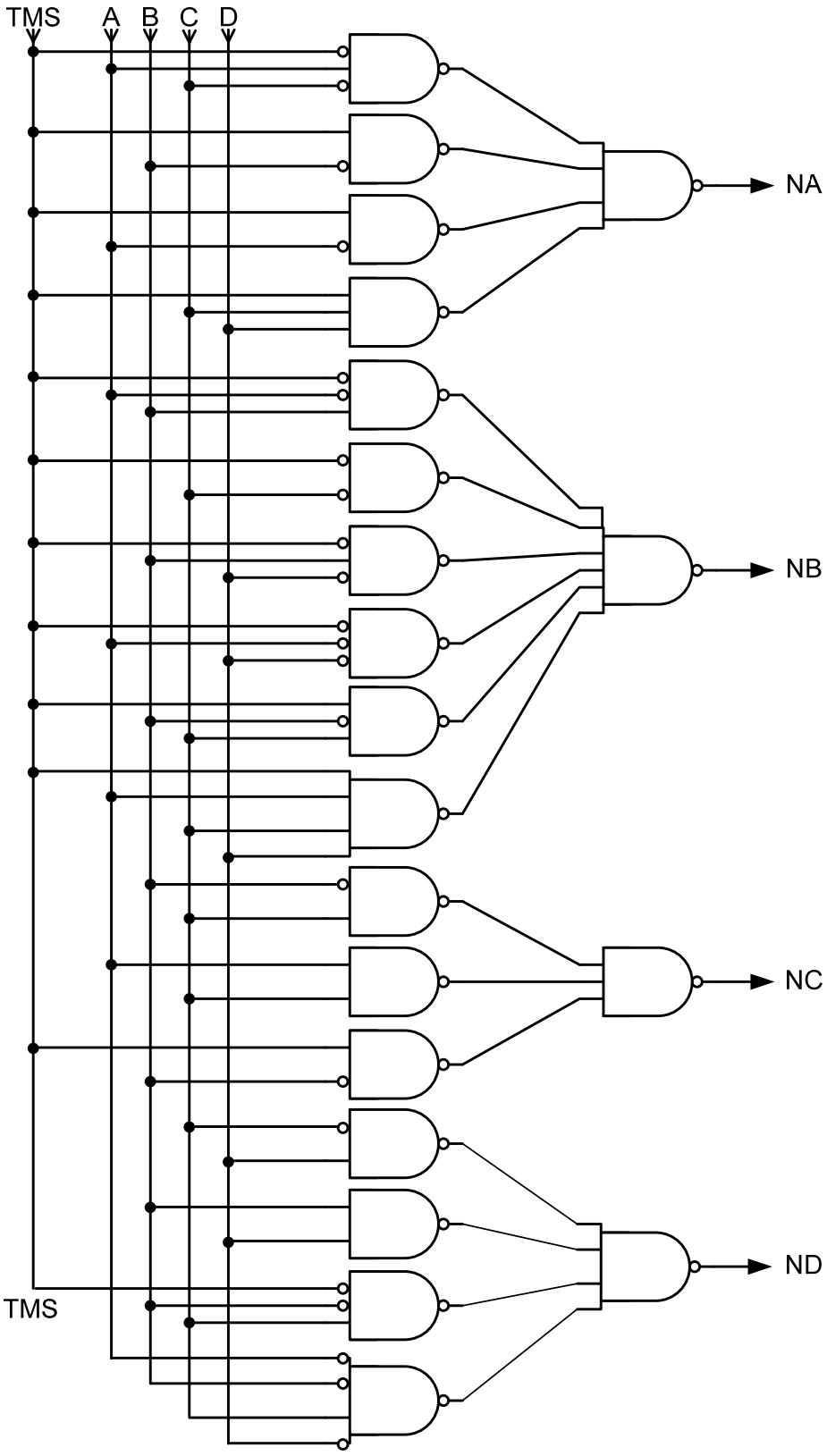
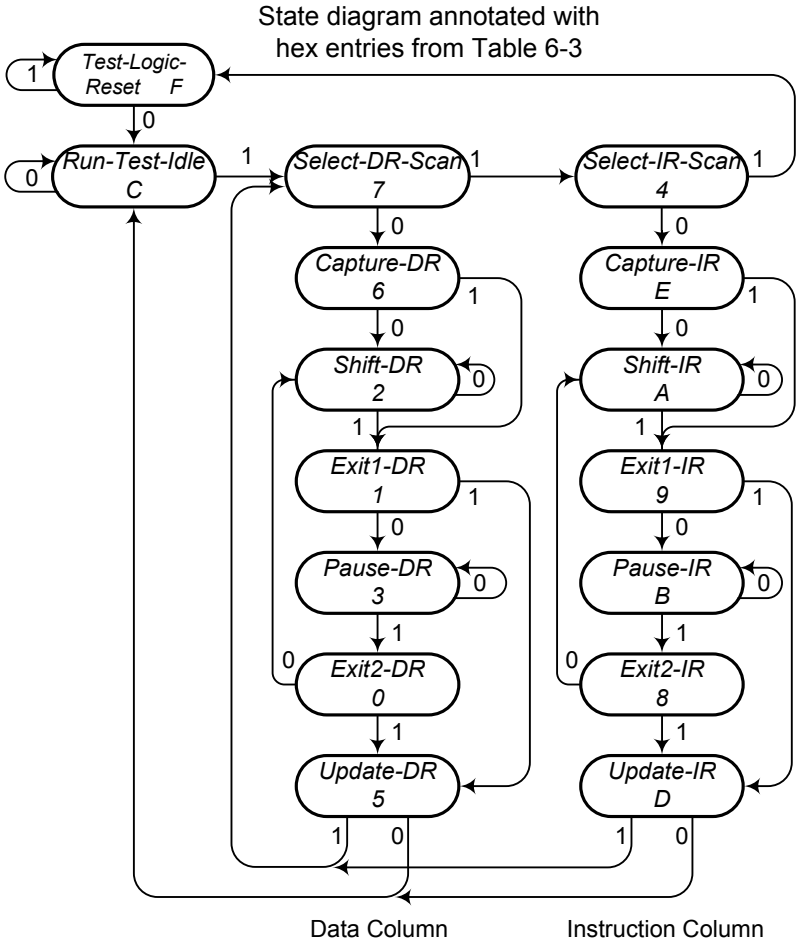


Figure 6-6—TAP controller implementation—next state logic

The assignment of controller states in the example implementation is given in Table 6-3.

Table 6-3—State assignments for example TAP controller

Controller State	DCBA (hex)
Exit2-DR	0
Exit1-DR	1
Shift-DR	2
Pause-DR	3
Select-IR-Scan	4
Update-DR	5
Capture-DR	6
Select-DR-Scan	7
Exit2-IR	8
Exit1-IR	9
Shift-IR	A
Pause-IR	B
Run-Test/Idle	C
Update-IR	D
Capture-IR	E
Test-Logic-Reset	F



The Boolean equations for the next state logic in Figure 6-5 and Figure 6-6 are as follows:

```
ND := DC* + DB + T*CB* + D*CB*A*
NC := CB* + CA + TB*
NB := T*BA* + T*C* + T*D*B + T*D*A* + TCB* + TDCA
NA := T*C*A + TB* + TA* + TDC
where
T = value present at TMS
```

Figure 6-7 shows the operation of this controller implementation through instruction and test data register scan cycles.

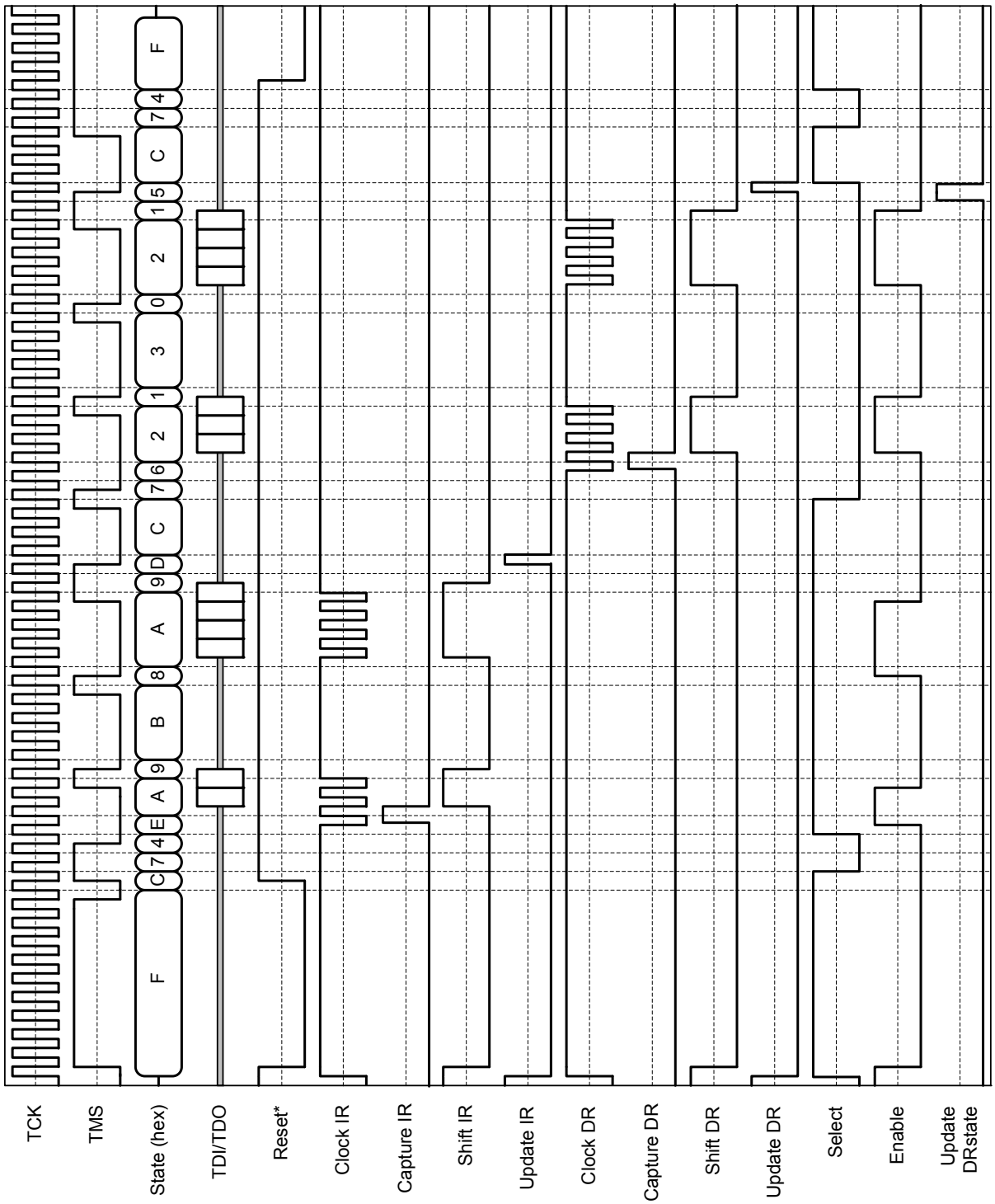


Figure 6-7—Operation of the example TAP controller

6.1.3 TAP controller initialization

6.1.3.1 Specifications

Rules

a) The TAP controller shall be forced into the *Test-Logic-Reset* controller state at power-up either by use of the TRST* signal or as a result of circuitry built into the test logic or both.

NOTE—If the TAP controller is to be reset at power-up using TRST*, the design of the assembled system has to apply a logic 0 to TRST* when power is applied. Similarly, where the TAP controller is to be reset using TRST* after enabling of compliance to this standard as described in 4.8, the design of the assembled system has to apply a logic 0 to TRST* when compliance is enabled.

b) The TAP controller shall not be initialized by operation of any system input, such as a system reset.

c) Where a dedicated reset pin (TRST*) is provided to allow initialization of the TAP controller, initialization shall occur asynchronously (without dependence on TCK or any other clock) when the TRST* input changes to the low logic level.

d) Where the TAP controller is initialized at power-up by operation of circuitry built into the test logic, the result shall be equivalent to that which would be achieved by application of a logic 0 to a TRST* input.

6.1.3.2 Description

In a board design that contains multi-sourced nets, provisions have to be made to help ensure that at power-up any period of contention between drivers on the bus is kept within limits that cause no damage to the components on the board. Therefore, when boundary-scan circuitry is inserted between the on-chip system logic and package pins, it becomes essential that shortly after power-up, the test circuitry enters a state where output drivers are controlled by the system circuitry, i.e., the *Test-Logic-Reset* TAP controller state and the *Persistence-Off* TMP controller state (see 6.2.1).

NOTE—Clause 11 contains rules that define required behavior of the boundary-scan register during power-up.

While the TAP controller will synchronously enter the *Test-Logic-Reset* controller state after five rising edges at TCK with TMS held high, the worst-case time taken to reach this state may exceed that at which damage could occur. Furthermore, it cannot be guaranteed that the clock will be running at the time at which power is applied to the board. Therefore, the "reset at power-up" requirement is included, and its use is intended to be only for power-up. In this standard, the term "TAP_POR*" and a signal named TAP_POR* in example implementation figures are used to refer to whatever mechanism is used to reset the TAP (and other test components required to be reset at power-up).

The requirement can be met in a variety of ways, for example, by inclusion of a power-up reset generator within the integrated circuit, or by asymmetric design of the latches or registers used to construct the TAP controller. It also could be met by inclusion of a dedicated TRST* pin for the TAP controller. However, a system reset cannot also be used to initialize the TAP controller since this would compromise the ability to test system interconnections at the board level using the boundary-scan circuitry. In some systems, it also may be possible to use the independence of the system and test resets to allow sampling and examination of data after a system failure. This would require that the test logic be reset before re-initialization of the on-chip system logic.

Where a power-up reset facility is provided within the component, this can be used to initialize both the system and the test logic, for example, as shown in Figure 6-8.

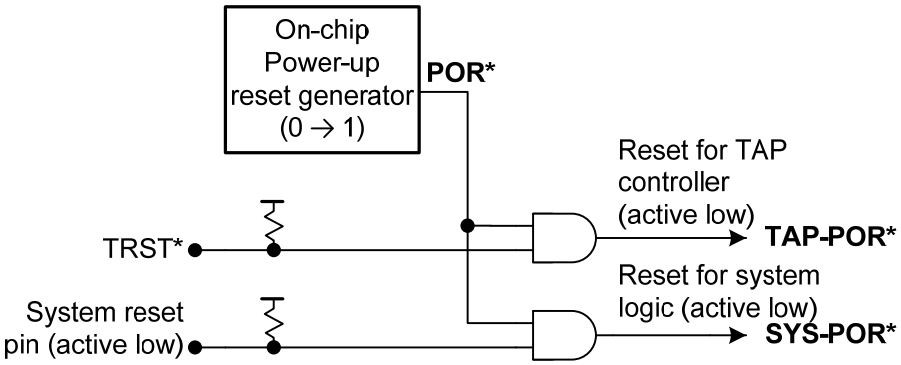


Figure 6-8—Example implementation of the power-up reset for on-chip system and test logic

6.2 Test mode persistence (TMP) controller

The optional test mode persistence (TMP) controller provides a means to place and hold the component in test mode, enabling on-chip tests and on-chip assisted external board- or system-level functional tests while on a board or in a system. For the purpose of this standard, test mode means that the component pins are controlled from the boundary-scan register, and the system logic is held in a safe state, as needed, while performing tests. Note that if *all* of the on-chip system logic is in a reset state, then the ability to perform on-chip tests will not be achieved and it is the responsibility of the component designer to determine the proper way to protect the component during the *Persistence-On* state while allowing any desired design-specific tests to proceed.

This capability may also be used to prevent a component from attempting to return to functional operation when a non-test mode instruction, such as *BYPASS* or *PRELOAD,* is inserted between test mode instructions such as *EXTEST* or *RUNBIST*.

During board or system test, the TMP controller provides control over which components are in test mode and which are not, independent of the active instruction. Such control may keep an integrated circuit, board, or system safe until the circuit under test is either powered down or a proper reset sequence can be performed to safely bring the integrated circuit, board, or system out of test and to make it ready for other operations.

When the optional TMP controller is implemented, the TAP_POR* reset and the *Test-Logic_Reset* TAP controller reset are not equivalent. The TMP controller will keep the component in test mode even when the *Test-Logic-Reset* state is entered by the TAP controller state machine transitioning, and it may prevent the reset of selected parts of the test logic. Assertion of TAP-POR* will always reset both the TAP controller state machine and the TMP controller state machine, thereby resetting all of the test logic and placing the component back in functional mode.

The resources required to implement the optional TMP controller include a simple state machine (see Figure 6-9), the three instructions *CLAMP_HOLD*, *CLAMP_RELEASE*, and *TMP_STATUS* (see 8.20), and a 2 bit TMP status register (see Clause 16).

6.2.1 TMP controller state diagram

6.2.1.1 Specifications

Rules

a) The state diagram for the TMP controller shall be as shown in Figure 6-9.

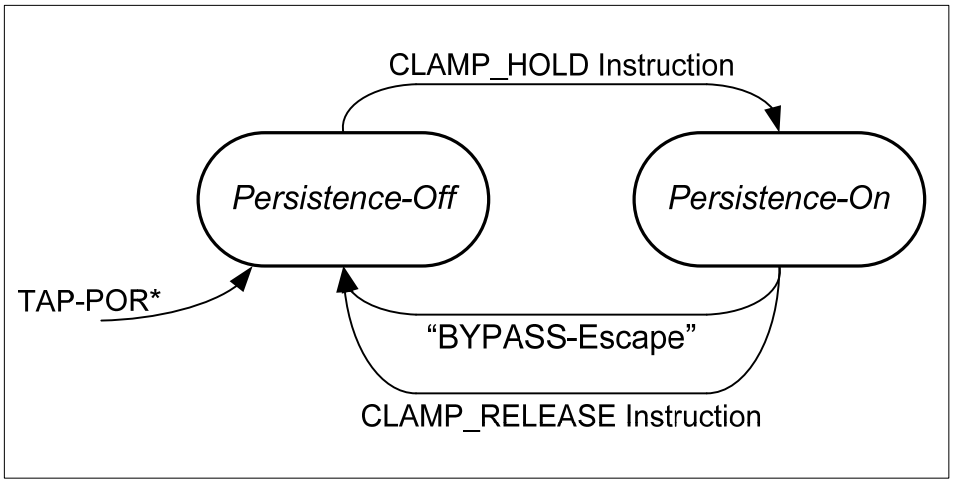


Figure 6-9—Persistence controller state diagram

b) The TMP controller shall change state only in response to the following events:
   1) Any rising edge of TCK when either the *CLAMP_HOLD* or the *CLAMP_RELEASE* instruction is active (see 8.20)
   2) A transition to logic 0 at the TRST* input (if provided, see Figure 6-8)
   3) On-chip reset at power-up (i.e., POR; if provided, see Figure 6-8)
   4) The rising edge of TCK when the bypass-escape bit of the TMP status register is asserted (it contains a 1), and the *BYPASS* instruction is in the Instruction Register and the TAP controller is in the *Update-IR* TAP controller state (the transition is labeled "BYPASS-Escape" in Figure 6-9).

NOTE 1—Either the *IDCODE* instruction or the *BYPASS* instruction is forced into the Instruction register by entering the *Test-Logic-Reset* TAP controller state according to rule e) and rule f) in 7.2.1. This is not a condition for resetting the TMP controller state since the TAP controller state does not pass through *Update-IR*.

NOTE 2—Any opcode specifically identified to be *BYPASS*, or any that defaults to *BYPASS*, will satisfy this "BYPASS-Escape" option.

NOTE 3—The TMP status register is selected for scanning by the *CLAMP_HOLD* instruction. See 8.20 for the instruction definition and Clause 16 for the register definition.

c) The state of the TMP controller shall not be altered by the operation of any system input, including a system reset input.

Recommendations

d) When the TMP controller is provided, the opcodes for the *BYPASS* instruction should include the all 0s opcode in addition to the required all 1s opcode.

6.2.1.2 Description

The optional TMP controller element allows the state of the signals driven from component pins to be determined from the boundary-scan register (and from the effects of the initialization instructions, when provided) while any other instruction that would normally return the control of the pins to the system logic is active.

In addition, the optional TMP controller element allows the state of inputs to the system logic to be determined from the boundary-scan register where control-and-observe boundary cells are provided (see 11.5). Other inputs to the

system logic will still reflect the value received from the external logic, and the component designer can provide control of such inputs if they could adversely affect the system logic or alter a test to be run in the component.

The TMP controller has two states: *Persistence-Off* and *Persistence-On*, as shown in Figure 6-9. Either the TRST* or on-chip Power-On Reset (POR) transitions must be provided, and they are asynchronous (un-clocked) events. The other transitions are synchronous and occur on the rising edge of TCK. The "BYPASS-Escape" transition is taken only when all of the following are true:

— The TAP controller state is *Update-IR*.
— The *BYPASS* instruction is the new instruction.
— The bypass-escape bit of the TMP status register is set to its default value of 1.

Note that this assumes a half-TCK cycle for decoding the *BYPASS* instruction once it is loaded into the IR update latches. The device designer is responsible for ensuring that the design meets timing requirements.

The TMP status register bypass-escape bit allows a component to escape the test mode if setting the TMP controller to *Persistence-On* state were to break the TDI/TDO path of the scan chains and prevent loading the *CLAMP_RELEASE* instruction. A broken scan chain will normally act as either stuck-at 0 or stuck-at 1, and if these are both (all 0s and all 1s) encoded as the *BYPASS* instruction, then performing an instruction register scan and update would force the TMP controller to the *Persistence-Off* state if the TMP status register bypass-escape bit were in its default state of 1. This will occur any time that an opcode that decodes to the *BYPASS* instruction is in the instruction register and the TAP controller is in the *Update-IR* state. This may be referred to as a "BYPASS-Escape."

Figure 6-10 illustrates a possible TMP controller design that meets the requirements of this clause. For an example design of the TMP status register, see Figure 6-1.

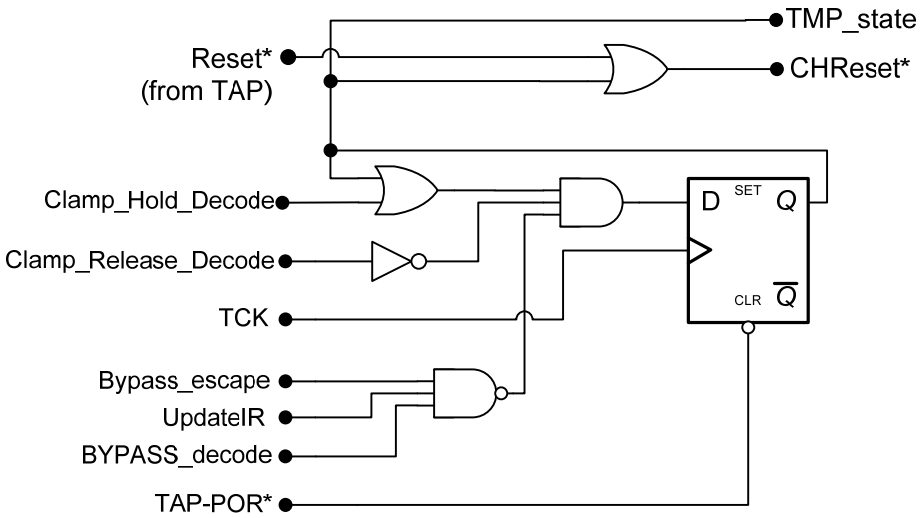


Figure 6-10—Test mode persistence (TMP) controller example design

Figure 6-10 also shows the generation of the CHReset* signal from the TAP Reset* signal (see Figure 6-5). The CHReset* signal will be used by all resettable boundary-scan register control cells when the TMP controller is provided. It also may be selected for use by design-specific test data registers. Figure 6-8 shows the generation of the TAP_POR* signal, an input to the TAP controller, from the TRST* or on-chip power-up reset generation circuit.

A prime use of the TMP controller is to override the various mode controls for the boundary-scan cells. Since there are multiple mode signals (Mode1 through Mode7) defined in the tables in Clause 11, the effect of the TMP controller on the mode signals is shown in those tables instead of in Figure 6-10.

6.2.2 TMP controller operation

6.2.2.1 Specifications

Rules

a) When an instruction is active that causes the component pins to be controlled by the test logic, the TMP controller state shall not alter the behavior of the instruction.

NOTE 1—For the instructions defined in this standard, this includes *EXTEST, INTEST, RUNBIST, CLAMP, HIGHZ, INIT_RUN*, and the two instructions: *CLAMP_HOLD* and *CLAMP_RELEASE*.

b) When a design-specific instruction is active that is normally defined as interfering with the flow of signals through the component pins to and from the system logic, then the TMP controller state of *Persistence-Off* shall not alter the behavior of the instruction.

c) When any instruction is active that is normally defined as *not* interfering with the flow of signals through the component pins to and from the system logic, then the TMP controller state of *Persistence-On* shall cause the state of signals driven from all system output pins provided with boundary cells in any included boundary-scan register segments (see 9.4) to be controlled by the data in those cells.

NOTE 2—The affected instructions defined in this standard include *BYPASS, IDCODE, USERCODE, PRELOAD, SAMPLE, INIT_SETUP,* and *IC_RESET*.

d) When the TMP controller state is *Persistence-On*, the TAP controller state of *Test-Logic-Reset* shall reset the Instruction Register as required in rule e) of 7.2.1, but it shall not reset any of the test logic controlling the component pins, including boundary-scan register cells, the results of the initialization instructions, and the state of any cells controlling excludable boundary register segments (see 9.4).

e) When the TMP controller state is *Persistence-On*, the boundary cells and the on-chip system logic shall be controlled such that damage or other undesired conditions cannot occur as a result of signals received at the system input or system clock input pins.

NOTE 3—If, during the *Persistence-On* state, some input activity could cause unsafe states internal to the device, then appropriate portions of the on-chip system logic can be placed in a reset or "hold" state.

Permissions

f) While a design-specific instruction or group of design-specific instructions is active that requires use of specific I/O pins, then the instruction decode or a decode of the associated test data register may override the signal or signals placing those specific I/O in test mode when the TMP controller is in the *Persistence-On* state.

6.2.2.2 Description

The optional TMP controller will hold the device in the test mode (that is, the I/O controlled by the boundary-scan register) and, as needed, hold the system logic in a safe state for performing tests. This state is maintained independent of any instruction that will be executed while it is set, giving the test engineer greater flexibility and control of the multiple, sequential test processes that may be defined at various packaging levels. This capability is specifically provided for components with design-specific test data registers (TDRs) controlling internal tests and ICs that support initialization, but it should be useful in many other situations. This controller enables safe use of on-chip test capabilities when the IC is in-system by preventing the I/O from being reconnected to system logic that may be in an indeterminate state. This controller also reduces unnecessary reloading of the initialization data and boundary-scan register between IEEE 1149.1-based board tests.

One implication of the TMP controller is that the component designer cannot rely on the I/O being in functional mode when executing a design-specific instruction, even if that instruction is defined to leave the I/O in functional

mode. On-chip, design-specific test functions such as a pseudo-random binary sequence (PRBS) type of built-in test may require specific pins to be functional even if the remaining I/O pins are in test mode while the PRBS is executed. In that case, permission f) of this clause allows the IC designer to override the "mode" signals to the appropriate boundary-scan cells while such a design-specific instruction and test are active, so that the pins needed are forced to functional mode regardless of the presence or state of the TMP controller. When the design-specific instruction or group of instructions are no longer active, the override must be removed.

A second implication of the TMP controller is that the IC designer has an additional option for resetting TDRs. In addition to POR or TRST* (whichever is used to reset the TAP) and the TAP Reset* decode of the *Test-Logic-Reset* (TLR) TAP controller state, there is a copy of Reset* called CHReset* (see Figure 6-10) that has been gated with the TMP controller state. If this signal is used, the TDR will not reset in the TAP controller TLR state if the TMP controller is in the *Persistence-On* state.

The TMP controller preserves the content and effect of the initialization data, boundary, and possibly other test data registers when the *Test-Logic-Reset* TAP controller state is entered. In large part, this preservation of data is to prevent the I/O of the component from changing. This also reduces the need to reload these registers when resets are encountered.

The TMP controller *Persistence-On* state also requires that the internal logic, now disconnected from the I/O, be maintained in a safe state, just as with *CLAMP* and any standard instruction that forces the test mode. Board testing using this standard can leave a complex IC's internal logic in an unpredictable and potentially destructive condition. Entering the *Test-Logic-Reset* state in between board tests may not be sufficient to restore the component back to a safe or functional mode. The TMP controller enables the test engineer to set persistence prior to board testing and then subsequently release the component back into functional mode in a controlled manner independent of other instructions executed in the meantime.

The effect of the TMP controller on the I/O pin behavior during the standard instructions is shown in Table 8-5.

Design-specific TDRs may access one or more on-chip test IP blocks such as BIST engines and system monitors. The instructions, which are used to select these target TDRs, are responsible for setting the pins to either a functional mode or a test mode. This requires the designer to partition on-chip test functions onto individually distinct TDRs, one or more associated with an instruction that forces test mode, and one or more for functional mode, or provide two instructions for the same TDR, one of which forces test mode and the other does not. Once made, there is no flexibility in these assignments. Selections were made by the component designer based on the IP block's purpose and how the pins should behave during manufacturing testing. However, some test IP blocks require the I/O mode to be set based on the environment the IC is in. For instance, for a particular test, the I/O may need to be in functional mode on an IC tester but in test mode when the IC is soldered in a system. Figure 6-11 shows an example IC with four on-chip IP blocks on a single design-specific TDR.

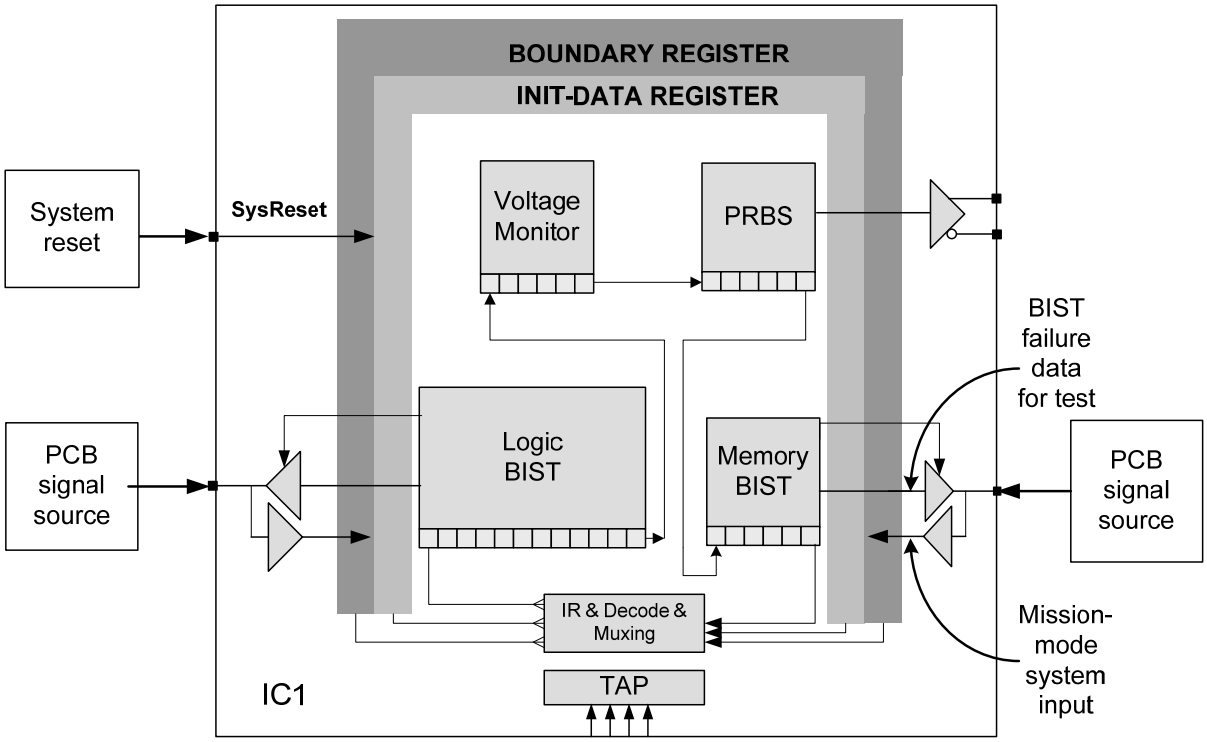


Figure 6-11—Example IC, which benefits from test mode persistence

On-chip logic BIST functions will often have the effect of causing some or all of the mission mode I/O to toggle as the BIST sequences are executed. The I/O are shared between the BIST and mission mode. This may be very desirable when the IC is on the IC ATE. Figure 6-11 illustrates a component where both the Logic BIST and the Memory BIST may cause one or more shared I/O to change from a system input to an output during BIST. Such on-chip test IP blocks may intentionally use the I/O for streaming failure data over shared functional pins on the IC tester, but that could be undesirable for an in-system IC test where those pins are connected to other components. If the other component was a signal source not controlled during test, then allowing BIST to control the pin would create a potential conflict, an example of an obstacle that a test generation program must avoid. The TMP controller gives the test engineer the ability to put the component into test mode and avoid the potential conflict or other obstacle.

In another example, the component designer may have placed, say, a voltage monitor on the same TDR as other functions that would normally require test mode, but the instruction selecting this TDR is not a test mode instruction to allow monitoring of the voltage during system operation. The TMP controller would allow the test engineer to put the component into test mode as needed, while allowing mission mode operation of the monitor.

On-chip tests could also be affected by glitching or undefined system reset signals, which drive into the IC logic during testing in the system. Using the TMP controller to put the component I/O into test mode can block such resets.

In addition to maintaining the component in a test state between tests, or between a test and a controlled reset process, the TMP controller allows the system test engineer to put the component into test mode or mission mode, as needed, during system tests. This simplifies the component designer’s task when incorporating built-in tests and enhances testing in ways that are not always anticipated.

6.2.3 TMP controller initialization

6.2.3.1 Specifications

Rules

a) If a dedicated test reset pin (TRST*) is provided to initialize the TAP controller, then TRST* shall also force the TMP controller to the *Persistence-Off* state.

b) If the TAP controller is initialized at power-up by operation of circuitry built into the test logic (POR), then the same logic shall also force the TMP controller to the *Persistence-Off* state.

6.2.3.2 Description

The TMP controller must be reset at power-up by the same mechanism used to initialize the TAP controller: either an on-chip Power-On-Reset circuit or the TAP TRST* port.

7. Instruction register

The instruction register allows an instruction to be shifted into the design. The instruction is used to select the test to be performed or the test data register to be accessed or both. As will be discussed in Clause 8, a number of mandatory and optional instructions are defined by this standard. Further design-specific instructions can be added to extend the test logic functionality built into a component.

Optionally, the instruction register allows examination of design-specific information generated within the component.

This clause contains the design requirements for the instruction register.

7.1 Design and construction of the instruction register

The instruction register is a shift-register-based design that has an optional parallel input for register cells other than the two nearest to the serial output. The instruction shifted into the register is latched during the *Update-IR* TAP controller state.

7.1.1 Specifications

Rules

a) The instruction register shall include at least two shift-register-based cells capable of holding instruction data.
b) The instruction shifted into the instruction register shall be latched such that changes in the effect of an instruction occur only in the *Update-IR* and *Test-Logic-Reset* TAP controller states (see 7.2).
c) There shall be no inversion of data between the serial input and the serial output of the instruction register.
d) The two least significant instruction register cells (i.e., those nearest the serial output) shall load a fixed binary "01" pattern (the 1 into the least significant bit location) in the *Capture-IR* TAP controller state (see 7.2).

Recommendations

e) Where the parallel inputs of instruction register cells are not required to load design-specific information, then these cells should be designed to load fixed logic values (0 or 1) in the *Capture-IR* TAP controller state.

Permissions

f) Parallel inputs may be provided to instruction register cells (other than the two least significant cells) to permit capture of design-specific information in the *Capture-IR* TAP controller state.

7.1.2 Description

The parallel output from the instruction register is latched so the test logic is protected from the transient data patterns that will occur in its shift-register stages as new instruction data are entered. The latched parallel output is controlled such that it can change state only in the *Update-IR* and *Test-Logic-Reset* controller states. The timing and nature of these changes are discussed in detail in 7.2.

The minimum size (two instruction register cells) is necessary to meet rules stated elsewhere in this standard:

— The instruction register must allow selection of the bypass register.
— The instruction register must allow access to the boundary-scan register in at least three configurations (*EXTEST*, *PRELOAD*, and *SAMPLE*—see 8.2).

It is permissible [see permission h) of 8.1.1] for instructions to share binary codes provided that none of the rules that specify the merged instructions is violated. In earlier editions of this standard, *SAMPLE* and *PRELOAD* were specified as a merged instruction—*SAMPLE/PRELOAD*—such that the four instructions mandated by this standard—*BYPASS*, *EXTEST*, *PRELOAD*, and *SAMPLE*—could be implemented using three binary codes, leaving the fourth achievable with a minimum 2 bit instruction register available for implementation of an optional instruction.

In addition, fault isolation of the board-level serial test data path shall be supported. This is achieved by loading a constant binary "01" pattern into the least significant bits of the instruction register at the start of the instruction-scan cycle.

The inclusion of the optional design-specific data inputs to the instruction register allows key data signals within the device to be examined at the start of testing, with future test actions potentially depending on the design-specific information gathered. Where the parallel inputs to instruction register cells are not used for design-specific information, it is recommended that these cells be designed to load a fixed logic value (0 or 1) during the *Capture-IR* TAP controller state and that that value be documented in the BSDL to enhance component debug.

7.2 Instruction register operation

7.2.1 Specifications

Rules

a) The behavior of the instruction register in each TAP controller state shall be as defined in Table 7-1.

b) This rule is moved to rule e) in 8.1.1 in this version of the standard.

c) All operations of shift-register stages shall occur on the rising edge of TCK after entry into a controller state.

Table 7-1—Instruction register operation in each controller state

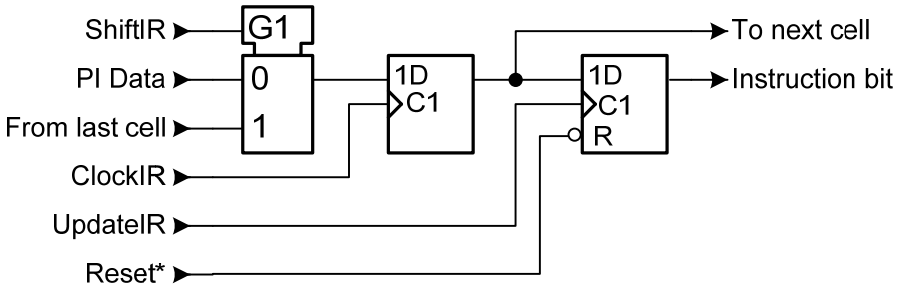
Controller state	Shift-Register stage	Parallel output
Test-Logic-Reset	Undefined	Reset to the *IDCODE* (or *BYPASS*) instruction
Capture-IR	Load 01 into bits closest to TDO and, optionally, design-specific data or fixed values into other bits closer to TDI	Retain last state
Shift-IR	Shift toward serial output	Retain last state
Exit1-IR *Exit2-IR* *Pause-IR*	Retain last state	Retain last state
Update-IR	Retain last state	Load from shift-register
All other states	Undefined	Retain last state

d) The data present at the parallel output of the instruction register shall be latched from the shift-register stage on the falling edge of TCK in the *Update-IR* controller state.

e) After entry into the *Test-Logic-Reset* controller state as a result of the clocked operation of the TAP controller, the *IDCODE* instruction (or, if there is no device identification register, the *BYPASS* instruction) shall be latched onto the instruction register output on the falling edge of TCK.

f) If the TRST* input is provided and a low signal is applied to the input, the latched instruction shall change asynchronously to *IDCODE* (or, if no device identification register is provided, to *BYPASS*).

7.2.2 Description



NOTE—The parallel output flip-flop in this figure is provided with a reset input. To meet rule e) and rule f) of 7.2.1), some or all instruction register cells will require use of a set, rather than a reset, input.

Figure 7-1—Instruction register cell (gated clock)

Figure 7-1 shows an implementation of an instruction register cell that satisfies these requirements and operates in response to the signals generated by the example TAP controller design contained in 6.1.2:

— The clock input (*Clock-IR*) to the register in the serial path is applied only during the *Capture-IR* and *Shift-IR* TAP controller states. The clock input (*Update-IR*) to the hold register is applied only during the *Update-IR* TAP controller state.

— The parallel output (labeled Instruction bit) is updated at the end of the instruction-scan cycle during the *Update-IR* controller state. This must occur on the falling edge of TCK because a change in the latched instruction can result in a change at system output pins due to the operation of the boundary-scan register. Such changes must occur on the falling edge of TCK as defined in Clause 11. Note that in Figure 7-1, an edge-triggered flip-flop is provided adjacent to the shift-register stage to meet this requirement. Alternative implementations, for example, where a level-operated latch is used or the storage element follows (rather than precedes) the instruction decoding logic (see Figure 7-2), are permissible.

— The parallel output is reset in the *Test-Logic-Reset* controller state as a result of a logic 0 received at the Reset* input of the cell. Referring to Figure 6-5 and Figure 6-6, notice that a low Reset* signal will be generated on the falling edge of TCK after entry into the *Test-Logic-Reset* controller state under control of TMS and TCK (TRST* held at 1). The parallel output of the instruction register will change on the falling edge of TCK, as is the case in the *Update-IR* controller state. In contrast, when a logic 0 is applied to TRST*, Reset* consequently is asserted (low) (see Figure 6-5) and the change at the parallel output occurs immediately, irrespective of the state of TMS or TCK. Note that some cells will need to be designed such that the parallel output is set high during this controller state so that the value of the *IDCODE* (or *BYPASS*) instruction is loaded onto the complete register's outputs as required by rule e) of 7.2.1.

— Application of a 0 at TRST* causes the parallel output to be asynchronously set low. Again, some cells may need to be designed to be set high by TRST* such that the value of the *IDCODE* (or *BYPASS*) instruction is forced onto the register's outputs.

Note that the parallel data inputs to the two least significant stages (instruction register stages 0 and 1) must be tied to fixed logic levels (1 for the least significant bit, 0 for the next-least significant bit).

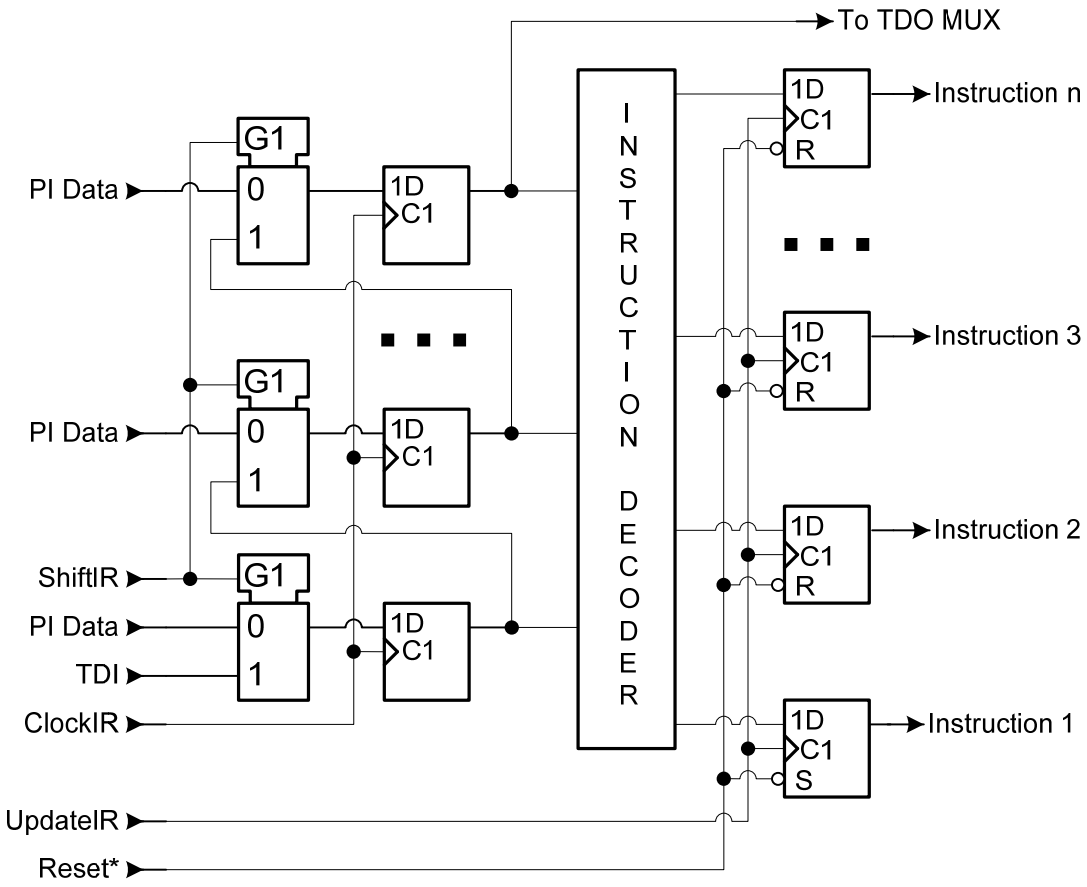


Figure 7-2—Instruction register with decoder between shift and update stages

Figure 7-2 illustrates a common implementation of the instruction register and its decoding. The advantage is that the instruction decodes are now updated by a clock and are therefore glitch-free. The number of update flops or latches is now the same as the number of instructions, not the number of instruction register bits. Note that "Instruction 1" is set "on" by the "Reset*" signal; this would be the mandatory *BYPASS* or *IDCODE* instruction decode.

8. Instructions

The instruction register allows instructions to be entered serially into the test logic during an instruction-register scan cycle. This clause defines the minimum range of instructions that must be supplied and the operations that occur in response to those instructions. Optional instructions and the resulting operation of the test logic are also defined, together with the requirements for extensions to the instruction set defined in this standard.

8.1 Response of the test logic to instructions

8.1.1 Specifications

Rules

a) Each instruction shall completely define the set of test data register(s) that operate or (where required) interact with the on-chip system logic while the instruction is active.
b) Test data registers that are not part of the set that may operate as defined by the active instruction shall be controlled such that they do not interfere with the operation of the on-chip system logic or the defined set of test data registers.
c) Each instruction shall select a single serial test data register path to be enabled to shift data between TDI and TDO in the *Shift-DR* controller state (as defined in Table 6-2).
d) Instruction binary codes that are not otherwise required to provide control of test logic shall be equivalent to the *BYPASS* instruction (see 8.4).
e) All test logic responses to an active instruction shall terminate when a different instruction is transferred to the parallel output of the instruction register (i.e., in the *Update-IR* or *Test-Logic-Reset* controller states) unless the new instruction supports the same test logic response.

Recommendations

f) Use of the binary code {000…0} should be avoided for instructions that disrupt normal (i.e., nontest) operation of the component.

NOTE—Earlier editions of this standard mandated that a binary code for the *EXTEST* instruction be {000...0} (i.e., a logic 0 is loaded into every instruction register cell). While use of this binary code for *EXTEST* or other test mode instruction is neither mandated nor prohibited, it could create problems for a system. A stuck-at-zero fault condition at a component's TDI pin could result in unexpected selection of *EXTEST* and consequent removal of the component from normal service.

Permissions

g) The mode of operation of a test data register may be defined by a combination of the active instruction and further control information contained in test data registers.
h) Two or more instructions may share a single binary code provided that all of the rules for the separate instructions are met.

8.1.2 Description

An instruction is considered to be active from the time it is transferred to the parallel outputs of the instruction register until a different instruction is transferred to the parallel outputs of the instruction register. Such a transfer may take place in either the *Update-IR* or the *Test-Logic-Reset* TAP controller states. The active instruction is decoded in order to achieve two key functions.

First, an instruction selects the serial test data register path that is used to shift data between TDI and TDO during data register scanning. Note that a particular instruction may result in a single test data register being connected

between TDI and TDO or in several test data registers being serially interconnected between TDI and TDO, although in that case the set of serially concatenated test data registers must be given a unique name (for an example, see 9.2).

Second, each instruction defines the set of test data registers that may be used while the instruction is active. Other test data registers should be controlled such that they cannot interfere with the operation of the on-chip system logic or other test data registers. Several registers may be set into test modes simultaneously (for an example, see 9.2).

Rule d) of 8.1.1 requires that every pattern of 1s and 0s that can be shifted into the instruction register produces a defined response and, in particular, that a test data register is connected between TDI and TDO for every possible instruction binary code.

Rule e) and permission g) of 8.1.1 require that the operation of the test logic is determined only by the controller states, the current instruction, and data currently in operating test data registers. The intent is that there is no possibility that a sequential process resulting from any prior instruction (e.g., execution of an internal self-test) might interfere with subsequent instructions once the prior instruction is removed. The circuit under test may not be in a known state if a new instruction is loaded before the previous one has run to completion.

Note that there may be more than one instruction that assumes a specific test logic response, and these may succeed each other without causing the response to terminate. A simple example would be instructions that cause the I/O to be controlled by the boundary-scan register (*EXTEST, CLAMP*, etc.) These may succeed each other without any visible effect on the I/O. In the same way, a sequential test process may be initiated by one instruction and monitored by another without terminating the process when the monitoring instruction is made active.

Rule e) of 8.1.1 also implies that there is no memory in the instruction decoder, that the decoding of the instruction register is strictly static (combinatorial).

Permission h) of 8.1.1 allows for the merging of instructions to operate under a single binary code when their respective behaviors are not mutually exclusive. A prime example of such merging would be a sharing of a single binary code between *SAMPLE* and *PRELOAD*. The resulting merged behavior, which could be called *SAMPLE/PRELOAD*, would be fully equivalent to that mandated in earlier editions of this standard.

8.2 Public instructions

8.2.1 Specifications

Rules

a) Public instructions shall be available for use by purchasers of a component.
b) The following public instructions shall be provided in all components claiming conformance to this standard: *BYPASS*, *SAMPLE*, *PRELOAD*, and *EXTEST* (see 8.4 through 8.8, respectively).
c) If the optional device identification register is included in a component, the *IDCODE* instruction shall be provided.
d) If the optional device identification register is included in a user-programmable component that does not allow the programming via the test logic defined by this standard, then the *USERCODE* instruction shall be provided.
e) The binary codes for the *BYPASS* instruction shall be as defined in 8.4.

Recommendations

f) The following instructions should be supported:
    1) IDCODE
    2) CLAMP and HIGHZ

3) IC_RESET
4) CLAMP_HOLD, CLAMP_RELEASE, and TMP_STATUS

g) If a component has programmable I/O, it should additionally support *INIT_SETUP, INIT_SETUP_CLAMP,* and *INIT_RUN,* as needed.

Permissions

h) A design may offer public instructions in addition to those defined in this standard to give the device purchaser access to design-specific features.

i) Where binary codes for public instructions are not defined by this standard, they may be assigned as required for the particular design.

8.2.2 Description

The public instructions provide the component purchaser with access to test features (e.g., go/no-go self-test testing of the component) and to board interconnect test via the boundary-scan register. The purchaser expects that the results of such tests will be independent of the variant of the component installed in a particular board, of the source of the component, etc. An exception, of course, is when the test results are intended to distinguish the variant, etc., as would be the case if the *IDCODE* instruction were used (see 8.13).

In addition to the mandatory instructions, which may be sufficient for a very simple component, a number of instructions are recommended. In general, these instructions allow more sophisticated control of the component during board and system tests, and support reuse of internal tests that are controlled or at least initialized from the test logic in board and system test environments.

The *IDCODE* instruction is mandatory if the device identification register is included, but both are recommended. This register allows test software to identify variants of the component and adjust the tests accordingly. (See 8.13.)

CLAMP and *HIGHZ* are recommended to give the test engineers the ability to isolate components on the board or system being tested, and to give the test equipment control of board nets without the risk of damage inherent in overdriving active signals. (See 8.11 and 8.16.)

IC_RESET instruction is mandatory if the reset selection register is included, and is recommended to give the tester programmable control, possibly with greater resolution than supported by the component reset inputs, of the resets and related signals such as the power controls in the component. The reset nets on a board may not be accessible by the test equipment in a board or system environment. (See 8.21.)

The *CLAMP_HOLD*, *CLAMP_RELEASE*, and *TMP_STATUS* instructions are mandatory if the TMP controller and TMP status register are included, and are recommended to provide an additional and highly valuable means for test engineers to hold a component in the test mode of operation, to help avoid spurious system operation in the test environment, and to isolate the component during board and system tests. (See 6.2 and 8.20.)

The *INIT_SETUP* and *INIT_SETUP_CLAMP* instructions are mandatory if the initialization data register is provided, and the *INIT_RUN* instruction is mandatory if the initialization status register is provided. All are recommended, as appropriate for the specific design, to prepare a component for board test where a simple power-up reset is inadequate. They allow initializing programmable I/O so that they are configured correctly for interconnect test at the board level, and other initializations (such as shutting down clock distribution or bypassing a Phase-Locked Loop) to put the component system logic into a safe state for board test, as needed. (See 8.17.)

The binary code of an instruction is the sequence of data bits shifted serially into the instruction register from TDI during the *Shift-IR* controller state.

8.3 Private instructions

8.3.1 Specifications

Rules

a) If private instructions are utilized in a component, the vendor shall clearly identify any instruction binary codes that, if selected, could cause damaging operation of the component by designating the instruction as private in the BSDL.

Permissions

b) The public instructions may be supplemented with private instructions intended solely for the use of the component manufacturer.

c) The operation of private instructions may be undocumented.

8.3.2 Description

Private instructions allow the component manufacturer to use the TAP and test logic to gain access to test features embedded in the design for design verification, production testing, or fault diagnosis, which are not intended for use once the component is placed on a board or in a system. The component manufacturer may require tests performed using these features to give results that differ between variants of the component, for example, that would render documentation and use by component purchasers difficult.

Also, some instructions may even cause a component to operate in a manner that could be potentially destructive of this or other components on a board or in a system. For example, if an instruction causes component inputs to become outputs for test data, etc., then damage may result if the instruction is selected while the component is surrounded by other components on an assembled board. The vendor must therefore clearly identify as private any instruction binary codes that may cause dangerous operation if used by the component purchaser.

8.4 *BYPASS* instruction

The bypass register contains a single shift-register stage and is used to provide a minimum-length serial path between the TDI and the TDO pins of a component when no test operation of that component is required. This allows more rapid movement of test data to and from other components on a board that are required to perform test operations.

8.4.1 Specifications

Rules

a) Each component shall provide a *BYPASS* instruction.

b) A binary code for the *BYPASS* instruction shall be {111...1} (i.e., a logic 1 entered into every instruction register cell).

c) The *BYPASS* instruction shall select the bypass register to be connected for serial access between TDI and TDO in the *Shift-DR* controller state.

d) When the *BYPASS* instruction is selected, all test data registers that can operate in either system or test modes shall perform their system function.

e) When the *BYPASS* instruction is selected, then:

   1) If the TMP controller is either not provided or in the *Persistence-Off* state, the operation of the test logic other than the optional reset selection register and its associated logic (shown in Clause 17) shall have no effect on the operation of the on-chip system logic or on the flow of signals between the system pins and the on-chip system logic.

2) If the TMP controller is provided and in the *Persistence-On* state, then the state of all signals driven from system output pins shall be completely defined by the data held in the boundary-scan register.

Permissions

f) The *BYPASS* instruction may have binary codes in addition to that defined in rule b) of 8.4.1.

Recommendations

g) A binary code for the *BYPASS* instruction should be {000...0} (i.e., a logic 0 entered into every instruction register cell).

8.4.2 Description

The *BYPASS* instruction can be entered by holding TDI at a constant high value and completing an instruction-scan cycle. The demands on the host test system consequently are reduced in cases where access is required, say, to only component number 57 on a 100 component board. In this case, the overall instruction pattern that shall be shifted into the design consists of a background of 1s with a small field of specific instruction data.

Note also that, since the TDI input is designed such that when it is not terminated it behaves as though a high signal were being applied, an open-circuit fault in the serial board-level test data path will cause the bypass register to be selected after an instruction-scan cycle. Therefore, no unwanted interference with the operation of the on-chip system logic can occur.

Where no device identification register is provided in a component, then the *BYPASS* instruction is forced into the latches at the parallel outputs of the instruction register during the *Test-Logic-Reset* controller state. This means that after a reset, a complete serial path through either the bypass or the device identification register is established.

A consideration for usage is that if *BYPASS* is operated in some components while test mode instructions (e.g., *EXTEST*) are operated in others, the normal system-logic operation of those components that are operating *BYPASS* may conflict with the test operation of the others. Therefore, careful analysis of interactions is necessary.

A further consideration is that if the TMP controller is provided, scanning in any instruction value that decodes to the *BYPASS* instruction will cause the TMP controller to revert to the *Persistence-Off* state. Use of the *CLAMP* instruction instead when the TMP controller is in the *Persistence-On* state will avoid this.

8.5 Boundary-scan register instructions

As discussed in Clause 1, the boundary-scan register is composed of cells connected between the on-chip system logic and the component's system input and output pins. This subclause is included to provide an overview of the structure and operation of the boundary-scan register that will assist the reader in understanding the specifications for the mandatory and optional instructions that make use of the boundary-scan register.

Design requirements for the boundary-scan register instructions are contained in 8.6 to 8.11. General requirements for the design of TDRs are contained in Clause 9, and additional specific requirements for the design of the boundary-scan register and boundary-scan register cells are contained in Clause 11.

8.5.1 Overview of the operation of the boundary-scan register

The boundary-scan register is a shift-register-based structure that includes a variety of different cell designs matched onto the requirements of the particular component. Different cell designs are used according to the type of system pin concerned (input, output, three-state, bidirectional) and according to the set of boundary-scan instructions supported.

A simplified view of a boundary-scan register is shown in Figure 8-1.

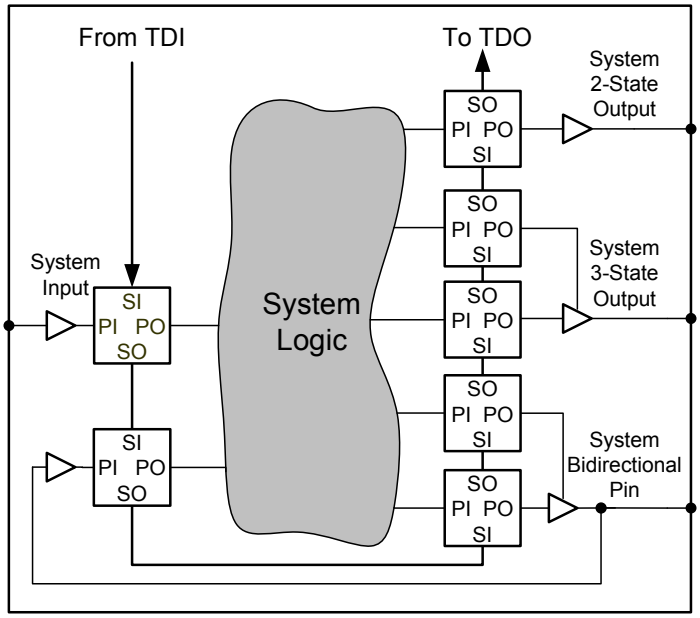


Figure 8-1—Simplified view of the boundary-scan register

An example implementation for a cell that could be used in each location shown in Figure 8-1 is given in Figure 8-2.

The connections labeled PI, PO, SI, and SO in Figure 8-2 are connected to adjacent cells, the on-chip system logic, and the system pins as shown in Figure 8-1. Like all the cells shown in this standard, the cell shown in Figure 8-2 is designed to respond to the *Clock-DR*, *Shift-DR*, and *Update-DR* signals generated by the example TAP controller implementation shown in Figure 6-5 and Figure 6-6. The Mode input shall be controlled according to the type of pin connected to the cell (input, output, etc.) and the specific instruction selected.

Use of this cell design, with appropriate signals supplied to the Mode input of each cell, will result in a component that supports the *SAMPLE*, *PRELOAD*, *EXTEST*, and *INTEST* instructions. As will be discussed in Clause 11, other cell designs are possible that meet the requirements of this standard for different sets of instructions. For example, in Figure 8-2:

— R2 may be either a flip-flop (as shown) or a latch.
— R2 is optional for cells that feed data from a system pin to the on-chip system logic, e.g., the cells at system input pins. The lower input to M2 would, in such cases, be fed directly from the output of R1.
— If the *INTEST* instruction were not supported, R2 and M2 could be omitted from cells that feed data from a system pin to the on-chip system logic. The input labeled PI then would be connected directly to the output labeled PO.

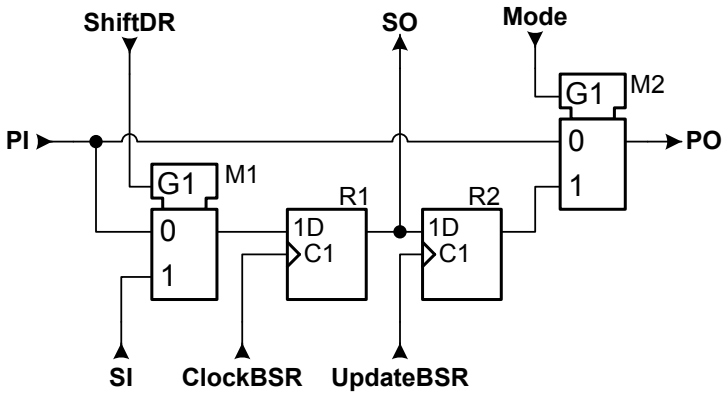


Figure 8-2—Example boundary-scan register cell design

8.5.2 Specifications for boundary-scan register instructions

The specifications for boundary-scan instructions given in the following subclauses of this clause define:

a) Whether the instruction is mandatory or optional.
b) Which test data registers can be connected in the serial path between TDI and TDO.
c) The restrictions (if any) on the choice of binary codes for each instruction (i.e., the patterns of 1s and 0s that, when shifted into the instruction register, cause the instruction to be selected).
d) The flow of data among the component's system pins, the boundary-scan register cells, and the on-chip system logic.

The specifications are supported by descriptive text that includes a version of Figure 8-3 that shows one input and one output for a component. The solid bold lines in later copies of this figure show the mandatory data flows for each instruction.

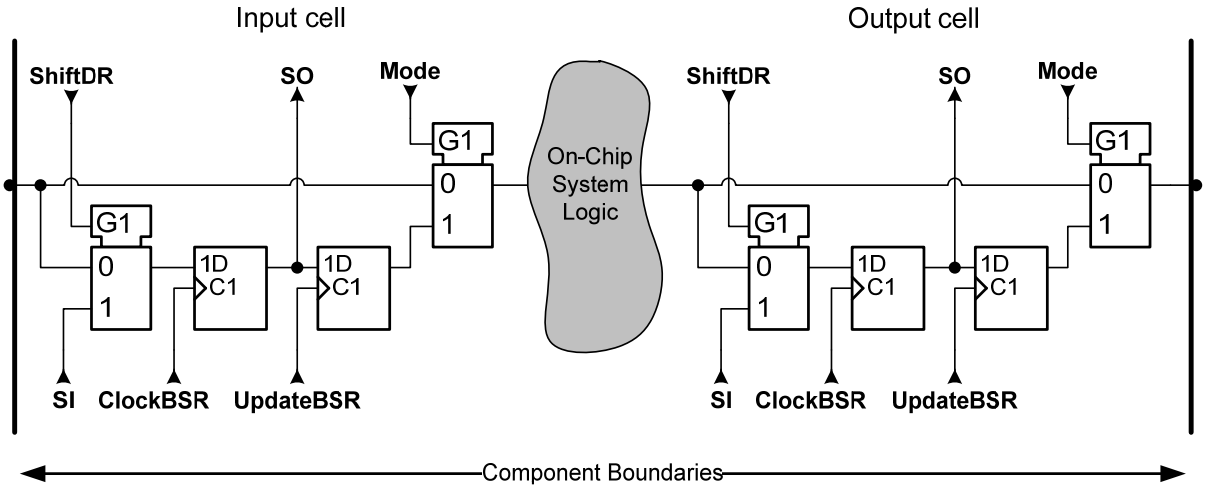


Figure 8-3—Figure used to illustrate boundary-scan instructions

8.6 *SAMPLE* instruction

The mandatory *SAMPLE* instruction allows a snapshot of the normal operation of the component to be taken and examined.

8.6.1 Specifications

Rules

a) Each component shall provide a *SAMPLE* instruction.

b) The *SAMPLE* instruction shall select *only* the boundary-scan register to be connected for serial access between TDI and TDO in the *Shift-DR* controller state (i.e., no other test data register may be connected in series with the boundary-scan register).

c) When the *SAMPLE* instruction is selected, then:

   1) If the TMP controller is either not provided or in the *Persistence-Off* state, the operation of the test logic other than the optional reset selection register and its associated logic (shown in Clause 17) shall have no effect on the operation of the on-chip system logic or on the flow of signals between the system pins and the on-chip system logic.

   2) If the TMP controller is provided and in the *Persistence-On* state, then the state of all signals driven from system output pins shall be completely defined by the data held in the boundary-scan register.

d) When the *SAMPLE* instruction is selected, the states of all signals flowing from the on-chip system logic or through system pins (input or output) shall be loaded into the boundary-scan register on the rising edge of TCK in the *Capture-DR* controller state.

NOTE—The intent of this rule is to specify when the loading action should occur. Detailed specifications for the choices of signal values to be loaded are provided in rule f) of 11.5.1 and rule h) of 11.6.1, respectively, for system logic inputs and system logic outputs.

Recommendations

e) Where each of *SAMPLE* and *PRELOAD* implements the functionality of the other, they should share a common binary value(s).

Permissions

f) When the *SAMPLE* instruction is selected, parallel output registers/latches included in boundary-scan register cells may load the data held in the associated shift-register stage on the falling edge of TCK in the *Update-DR* controller state.

g) The binary value(s) for the *SAMPLE* instruction may be selected by the component designer.

8.6.2 Description

The *SAMPLE* instruction allows a snapshot to be taken of the states of the component's input and output signals without interfering with the normal operation of the assembled board. The snapshot is taken on the rising edge of TCK in the *Capture-DR* controller state, and the data can then be viewed by shifting through the component's TDO output. Note that, depending on the configuration of components on a board, the *SAMPLE* instruction may not capture correct data prior to the completion of any initialization using the optional initialization instructions.

Example applications of the *SAMPLE* capability are:

— To provide an analog to the guided-probing process performed on an assembled board during functional test diagnosis, but without the need for physical contact.

— Verification of the interaction between components during normal functional operation.

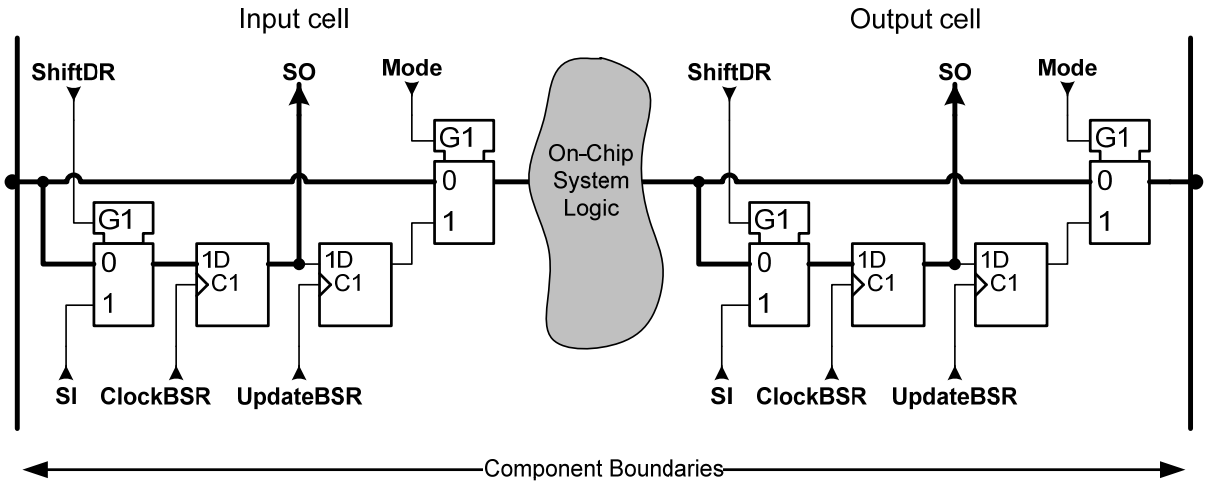


Figure 8-4—Data flow for the *SAMPLE* instruction

The flow of data for the *SAMPLE* instruction is shown in Figure 8-4. As can be seen, *SAMPLE* can be used without causing interference to the normal operation of the on-chip system logic. Data received at system input pins is supplied without modification to the on-chip system logic; data from the on-chip system logic is driven without modification through the system output pins; etc. For the example boundary-scan register cell design given in Figure 8-2, this is achieved by holding the Mode input at 0 when the *SAMPLE* instruction is selected.

NOTE 1— At output pins, the signal samples may be either that output from the component or that output from the on-chip system logic.

NOTE 2—A component may be designed such that the *SAMPLE* and *PRELOAD* instructions are combined by assigning the same binary code to both. While *SAMPLE* captures data into the boundary-scan register and allows this to be shifted out through TDO for examination, a specific use of data shifted in at TDI is not mandated. In contrast, *PRELOAD* shifts data into the boundary-scan register through TDI such that it can be loaded into the register's parallel output registers/latches in advance of selecting an instruction (such as *EXTEST*) that supplies the data held in these registers/latches to the component's output pins. The data captured into the boundary-scan register before shifting is not defined. The mutual exclusivity of these behaviors permits the instructions to be merged where desired [see permission h) of 8.1.1, permission f) of 8.6.1, and permission f) of 8.7.1]. Furthermore, where *SAMPLE* and *PRELOAD* instructions are merged in this fashion, by moving the TAP controller through the state sequence *Capture-DR* → *Exit1-DR* → *Update-DR* while the merged *SAMPLE/PRELOAD* instruction is selected, the state of the signals flowing into and out of the on-chip system logic at the time of sampling can be loaded onto the latched parallel output of the boundary-scan shift register.

8.7 *PRELOAD* instruction

The mandatory *PRELOAD* instruction allows data values to be loaded onto the latched parallel outputs of the boundary-scan shift register before selection of the other boundary-scan test instructions.

8.7.1 Specifications

Rules

a) Each component shall provide a *PRELOAD* instruction.
b) The *PRELOAD* instruction shall select *only* the boundary-scan register to be connected for serial access between TDI and TDO in the *Shift-DR* controller state (i.e., no other test data register may be connected in series with the boundary-scan register).
c) When the *PRELOAD* instruction is selected, then:

1) If the TMP controller is either not provided or in the *Persistence-Off* state, the operation of the test logic other than the optional reset selection register and its associated logic (shown in Clause 17) shall have no effect on the operation of the on-chip system logic or on the flow of signals between the system pins and the on-chip system logic.
2) If the TMP controller is provided and in the *Persistence-On* state, then the state of all signals driven from system output pins shall be completely defined by the data held in the boundary-scan register.

d) When the *PRELOAD* instruction is selected, parallel output registers/latches included in boundary-scan register cells shall load the data held in the associated shift-register stage on the falling edge of TCK in the *Update-DR* controller state.

Recommendations

e) Where *SAMPLE* and *PRELOAD* each implement the functionality of the other, they should share a common binary value(s).

Permissions

f) When the *PRELOAD* instruction is selected, the states of all signals flowing through system pins (input or output) may be loaded into the boundary-scan register on the rising edge of TCK in the *Capture-DR* controller state.

g) The binary value(s) for the *PRELOAD* instruction may be selected by the component designer.

8.7.2 Description

The *PRELOAD* instruction allows scanning of the boundary-scan register without causing interference to the normal operation of the on-chip system logic. It thus allows an initial data pattern to be placed at the latched parallel outputs of boundary-scan register cells (e.g., as provided in the cells connected to system output pins) before the selection of another boundary-scan test operation. For example, before the selection of the *EXTEST* instruction, data can be loaded onto the latched parallel outputs using *PRELOAD*. As soon as the *EXTEST* instruction has been transferred to the parallel output of the instruction register, the preloaded data are driven through the system output pins. Known data, consistent at the board level, is thereby driven immediately when the *EXTEST* instruction is entered; without performing a *PRELOAD* instruction first, indeterminate data would be driven from the system output pins until the first scan sequence had been completed.

The flow of data for the *PRELOAD* instruction is shown in Figure 8-5. Data received at system input pins are supplied without modification to the on-chip system logic; data from the on-chip system logic is driven without modification through the system output pins; etc. For the example boundary-scan register cell design given in Figure 8-2, this is achieved by holding the Mode input at 0 when the *PRELOAD* instruction is selected.

NOTE—A component may be designed such that the *SAMPLE* and *PRELOAD* instructions are combined by assigning the same binary code to both. While *SAMPLE* captures data into the boundary-scan register and allows this to be shifted out through TDO for examination, a specific use of data shifted in at TDI is not mandated. In contrast, *PRELOAD* shifts data into the boundary-scan register through TDI such that it can be loaded into the register's parallel output registers/latches in advance of selecting an instruction (such as *EXTEST*) that supplies the data held in these registers/latches to the component's output pins. The data captured into the boundary-scan register before shifting is not defined. The mutual exclusivity of these behaviors permits the instructions to be merged where desired [see permission h) of 8.1.1, permission f) of 8.6.1, and permission f) of 8.7.1]. Furthermore, where *SAMPLE* and *PRELOAD* instructions are merged in this fashion, by moving the TAP controller through the state sequence *Capture-DR* → *Exit1-DR* → *Update-DR* while the merged *SAMPLE/PRELOAD* instruction is selected, the state of the signals flowing into and out of the on-chip system logic at the time of sampling can be loaded onto the latched parallel output of the boundary-scan shift register.

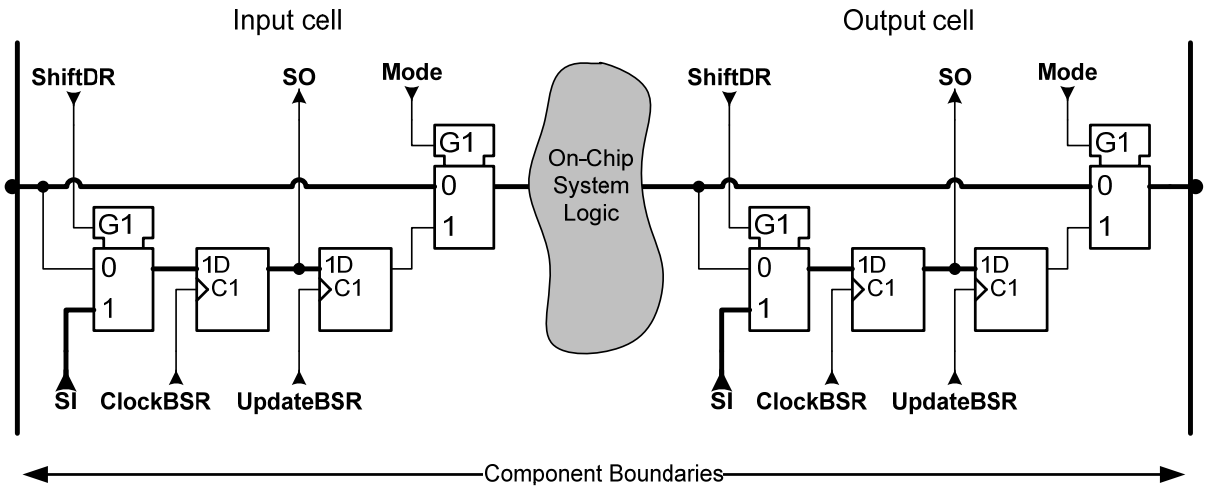


Figure 8-5—Data flow for the *PRELOAD* instruction

8.8 *EXTEST* instruction

The mandatory *EXTEST* instruction allows testing of off-chip circuitry and board-level interconnections. Data typically would be loaded onto the latched parallel outputs of boundary-scan shift-register stages using the *PRELOAD* instruction before selection of the *EXTEST* instruction.

NOTE—After use of the EXTEST instruction, the on-chip system logic may be in an indeterminate state that will persist until a system reset is applied. Therefore, the on-chip system logic may need to be reset on return to normal (i.e., nontest) operation.

8.8.1 Specifications

Rules

a) Each component shall provide an *EXTEST* instruction.

b) The *EXTEST* instruction shall select *only* the boundary-scan register to be connected for serial access between TDI and TDO in the *Shift-DR* controller state (i.e., no other test data register may be connected in series with the boundary-scan register).

c) While the *EXTEST* instruction is selected, the on-chip system logic shall be controlled such that it cannot be damaged as a result of signals received at the system input or system clock input pins.

   NOTE 1—This might be achieved by placing the on-chip system logic in a reset or "hold" state while the *EXTEST* instruction is selected.

d) When the *EXTEST* instruction is selected, the state of all signals driven from system output pins controlled by the boundary-scan register or included boundary-scan register segments:
   1) Shall be defined by the data held in the boundary-scan register
   2) Shall change only on the falling edge of TCK in the *Update-DR* controller state
   3) For system output pins controlled by excluded boundary-scan register segments, shall be defined by the system logic

e) When the *EXTEST* instruction is selected, the state of all signals received at system input pins shall be loaded into the boundary-scan register on the rising edge of TCK in the *Capture-DR* controller state.

Recommendations

f) The data loaded into boundary-scan register cells located at system output pins (two-state, three-state, or bidirectional) in the *Capture-DR* controller state when the *EXTEST* instruction is selected should be independent of the operation of the on-chip system logic.

g) A value should be defined for each boundary-scan register cell that, when the *EXTEST* instruction is selected, will permit all component outputs to be overdriven simultaneously for an indefinite period without risk of damage to the component.

NOTE 2—This is easily achieved if all outputs can be set to an inactive drive state by previous use of the *PRELOAD* instruction.

Permissions

h) The binary value(s) for the *EXTEST* instruction may be selected by the component designer.

8.8.2 Description

The *EXTEST* instruction allows circuitry external to the component package—typically the board interconnect—to be tested. Boundary-scan register cells at output pins are used to apply test stimuli, while those at input pins capture test results. Note that, depending on the configuration of components on a board, the *EXTEST* instruction may not capture correct data prior to the completion of any initialization using the optional initialization instructions.

This instruction also allows testing of blocks of components that do not themselves incorporate boundary-scan registers. The flow of data through the boundary-scan register cells in this configuration is shown in Figure 8-6. For example, at input pins, data are first captured into the shift-register path and then shifted out of the component for examination; at output pins, data shifted into the component are applied to the external interconnection.

Typically, the first test stimulus to be applied using the *EXTEST* instruction will be shifted into the boundary-scan register using the *PRELOAD* instruction. Thus, when the change to the *EXTEST* instruction takes place in the *Update-IR* controller state, known data will be driven immediately from the component onto its external connections. The stimuli for the next test will be shifted in while the results from the current test are shifted out. That is, the two shift operations are overlapped. Note that while the results from the final test are shifted out, a determinate set of data must be shifted in that will leave the board in a consistent state at the end of the shifting process. This can be achieved by again shifting the stimuli for test *N* (or indeed any other test) into the boundary-scan register, or by shifting in a “safe” state.

The *EXTEST* instruction also allows component outputs to be set to a state that minimizes the risk of damage when overdriven during in-circuit testing [see recommendation g) in 8.8.1]. Such testing may be used where not all components on an assembled board are testable via boundary scan.

Note that the boundary-scan register cells located at input pins may optionally be designed to allow signals to be driven into the on-chip system logic when the *EXTEST* instruction is selected. This allows the component designer to define values to be established at the system logic inputs, preventing incorrect operation in response to noise signals arriving from the board-level interconnect. The values driven may be constant for the duration that *EXTEST* is selected (e.g., by including a blocking gate at the input to the system logic) or may be loaded serially through the boundary-scan register, as shown in Figure 8-6.

Recommendation f) of 8.8.1, where followed, helps ensure that data shifted out of the component in response to the *EXTEST* instruction is not altered by the presence of faults in the on-chip system logic. This simplifies diagnosis since any errors in the output bit stream will most likely be caused only by faults in off-chip circuitry, in board-level interconnections, or in the boundary-scan registers used to apply the test.

While the *EXTEST* instruction is selected, the on-chip system logic may receive input signals that differ significantly from those expected during normal (nontest) operation. Rule c) of 8.8.1 places the responsibility for correct handling

of this situation on the component designer. If the on-chip system logic can tolerate any permutation of input signals that is received, then no specific design changes are required to meet this rule. (An example here would be the case where the on-chip system logic is entirely combinational.) However, for some components, there may be input sequences that could place the on-chip system logic in a state where damage may result. In these cases, it is the responsibility of the designer to prevent the on-chip system logic from processing these input sequences while the *EXTEST* instruction is selected. As noted, this may be achieved by placing the on-chip system logic into a reset or "hold" state.

Alternatively, the data held in the boundary-scan register may be presented to the on-chip system logic while the *EXTEST* instruction is selected. Note that where this is the case, rule e) of 11.3.1 prohibits the imposition of any restriction on the logic values that may be driven to the on-chip system logic.

Note that while earlier editions of this standard mandated that a binary code for *EXTEST* be {000…0}, the use of this binary code for *EXTEST* and all other test mode instructions has been deprecated [see recommendation f) of 8.1.1 and the associated NOTE].

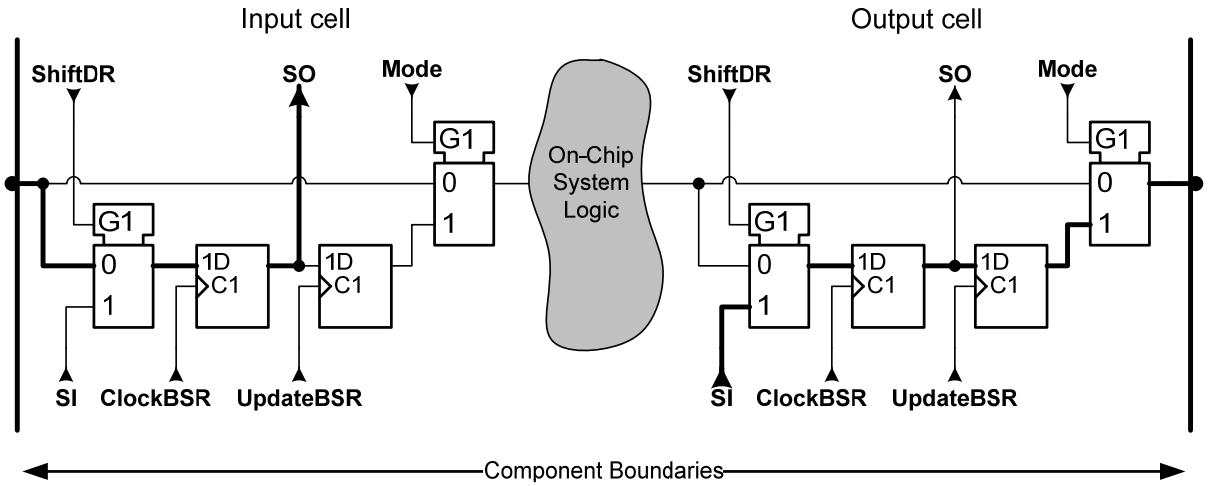


Figure 8-6—Data flow for the *EXTEST* instruction

8.9 *INTEST* instruction

The optional *INTEST* instruction is one of two instructions (the other is *RUNBIST*) defined by this standard that allow testing of the on-chip system logic while the component is assembled on the board. Using the *INTEST* instruction, test stimuli are shifted into the boundary-scan register one at a time and applied to the on-chip system logic. The test results are captured into the boundary-scan register and are examined by subsequent shifting. Data typically would be loaded onto the latched parallel outputs of boundary-scan shift-register stages using the *PRELOAD* instruction before selection of the *INTEST* instruction.

The following rules apply where the *INTEST* instruction is provided.

NOTE—After use of the *INTEST* instruction, the on-chip system logic may be in an indeterminate state that will persist until a system reset is applied. Therefore, the on-chip system logic may need to be reset on return to normal (i.e., nontest) operation.

8.9.1 Specifications

Rules

a) The *INTEST* instruction shall select only the boundary-scan register to be connected for serial access between TDI and TDO in the *Shift-DR* controller state (i.e., no other test data register may be connected in series with the boundary-scan register).

b) The on-chip system logic shall be capable of single-step operation while the *INTEST* instruction is selected.

c) When the *INTEST* instruction is selected, all system outputs from the component shall be defined as follows:

   1) The state of all signals driven from system output pins controlled by the boundary-scan register or included boundary-scan register segments shall be defined by the data held in the boundary-scan register and shall change only on the falling edge of TCK in the *Update-DR* controller state, and system output pins controlled by excluded boundary-scan register segments shall be defined by the system logic.

   2) All outputs from the component (including those that are two-state outputs) except any outputs that are powered-down shall be placed in an inactive drive state (e.g., high-impedance) on selection of the *INTEST* instruction.

d) When the *INTEST* instruction is selected, the state of all nonclock signals driven into the system logic from the boundary-scan register shall be completely defined by the data held in the register.

e) When the *INTEST* instruction is selected, the state of all signals output from the system logic to the boundary-scan register shall be loaded into the register on the rising edge of TCK in the *Capture-DR* controller state.

Recommendations

f) For boundary-scan register cells located at system input pins (clock or nonclock) or at bidirectional pins configured as inputs, the data loaded in the *Capture-DR* controller state when the *INTEST* instruction is selected should be independent of the operation of off-chip circuitry or board-level interconnections.

g) The number of system clocks used for the *INTEST* instruction should be minimized, or where practical, TCK should be used in lieu of any system clocks.

Permissions

h) The binary value(s) for the *INTEST* instruction may be selected by the component designer.

8.9.2 Description

The *INTEST* instruction allows static (slow-speed) testing of the on-chip system logic, with each test pattern and response being shifted through the boundary-scan register. The *INTEST* instruction requires that the on-chip system logic can be operated in a single-step mode, where the circuitry moves one step forward in its operation each time shifting of the boundary-scan register is completed.

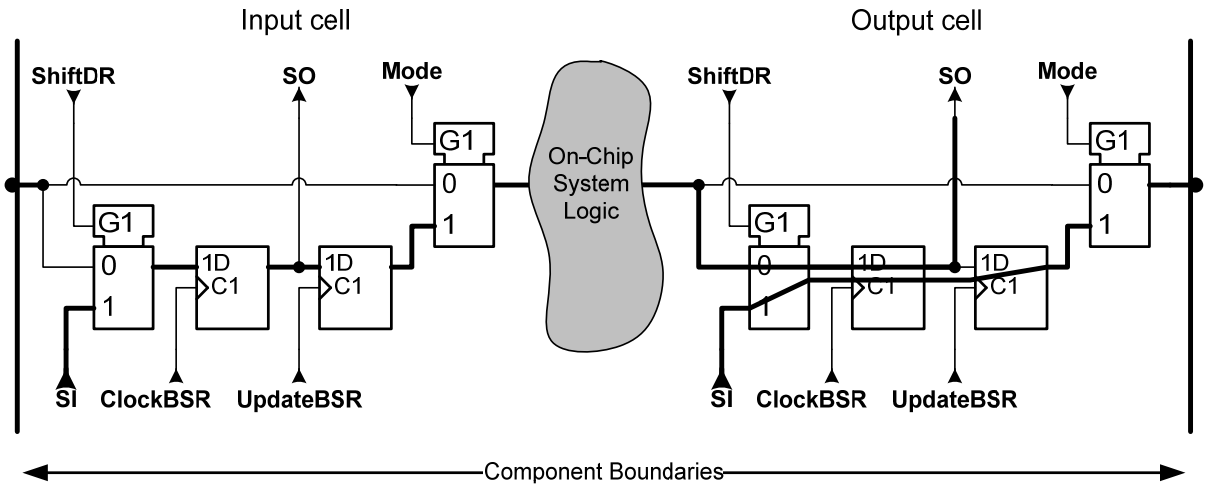


Figure 8-7—Data flow for the *INTEST* instruction

The flow of data through the boundary-scan register cells while the instruction is selected is shown by the bold paths in Figure 8-7. The topmost bold path through the cell at the output pin is that taken by the results of the test of the on-chip system logic; the lowermost path is that taken by the data to be held at the pin while the test is applied. Note that, for each test, the latched parallel output of the boundary-scan register cell at the system output pin is updated from data shifted in *before* the state of the shift-register is overwritten with the test response.

While the *INTEST* instruction is selected, the boundary-scan register assumes the role of the ATE system used for stand-alone component testing. Cells at nonclock system input pins are used to apply the test stimulus, while those at system output pins capture the response. Stimuli and responses are moved into and out of the circuit by shifting the boundary-scan register. Note that this requires that the boundary-scan register cells located at system input pins are able to drive signals into the on-chip system logic.

Typically, the on-chip system logic will receive a sequence of clock events between application of the stimulus and capture of the response such that single-step operation is achieved. The specification of boundary-scan register cells for system clock input pins allows the clocks for the on-chip system logic to be obtained in several ways while the *INTEST* instruction is selected. The following are offered as examples:

a) The signals received at system clock pins can be fed directly to the on-chip system logic as during normal operation of the component. Where this option is selected, the component shall be designed so that precisely one single step of operation of the on-chip system logic occurs while, at least, a specified minimum number of TCK cycles are applied during the *Run-Test/Idle* controller state. The component shall be designed so that only one single step of operation is performed whether or not more than the specified minimum number of TCK cycles is applied while the TAP controller is in the *Run-Test/Idle* controller state. This may, for example, require that clock signals coming into the component be gated before application to the on-chip system logic. In this way, operation of the on-chip system logic can be inhibited while test data are shifted through the boundary-scan register. Figure 8-8 illustrates how the system clock applied to the component should be controlled during testing of the on-chip system logic using the *INTEST* instruction.

   While Figure 8-8 illustrates a situation in which the system clock is a single positive-going pulse, rule b) of 8.9.1 can be generalized to apply to components that employ multiple clock cycles for each step of operation or that have several clock input pins at which multiphase clock signals are received. Note that, while Figure 8-8 shows entry into the *Run-Test/Idle* controller state from the *Update-DR* controller state, clock pulses also would be applied to the on-chip system logic if the *Run-Test/Idle* controller state were entered from the *Update-IR* controller state.

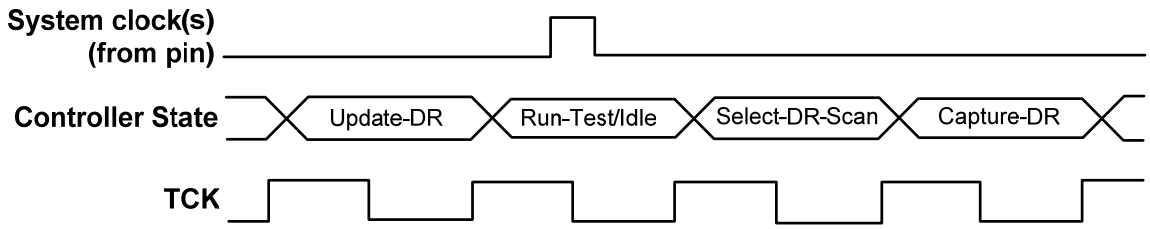


Figure 8-8—Control of applied system clock during *INTEST*

b) The on-chip system logic can be supplied with clock signals derived from TCK in the *Run-Test/Idle* controller state. In all other controller states, the clocks should not change state. Figure 8-9 shows a derived clock signal where the on-chip system logic responds to rising clock edges, for example.

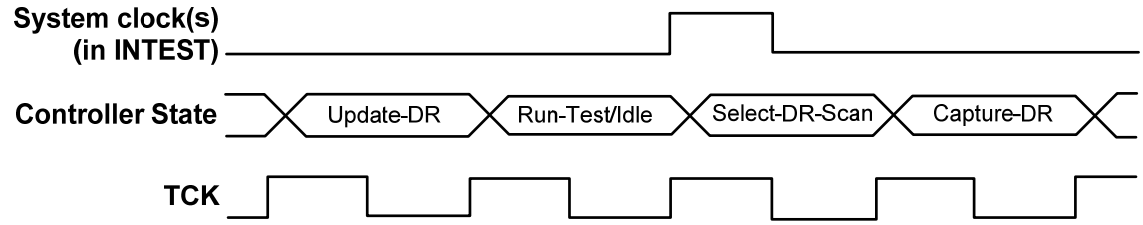


Figure 8-9—Use of TCK as clock for on-chip system logic during *INTEST*

c) Circuitry may be built into the component that, on entry into the *Run-Test/Idle* controller state, allows the on-chip system logic to complete one step of operation. For example, if the component were a microprocessor, it would be permitted to complete a single processing cycle, for example, by internal generation of a pulse on the hold signal. In this case, the clock(s) applied at the system clock pin(s) during the test could be free-running.

d) Clock signals can be shifted in via the boundary-scan path in the same manner in which nonclock signals for the on-chip system logic are supplied. Note that this will require the boundary-scan register to be shifted for each distinct clock signal state (e.g., twice for a single-phase clock).

NOTE—This may be a hazard-prone operation for certain circuit designs.

While the *INTEST* instruction is selected, the state of system output pins is determined by the test logic. There are two options. First, the pin state may be determined by the data held in the boundary-scan register, shifted onto the latched parallel outputs of the register during each pass through the scan sequence for the register. If there are excluded segments, then the state of system output pins controlled by excluded boundary-scan register segments will be defined by the system logic. Second, every system output pin may be forced to an inactive drive state (e.g., high-impedance). If an output is in a powered-down domain, then it remains powered-down. This supplies surrounding components on an assembled board with predictable signal levels while the on-chip system logic test is in progress. Typically, a consistent set of data values would be shifted into the appropriate stages of the boundary-scan register using the *PRELOAD* instruction before the selection of the *INTEST* instruction. This data pattern is then reloaded each time a new *INTEST* test pattern is shifted into the boundary-scan register.

Recommendation f) of 8.9.1, where followed, helps ensure that data shifted out of the component in response to the *INTEST* instruction is not altered by the presence of faults in off-chip system logic, board-level interconnections, and so on. This simplifies diagnosis since any errors in the output bit stream can be caused only by faults in the on-chip system logic or in the boundary-scan register.

8.10 *RUNBIST* instruction

The optional *RUNBIST* instruction causes execution of a self-contained self-test of the component. Use of the instruction allows a component user to determine the health of the component without the need to load complex data patterns and without the need for single-step operation (as required for the *INTEST* instruction). While the *RUNBIST* instruction is selected, the state of all system output pins is determined by the test logic. There are two options. First, the pin state may be determined by the data held in the boundary-scan register, shifted onto the latched parallel outputs of the register during each pass through the scan sequence for the register. Second, every system output pin may be forced to an inactive drive state (e.g., high-impedance).

The following rules apply where the *RUNBIST* instruction is provided.

NOTE—After use of the *RUNBIST* instruction, the on-chip system logic may be in an indeterminate state that will persist until a system reset is applied. Therefore, the on-chip system logic may need to be reset on return to normal (i.e., nontest) operation.

8.10.1 Specifications

Rules

a) When the *RUNBIST* instruction is selected, the test data register into which the results of the self-test(s) will be loaded shall be connected for serial access between TDI and TDO in the *Shift-DR* controller state.

b) Self-test mode(s) of operation accessed through the *RUNBIST* instruction shall execute *only* in the *Run-Test/Idle* controller state.

c) Where a test data register is required to be initialized before execution of the self-test, this shall occur at the start of the self-test without any requirement to shift data into the component (i.e., there shall be no requirement to enter seed values into any test data register).

   NOTE—As per rule k1) in this subclause, the boundary-scan register may (optionally) need to be initialized to define the state of signals driven from system output pins. However, this value should not be used as a seed for the self-test operation since it may be dependent on the board design.

d) A duration shall be specified for the test executed in response to the *RUNBIST* instruction (e.g., a number of rising edges of TCK or the system clock).

e) The result of the self-test(s) executed in response to the *RUNBIST* instruction shall be loaded into the test data register connected between TDI and TDO no later than the rising edge of TCK in the *Capture-DR* controller state.

f) After the specified minimum duration, the test result observed by loading and shifting of the test data register selected by the *RUNBIST* instruction shall be constant regardless of when the *Capture-DR* controller state is entered.

g) Use of the *RUNBIST* instruction shall give the same result in all versions of a component.

h) Data shifted out of a component after completion of execution of a self-test accessed using the *RUNBIST* instruction shall be independent of the operation of off-chip circuitry or board-level interconnections.

i) All stages of the test data register selected by the *RUNBIST* instruction shall be set to determinate logic states (0 or 1) no later than the rising edge of TCK in the *Capture-DR* controller state.

j) The component shall be designed such that results of self-tests executed in response to the *RUNBIST* instruction are not affected by signals received at nonclock system input pins.

k) When the *RUNBIST* instruction is selected, all system outputs from the component shall be defined as follows:

   1) The state of all signals driven from system output pins controlled by the boundary-scan register or included boundary-scan register segments shall be defined by the data held in the boundary-scan register and shall change only on the falling edge of TCK in the *Update-DR* controller state, and system output pins controlled by excluded boundary-scan register segments shall be defined by the system logic.

2) All outputs from the component (including those that are two-state outputs) except any outputs that are powered-down shall be placed in an inactive drive state (e.g., high-impedance) on selection of the *RUNBIST* instruction.

l) The states of the parallel output registers or latches in boundary-scan register cells located at system output pins (two-state, three-state, or bidirectional) shall not change while the *RUNBIST* instruction is selected, unless the associated pin has been placed in an inactive drive state (e.g., high-impedance) as defined in rule k2) in this subclause.

Recommendations

m) Removed for this version of the standard.

Permissions

n) The binary value(s) for the *RUNBIST* instruction may be selected by the component designer.

o) Where a component includes multiple self-test functions, these may be executed either concurrently or in a sequence determined by the component manufacturer in response to the *RUNBIST* instruction. In the latter case, all sequencing should be taken care of within the component itself without requiring the alteration of the instruction register contents.

p) Additional public instructions may be provided to give a component user access to individual self-test functions within a component.

q) The test data register connected between TDI and TDO when the *RUNBIST* instruction is selected may be the boundary-scan register.

r) While the *RUNBIST* instruction is selected, the boundary-scan register may act as a pattern generator or signature compactor in the *Run-Test/Idle* controller state provided rule l) of this subclause is met.

8.10.2 Description

The *RUNBIST* instruction provides the component purchaser with a means of running a self-test function within the component as a result of a single instruction. This permits all components on a board that offer the *RUNBIST* instruction to execute their self-tests concurrently, providing a rapid health check for the assembled board. Note, however, that the component manufacturer can include further private or public instructions to give access to individual self-test functions one at a time or to self-test functions that are not invoked by the *RUNBIST* instruction.

The sequence of steps required for completion of the execution of *RUNBIST* can be defined as:

a) (Optional) initialization of the boundary-scan register (for example, via *PRELOAD*). This is required if the pin state during BIST is to be determined by the data in the latched parallel outputs of the register.

b) Initiate BIST: scan the *RUNBIST* instruction into the instruction register.

c) Execute BIST: cause the TAP controller to remain in its *Run-Test/Idle* controller state for the duration required for completion of the execution of BIST.

d) Evaluate BIST results: bring the TAP controller to the *Shift-DR* controller state and scan out the test results (e.g., a signature) from the register connected to TDI and TDO by the *RUNBIST* instruction.

While the test is proceeding, the test logic defines the outputs from the component. As for the *INTEST* instruction, two options are available:

— The pin state may be determined by the data held in the boundary-scan register.

— Every system output pin may be forced to an inactive drive state (e.g., high-impedance).

Where the former option is selected, the data values driven through the system output pins are fixed at the time the *RUNBIST* instruction is selected, based on data held in the boundary-scan register at that time. (These data may have been preloaded using the *PRELOAD* instruction.) The boundary-scan register is controlled such that the data held in

the latched parallel outputs of cells that feed system output pins does not change while the *RUNBIST* instruction is selected. Referring to Figure 8-2, this might, for example, be achieved by holding the *Update-DR* signal at 0 while the *RUNBIST* instruction is selected. The Mode signal would be held at 1. If there are excluded segments in the boundary-scan register, then the output pins associated with those segments will be controlled by the system logic.

Boundary-scan register cells also may be used to hold programmed signal values at inputs to the on-chip system logic while the self-test is executing (again, as shown in Figure 8-7). Alternatively, boundary-scan register cells located at nonclock system logic inputs can be designed to act as a source of self-test data for the on-chip system logic. Similarly, boundary-scan register cells located at system logic outputs can act as compactors for the results of the self-test.

The specification of boundary-scan register cells for system clock input pins allows the clocks for the on-chip system logic to be obtained in one of two ways while the *RUNBIST* instruction is selected:

— The signals received at system clock pins can be fed directly to the on-chip system logic as during normal operation of the component. Where this is done, the component shall be designed such that the self-test executes *only* in the *Run-Test/Idle* controller state. The clock may, however, be active in other controller states.
— The on-chip system logic can be supplied with clock signals derived from TCK in the *Run-Test/Idle* controller state. In all other controller states, the clocks should not change state.

The rules relating to the duration of a self-test executed in response to the *RUNBIST* instruction [rule d) and rule f) in 8.10.1)] require that sufficient clock edges are applied to allow completion of self-tests executed concurrently in different components on an assembled board. Thus, in a product containing components with self-test lengths of 1000, 5000, 10 000, and 50 000 rising clock edges on TCK, the complete board shall be left in the *Run-Test/Idle* controller state for at least 50 000 rising clock edges to allow time for all tests to complete satisfactorily. Tests that complete before 50 000 clock edges have been applied will hold their results until they are accessed.

Rule g) of 8.10.1 requires that the test for an assembled board be independent of the versions of the components mounted on it. This is an important consideration when working in a maintenance or repair environment, where the versions of the components used on a board may not be known. The rule can be met by forming the exclusive-OR of the result from execution of the *RUNBIST* instruction with a fixed (version-dependent) pattern. The output from this function would become the result loaded into the boundary-scan register or the other test data register connected between TDI and TDO.

Rule h) of 8.10.1 requires that data shifted out of the component in response to the *RUNBIST* instruction is not altered by the presence of faults in off-chip system logic, board-level interconnections, and so on. This simplifies diagnosis since any errors in the output bit stream can be caused only by faults in the on-chip system logic or in the test data register connected in the path between TDI and TDO.

8.11 *CLAMP* instruction

The optional *CLAMP* instruction allows the state of the signals driven from component pins to be determined from the boundary-scan register, while the bypass register is selected as the serial path between TDI and TDO. The signals driven from the component pins will not change while the *CLAMP* instruction is selected with the exception that if the boundary-scan register is segmented, then the signals driven from the component pins controlled by an excluded segment will be determined by the system logic.

The following rules apply where the *CLAMP* instruction is provided.

NOTE—After use of the *CLAMP* instruction, the on-chip system logic may be in an indeterminate state that will persist until a system reset is applied. Therefore, the on-chip system logic may need to be reset on return to normal (i.e., nontest) operation. This can be accomplished with the optional *IC_RESET* instruction.

8.11.1 Specifications

Rules

a) The *CLAMP* instruction shall select the bypass register to be connected for serial access between TDI and TDO in the *Shift-DR* controller state.

NOTE 1—The bypass register will behave fully as defined in Clause 10 while the *CLAMP* instruction is selected. Therefore, it will load a logic 0 during the *Capture-DR* controller state and shift data during the *Shift-DR* controller state.

b) When the *CLAMP* instruction is selected, the state of all signals driven from system output pins controlled by the boundary-scan register or included boundary-scan register segments shall be defined by the data held in the boundary-scan register, and system output pins controlled by excluded boundary-scan register segments shall be defined by the system logic.

NOTE 2—For example, this boundary-scan register data may be shifted into the boundary-scan register by previous use of the *PRELOAD* instruction.

c) The states of the update stages in boundary-scan register cells located at system logic outputs (for two-state, three-state, or bidirectional pins) shall not change while the *CLAMP* instruction is selected.

d) When the *CLAMP* instruction is selected, the on-chip system logic shall be controlled such that it cannot be damaged as a result of signals received at the system input or system clock input pins.

NOTE 3—This might be achieved by placing the on-chip system logic in a reset or "hold" state while the *CLAMP* instruction is selected.

Permissions

e) The binary value(s) for the *CLAMP* instruction may be selected by the component designer.

8.11.2 Description

During testing of a particular IC or a cluster of ICs on a loaded printed circuit board, it may be necessary to place static "guarding" values on signals that control operation of components not involved in the test—for example, to place it in a state where it cannot respond to signals received from the logic under test, or to avoid driver contention when testing other components or performing an in-circuit test.

The *EXTEST* instruction could be used for this purpose. This instruction would be loaded serially into the ICs that drive the signals on which "guarding" values are required. The required signal values would be loaded as a part of the complete serial data stream shifted into the board-level path both at the start of the test and each time a new test pattern is entered. A limitation of this approach is that the length of the data pattern to be shifted for each test is increased by inclusion of the boundary-scan registers in the ICs involved in the "guarding" process. As a result, the test application rate is reduced.

The optional *CLAMP* instruction allows "guarding" values to be applied using the boundary-scan registers or included boundary-scan register segments of the appropriate ICs, but it does not retain these registers in the serial path during test application. In a case in which the *CLAMP* instruction is used to create "guarding," the following process would be used:

NOTE—It is presumed in the following description that every component implements the optional *CLAMP* instruction.

a) Before the test, the *PRELOAD* instruction would be loaded into all ICs that will provide "guarding" signals during the upcoming test. Call this group of ICs *G*. If test set-up data are required in ICs not in *G* (i.e., in those ICs that will participate actively in the upcoming test), the *PRELOAD* instruction also may be loaded into these ICs at this time.

b) Shift the "guarding" pattern into all relevant boundary-scan register cells of the ICs in *G*. Any test set-up data required for the ICs to be tested also are loaded.

c) From this point on, until the test is concluded, every time instructions are to be scanned into devices on the board, enter the *CLAMP* instruction into the ICs in *G*. As long as the *CLAMP* instruction is maintained as the active instruction in the ICs of *G*, the output signal values of these ICs will be determined by the "guarding" data in their boundary-scan registers or included boundary-scan register segments. Also, as a consequence of the use of the *CLAMP* instruction, the ICs in *G* all have their bypass registers selected throughout the test; thus, they contribute very little to the overall test time.

When the boundary-scan register is segmented, then the guarding is also segmented. By excluding a boundary-scan register segment, the component outputs controlled by the boundary cells in that segment will stay in their normal mode and be controlled by the system logic.

8.12 Device identification register instructions

Use of the optional device identification register allows a code to be serially read from the component that shows:

— The manufacturer's identity
— The part number
— The version number for the part

The device identification register is selected for scan by two standard instructions: *IDCODE* and *USERCODE*. These instructions are defined in 8.13 and 8.14. Use of the *IDCODE* instruction will provide information on the base component, while use of the *USERCODE* instruction will provide information on the particular programming of a programmable component [e.g., a fuse-programmable logic device or random access memory (RAM)-based, field-programmable gate array].

8.13 *IDCODE* instruction

8.13.1 Specifications

The following rules apply when the *IDCODE* instruction is provided.

Rules

a) Where a device identification register is included in the design, the component shall provide an *IDCODE* instruction.

b) The *IDCODE* instruction shall select *only* the device identification register to be connected for serial access between TDI and TDO in the *Shift-DR* controller state (i.e., no other test data register may be connected in series with the device identification register).

c) When the *IDCODE* instruction is selected, the device identification code shall be loaded into the device identification register on the rising edge of TCK after entry into the *Capture-DR* controller state.

d) When the *IDCODE* instruction is selected, all test data registers that can operate in either system or test modes shall perform their system function.

e) When the *IDCODE* instruction is selected, then:

1) If the TMP controller is either not provided or in the *Persistence-Off* state, the operation of the test logic other than the optional reset selection register and its associated logic (shown in Clause 17) shall have no effect on the operation of the on-chip system logic or on the flow of signals between the system pins and the on-chip system logic.

2) If the TMP controller is provided and in the *Persistence-On* state, then the state of all signals driven from system output pins shall be completely defined by the data held in the boundary-scan register.

Permissions

f) The binary value(s) for the *IDCODE* instruction may be selected by the component designer.

8.13.2 Description

Where a device identification register is included in a component design, the *IDCODE* instruction is forced into the instruction register's parallel output latches during the *Test-Logic-Reset* controller state. This allows the device identification register to be selected by manipulation of the broadcast TMS and TCK signals, as well as by a conventional instruction register scan operation. The importance of this means of selecting access to the device identification register is that it permits blind interrogation of the components assembled onto a printed circuit board, and so on. Thus, in circumstances where the component population may vary (e.g., due to second sourcing of components), it is possible to determine which components exist in a product.

The purpose of the *IDCODE* instruction is to verify that the device at a board location matches the BSDL used to generate the test. (See the rules for coding device information given in Clause 12.) If the *IDCODE* value read during test does not match the expected device identification register result, that could mean:

a) The wrong device with a compatible pin layout was located at that board location.
b) A different version of the correct device was located at that board location, as indicated by a mismatch in the version field of the device identification register, but with a matching value in the part number field.
c) A functionally compatible device from a different manufacturer was placed at that board location as indicated by a mismatch of the manufacturer's identification field.

Such results would require the board test engineer to investigate the device change and determine whether a correct part is mounted on the board and whether the current test patterns are still valid. If it is a new version or a second-sourced version of the component, the board test engineer needs to verify that the new version or manufacturer is acceptable in the current application. If any of the fields of the device identification register change, then the device may have changes in the test features documented in BSDL that would require generation of new board-level test patterns.

If just the version field changes, a device may have been revised by the manufacturer in a way that is of no concern to either the functional definition of the board or the test logic and, therefore, can be tested with the existing test patterns and shipped to end users as an acceptable variant. In such a case, either version of the device is acceptable and allowed on the board. There is no requirement that all variants of the device mission mode logic be accompanied by a version code change, although version code changes are recommended for significant mission mode variants where the change is great enough that one or another variant might not be acceptable in a specific application.

If only the manufacturer code changes, the test engineer needs to confirm that the devices from the alternative source manufacturer are acceptable. If so, then either of the device choices is acceptable on the board. The BSDL files for the two devices can be compared to verify that the test logic is the same, as is required for a true second source.

Board test engineers can decide which device identification register results are allowed and update their tests to behave accordingly.

8.14 *USERCODE* instruction

8.14.1 Specifications

The following rules apply when the *USERCODE* instruction is provided.

Rules

a) Where a device identification register is included in the design and the component is user-programmable, the component shall provide a *USERCODE* instruction.

b) The *USERCODE* instruction shall select the device identification register to be connected for serial access between TDI and TDO in the *Shift-DR* controller state (i.e., no other test data register may be connected in series with the device identification register).

c) When the *USERCODE* instruction is selected, the 32-bit user-programmable identification code shall be loaded into the device identification register on the rising edge of TCK after entry into the *Capture-DR* controller state.

d) When the *USERCODE* instruction is selected, all test data registers that can operate in either system or test modes shall perform their system function.

e) When the *USERCODE* instruction is selected, then:

   1) If the TMP controller is either not provided or in the *Persistence-Off* state, the operation of the test logic other than the optional reset selection register and its associated logic (shown in Clause 17) shall have no effect on the operation of the on-chip system logic or on the flow of signals between the system pins and the on-chip system logic.

   2) If the TMP controller is provided and in the *Persistence-On* state, then the state of all signals driven from system output pins shall be completely defined by the data held in the boundary-scan register.

Permissions

f) The binary value(s) for the *USERCODE* instruction may be selected by the component designer.

8.14.2 Description

The *USERCODE* instruction allows a user-programmable identification code to be loaded and shifted out for examination. This instruction is required only for programmable components, in which the programming cannot otherwise be determined through use of the test logic. The instruction allows the programmed function of the component to be determined and verified. This standard does not mandate the 32-bit field content, but that value should provide identifying information that will be unique to a given application of the device. For example, it could contain a version number and firmware identification code unique to the content provider and the application it serves.

The *USERCODE* instruction is intended for use in components that go through a two-stage manufacturing process, typically performed by two different companies. The first stage creates programmable functional logic, along with some fixed and some programmable test logic. The fixed test logic would include both the *IDCODE* and *USERCODE* instructions; and a device identification register loading a fixed value for *IDCODE* and a programmable value for *USERCODE,* which could be implemented in fuses or other component-level programming technology. The boundary-scan register is fixed in length, although the function of some of the boundary-scan register may be programmable. The *USERCODE* value is programmed during the second step of manufacturing, which also defines device functionality.

The BSDL provided after the first stage would include the device identification register values for the initial manufacturer and all X's for the *USERCODE* register value. The BSDL after the second stage would include a unique value for the *USERCODE* (along with any other changes required by the programming, such as a modified function of some of the boundary-scan cells), all of which is completely defined by the second-stage manufacturer with the intent that no two different devices that are "socket compatible" have the same *USERCODE* register value.

Multiple USERCODE register values can be defined in a BSDL. The presence of multiple USERCODE values in the BSDL should be interpreted as each value represents a programming of the base component that is compatible with the BSDL.

8.15 *ECIDCODE* instruction

The electronic chip identification (ECID) is a "serial number" for the silicon die. If only one die is in a package, then it becomes the serial number of the packaged die, or component. If there are more than one die in a package, and only one die has the TAP that acts for the packaged component, then at least the ECID value for the die with the common TAP on it would be reported. Other die in the package could have their ECID values retrieved by other means, or the ECID registers of each die could be connected in series and treated, from an external point of view, as a single, extended ECID value. Finally, an excludable segment structure in the ECID register could be used to retrieve the multiple ECID values from one die at a time.

The following rules apply when the *ECIDCODE* instruction is provided.

8.15.1 Specifications

Rules

a) If a vendor-defined public electronic chip identification code is embedded in the component, the component shall provide an *ECIDCODE* instruction.

b) The *ECIDCODE* instruction shall select the ECID register to be connected for serial access between TDI and TDO in the *Shift-DR* TAP controller state.

c) When the *ECIDCODE* instruction is selected, the electronic chip identification code embedded in the component shall be retrieved, if it has not been retrieved already, and:

   1) If the retrieval is not complete, a value of all 1 (111...1) shall be loaded into the ECID register no later than on the rising edge of TCK after entry into the *Capture-DR* TAP controller state.

   2) If the retrieval is complete, the retrieved value shall be loaded into the ECID register no later than on the rising edge of TCK after entry into the *Capture-DR* TAP controller state.

d) Retrieval of the electronic chip identification code shall not be dependent on external digital inputs to the component other than specified system clocks and those digital inputs required to operate the test logic (TAP ports, including TCK, and compliance-enable ports).

e) When ECID retrieval requires a defined procedure, the procedure shall be provided using the PDL (see Annex C).

NOTE 1—Rule c) through rule e) do not preclude a more complex retrieval process requiring additional design-specific instructions and time. Such a procedure can be described using PDL (see Annex C). It also does not preclude a design where the ECID is immediately available to be read, perhaps by automatic retrieval at power-up. It does require the *ECIDCODE* instruction to retrieve the value, if necessary, and to return either a valid electronic chip identification value or a specific invalid value, which would permit polling in addition to simple verification of completion.

f) Any public embedded electronic chip identification shall not have a value of all 1.

NOTE 2—This value (all 1) is interpreted as the retrieval process is not complete.

g) When the *ECIDCODE* instruction is selected, all design-specific test data registers that can operate in either system or test modes shall perform their system function.

h) When the *ECIDCODE* instruction is selected, then:

   1) If the TMP controller is either not provided or in the *Persistence-Off* state, the operation of the test logic other than the optional reset selection register and its associated logic (shown in Clause 17) shall have no effect on the operation of the on-chip system logic or on the flow of signals between the system pins and the on-chip system logic.

   2) If the TMP controller is provided and in the *Persistence-On* state, then the state of all signals driven from system output pins shall be completely defined by the data held in the boundary-scan register.

Recommendations

i) The number of system clocks used for the retrieval of the electronic chip identification code should be minimized or, where practical, TCK should be used in lieu of any system clocks.

Permissions

j) The binary value(s) for the *ECIDCODE* instruction may be selected by the component designer.

8.15.2 Description

The optional *ECIDCODE* instruction allows an identification code, unique to each integrated circuit die of a specific type, to be loaded and shifted out for examination. The device identification code and possible user code provided through the device identification register are used to identify the type and operation of a component, and the electronic chip identification provided through the ECID register is used to identify the specific instance of that type.

The mechanism required to retrieve the electronic chip identification value may vary, and nothing in this standard is intended to restrict the underlying mechanism for storing and retrieving the electronic chip identification beyond the constraints in rule d) in 8.15.1. To support the various retrieval mechanisms, a PDL procedure (see Annex C) named "ecid" can be optionally provided by the component designer to document a sequence of operations or to document just the time required for an on-chip mechanism to retrieve the value.

In any case, if the retrieval process is not complete before the TAP controller reaches the *Capture-DR* state in the *ECIDCODE* instruction, then the specified value of all 1 is captured to indicate that the electronic chip identification value is not yet available. This supports polling the ecid register when the time required is not known exactly.

8.16 *HIGHZ* instruction

Use of the optional *HIGHZ* instruction places the component in a state in which *all* of its system logic outputs are placed in an inactive drive state (e.g., high impedance). In this state, an in-circuit test system may drive signals onto the connections normally driven by a component output without incurring the risk of damage to the component.

The following rules apply where the *HIGHZ* instruction is provided.

NOTE—After use of the *HIGHZ* instruction, the on-chip system logic may be in an indeterminate state that will persist until a system reset is applied. Therefore, the on-chip system logic may need to be reset on return to normal (i.e., nontest) operation.

8.16.1 Specifications

Rules

a) The *HIGHZ* instruction shall select the bypass register to be connected for serial access between TDI and TDO in the *Shift-DR* controller state.

NOTE 1—The bypass register will behave fully as defined in Clause 10 while the *HIGHZ* instruction is selected. Therefore, it will load a logic 0 during the *Capture-DR* controller state and shift data during the *Shift-DR* controller state.

b) When the *HIGHZ* instruction is selected, all system logic outputs (including two-state and three-state outputs and bidirectional pins) of the component shall immediately be placed in an inactive drive state.

NOTE 2—If present, pull-up, pull-down, and keeper circuits are not required to be disabled.

NOTE 3—Any portion of the outputs that are powered-down are considered to be in an inactive drive state.

NOTE 4—The *HIGHZ* instruction is not affected by the state of the TMP controller.

NOTE 5—Per rule k) of 11.2.1, there can be no consequential change in the states of the parallel output registers or latches in boundary-scan register cells. For example, on leaving the *HIGHZ* instruction and selecting the *EXTEST* instruction, the data held in the boundary-scan register before selection of the *HIGHZ* instruction should be applied to the system output pins.

c) When the *HIGHZ* instruction is selected, the on-chip system logic shall be controlled such that it cannot be damaged as a result of signals received at the system input or system clock input pins.

NOTE 6—This might be achieved by placing the on-chip system logic in a reset or "hold" state while the *HIGHZ* instruction is selected.

Permissions

d) The binary value(s) for the *HIGHZ* instruction may be selected by the component designer.

8.16.2 Description

On boards where not all the components are compatible with this standard, a need will continue to exist to use in-circuit test techniques in which test signals from an ATE system are driven into internal connections of the assembled board. To allow this to be done without risk of damage to the components that normally would control these connections, components should be designed such that their system logic output pins can be placed in an inactive drive state while in-circuit testing proceeds. On a component compatible with this standard, provision of the *HIGHZ* instruction allows such a state to be entered by use of the TAP. (On components that do not comply with this standard, this typically would be achieved using a dedicated test-control pin.)

A further use of the *HIGHZ* instruction is to allow a source of test data to be connected to one or more signals internal to a loaded board in place of the normal driver(s). An example application is shown in Figure 8-10.

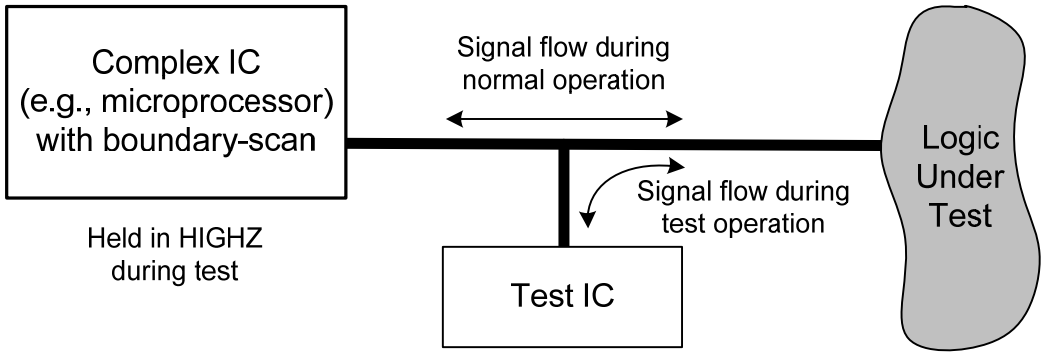


Figure 8-10—Use of the *HIGHZ* instruction

During normal operation, the outputs of the component under test would be in an inactive drive state (e.g., high-impedance), while the outputs of the processor would be active. During testing, the *HIGHZ* instruction is entered into the processor with the result that its outputs enter the inactive drive state. The component under test then can be enabled to drive the connections into the logic under test (which might, for example, be an array of memory integrated circuits).

Note that, where the system requirement is for a two-state output pin and both logic states are actively driven, a three-state buffer will have to be provided purely to allow entry into the inactive state when the *HIGHZ* instruction is selected. The enable input to this buffer will be supplied directly from the instruction decoder, as illustrated in Figure 8-11. (In Figure 8-11, the signal from the instruction decoder would be logic 1 other than when the *HIGHZ* instruction is selected.) No boundary-scan register cell is required in this signal path.

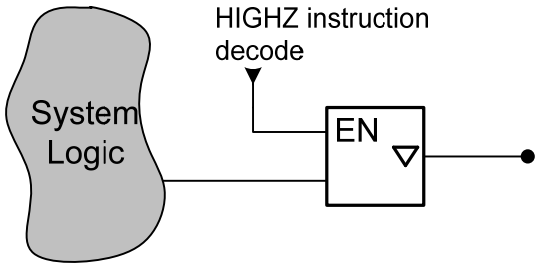


Figure 8-11—Provision of *HIGHZ* at a two-state pin

For a two-state pin where only one state is driven actively, the output should be forced into the inactively driven state when the *HIGHZ* instruction is selected. For example, the output pull-down transistor for an open-collector output would be forced off.

8.17 Component initialization instructions and procedures

At the time the IEEE Std 1149.1 was first created, the input and output circuits of components were mostly fixed TTL and CMOS I/O. This standard previously required that, after power was applied, application of the optional TRST* signal and application of the optional compliance-enable pins was the only initialization process required to prepare the component for board test.

Newer input and output circuits are often programmable, and if incorrectly programmed (or not programmed at all), they may not detect or drive the expected values during a board test or, in extreme cases, may even be damaged.

In addition, internal blocks such as phase-locked loops (PLLs) and power control circuitry may need to be placed in specific states to keep the component cool and safe in the board test environment. For example, it is not unusual for a board test to occur before heat sinks are added to components on the board, placing particularly strict requirements on component power dissipation.

Multiple solutions to providing more complex initialization processes have been used, but they are neither standardized nor automatic. They require manual changes to the generated board tests and typically lack sufficient status and correlation to achieve successful *EXTEST*-based interconnect testing and diagnosis. This 2013 version of the standard introduces optional instructions that, if provided in a component, will allow test software to automatically initialize the compliant components on a board, simplifying test preparation and execution. This standard specifies a set of rules permitting sufficient observation of the initialization process and, when appropriate, to detect a failure in achieving initialization (possibly due to a board-level fault) prior to the component going into *EXTEST*.

Note that while the initialization instructions are intended to be used after power-up and before *EXTEST,* they may be used any time they are needed.

The following rules apply when any initialization instruction is provided.

8.17.1 Specifications

Rules

a) If an orderly initialization process beyond a simple application of the TRST* TAP signal or other Power-On Reset is required for correct component operation in boundary-scan testing, the component shall provide either the pair of *INIT_SETUP* and *INIT_SETUP_CLAMP* instructions, the *INIT_RUN* instruction only, or all three instructions.

b) Compliance-enable and TAP pins shall operate as specified by this standard without programming by the initialization process defined in these rules.

c) When the initialization instructions are provided, initialization procedures shall be provided for the execution of the initialization process using the PDL defined in Annex C.

d) After the initialization procedures have completed successfully, the resulting system logic state and programmable analog characteristics of the system pins shall remain active as long as any instruction that interferes with the flow of signals between the system pins and the on-chip system logic is active, or for as long as the TMP controller remains in the *Persistence-On* state.

NOTE 1—Of the instructions defined in this standard, this includes *EXTEST, CLAMP, CLAMP_HOLD, CLAMP_RELEASE, HIGHZ, RUNBIST, INTEST*, *INIT_SETUP_CLAMP,* and *INIT_RUN*. This rule implies that if the TMP controller is not in the *Persistence-On* state, and an instruction has been made active that does not interfere with the flow of signals between system pins and the on-chip system logic, the initialization process must be run again before selecting an instruction that interferes with the on-chip system logic and the flow of signals between the system pins and the on-chip system logic.

e) Any changes to the programmable analog characteristics of system pins, and any changes (including a change to power status) to the system logic, specified by the *INIT_SETUP* or the *INIT_SETUP_CLAMP* instruction shall take effect upon or following the falling edge of TCK in the:

1) *Update-DR* TAP controller state when the *INIT_SETUP* or *INIT_SETUP_CLAMP* instruction is active.

2) *Update-IR* TAP controller state when a subsequent instruction that interferes with the flow of signals between the system pins and the on-chip system logic is made active.

NOTE 2—Of the instructions defined in this standard, this includes *EXTEST, CLAMP, CLAMP_HOLD, CLAMP_RELEASE, HIGHZ, RUNBIST, INTEST, INIT_SETUP_CLAMP,* and *INIT_RUN*.

Permissions

f) System clocks may be used by the initialization instructions as specified in the initialization procedures.

Recommendations

g) Where domains on a component may not be ready for test after power is applied, and status and/or controls are provided to allow these domains to be made ready for test, these controls should be capable of being set and the status capable of being read by the *INIT_SETUP* and *INIT_SETUP_CLAMP* instructions.

h) Where system clocks are used in the initialization procedures, the number of system clocks required should be minimized.

i) Where system clocks are used in the initialization procedures, they should be capable of being shut off at the end of the initialization process without loss of the results of the initialization process.

8.17.2 Description

The *INIT_SETUP* instruction supplies any information required by the initialization. The information to be supplied is determined by both the component and the board on which the component is used.

The *INIT_SETUP* instruction does not interfere with the flow of signals between the system pins and the on-chip system logic (that is, it is not a test mode instruction), but it can still be disruptive. The analog characteristics of the I/O can be changed by the *INIT_SETUP* instruction while the I/O are still controlled by the system logic. In addition, other effects of the information supplied by the *INIT_SETUP* instruction, such as powering-up domains, including boundary-scan register domains containing I/O, turning PLLs ON or OFF, can take effect immediately. So while the instruction is not a test mode instruction, it can be disruptive to both the I/O and the system logic.

Because conditions on the board may be indeterminate during the initialization process, the *INIT_SETUP_CLAMP* instruction is required whenever the *INIT_SETUP* instruction is provided. This instruction enforces a *CLAMP* behavior, controlling the system pins from the boundary-scan register (that is, it is a test mode instruction). While a

PRELOAD instruction will normally be required before invoking this instruction, it does provide the ability to mask most effects of initialization from other components on a board, when necessary.

In the case where the *INIT_SETUP* or *INIT_SETUP_CLAMP* instruction is used to control excludable segments (see 9.4) and will need to make the domains containing those segments ready for test, the test data register selected by the *INIT_SETUP* or *INIT_SETUP_CLAMP* instruction may need to be scanned multiple times.

The *INIT_RUN* instruction initiates and provides the time for an internally controlled sequential (state machine) initialization process, optionally using information provided by the *INIT_SETUP* instruction. The *INIT_RUN* instruction does interfere with the flow of signals between the system pins and the on-chip system logic. The initialization procedures must wait a specified amount of time to finish (in terms of a number of clock cycles or actual time), or poll initialization status to determine completion. What happens during the execution of the sequential machine initiated by the *INIT_RUN* instruction is the responsibility of the component designer and is not restricted to programming input and output circuits. If a sequential initialization process is not required, then the *INIT_RUN* instruction need not be provided.

The *INIT_SETUP* instruction, if provided, must be run prior to any instruction that interferes with the flow of signals between the system pins and the on-chip system logic. (Of the instructions defined in this standard, this includes *EXTEST, CLAMP, CLAMP_HOLD, CLAMP_RELEASE, HIGHZ, RUNBIST, INTEST*, *INIT_SETUP_CLAMP,* and *INIT_RUN*.) Given the programmability of some I/O circuits in current use, their analog characteristics can vary from board to board, or even between two uses of a component on a board, so the data provided by the initialization procedures and loaded by the *INIT_SETUP* or *INIT_SETUP_CLAMP* instruction may vary from use to use, and from board to board.

The *INIT_RUN* instruction, if provided, must be run prior to any other instruction that interferes with the flow of signals between the system pins and the on-chip system logic other than the *INIT_SETUP_CLAMP* instruction. The *INIT_SETUP* and *INIT_SETUP_CLAMP* instruction pair and the *INIT_RUN* instruction are each optional and independent. One can exist without the other. If *INIT_RUN* is provided and requires parameters, then all three instructions must be provided.

The initialization processes are expected to leave the component in a controlled test mode state where the system logic is cool and safe, the input and output pin analog characteristics are set appropriately for boundary-scan testing of the board under test, and the I/O are controlled from the boundary-scan register. While not required by this standard, a component designer may wish to use the initialization instruction decodes as a “warning” to the system logic that a test mode is about to ensue, and block any system logic activity that would interfere with test initialization. Alternatively, a component designer could use an initialization process to prepare the component for some built-in test.

Table 8-1 and Table 8-3 illustrate typical initialization sequences for a board with a mixture of components having varying initialization requirements. It is assumed that the board has been powered up and all compliance signal values applied. Steps labeled 1 through 8 would be a normal board interconnect test without any initialization requirement; steps labeled Init1 through Init5 are added to support initialization. As shown in the supplementary Table 8-2 and Table 8-4, this sequence can be further modified if there are powered-down and/or excludable register segments in the boundary-scan register that need to be powered-up or included.

Table 8-1—Typical initialization sequence, deferred test mode

Step	Action	Device 1 has no *INIT_SETUP* or *INIT_RUN*	Device 2 has only *INIT_SETUP*, *INIT_SETUP_CLAMP*	Device 3 has only *INIT_RUN*	Device 4 has *INIT_SETUP*, *INIT_SETUP_CLAMP* and *INIT_RUN*
1	TLR:	Initialize TAP; component I/O connected in mission mode.			
2	IRScan:	Load *PRELOAD* instruction to get ready for *EXTEST*.			
3	DRScan (NOTE 1):	Shift first pattern of test data into boundary-scan register; ignore data out.			
Init1	**IRScan**: Get ready for setup data	Load *BYPASS*.	Load *INIT_SETUP*.	Load *BYPASS*.	Load *INIT_SETUP*.
Init2	**DRScan** (NOTE 1): Load setup data		Shift Device2 setup data.		Shift Device4 setup data.
Init3	**IRScan** (NOTE 2 and NOTE 3): Complete init process	Load *CLAMP* or *EXTEST* (start test mode).	Load *CLAMP* or *INIT_SETUP_CLAMP* (start test-mode).	Load *INIT_RUN* (start test mode).	Load *INIT_RUN* (start test mode).
Init4	**DRScan**: Read out status	If *EXTEST*, load in same data as step 3, ignore data out.	If *INIT_SETUP_CLAMP* load in same data as step Init2, ignore data out.	Read Device3 status.	Read Device4 status.
Init5	Loop:			If not “Done” go to step Init4. If “Error”, exit.	
4	**IRScan**: Start test	Load *EXTEST*.			
5	DRScan:	Shift next test data vector into boundary-scan register; save current response data out.			
6	Loop:	Return to step 5 for each remaining pattern.			
7	DRScan:	Shift safe data into boundary-scan register; save last response data out.			
8	TLR:	End of test; I/O returned to mission mode, but in undefined state.			

NOTE 1—Two or more scans (see Table 8-2) may be needed if excludable or selectable segments (see 9.4) need to be included or selected, or domains need to be enabled.

NOTE 2—Start of test mode across the board.

NOTE 3—May require delay (wait) during execution.

If the boundary-scan register contains segments that are powered down and/or excluded when the component and board are powered up, then step 3 in Table 8-1 can be expanded as shown in Table 8-2 to bring those segments into the boundary-scan register. (See 9.4 for the definition of excludable segments.) In this case, this is done while the component and board are still in the mission mode, and it assumes that including those segments while in mission mode will not cause three-state conflicts or other problems on the board. The *PRELOAD* instruction is active in all components through and after this sequence.

Table 8-2—Including boundary-scan segments in mission mode

Step	Action	Device 1 has no *INIT_SETUP* or *INIT_RUN*	Device 2 has only *INIT_SETUP*, *INIT_SETUP_CLAMP*	Device 3 has only *INIT_RUN*	Device 4 has *INIT_SETUP*, *INIT_SETUP_CLAMP* and *INIT_RUN*
3.1	DRScan:	Load appropriate domain-control cells with 1.			
3.2	DRScan:	After power-up wait, if needed; check all appropriate segment-select cells capture 1.			
3.3	DRScan:	Load appropriate segment-select cells with 1.			
3.4	DRScan:	Shift first interconnect pattern of test data into boundary-scan register; ignore data out.			

In some cases, the initialization process itself, or the process of powering-up and including boundary-scan segments, may create three-state conflicts or other problems on the board. In this case, these problems may be avoidable if the components on the board are taken out of mission mode and put into test mode before such changes take place. Table 8-3 shows a typical sequence that would achieve this goal.

Table 8-3—Typical initialization sequence, immediate test mode

Step	Action	Device 1 has no *INIT_SETUP* or *INIT_RUN*	Device 2 has only *INIT_SETUP*, *INIT_SETUP_CLAMP*	Device 3 has only *INIT_RUN*	Device 4 has *INIT_SETUP*, *INIT_SETUP_CLAMP* and *INIT_RUN*
1	TLR:	Initialize TAP; component I/O connected in mission mode.			
2	IRScan:	Load *PRELOAD* instruction to get ready for *EXTEST*.			
3	DRScan (NOTE 1):	Shift first interconnect pattern of test data into boundary-scan register; ignore data out.			
Init1	**IRScan** (NOTE 2): Get ready for setup data	Load *CLAMP* or *EXTEST*, (start test mode).	Load *INIT_SETUP_CLAMP* (start test mode).	Load *CLAMP* or *EXTEST* (start test mode).	Load *INIT_SETUP_CLAMP* (start test mode).
Init2	**DRScan** (NOTE 1): Load setup data	If *EXTEST*, load in same data as step 3, ignore data out.	Shift Device2 setup data.	If *EXTEST*, load in same data as step 3, ignore data out.	Shift Device4 setup data.
Init3	**IRScan** (NOTE 3): Complete init process	Load *CLAMP* or *EXTEST*.	Load *CLAMP* or *INIT_SETUP_CLAMP*.	Load *INIT_RUN*.	Load *INIT_RUN*.
Init4	**DRScan**: Read out status	If *EXTEST*, load in same data as step 3, ignore data out.	If *INIT_SETUP_CLAMP* load in same data as step Init2, ignore data out.	Read Device3 status.	Read Device4 status.
Init5	Loop:			If not “Done” go to step Init4. If “Error”, exit.	
4	**IRScan**: Start test	Load *EXTEST*.			
5	DRScan:	Shift next test data vector into boundary-scan register; save current response data out.			
6	Loop:	Return to step 5 for each remaining pattern.			
7	DRScan:	Shift safe data into boundary-scan register; save last response data out.			

Step	Action	Device 1 has no *INIT_SETUP* or *INIT_RUN*	Device 2 has only *INIT_SETUP*, *INIT_SETUP_CLAMP*	Device 3 has only *INIT_RUN*	Device 4 has *INIT_SETUP*, *INIT_SETUP_CLAMP* and *INIT_RUN*
8	TLR:	End of test; I/O returned to mission mode, but in undefined state.			

NOTE 1—Two or more scans may be needed if excludable or selectable segments need to be included or selected, or domains need to be enabled; see Table 8-4.

NOTE 2—Start of test mode across the board.

NOTE 3—May require delay (wait) during execution.

If the boundary-scan register contains segments that are powered down and/or excluded when the component and board are powered up, then Step 3 in Table 8-3 can be expanded as shown in Table 8-4 to bring those segments into the boundary-scan register. (See 9.4 for the definition of excludable segments.) In this case, this is done while the component and board are in the test mode. The *PRELOAD* instruction is active in all components, but the *EXTEST* instruction is immediately made active to initiate test mode, which will be in effect when continuing with step Init1 in Table 8-3.

Table 8-4—Including boundary-scan segments in test mode

Step	Action	Device 1 has no *INIT_SETUP* or *INIT_RUN*	Device 2 has only *INIT_SETUP*, *INIT_SETUP_CLAMP*	Device 3 has only *INIT_RUN*	Device 4 has *INIT_SETUP*, *INIT_SETUP_CLAMP*, and *INIT_RUN*
3.1	DRScan:	Shift first interconnect pattern of test data into boundary-scan register; ignore data out.			
3.2	IRScan:	Load *CLAMP* or *EXTEST* instruction (start test mode). *EXTEST* required for components with excludable boundary-scan register segments.			
3.3	DRScan:	Load appropriate domain-control cells with 1.			
3.4	DRScan:	After power-up wait, if needed; check all appropriate segment-select cells capture 1.			
3.5	DRScan:	Load appropriate segment-select cells with 1.			
3.6	DRScan:	Because boundary-scan register has changed length, shift first interconnect pattern of test data into selected boundary-scan registers again; ignore data out.			

8.18 *INIT_SETUP* and *INIT_SETUP_CLAMP* instructions

8.18.1 Specifications

Rules

a) The *INIT_SETUP* and *INIT_SETUP_CLAMP* instructions shall both be provided when the initialization process requires parameterization, including parameters for setting input and output characteristics, for controlling excludable domains, or for controlling the sequential execution of the initialization process.

b) The *INIT_SETUP* and *INIT_SETUP_CLAMP* instructions shall select the initialization data register (see Clause 14) to be connected for serial access between TDI and TDO in the *Shift-DR* TAP controller state.

c) When the *INIT_SETUP* instruction is active, all public test data registers that can operate in either system or test modes, other than the initialization data register, shall perform their system functions.

d) When the *INIT_SETUP* instruction is active and the TMP controller is either not provided or in the *Persistence-Off* state, the operation of the test logic other than the optional reset selection register and its

associated logic (shown in Clause 17) shall have no effect on the flow of signals between the system pins and the on-chip system logic.

e) When the *INIT_SETUP* instruction is active and the TMP controller is provided and in the *Persistence-On* state, or the *INIT_SETUP_CLAMP* instruction is active, the state of all signals driven from system output pins shall be completely defined by the data held in the boundary-scan register.

Permissions

f) The binary value(s) for the *INIT_SETUP* and *INIT_SETUP_CLAMP* instructions may be selected by the component designer.

8.18.2 Description

As described above, the *INIT_SETUP* and *INIT_SETUP_CLAMP* instructions supply the data needed for initializing the component on the board by loading the initialization data register. The data values may change from board to board and even from use to use of a component on a single board.

The *INIT_SETUP* instruction does not interfere with the operation of the component input and output circuits and should be run before any such instruction is run. *INIT_SETUP_CLAMP,* on the other hand, forces *CLAMP* type behavior, putting the component into test mode. This will normally require a *PRELOAD* instruction first. The difference is to allow a test engineer to control exactly when the component transitions from mission to test mode. For example, some power domains may need to be powered up and boundary or initialization data register segments may need to be included. Such changes can have an effect on the board, and if that becomes a problem, the test engineer can switch from *INIT_SETUP* to *INIT_SETUP_CLAMP* to mask undesirable effects of initialization on a board. Normally, after *INIT_SETUP,* the component would not go to test mode until a test mode instruction such as *EXTEST* or *INIT_RUN* was made active.

It is recognized that, sometimes, initialization data are provided through component pins. These may be tied to a high or low value on the board, or controlled by another source on or off the board. Such pins have not had a definition in this standard, nor any standardized way of monitoring them to verify that they are set as expected for the board. With the allowance for monitoring virtually any pin with redundant observe-only boundary-scan register cells, these pins may be monitored during test in the boundary-scan register. Alternatively, with the capture capability of the initialization data register (see Clause 14), these pins may be checked and verified prior to initialization and scan the boundary-scan register, when required.

8.19 *INIT_RUN* instruction

8.19.1 Specifications

Rules

a) If the initialization process requires the execution of a sequential process within the component requiring some time before any test instruction controlling the input and output circuits can be used, then the *INIT_RUN* instruction shall be provided.

b) The *INIT_RUN* instruction shall select the initialization status register (see Clause 15) to be connected for serial access between TDI and TDO in the *Shift-DR* controller state.

c) While the *INIT_RUN* instruction is active, the sequential initialization process built into the component shall execute.

d) While the *INIT_RUN* instruction is active, the on-chip system logic shall be controlled such that it cannot be damaged as a result of signals received at the system input or system clock input pins.

e) When the *INIT_RUN* instruction is active, all system outputs from the component shall be defined by data held in the boundary-scan register, whose analog characteristics may have been defined by data held in the initialization data register.

f) A maximum duration shall be specified for the initialization process executed in response to the *INIT_RUN* instruction (e.g., absolute time or a number of rising edges of TCK).

Recommendations

g) After the successful completion of *INIT_RUN*, and when the active instruction no longer requires control of the input and output circuits, the device system logic should remain in the initialized state until another test or a functional reset process is initiated.

NOTE—This is to protect the component and other components on the board, and it does not eliminate the need to re-run initialization after an instruction that does not control the input and output circuits.

Permissions

h) At any time while the *INIT_RUN* instruction is active, the initialization status register may capture data in the *Capture-DR* TAP controller state and be scanned out in the *Shift-DR* TAP controller state.

i) The binary value(s) for the *INIT_RUN* instruction may be selected by the component designer.

8.19.2 Description

Sometimes, complex input and output circuits will require sequential initialization. More often, the system logic will require a sequential process to prepare it for the board test. For example, in order to minimize power consumption, PLLs or internal clock distribution may need to be turned OFF, various internal power domains shut down or powered up, and other steps taken to put the system logic into a protected state for the board test.

A state machine controlling such a process can be designed into the component by the designer. The *INIT_RUN* instruction is then used to automatically initiate and time the execution of that state machine prior to the board test. Alternatively, a PDL procedure “init_run” (see Annex C) could be provided by the component designer, and it could make use of multiple instructions (such as the optional *IC_RESET* instruction) to properly sequence the system logic to a safe state, as needed. In this case, the *INIT_RUN* instruction itself may not even be needed, and the analog characteristics of the I/O set by the *INIT_SETUP* instruction would take effect when the first test instruction (*EXTEST*, *CLAMP_HOLD*, etc.) becomes active.

The test software would normally run *INIT_SETUP* as the last non-test instruction followed by *INIT_RUN* as the first test instruction, although this could be altered by the provided PDL procedures.

While the *INIT_SETUP_CLAMP* or *INIT_RUN* instructions are selected, the logic value of all signals driven from system output pins will be completely defined by the data held in the boundary-scan register and will not change during the *Update-DR* state.

If the initialization process successfully completes on a component, then the system logic is assumed to be in a safe state with the output circuits ready for boundary-scan testing to occur. This is based on the assumed correctness of any values loaded into the initialization data register by the *INIT_SETUP* or *INIT_SETUP_CLAMP* instruction.

When the initialization process does not correctly complete on a component, then no instruction requiring control of the input and output circuits can be expected to operate correctly. Of the instructions defined in this standard, this includes *EXTEST, SAMPLE, CLAMP, CLAMP_HOLD, CLAMP_RELEASE, HIGHZ*, *RUNBIST* and *INTEST*. *BYPASS, IDCODE,* and *USERCODE* are expected to work without initialization. *PRELOAD* is expected to work, but it may not produce expected results if the boundary-scan register has excluded segments. In addition, the component system logic may not be in a safe state for the test.

The results of initialization are expected to remain in effect as long as the active instruction is one that interferes with system operation and the flow of signals between the system pins and the on-chip system logic (such as *EXTEST*, *CLAMP, CLAMP_HOLD, CLAMP_RELEASE, HIGHZ, INIT_SETUP_CLAMP, INIT_RUN*, *RUNBIST*, or

INTEST). The results are also expected to remain in effect if the TMP controller is provided and is in the *Persistence-On* state.

When the TMP controller is either not provided or is in the *Persistence-Off* state, loading an instruction that does not interfere with system operation (such as *PRELOAD, IDCODE, BYPASS*, etc.), or asserting the *Test-Logic-Reset* state, creates an assumption that the mission mode logic has taken control of those aspects of the system logic that were defined by the execution of the *INIT_RUN* instruction and, possibly, the analog characteristics of the I/O. Thus, in this situation, at least the *INIT_RUN* instruction or the "init_run" PDL procedure would have to be re-run to take back control of the system logic and the input and output circuits prior to running other standard instructions that interfere with system operation and the flow of signals between the system pins and the on-chip system logic.

The initialization process must put the system logic into a safe and test-ready state, which could prevent actions such as thermal runaway or large power consumption, as well as reducing system noise for sensitive tests, and so on. Whenever possible, the system logic should remain in that state, regardless of other test activity, until some explicit form of system reset takes control. Similarly, the analog controls of the I/Os may be released when the active instruction is one that does not block the flow of signals between the system logic and the system pins, or may stay in the same state until new instructions or functional operation change the values. There is some ambiguity as to when the initialized state of the I/Os and system logic end. Therefore, if the initialized state is required for tests after an instruction is made active that does not interfere with system operation or the flow of signals between the I/O and the system logic (such as *BYPASS* or *PRELOAD*), then the component must be re-initialized.

8.20 *CLAMP_HOLD*, *CLAMP_RELEASE*, and *TMP_STATUS* instructions

The *CLAMP_HOLD* and *CLAMP_RELEASE* instructions, provided if and only if the optional TMP controller is provided, control the state of the test mode persistence (TMP) controller. The *TMP_STATUS* instruction, again provided if and only if the TMP controller is provided, captures the current TMP controller state and the current state of the bypass-escape bit in the TMP status register. The TMP controller, in turn, forces the state of the signals driven from component pins to be determined from the boundary-scan register even when other instructions that would normally not control the signals at the pins may be active. The *CLAMP_HOLD* and *CLAMP_RELEASE* instruction pair departs from other instructions in this standard in that they set or clear a controller state, which modifies the behavior of the *Test-Logic-Reset* TAP controller state and many instructions.

The following rules apply when the *CLAMP_HOLD*, *CLAMP_RELEASE,* and *TMP_STATUS* instructions are provided.

8.20.1 Specifications

Rules

a) If and only if the TMP controller (described in 6.2) is provided per permission d) of 5.1.1, then the *CLAMP_HOLD*, *CLAMP_RELEASE,* and *TMP_STATUS* instructions shall be provided.

b) The *CLAMP_HOLD, CLAMP_RELEASE,* and *TMP_STATUS* instructions shall select the TMP status register to be connected for serial access between TDI and TDO in the *Shift-DR* controller state (see Clause 16).

c) When either the *CLAMP_HOLD* or the *CLAMP_RELEASE* instruction is selected, the state of signals driven from all system output pins provided with boundary cells in any included boundary-scan register segments shall be controlled by the data in those cells.

d) The *CLAMP_HOLD* instruction shall set the TMP controller state to *Persistence-On*.

e) The *CLAMP_RELEASE* instruction shall set the TMP controller state to *Persistence-Off*.

NOTE 1—Since the *CLAMP_HOLD* and *CLAMP_RELEASE* instructions already control the component pins from the boundary-scan register, the change in the TMP controller state is not immediately observable at the pins and thus the change of TMP controller state can be performed at any time while the respective instructions are selected. The change must occur on a rising edge of TCK, as required by the TMP controller rules.

f) When the *TMP_STATUS* instruction is selected, then:
   1) If the TMP controller is in the *Persistence-Off* state, the operation of the test logic other than the optional reset selection register and its associated logic (shown in Clause 17) shall have no effect on the operation of the on-chip system logic or on the flow of signals between the system pins and the on-chip system logic.
   2) If the TMP controller is in the *Persistence-On* state, then the state of all signals driven from system output pins shall be completely defined by the data held in the boundary-scan register.

g) The states of the update stages in boundary-scan register cells located at system logic outputs (for two-state, three-state, or bidirectional pins) shall not change while any of the *CLAMP_HOLD*, *CLAMP_RELEASE,* or *TMP_STATUS* instructions are selected.

h) The on-chip system logic shall be controlled such that it cannot be damaged as a result of signals received at the system input or system clock input pins while either the *CLAMP_HOLD* or *CLAMP_RELEASE* instruction is selected.

NOTE 2—This might be achieved by placing appropriate portions of the on-chip system logic in a reset or "hold" state while the *CLAMP_HOLD* or *CLAMP_RELEASE* instruction is selected.

i) The opcodes chosen for the *CLAMP_HOLD* and *CLAMP_RELEASE* instructions shall not include the all 0 opcode.

NOTE 3—The all 0 opcode could be presented to a device by a stuck-at-zero fault on a device TDI pin. As with any intrusive instruction, the *CLAMP_HOLD* and *CLAMP_RELEASE* instruction should not be selected by this event.

Permissions

j) The binary value(s) for the *CLAMP_HOLD*, *CLAMP_RELEASE,* and *TMP_STATUS* instructions may be selected by the component designer.

8.20.2 Description

The *CLAMP_HOLD* and *CLAMP_RELEASE* instructions exist solely to control the TMP controller state (see 6.2). These instructions do not need to perform a data register scan to change the TMP controller state. When the *Persistence-On* state is set, the device I/O and boundary-scan register remain in test mode regardless of the active instruction or the *Test-Logic-Reset* state. The *TMP_STATUS* instruction simply observes the current state of the TMP controller status without causing a change in the test mode state of the component. All three instructions are required if the TMP controller is provided, and they must not be provided otherwise (they are reserved keywords in BSDL).

All three instructions select the TMP status register for scanning, which includes a read-only TMP-status bit, which observes the TMP controller state, and the write only bypass-escape bit, which enables or disables the bypass-escape transition of the TMP controller state machine. See Clause 16 for details on the TMP status register.

When the TMP controller is not provided, a device would enter test mode only when the active instruction takes control of the device I/O away from the system logic (i.e., *EXTEST, CLAMP, HIGHZ, RUNBIST, INIT_SETUP_CLAMP, INIT_RUN,* etc.) and the device leaves test mode when the active instruction does not control the device I/O (i.e., *PRELEAD, SAMPLE, IDCODE, BYPASS, INIT_SETUP*, etc.). The *Test-Logic-Reset* TAP controller state forces the *IDCODE* or *BYPASS* instruction, causing the device to leave test mode. This is the "traditional" behavior of a device conforming to this standard since its inception in 1990.

When the TMP controller is provided, a device that was switched from its normal activities to test mode will persist in test mode rather than enter into an undefined and potentially destructive state that could occur by reconnecting the I/O pins to the device's system logic. Since the state of the internal logic of a device may be completely undefined as a result of entering test mode, it could be risky to reconnect this logic to the I/O pins and attempt to resume functional operation. The rules in this standard require that the system logic be maintained in a state designed to be

safe by the component designer while the component is in test mode. The persistent clamped I/O state gives test engineers the ability to control the unintended consequences of testing.

In addition, certain types of internal tests selectable by design-specific instructions may share some I/O pins to allow an IC tester to monitor the test. If this test may also be performed in situ, on a fully populated board, it may be necessary to block those signals to prevent them from causing undesired effects in other devices on the board. The TMP controller allows the test engineer to put such a device on a board into a safe state for running such tests.

For the boundary-scan register cells defined in this standard (**BC_1**, **BC_2**, etc.), control of the I/O is dominantly exercised through one or another of the "Mode" signals defined in the tables provided in Clause 11, which include entries for the TMP controller *Persistence-On* state and for excluded boundary-scan register segments. In addition, some of the non-data characteristics of the I/O may be controlled in the test mode by the results of running the *INIT_SETUP, INIT_SETUP_CLAMP,* and *INIT_RUN* instructions.

When the *Persistence-On* state is set, instructions that would otherwise assert test mode will behave as specified. *HIGHZ*, for instance, will shift the I/O controlled by included boundary-scan register segments from a *CLAMP* to a *HIGHZ* behavior while active. Instructions that would otherwise not assert test mode will behave as specified except that the I/O controlled by included boundary-scan register segments will remain controlled by the test logic. This behavior is shown in Table 8-5.

Table 8-5—I/O pin behavior for TMP controller states

I/O pin behavior per instruction based on TMP controller state		
Instruction	**Behavior when *Persistence-Off***	**Behavior when *Persistence-On***
IDCODE/USERCODE	Mission pins connected to system logic	Pins controlled by boundary-scan register content
BYPASS	Mission pins connected to system logic	Pins controlled by boundary-scan register content
PRELOAD	Mission pins connected to system logic	Pins controlled by boundary-scan register content but content could change at *Update-DR* (emulates *EXTEST*)
SAMPLE with permission f) of 8.6.1	Mission pins connected to system logic, emulates *PRELOAD*	Pins controlled by boundary-scan register content but content could change at *Update-DR* (emulates *EXTEST*)
SAMPLE without permission f) of 8.6.1	Mission pins connected to system logic, does not emulate *PRELOAD*	Pins controlled by boundary-scan register content
EXTEST	IEEE 1149.1 *EXTEST*	IEEE 1149.1 *EXTEST*
INTEST with *EXTEST* output behavior; rule c1) of 8.9.1	Pins controlled by boundary-scan register content; emulates *EXTEST* on outputs only	Pins controlled by boundary-scan register content; emulates *EXTEST* on outputs only
INTEST with *CLAMP* output behavior; rule c2) of 8.9.1	Pins controlled by boundary-scan register content	Pins controlled by boundary-scan register content
CLAMP	Pins controlled by boundary-scan register content	Pins controlled by boundary-scan register content
HIGHZ	All pins high impedance	All pins high impedance (does not change boundary-scan register content)
RUNBIST with *HIGHZ* behavior; rule k2) of 8.10.1	All pins high impedance	All pins high impedance (does not change boundary-scan register content)
RUNBIST with *CLAMP* behavior; rule k1) of 8.10.1	Pins controlled by boundary-scan register content	Pins controlled by boundary-scan register content
INIT_SETUP	Mission pins connected to system logic	Pins controlled by boundary-scan register content

I/O pin behavior per instruction based on TMP controller state		
Instruction	**Behavior when *Persistence-Off***	**Behavior when *Persistence-On***
INIT_SETUP_CLAMP	Pins controlled by boundary-scan register content	Pins controlled by boundary-scan register content
INIT_RUN	Pins controlled by boundary-scan register content	Pins controlled by boundary-scan register content
CLAMP_HOLD	Pins controlled by boundary-scan register content	Pins controlled by boundary-scan register content
CLAMP_RELEASE	Pins controlled by boundary-scan register content	Pins controlled by boundary-scan register content
TMP_STATUS	Mission pins connected to system logic	Pins controlled by boundary-scan register content
IC_RESET	Mission pins connected to system logic	Pins controlled by boundary-scan register content

In addition, the behavior of the *Test-Logic-Reset* TAP controller state is significantly changed when the *Persistence-On* state is set. The forced setting of the *IDCODE* or *BYPASS* opcode in the update portion of the instruction register still takes place, but the resetting of any TDR or other test logic that affects control of the I/O must be blocked. This specifically applies, but is not limited, to the setting or clearing of the update flip-flop of a boundary-scan register control cell as described in part 2 of permission h) of 11.3.1 and illustrated in the gated-clock design in Figure 8-12.

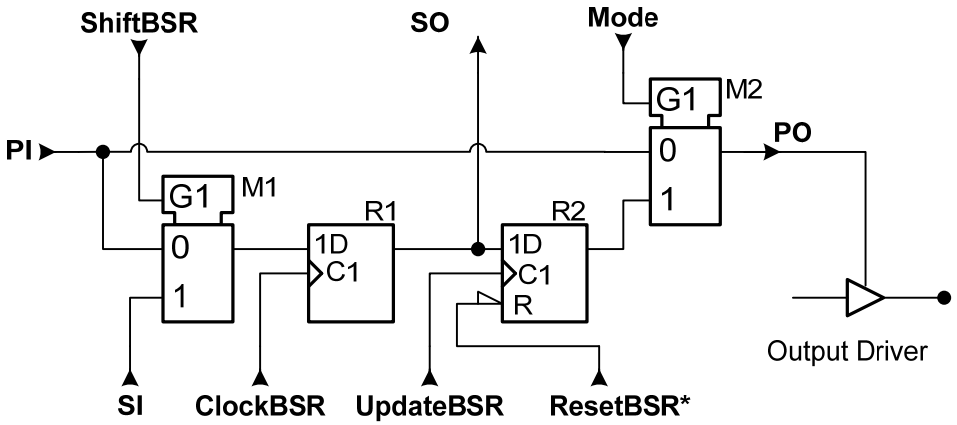


Figure 8-12—Boundary-scan register control cell with a reset on the update flip-flop R2

If the ResetBSR* signal shown in Figure 8-12 were connected to the TAP Controller output signal "Reset*", then entry into the *Test-Logic-Reset* TAP controller state will clear (or set) the Update flip-flop "R2" of a control cell to the state that will disable the driver [see rule p) of 11.6.1]. This would produce the noncompliant effect of a visible change at the I/O pins if the TMP controller was in the *Persistence-On* state. One function of the TMP controller is to intercept the "Reset*" signal from the TAP controller to the TDRs controlling the I/O. See Figure 6-10 for an implementation example including the generation of the modified reset signal "CHReset*". When the TMP controller exists, the "CHReset*" signal is connected to the boundary-scan register reset ("ResetBSR*") signal and other TDRs controlling the I/O. In this way, the boundary-scan register cells controlling data to I/O pins, and the effects of the initialization instructions controlling the other characteristics of the I/O pins, are maintained across the *Test-Logic-Reset* TAP controller state. Note that either TRST* or on-chip power-up reset generation will continue to reset all of the test logic, including the TMP controller state.

8.21 *IC_RESET* instruction

The purpose of the optional *IC_RESET* instruction is to provide a means to control reset and related signals to the system logic using the TAP. This instruction selects the reset selection register (see Clause 17).

Complex components may have multiple internal reset functions that they can perform. These are normally initiated by pulsing one or another I/O pin, although some reset functions may be initiated within the system logic, such as between two IP blocks on an SOC (system-on-a-chip). The *IC_RESET* instruction and its associated reset selection register (see Clause 17) allow control of system reset functions through the TAP, including blocking undesired resets to the system logic during testing.

The term "system reset" is defined loosely, and a given reset signal to the system logic could cause any number of actions to occur, from preparing some part of the component for functional operation, to forcing a region into a state ready for a BIST, to forcing some functional block off-line.

The following rules apply when the *IC_RESET* instruction is provided.

8.21.1 Specifications

Rules

a) When the *IC_RESET* instruction is selected, the reset selection register shall be connected for serial access between TDI and TDO in the *Shift-DR* controller state (see Clause 17).

b) When the *IC_RESET* instruction is selected, any changes of the reset function signal(s) to the system logic caused by the test logic shall be asserted or de-asserted on the falling edge of TCK in the *Update-DR* TAP controller state.

NOTE 1—This rule does not constrain timing when the reset selection register does not control the reset signals to the system logic. See Clause 17.

c) When the *IC_RESET* instruction is selected and the optional TMP controller is not provided or is in the *Persistence-Off* state, the operation of the test logic other than the optional reset selection register and its associated logic (shown in Clause 17) shall have no effect on the flow of signals between the system pins and the on-chip system logic.

NOTE 2—This rule does not constrain the response of the system logic to reset signals controlled by the reset selection register. Such a response might, itself, affect the flow of signals to and from the system logic.

d) When the *IC_RESET* instruction is selected and the optional TMP controller is provided and in the *Persistence-On* state, the state of signals driven from all system output pins provided with boundary cells in any included boundary-scan register segment shall be controlled by the data in those cells.

e) When the optional TMP controller is provided and in the *Persistence-On* state, any system reset function initiated by the *IC_RESET* instruction shall be designed so that it does not alter the data held in the boundary-scan register nor alter the characteristics of the component I/O established by execution of the *INIT_SETUP, INIT_SETUP_CLAMP,* and *INIT_RUN* instructions.

NOTE 3—This rule may be met by using the TMP controller *Persistence-On* state to block propagation of any reset signal that would otherwise violate this rule. In other words, some internal reset signals may be defined as valid only if the TMP controller is in the *Persistence-Off* state.

f) The *IC_RESET* instruction shall not affect the operation of the TAP TRST* pin, any internally generated POR* signal to the TAP (see Figure 6-8), nor either the Reset* or CHReset* outputs of the TAP and TMP controller, respectively, nor any other test logic.

Recommendations

g) All functional reset sources to the component should be controlled by the *IC_RESET* instruction.

NOTE 4—Functional reset sources include I/O pins and internally generated reset signals.

h) Where practical, independent system reset functions should be controlled separately, even if controlled in parallel from a single reset input to the component.

i) The binary logic value(s) for the *IC_RESET* instruction should not include the all 0 value.

Permissions

j) The binary logic value(s) for the *IC_RESET* instruction may be selected by the component designer.

k) Additional design-specific instructions may be provided to give the component users access to system reset functions within a component.

l) System clocks may be specified when the system logic requires such clocks to complete the reset action.

8.21.2 Description

The *IC_RESET* instruction is an optional instruction providing a standard means for the test logic to control one or more reset signals to the system logic without requiring access to the reset pins on the board. This may be desirable in a test environment where the boundary-scan register may intercept the board reset signals normally connected to the component I/O, blocking their execution, or where the reset pins are not accessible to the tester. The *IC_RESET* instruction may help maintain safe board or system operation after testing is complete without the need of a manual power cycle or external system reset intervention.

While a given reset signal may cause an initialization state machine in the functional logic to run and initialize some portion of the system logic, the core concepts of reset and initialization are separate and distinct. The purpose of this instruction is to provide test control of the system and other reset inputs to the system logic, regardless of the response of the system logic to the system inputs. The initialization instructions are provided to allow test control of component initialization for a test, at least for a board test. On the other hand, the reset signal to the system logic can initiate a sequencer in the system logic that does initialize the component for functional operation, if it is designed that way.

Note that this instruction is not allowed to control reset signals to the test logic. Other mechanisms for that are defined in this standard, and the test and system logic should be kept as separate as practical. In particular, neither the TRST* test reset pin nor the internal Reset* or CHReset* signals generated by the TAP and TMP controllers may be controlled by this instruction. In addition, if there is a POR signal generated internal to the component, and it resets either or both the system and the test logic, the POR signal to the system logic may be controlled but the POR signal to the test logic may not be controlled by this instruction, all per rule f) of 8.21.1.

The action of the *IC_RESET* instruction is simply to control the reset signals to the system logic. There are no requirements or constraints placed on the system response to the reset signals so controlled. Often, system resets are designed so that one action or set of actions is taken on the assertion of the reset signal, and other actions taken on the de-assertion of the reset signal. The *IC_RESET* instruction allows that same behavior; the reset signal will be asserted in the *Update-DR* TAP controller state and de-asserted in another *Update-DR* state. It is not required that the entire system reset process take place while the reset signal is asserted, only that the appropriate reset functions are initiated. If both edges of the reset signal are required for proper reset operation, with perhaps a minimum delay between the edges, then the designer can use a PDL procedure to document such requirements (see Annex C) or the second edge can be generated with system logic.

One requirement that the component designer must be aware of is that the system reset functions must honor the *Persistence-On* state of the TMP controller, when it is included, by not altering either the data or the characteristics controlling the component I/O. There are situations that arise during system-level testing where the I/O of a component must be maintained at logic values under test logic control while performing an internal system logic test

such as memory built-in self-test (MEMBIST). If a reset function is needed prior to initiating MEMBIST, then the I/O logic levels and characteristics should be maintained during the reset to maintain proper test control in the system.

If this requirement to honor the TMP controller state cannot otherwise be met, then the reset signal to that logic controlling the I/O should be blocked by the TMP controller *Persistence-On* state to preserve the test mode state of the I/O when the controller is in the *Persistence-On* state.

In many cases, the component designer is not aware of the board- or system-level test requirements. To provide better control of resets from the test logic, the component designer should provide separate control of all reset input pins or other functional reset sources and additional separate control of internal reset functions, even if such functions are all activated in parallel by the system reset I/O pin or are generated from an internal circuit.

9. Test data registers

The test logic architecture contains a minimum of two test data registers (TDRs): the bypass and boundary-scan registers. In addition, the designs of other standard optional test data registers are defined: the device identification, electronic chip identification, initialization data, initialization status, TMP status, and reset selection registers.

The architecture is extensible beyond the TDRs specified in this standard to allow access to any test-support features embedded in the design. These features might include scan-test, self-test registers, or access to key registers in the design (for example, via scannable shadow registers). Additional test data registers need not be intended for public access and use; in which case, the instruction selecting them for scan must be clearly documented as "Private" and the test data register need not follow all of the rules of this clause.

Each named TDR complying with this standard has a defined length after reset [see rule c) and rule d) of 9.2.1] and can be accessed using one or more instructions. The registers can, where appropriate, share circuitry and can be concatenated to form further TDRs, provided that each distinct combination is given a new name (thus, allowing it to meet the defined length requirement).

Excludable and selectable register segments can exist within a TDR as explained in 9.4. The definition of excludable segments provides a standardized implementation of register segments within various power or other domains that may be individually activated or inactivated. The definition of selectable segments provides for a standardized description of register structures like those found in IEEE Std 1500™.[6]

This clause defines the common design requirements for all publically accessible test data registers incorporated in the test logic architecture defined by this standard. Specific design requirements for the bypass, boundary-scan, device identification, electronic chip identification, initialization data, initialization status, TMP control, and reset selection registers are contained in Clause 10 through Clause 12 and Clause 14 through Clause 17, respectively. Subclauses B.8.18 through B.8.21 provide the syntax for describing test data registers, and Annex C provides a language for automating procedural access to the registers.

9.1 Provision of test data registers

9.1.1 Specifications

Rules

a) The group of test data registers shall include, as a minimum, a bypass register and a boundary-scan register designed according to the requirements contained in this clause and in Clause 10 and Clause 11, respectively.
b) Where a device identification register is included in the group of test data registers, it shall be designed according to the requirements contained in this clause and in Clause 12.
c) Where an electronic chip identification is included in the group of test data registers, it shall be designed according to the requirements contained in this clause and in Clause 13.
d) Where an initialization data register is included in the group of test data registers, it shall be designed according to the requirements contained in this clause and in Clause 14.
e) Where an initialization status register is included in the group of test data registers, it shall be designed according to the requirements contained in this clause and in Clause 15.
f) Where a TMP status register is included in the group of test data registers, it shall be designed according to the requirements contained in this clause and in Clause 16.

[6] IEEE publications are available from The Institute of Electrical and Electronics Engineers (http://standards.ieee.org/).

g) Where a reset selection register is included in the group of test data registers, it shall be designed according to the requirements contained in this clause and in Clause 17.

h) All other public test data registers shall be designed according to the requirements contained in this clause.

Permissions

i) Design-specific test data registers may be provided within the group of test data registers to give access to design-specific testability features.

j) Design-specific test data registers may (but need not) be publicly accessible.

NOTE—Any register that is not publicly accessible must be accessed by a private instruction as documented in BSDL using the **INSTRUCTION_PRIVATE** attribute. See B.8.11.

Recommendations

k) Each design-specific test data register should be accessed by a single instruction in order that it can be more easily used with the PDL as described in Annex C.

9.1.2 Description

Figure 9-1 shows the bypass, boundary-scan, and optional test data registers realized as a set of shift-register based elements connected in parallel between a common serial input and a common serial output. Selection of the register that forms the serial path at a given time is controlled from the instruction register. In Figure 9-1, this is shown to be achieved using a multiplexer; however, other implementations are possible.

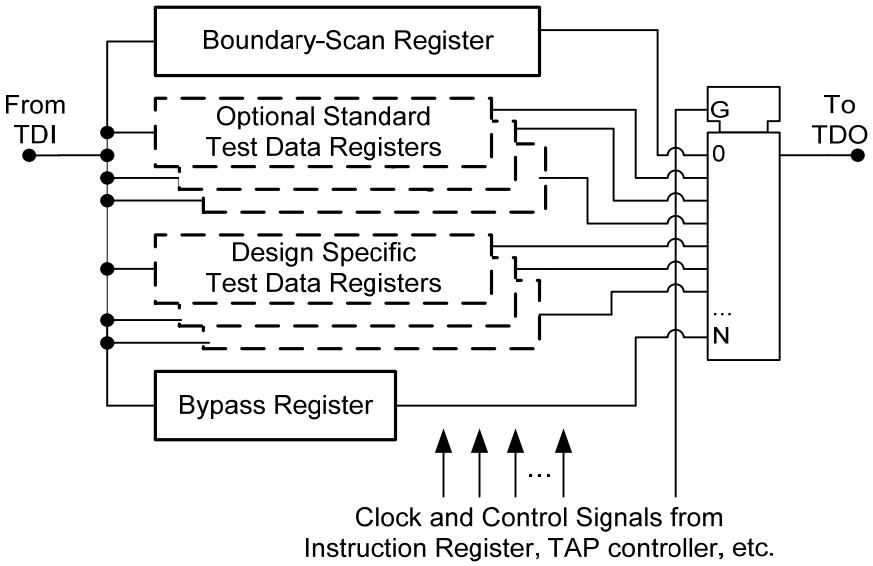


Figure 9-1—Implementation of the group of test data registers

The registers shown in Figure 9-1 are briefly described as follows.

Bypass register

This provides a single-bit, which is the minimum length for any TDR serial connection through the circuit when none of the other test data registers is selected. This register can, for example, be used to allow test data to flow

through a particular component to other components in a product without affecting the normal operation of the particular component. The specification for the bypass register is contained in Clause 10.

Boundary-scan register

This allows testing of board interconnections, detecting typical production defects such as opens, shorts, and so on. It also allows access to the inputs and outputs of components when testing their system logic or sampling of signals flowing through the system inputs and outputs. The specification for the boundary-scan register is contained in Clause 11.

9.1.2.1 Optional standard test data registers

This group of registers is defined in this standard and may include any of the following.

Device identification register

This is an optional test data register that allows the manufacturer, part number, and variant of a component to be shifted out. If this register is included, it must conform to the specification contained in Clause 12.

Electronic chip identification register

This is an optional test data register that allows an identifier unique to this instance of the component to be determined. If this register is included, it must conform to the specifications contained in Clause 13.

Initialization data register

This is an optional test data register that provides parameters for initializing programmable I/O and other circuits requiring initialization prior to boundary-scan interconnect testing. If this register is included, then it must conform to the specifications contained in Clause 14.

Initialization status register

This is an optional test data register that observes the status of the initialization process. If this register is included, then it must conform to the specifications contained in Clause 15.

TMP status register

This is an optional test data register that enables a special transition of the TMP controller state to *Persistence-Off*. If this register is included, then it must conform to the specifications contained in Clause 16. This register is not, and should not be confused with, the Bypass Register.

Reset selection register

This is an optional test data register that selects one or more possible functional reset operations to be performed. If this register is included, then it must conform to the specifications contained in Clause 17.

9.1.2.2 Design-specific test data registers

These optional registers allow access to design-specific test support features in the integrated circuit such as self-tests, scan paths, and so on. They may or may not be made available for public use and access, as the component designer wishes. If they are made available for public use, then they must conform to the specifications in Clause 9.

9.2 Design and construction of test data registers

9.2.1 Specifications

Rules

a) Each test data register shall be given a unique name.

b) The design of each test data register shall be such that, when data are shifted through it, data applied to TDI appears without inversion at TDO after a number of full TCK cycles equal to the current length of the register when the TAP controller is in the *Shift-DR* state.

c) The length of each mandatory or optional standard test data register defined in this standard, or the length of each segment used to assemble such a register, shall be fixed, independent of the instruction by which the test data register is accessed.

d) For programmable components, the lengths of each mandatory or optional standard test data register defined in this standard and the length of all segments of which such registers are assembled shall be independent of the way the component is programmed.

NOTE 1—Most standard test data registers may vary in length, by switching fixed-length segments into and out of the register, for both fixed and programmable components. See permission k) in this subclause and 9.4.

e) Segments used to assemble a mandatory or optional standard test data register shall not overlap and shall not be contained within another segment.

f) Each test data register and each test data register segment shall have a minimum length of one.

g) Each test data register cell shall be able to respond to the TAP controller in accordance with the rules in Clause 9 any time compliance has been established.

Permissions

h) A test data register may be constructed from segments, including segments also used in one or more other registers, provided that the resulting design complies fully with the rules in this standard.

NOTE 2—Per rule a) of this subclause, the resulting combination must be given a name distinct from those of the registers from which it is constructed.

i) Circuitry (including the shift-register paths) in the various test data registers included in a design may be shared between test data registers provided that the rules contained in this standard are met.

j) Unless specifically prohibited by this standard, circuitry contained in test data registers may be used to perform system functions when test operation is not required.

k) A test data register may be constructed from segments that are switched between excluded from or included in the register, and from sets of segments where one segment of each set is selected for inclusion in the register.

NOTE 3—The rules for controlling excludable and selectable segments are detailed in 9.4.

Recommendations

l) Design-specific test data registers should use the logical interface using ungated clocks as defined in Table 9-1.

Table 9-1—Recommended TDR interface for design specific TDRs

Signal	Source	Shared/Unique	Comment
TCK (NOTE 1 and NOTE 5)	from TAP	Common signal	
SI_<TDR> (NOTE 1 and NOTE 5)	TDI or previous segment	Shift chaining input	Changes state on rise of TCK
Shift_<TDR> (NOTE 2 and NOTE 5)	TAP State Decode	Unique control input	Changes state on fall of TCK
Capture_<TDR> (NOTE 2 and NOTE 5)	TAP State Decode	Unique control input	Changes state on fall of TCK
Update_<TDR> (NOTE 2 and NOTE 5)	TAP State Decode	Unique control input	Changes state on rise of TCK
SO_<TDR> (NOTE 3 and NOTE 5)	Scan output from register to next segment or TDO multiplexer	Shift chaining output	Changes state on rise of TCK
Reset*_<TDR> (NOTE 2 and NOTE 5)	from TAP	Common signal	TAP_POR*, TRST*, Reset*, or CHReset*

NOTE 1—See 4.2 and 4.4.

NOTE 2—See Figure 9-4.

NOTE 3—This signal is an output from the TDR back to the TAP and will be multiplexed with other register scan outputs to form TDO.

NOTE 4—This signal, the optional reset signal to the update stages, could be any of Reset*, CHReset*, TRST*, or TAP_POR*. These reset signals are not normally gated with the <TDR> decode.

NOTE 5—"_<TDR>" is a unique label for the register and normally implies a signal qualified by the instruction that selects the register.

9.2.2 Description

While the example implementations contained in this standard show the various test data registers to be separate physical entities, circuitry may be shared between the test data registers provided the rules contained in this standard are met. For example, this would allow the device identification register and the bypass register to share shift-register stages; in which case, the requirements of this standard would be met by operating the common circuitry in two different modes—the device identification register mode and the bypass register mode. Except where identified specifically, the test data registers also may perform system functions and, thus, be a part of the on-chip system logic, when they are not required to perform test functions.

Rule c) of 9.2.1 requires that the length of any test data register defined in this standard, or the segments of which it is assembled, be fixed. The bypass, device identification, and the TMP status registers have specified and fixed lengths. For example, the device identification register shall always contain 32 stages no matter how or when it is accessed. Other test data registers defined in this standard may have segments that can be excluded but must have a defined length after the initialization of the test logic using either TRST* or TAP_POR*. Design-specific registers may have a variable length. Rule d) of 9.2.1 requires that the length of a test data register or its segments also be fixed independently of how a programmable component is programmed.

Table 9-1 defines a standard logical TDR interface that can support TDRs with most common design styles. It utilizes a combination of TAP ports, TAP controller state decodes gated by Instruction Register decodes (see Figure 9-4), and an ungated TCK. This is a logical (that is, useful at the Hardware Description Language level of

design) interface based on signals readily available in most implementations of this standard. While other TDR interfaces are compliant, adoption of this interface should allow simple, and perhaps automated, connection of TDRs to the TAP regardless of the source of the design containing the TDR.

Note that virtual registers are allowed (i.e., a named test data register may be built from circuitry shared with other test data registers or with the system logic). Therefore, requirements may exist for different segments of a single physical register to be accessed for different tests, by different instructions. In these cases, rule a) of 9.2.1 is met by assigning a unique name to each distinguishable register configuration (see Figure 9-2 and Table 9-2). These restrictions are necessary to help avoid unnecessary complication of the software used to generate and apply tests.

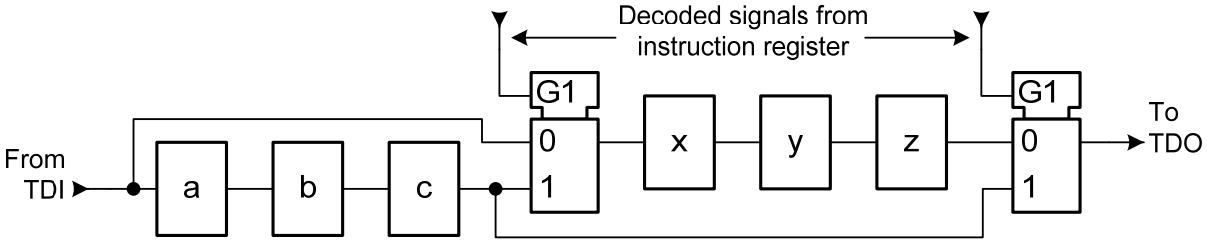


Figure 9-2—Construction of multiple test data registers from shared circuitry

Table 9-2—Naming of test data registers that share circuitry

Test data register name	Stages that form the register
WHOLE_REG	a, b, c, x, y, z
FRONT_REG	a, b, c
BACK_REG	x, y, z

9.2.3 TAP-to-TDR interface

The TAP decodes shown in Figure 6-5 are general and not gated for any specific register. In the simplest implementations of this standard, with only the boundary-scan, bypass, and perhaps the device identification registers, those signals may be used without any further gating. All registers would shift together, increasing power consumption and possibly noise, but as the bypass and device identification register are “read-only” without update latches or registers, the operation would be fully compliant.

In more complex situations, where there are additional optional standard and design-specific registers, then at a minimum the controls or gated clocks of the update latch must be qualified by an instruction decode signal. For several reasons, such as noise, power, and possible issues with TDRs that do not include the optional update latch or register, it is preferred practice to also gate the signals that clock or control capture and shifting of the shift stage of TDRs as well. This helps ensure that the TDR only changes states at appropriate times.

Throughout this standard, the example boundary-scan register cells all assume the gated clock TDR interface (state *Shift-DR* plus gated clocks *Clock-DR* and *Update-DR*). Figure 9-3 shows how, in all but the simplest implementation, the gated clocks would be further gated by an output of the instruction register decoding logic to create the signals *Clock-BSR* and *Update-BSR* that are then distributed to all of the boundary-scan register cells. [The instruction decodes shown implement permission f) of 8.6.1 and permission f) of 8.7.1, where both *SAMPLE* and *PRELOAD* cause the boundary-scan register to both capture and update.] Figure 9-3 serves as an example for other TDRs using the gated TCK approach.

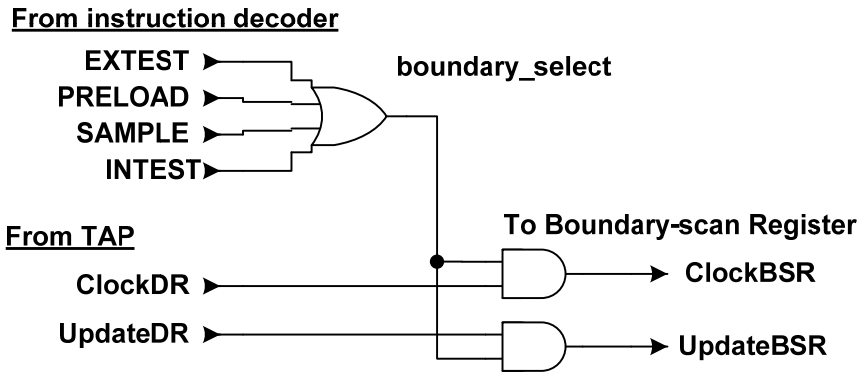


Figure 9-3—Gated-clock boundary-scan register gating

As use of this standard has evolved over the years, the original set of TAP outputs for TDRs, including the two gated clocks *Clock-DR* and *Update-DR*, has worked well for the boundary-scan register cells and the device identification and bypass registers defined in this standard and often is automatically inserted by synthesis tools. However, design-specific test data registers, especially where a TDR is synthesized or embedded in an IP block, have tended toward an interface that uses the ungated TCK. This creates a single clock domain for the test logic and simplifies the timing and synthesis of the circuits. It also allows the TDR to be specified in a hardware description language that is independent of the considerations required for gate-level synthesis and physical placement.

This standard now recommends a standard TDR interface (see Table 9-1) that uses the ungated TCK, especially if the TDR is part of a reusable logic block. This interface is also used in the design examples for some of the optional TDRs in this standard including the TMP status and reset selection registers. Figure 9-4 shows example gating logic for interfacing the general TAP outputs of Figure 6-5 with the TDR-specific standard TDR inputs.

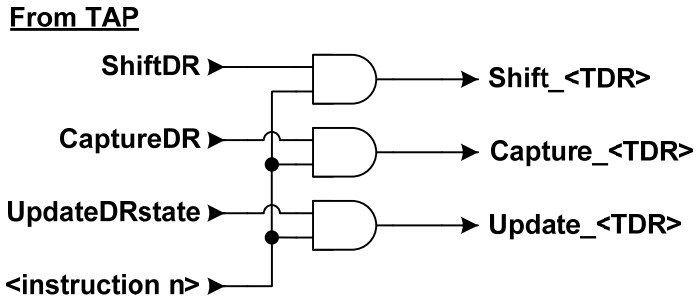


Figure 9-4—Test data register control gating

The <instruction n> input is the output from the instruction decoder (see Figure 7-2 for an example) for the instruction(s) selecting the TDR with the unique identifier <TDR>. This interface logic may be built into the TAP, into the TDR, or provided in a separate logical entity. In many cases, this logic will be created by the same tools that build the TAP and connect to TDRs.

9.2.4 Test data register cell design examples

Test data register cells come in essentially four basic forms. In every case, a cell will have a shift register stage that shifts on the rising edge of every TCK in the *Shift-DR* TAP controller state and holds the data stable at least until the instruction selecting the register is no longer active. The first form of the cell is just that shift register driving its data to the parallel output of the cell and is referred to here as a shift-only cell. The second form of the cell adds to the shift register the capability of capturing data from the cell parallel input on the rising edge of TCK in the *Capture-DR* TAP controller state and is referred to here as a capture cell. The third form of the cell adds to the shift register

stage the capability of transferring the data from the shift register to a hold latch or register on the falling edge of TCK in the *Update-DR* TAP controller state and is referred to here as an update cell. The fourth form adds to the shift register both the capture and update cell capabilities and is referred to here as a capture-update cell. An example of each of these forms is shown below.

When the update latch or register is present, it may optionally be set or reset by the *Test-Logic-Reset* TAP controller state (signal Reset*) or by the *Test-Logic-Reset* TAP controller state only when the TMP controller is in the *Persistence-Off* state (signal CHReset*), or by either the TRST* TAP port signal or the TAP_POR*, whichever initializes the TAP controller. Note that, in general, resets may not be needed and should be used only where needed, and with the appropriate reset signal. Experience has shown, for instance, that the *Test-Logic-Reset* TAP controller state may be entered frequently during testing, which may make Reset* a poor choice for resetting the TDR if the data it holds may be needed later. Where the update stage is used, the optional reset is shown in the examples.

The register may be shared with other test data registers, or it may have a system function when the test logic is not active. This will significantly change the implementation details and is not shown in the examples.

Additionally, there are different approaches to clocking the cells: using either an ungated TCK or a gated TCK such as the *Clock-DR* signal generated in the TAP. Signals supporting both clocking schemes are shown as outputs from the example TAP in Figure 6-5.

Gated-clock example TDR bit

Figure 9-5 shows a possible gate-level example of a control-update TDR cell built using the gated clock implementation that shifts only when selected. The apparent simplicity of this example stems from the fact that the appropriate TAP controller state decodes have been used to gate the two clocks within the TAP (see Figure 6-5). The TAP-to-TDR interface logic, equivalent to that shown for the boundary-scan register in Figure 9-3, is included in this figure outside the dashed box of the TDR cell, but again it could be implemented in the TAP, the TDR, or another block in between. Again, <instruction n> is the decode of the instruction that selects the TDR for scan, and an example of its generation is shown in Figure 7-2.

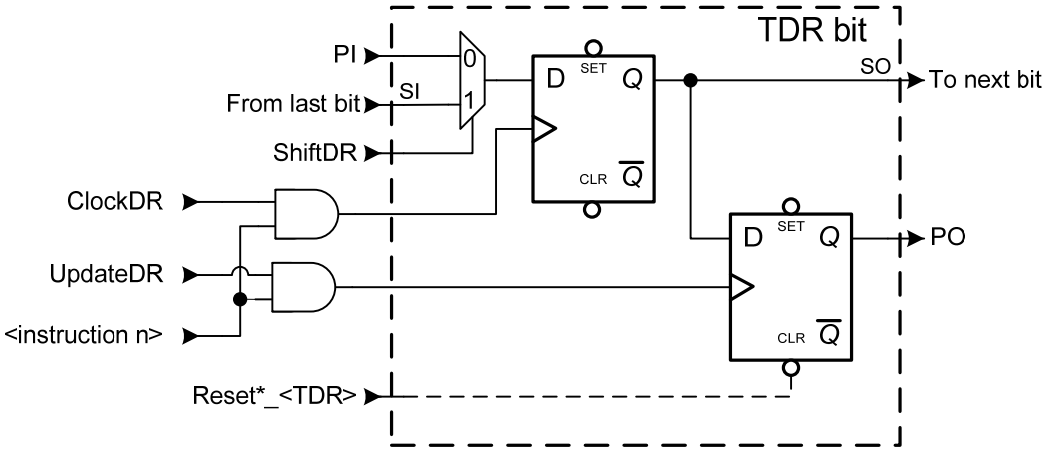


Figure 9-5—Capture-update TDR cell using gated clocks

Ungated-clock example TDR bits

This standard recommends a standard TDR interface using ungated TCK (see Table 9-1). Figure 9-6 through Figure 9-9 illustrate the use of this interface using the outputs of the TAP to TDR interface logic in Figure 9-4. For each figure, VHDL and Verilog code segments are also provided. The figures show one possible gate-level implementation. In each gate-level example, PI is the parallel input for capture, PO the parallel output to the test

logic, “From last bit” and “To next bit” are the scan chain through the TDR, and the other signals are as defined in Table 9-1.

Ignoring all the language structural requirements for inputs and outputs of entities or modules, segments of VHDL and Verilog for coding each form of the TDR cell with an ungated TCK as input are shown below. The optional reset is included if the update stage is included.

First is the capture-update (read-write) form with optional reset.

```
------------------------------------------------------------
   -- VHDL code segment, see Figure 9-6
   if (TCK'event and TCK='1') then
      if CaptureTdrBit = '1' then
         tdr_cap <= pi;       -- Capture parallel input
      elseif ShiftTdrBit = '1' then
         tdr_cap <= si;       -- Shift data TDI->TDO
      end if;

   if (Reset_tdr='0') then
       tdr_upd <= 0;
   elseif (TCK'event and TCK='0') then
      if (UpdateTdrBit='1')
         tdr_upd <= tdr_cap;  -- Update from shift/capture flop
      end if;
   end if;

   po <= tdr_upd;
   so <= tdr_cap;

------------------------------------------------------------
   // Verilog code segment, see Figure 9-6
   always @(posedge TCK)            // Shift/Capture flop
      if (CaptureTdrBit)
         tdr_cap <= pi;       // Capture parallel input
      else if (ShiftTdrBit)
         tdr_cap <= si;       // Shift data TDI->TDO

   always @(negedge TCK, negedge Reset_tdr)     // Update flop
      if (~Reset_tdr)
         tdr_upd <= 0;        // (Optional) reset of Update flop
      else if (UpdateTdrBit)
         tdr_upd <= tdr_cap;  // Update from shift/capture flop

   assign po = tdr_upd;
   assign so = tdr_cap;

------------------------------------------------------------------
```

Figure 9-6 illustrates this test data register capture-update cell and has an update register that can optionally be reset by the *Test-Logic-Reset* TAP controller state. This example includes essentially all of the capabilities listed above, other than sharing the register with system logic. Most test data register cells will be a subset of this one.

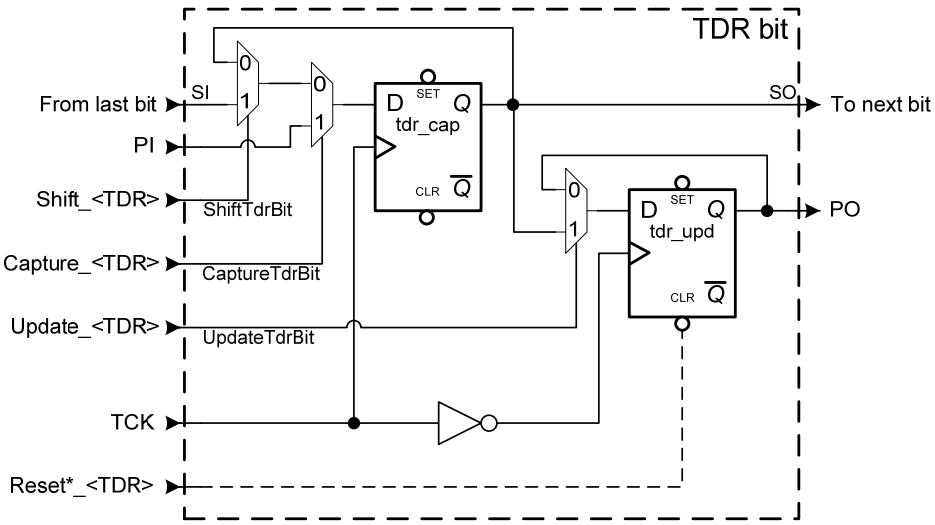


Figure 9-6—Capture-update TDR cell with nongated clock and optional reset

The second form is the "write-only" cell that does not capture PI data and is shown in Figure 9-7. The update register is usually provided so the PO data remain constant while the shift register is being shifted. The only difference is the simpler selection logic providing the data input to the shift register.

```
-------------------------------------------------------------
   -- VHDL code segment, see Figure 9-7
   if (TCK'event and TCK='1') then
      if ShiftTdrBit = '1' then
         tdr_cap <= si;       -- Shift data TDI->TDO
      end if;

   if (Reset_tdr='0') then
       tdr_upd <= 0;
   elseif (TCK'event and TCK='0') then
      if (UpdateTdrBit='1')
         tdr_upd <= tdr_cap;  -- Update from shift/capture flop
      end if;
   end if;

   po <= tdr_upd;
   so <= tdr_cap;

-----------------------------------------------------------------
   // Verilog code segment, see Figure 9-7
   always @(posedge TCK)            // Shift/Capture flop
      if (ShiftTdrBit)
         tdr_cap <= si;       // Shift data TDI->TDO

   always @(negedge TCK, negedge Reset_tdr)     // Update flop
      if (~Reset_tdr)
         tdr_upd <= 0;        // (Optional) Reset
      else if (UpdateTdrBit)
         tdr_upd <= tdr_cap;  // Update from shift/capture flop

   assign po = tdr_upd;
   assign so = tdr_cap;
-----------------------------------------------------------------
```

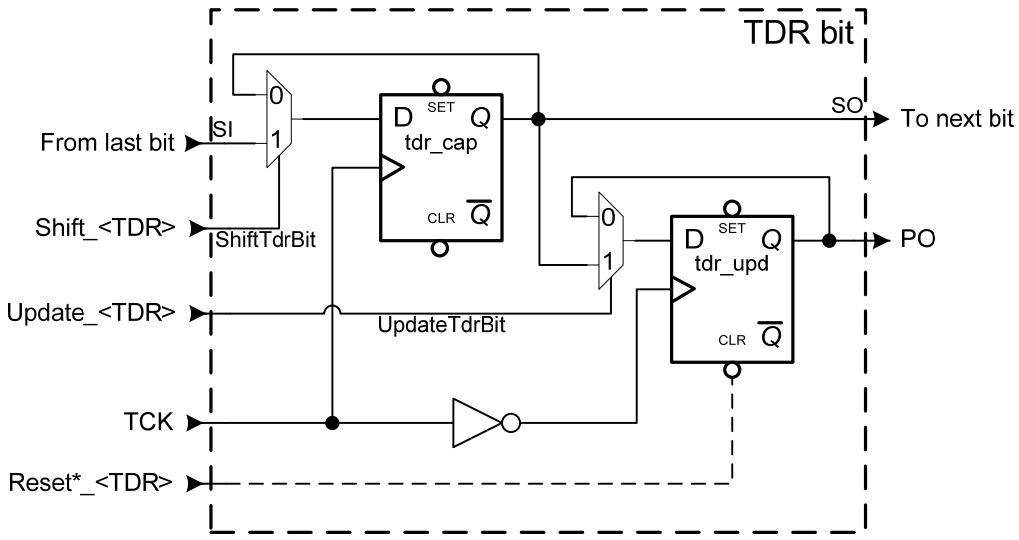


Figure 9-7—Update TDR cell without capture and with nongated clock and optional reset

One item of note on the first three example TDR cells, Figure 9-5 through Figure 9-7: the "Reset*_<TDR>" signal (Reset_tdr in the code segments) is not a signal gated by the TDR decode, but it may be sourced from any of four possible reset signals: Reset*, illustrated in Figure 6-5, will reset the update register any time the TAP controller enters the *Test-Logic-Reset* state. If the register is designed to be reset, but it is one of those that needs to be held when the TMP controller is provided and in the *Persistence-On* state, then the "CH-Reset*", illustrated in Figure 16-1, would be used instead. Alternatively, when the TDR only needs to be reset at power-up, the TRST* TAP port or TAP_POR*, whichever is used to reset the TAP controller, may be used. Again, TDRs should only be provided with a reset when needed.

The third form shows a "read-only" cell that captures data and may optionally drive the PO output from the scan stage and is shown in the code samples and Figure 9-8. Note that the parallel output data will not be stable during shift operations. The logic is simpler still.

```
-------------------------------------------------------------
   -- VHDL code segment, see Figure 9-8
   if (TCK'event and TCK='1') then
      if CaptureTdrBit = '1' then
         tdr_cap <= pi;       -- Capture parallel input
      elseif ShiftTdrBit = '1' then
         tdr_cap <= si;       -- Shift data TDI->TDO
      end if;

   -- po <= tdr_cap;          -- Optional PO without update
   so <= tdr_cap;
-------------------------------------------------------------
   // Verilog code segment, see Figure 9-8
   always @(posedge TCK)            // Shift/Capture flop
      if (CaptureTdrBit)
         tdr_cap <= pi;       // Capture parallel input
      else if (ShiftTdrBit)
         tdr_cap <= si;       // Shift data TDI->TDO

   // assign po = tdr_cap;    // Optional PO without update.
   assign so = tdr_cap;
-------------------------------------------------------------
```

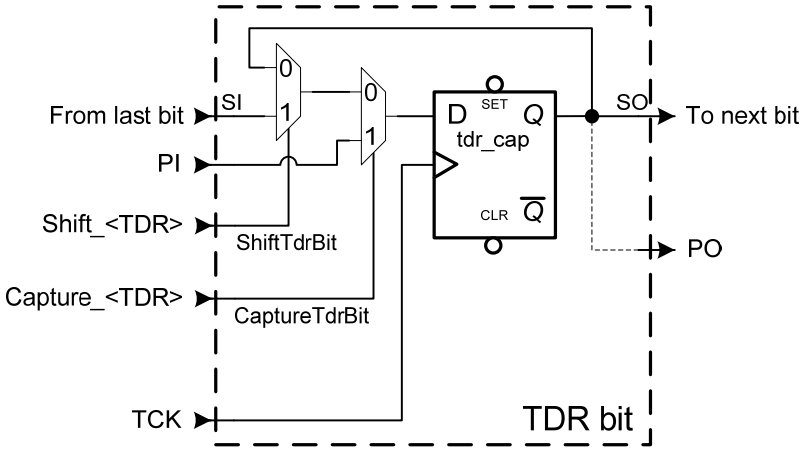


Figure 9-8—Capture TDR cell with nongated clock and without update stage

The fourth and final form is a single shift stage without an update stage. This can be used as a "scan-only" cell, or again as a "write-only" TDR where the PO output is not stable during shift operations. It is the simplest yet, as shown in the code and Figure 9-9.

```
-------------------------------------------------------------
   -- VHDL code segment, see Figure 9-9
   if (TCK'event and TCK='1') then
      if ShiftTdrBit = '1' then
         tdr_cap <= si;       -- Shift data TDI->TDO
      end if;

   po <= tdr_cap;
   so <= tdr_cap;

-----------------------------------------------------------------
   // Verilog code segment, see Figure 9-9
   always @(posedge TCK)            // Shift/Capture flop
      if (ShiftTdrBit)
         tdr_cap=si;                // Shift data TDI->TDO

   assign po = tdr_cap;
   assign so = tdr_cap;

-----------------------------------------------------------------
```

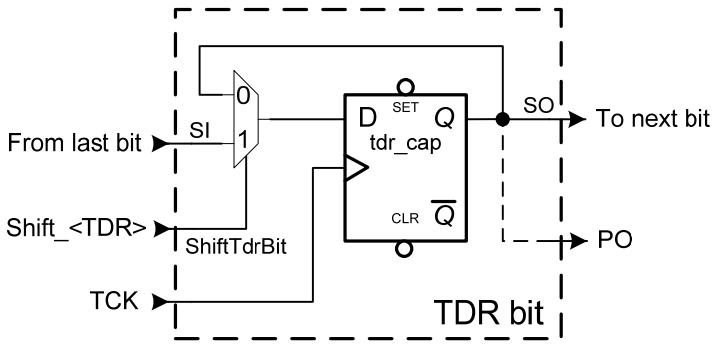


Figure 9-9—Shift-only TDR cell with nongated clock and without update stage.

Care should be taken when using the last two forms (Figure 9-8 and Figure 9-9) as the parallel outputs will toggle often during shifting. The test logic fed by PO must be designed to take this behavior into account and, if necessary, prevent problems.

There are two additional specialized variations on the capture-update cell documented in this standard: a self-monitoring cell that captures its own update latch so the current value can be observed, and a self-resetting cell, which will produce a pulse as its output instead of a level.

The self-monitoring cell is shown in Figure 9-10. When the cell with an Update capability does not otherwise require the capturing of the parallel input (PI), then the capture capability can be used to capture the value present in the Update stage during the TAP controller state *Capture-DR*. This improves the observability of the test logic, and it allows a user to verify the value actually being driven by the cell on the parallel output (PO) is as expected. For instance, if a reset has occurred, the value in the Update stage may not be the same as the value last shifted in. If there is a defect in the cell, this monitor can detect a stuck defect.

```
------------------------------------------------------------
   -- VHDL code segment, see Figure 9-10
   if (TCK'event and TCK='1') then
      if CaptureTdrBit = '1' then
         tdr_cap <= tdr_upd ;       -- Capture Update stage
      elseif ShiftTdrBit = '1' then
         tdr_cap <= si;       -- Shift data TDI->TDO
      end if;

   if (Reset_tdr='0') then
       tdr_upd <= 0;
   elseif (TCK'event and TCK='0') then
      if (UpdateTdrBit='1')
         tdr_upd <= tdr_cap;  -- Update from shift/capture flop
      end if;
   end if;

   po <= tdr_upd;
   so <= tdr_cap;

------------------------------------------------------------
   // Verilog code segment, see Figure 9-10
   always @(posedge TCK)            // Shift/Capture flop
      if (CaptureTdrBit)
         tdr_cap <= tdr_upd;        // Capture parallel input
      else if (ShiftTdrBit)
         tdr_cap <= si;       // Shift data TDI->TDO

   always @(negedge TCK, negedge Reset_tdr)     // Update flop
      if (~Reset_tdr)
         tdr_upd <= 0;        // (Optional) reset of Update flop
      else if (UpdateTdrBit)
         tdr_upd <= tdr_cap;  // Update from shift/capture flop

   assign po = tdr_upd;
   assign so = tdr_cap;

------------------------------------------------------------------
```

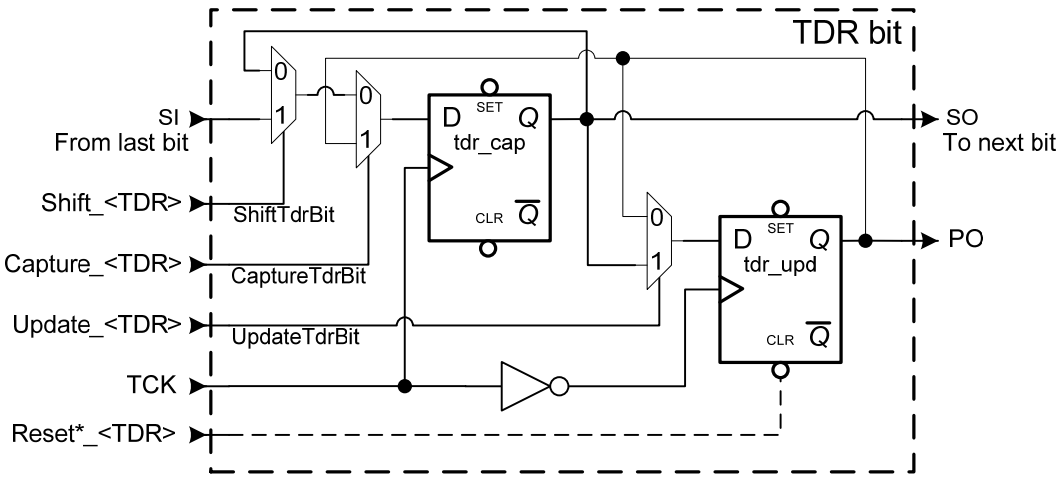


Figure 9-10—Self-monitoring TDR cell with update stage and nongated clocks

The self-resetting cell (also with self-monitoring capability) is shown in Figure 9-11. When this cell has a 1 value in the shift stage, it will produce a "0-1-0" pulse on PO and a "1-0-1" pulse on the inverse of PO (called Pulse1_PO and Pulse0_PO in Figure 9-11 and in the code examples). The pulse changes state on the consecutive falling edge of TCK. This signal can then be directly used to gate TCK to other registers clocked by TCK, either using actual clock gating or using the data-hold illustrated in these cells. Figure 9-12 shows the timing of the output pulse and the monitor stage in and around the *Update-DR* TAP controller state. Note that this cell would violate rule e) of 9.3.1 if PO were designed to be driven off-chip.

In order to support the expected behavior of this cell, the reset input is connected to the Reset* TAP controller output to clear the update and monitor cells at power-up and whenever the test logic is reset.

The self-monitor is designed, in this case, to capture the value on PO on the rising edge of TCK following the transfer of shift data to the Update stage on the falling edge of TCK. This should, unless there has been a reset in the meantime, reflect the value last shifted into the cell. If the monitoring is not desired, or the cell is required to capture other data on the PI input, the monitoring stage may be deleted.

```
-------------------------------------------------------------
   -- VHDL code segment, see Figure 9-11
   if (TCK'event and TCK='1') then
      if CaptureTdrBit = '1' then
         tdr_cap <= monitor;        -- Capture monitor stage
      elseif ShiftTdrBit = '1' then
         tdr_cap <= si;       -- Shift data TDI->TDO
      end if;

   if (Reset_tdr='0') then
       tdr_upd <= 0;
   elseif (TCK'event and TCK='0') then
  -- Update from shift/capture flop, clear after Update-DR
      tdr_upd <= UpdateTdrBit AND tdr_cap;
   end if;

   if (Reset_tdr='0') then
       monitor <= 0;
   elseif (TCK'event and TCK='1') then
      if (UpdateTdrBit='1')
         monitor <= tdr_upd;  -- Update from shift/capture flop
```

```
      end if;
   end if;

   Pulse1_PO <= tdr_upd;
   Pulse0_PO <= NOT tdr_upd;
   so <= tdr_cap;

------------------------------------------------------------
   // Verilog code segment, see Figure 9-11
   always @(posedge TCK)            // Shift/Capture flop
      if (CaptureTdrBit)
         tdr_cap <= monitor;        // Capture monitor stage
      else if (ShiftTdrBit)
         tdr_cap <= si;       // Shift data TDI->TDO

   always @(negedge TCK, negedge Reset_tdr)     // Update flop
      if (~Reset_tdr)
         tdr_upd <= 0;        // (Optional) reset of Update flop
   // Update from shift/capture flop, clear after Update-DR
      else
         tdr_upd <= UpdateTdrBit & tdr_cap;

   always @(posedge TCK, negedge Reset_tdr)     // Update flop
      if (~Reset_tdr)
         monitor <= 0;        // (Optional) reset of Update flop
      else if (UpdateTdrBit)
         monitor <= tdr_upd;  // Update from shift/capture flop

   assign Pulse1_PO = tdr_upd;
   assign Pulse0_PO = ~tdr_upd;
   assign so = tdr_cap;

------------------------------------------------------------------
```

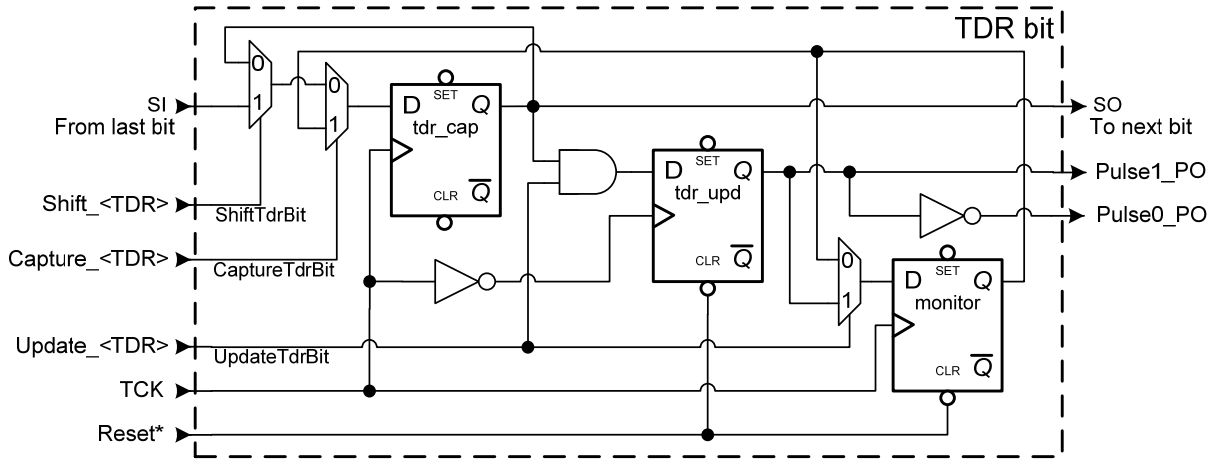


Figure 9-11—Self-resetting and self-monitoring TDR cell with nongated clocks

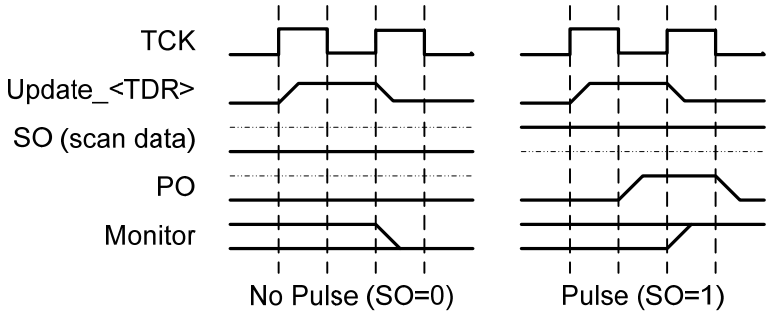


Figure 9-12—Timing of a self-resetting and self-monitoring TDR cell at *Update-DR*

9.3 Operation of test data registers

9.3.1 Specifications

Rules

a) Each instruction shall identify a test data register that will be serially connected between TDI and TDO.

NOTE 1—Use of PDL (see Annex C) creates a preference for one and only one instruction selecting a given TDR.

b) The test data register connected between TDI and TDO shall shift data one stage toward TDO after each rising edge of TCK in the *Shift-DR* controller state.

c) When the TAP controller is in the *Test-Logic-Reset* state and the TMP controller, if present, is in the *Persistence_off* state, all test data registers shall be set so that either they perform their system function (if one exists) or they do not interfere with the operation of the on-chip system logic.

d) Where a test data register is required to load data from a parallel input in response to the current instruction, these data shall be loaded on the rising edge of TCK in the *Capture-DR* controller state.

e) Test data registers enabled to drive data (signal values) off-chip shall be designed such that component outputs change only:

1) On the falling edge of TCK after entry to the *Update-DR*, *Update-IR*, *Run-Test/Idle*, or *Test-Logic-Reset* controller state as a result of signals applied at TCK and TMS.

NOTE 2—This may require that the register be provided with a latched parallel output.

2) Immediately on entry into the *Test-Logic-Reset* controller state as a result of a logic 0 being applied at TRST*.

f) Where a test data register is required to operate in response to the *RUNBIST* instruction, the required operation shall occur in the *Run-Test/Idle* controller state.

g) Where no operation of a selected test data register is required in a given controller state in response to the current instruction, the register shall retain its last state.

h) Test data registers that are not selected by the current instruction shall be set so that either they perform their system function (if one exists) or they do not interfere with the operation of the on-chip system logic.

i) Test data registers not enabled for shifting between TDI and TDO by the active instruction shall maintain their state in the *Update-DR* TAP controller state.

Permissions

j) In addition to the test data register enabled for shifting between TDI and TDO, an instruction may select further test data registers.

Recommendations

k) Test data registers not enabled for shifting between TDI and TDO by the active instruction should maintain their state in the *Capture-DR* and *Shift-DR* TAP controller states.

9.3.2 Description

These requirements define the correct test data register operation in conjunction with the TAP and the TAP controller.

When there are multiple power domains on a component, some of which may be powered down, then in order for the test data registers to comply with the above rules of operation, they must either be in a power domain that is always on or is always on whenever the power domain containing the TAP controller is on. The exception is excludable register segments per permission k) of 9.2.1, which have additional rules in 9.4.

Note that while an instruction may select the test operation of more than one test data register, there must be one and only one such selected test data register between TDI and TDO. All other selected test data registers retain their state in the *Shift-DR* controller state. One use of this capability is as follows.

Consider the case where several test data registers need to be accessed in sequence in order to establish the starting conditions for a test or to examine test results. As an example, Table 9-3 shows the sequence of events (starting from the *Test-Logic-Reset* controller state) that would be required if two design-specific test data registers and the boundary-scan register needed to be accessed in order to execute an instruction. Figure 9-13 shows the design of the group of test data registers for this example.

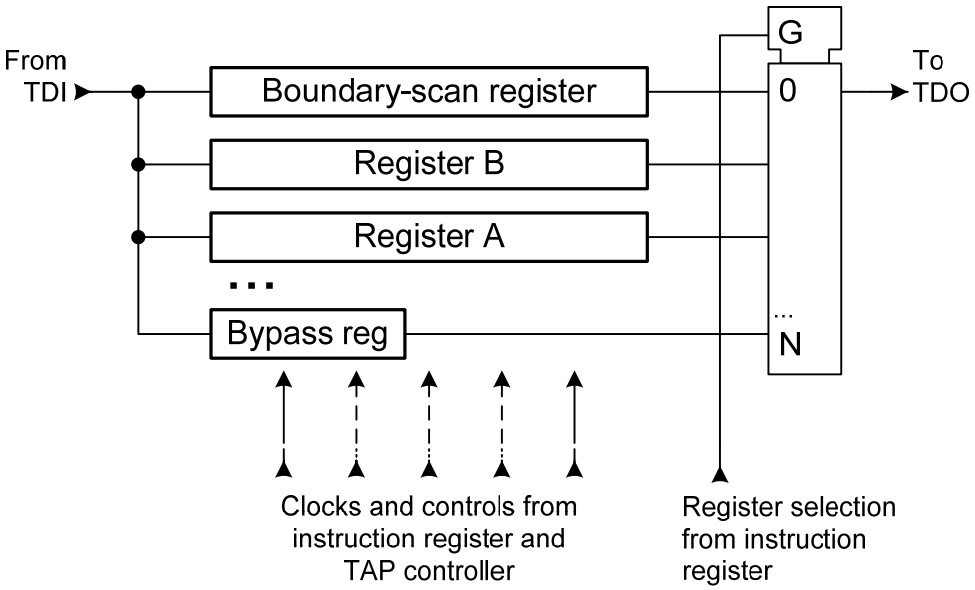


Figure 9-13—Example design containing two optional test data registers

In step 4, serial access to register B is required in order to set its initial condition as required for execution of the test. However, register A was set to its required initial condition in step 2, and so it is necessary to design the test logic such that register A can retain its state between steps 2 and 7 (when the self-test is executed) while register B and the boundary-scan register are accessed. Similarly, the test logic must be designed so that register B retains the initial condition set during step 4 until step 7 and so that the reverse sequence of events can occur after completion of execution of the self-test.

Note that the design of the test logic may require that test data registers be accessed in a fixed sequence in order to achieve the desired result. For example, the test logic may not allow register B to retain its state while register A is scanned.

Table 9-3—Sequential access to test data registers

Step	Action
0	Test logic inactive in the *Test-Logic-Reset* controller state.
1	Enter instruction that selects register A for connection between TDI and TDO.
2	Scan required initial values into register A.
3	Enter instruction that selects register B for connection between TDI and TDO and keeps register A in its test mode of operation.
4	Scan required initial values into register B. Register A retains its state.
5	Enter the test instruction that selects test operation of registers A and B and connects the boundary-scan register between TDI and TDO.
6	Scan required values for the component inputs and outputs into the boundary-scan register. Registers A and B retain their state.
7	Execute the instruction by entering the *Run-Test/Idle* controller state.
8	Enter instruction that selects register B for connection between TDI and TDO and keeps register A in its test mode of operation.
9	Scan test results out of register B. Register A retains its state.
10	Enter instruction that selects register A for connection between TDI and TDO.
11	Scan test results out of register A.

9.4 Design and control of test data register segments

A test data register may be constructed as a chain of multiple segments, some of which are always scanned while others, called excludable segments and selectable segments, and defined below, are scanned only in particular situations. Restrictions on the use of these segments for the test data registers defined in this standard are also defined below.

The simple ability to exclude or include a defined register segment is intended to support registers or portions of registers passing through power domains that may be powered down, but it is not restricted to that use. The boundary-scan, initialization data and status registers, ECID, and reset selection registers defined in this standard are limited to un-nested excludable segments. Other TDRs defined in this standard, such as the bypass, idcode, and TMP status registers, are not allowed to have excludable segments. Design-specific TDRs may have excludable and nested excludable segments (excludable segments completely within another excludable segment).

Design-specific TDRs may also have selectable segments, where encoded selection fields select one or more of several segments for parallel scan-in, and select exactly one segment for scan-out. This parallel structure is similar to the structure of the TDRs defined in this standard, where one of multiple, parallel TDRs is selected by the instruction register for scanning and is connected to TDO. (All may be connected to TDI.) This may be replicated on a smaller scale within a single TDR. A good, and well-defined example of this is the wrapper serial port (WSP) of IEEE Std 1500, with a wrapper instruction register (WIR) selecting one of several wrapper data registers (WDRs) to place in the scan chain. Only design-specific TDRs are allowed to have selectable segments.

The following aspects of this definition help ensure the integrity of the test data scan path:

— All excludable segments are initialized to the excluded state, and any encoded selection field is initialized to a specified state selecting a documented (public) segment.
— For simple excludable segments, provisions are made to capture whether the segment is ready to scan or not, so the segment would not be included when it could break the scan chain.
— Special register fields are defined in the BSDL Standard Package (see B.9) for use in a test data register to provide consistent control of the excludable segments:

i) A “segment-select” field to both capture a “ready to scan” indication and control the inclusion of a simple excludable segment.
ii) A zero-length “segment-start” field to mark the beginning of an excludable segment.
iii) A zero-length “segment-mux” field to mark the end of an excludable segment.

— The outputs of the segment-select field (for excludable segments) and the selection field (for selectable segments) are delayed to prevent a race condition between the update signals to the segments and the selection criteria gating those signals.

No additional instructions are necessary beyond the one that selects the test data register for scanning. In the initialized state, the TDR must still have a nonzero length, and that is the length used to determine the documented length of the register.

In the case of power domains that may be powered down (or other domain types that may need to be “made ready” before a TDR can be scanned), another special test data register cell (a “domain-control” cell) is provided. This cell is only used if the power (or other domain type) is controlled on-chip, and may be controlled through the TAP for test purposes. The latched parallel output of the domain-control cell must be designed to override the functional signal to the power or domain controller and cause it to power up the domain or otherwise make it ready for scanning.

The domain-control and segment-select fields and the encoded segment selection field may be in the same TDR as the excludable or selectable segments, or they may be in another TDR (such as the initialization data register) that acts to configure the component for test. All of the control fields and controlled segments must be in public (standard or design-specific) and documented TDRs so that tools can find them. Duplicate control fields, each having the same effect, are allowed in different public TDRs. Note that boundary-scan and initialization data register segment domain-control and segment-select fields are required to be in the same register as the segments, although duplicates are allowed elsewhere.

9.4.1 Specifications

Rules

a) Any excludable test data register segment taking advantage of permission k) of 9.2.1 shall be immediately preceded (closest to TDI) by a segment-select cell or a segment-start field and immediately followed by the associated segment-mux switching circuit.

NOTE 1—See B.8.21 for a full discussion on documenting these fields and segments.

b) No excludable segment shall appear in the bypass, device identification, or TMP status standard registers.
c) No additional excludable segment shall appear within an excludable segment in the boundary-scan, initialization data or status, ECID, or reset selection standard registers.
d) Any excludable segment that appears within another excludable or selectable segment shall be completely enclosed within that segment.
e) The segment-select cell for an excludable segment shall have capture, update, and reset capabilities (see the example in Figure 9-19).
f) The segment-select cell for an excludable segment shall load, at the rising edge of TCK in the *Capture-DR* TAP controller state, a signal indicating whether the segments are ready to scan (a logic 1) or not (a logic 0).
g) The latched parallel output of a segment-select cell for an excludable segment shall control the switching element after the end of the segment, and the value in the parallel output shall either exclude (0) or include (1) the segment.
h) When an excludable segment is dependent on an on-chip controller to make it ready to scan, a domain-control cell with a latched parallel output shall be provided.
i) The output of a domain-control cell shall be connected to the on-chip domain controller, and a logic 1 in the domain-control cell shall cause the domain controller to enable the domain for its test function.

NOTE 2—It is the intent that the domain-control signals to an on-chip power controller would have priority over mission mode signals just as IEEE 1149.1 test modes have priority over the mission mode. The power controller is permitted to not honor domain control signals only when doing so is impossible or could damage the component, and such cases would be documented by the component designer.

NOTE 3—In a case where a boundary-scan register segment and the associated I/O are in separately controlled power domains, then more than one domain controller could be designed to respond to a single override signal.

j) The domain-control cell shall have update and reset capabilities (see the example self-monitoring cell in Figure 9-10 or the example update cell in Figure 9-7).

k) For each domain-control override signal connected to an on-chip domain controller, the domain controller shall output a status signal to be captured in the segment-select cell and shall have a value of 1 when the domain is ready (regardless of the state of the signal from the domain-control cell) and a value of 0 when the domain is not ready.

l) Once the domain has been enabled for test, it shall remain enabled as long as the domain-control cell remains set to 1.

NOTE 4—If the on-chip domain-controller cannot respond at the same time to all of a set of override signals, then any established overrides must be maintained and any conflicting overrides ignored. An established override must be explicitly de-selected (with a TDR scan or a reset of the domain control cell) before requesting a different override when the two cannot be turned on at the same time. The fact that the override is being ignored is reflected in the status signal returned from the domain-controller to the segment-select cell.

m) Any selectable set of test data register segments taking advantage of permission k) of 9.2.1 shall be documented and the documentation shall include:

1) The association of each set with an encoded selection field or fields.
2) The encoded values required to select for scan-out each and every segment in the set.
3) The encoded values required to select for scanning each desired subset of segments in the set.

NOTE 5—See B.8.20 and B.8.21 for rules concerning documentation of excludable and selectable segments in BSDL.

n) A selectable set of segments shall not appear in the bypass, device identification, TMP status, boundary-scan, initialization data or status, or reset selection standard registers.

o) Any selectable set of segments that appears within another excludable or selectable segment shall be completely enclosed within that segment.

p) Each segment in a selectable set of segments shall receive the same scan-in data.

q) The selection field cells associated with a selectable set of segments shall have update and reset capabilities (see the example in Figure 9-19).

NOTE 6—Capture capabilities are optional, but self-monitoring cells would be highly desirable in order to scan out the current selection value.

r) The decoded values of the latched parallel output of the selection field cells associated with a selectable set of segments shall select the scan-out of no more than one segment for connection to the scan chain.

NOTE 7—There is no requirement that all possible decodes of the selection field be used; it may be one-hot encoded, for example. Unused decodes produce undefined results and could be used for “private,” undocumented segments, be reserved for future expansion, and so on.

s) To provide time for the test data register and segment cells to complete update operations, when a segment-select cell or segment selection field is scanned at the same time as the controlled segments, the latched parallel output of the segment-select cell or segment selection field shall be delayed to avoid race conditions in the operation and switching of the test data register segments, and the delay shall be no more than two TCK cycles.

NOTE 8—Figure 9-19 illustrates such a cell with a one TCK cycle delay. This rule explicitly excludes structures such as the WIR of IEEE Std 1500, which are not scanned at the same time as the WDRs that they select.

t) All segment-select, domain-control, and selection fields associated with excludable or selectable segments shall be dedicated test logic.

u) Each segment-select cell and domain-control cell shall be reset to a logic value of 0, and each selection field associated with a set of selectable segments shall be reset to a specified value by one of the following:

1) The *Test-Logic-Reset* TAP controller state when the TMP controller is provided and in the *Persistence-Off* state (i.e., the CHReset* signal shown in Figure 6-10).

2) The *Test-Logic-Reset* TAP controller state whether or not the TMP controller is provided (i.e., the Reset* signal shown in Figure 6-5).

3) The TRST* TAP port and/or the operation of power-up reset (also known as power-on reset or POR) circuitry built into the test logic, as used to reset the TAP controller (see 6.1.3).

NOTE 9—Each segment-selector, domain control, and selection field instance can be reset by a different choice; this rule does not require all such instances to use the same reset signal.

v) When excludable segments are excluded, or selectable segments are not selected for scan, such segments shall not respond to the *Update-DR* TAP controller state.

w) When excludable boundary-scan register segments are excluded, they shall have no effect on the flow of signals between the system pins and the on-chip system logic.

x) Any segment-select cell or selection field contained in an excludable or selectable segment shall be in a known state prior to the containing segment being included or selected.

Permissions

y) When a domain controller requires more than just the override signal from the domain controller cell to make the segment ready for scan (such as a fixed delay), such requirements may be documented in a PDL procedure. (See Annex C.)

z) When excludable segments are excluded, or selectable segments are not selected for scan, such segments may perform other system or test functions, and excluded segments may be powered down.

Recommendations

aa) When excludable segments are excluded, or selectable segments are not selected for scan, such segments should not respond to the *Capture-DR* and *Shift-DR* TAP controller states.

bb) Any segment-select cell contained in an excludable segment that can be powered down should automatically reset at power-up.

cc) Any domain incompatibilities or external requirements that would prevent all domains from being enabled at the same time should be documented by the component designer.

dd) If the selection field cells associated with a selectable set of segments do not otherwise capture data, they should be self-monitoring cells (see example Figure 9-10) in addition to the required capabilities.

9.4.2 Description

Excludable segments

Segments of a test data register are made excludable by means similar to that shown in Figure 9-14.

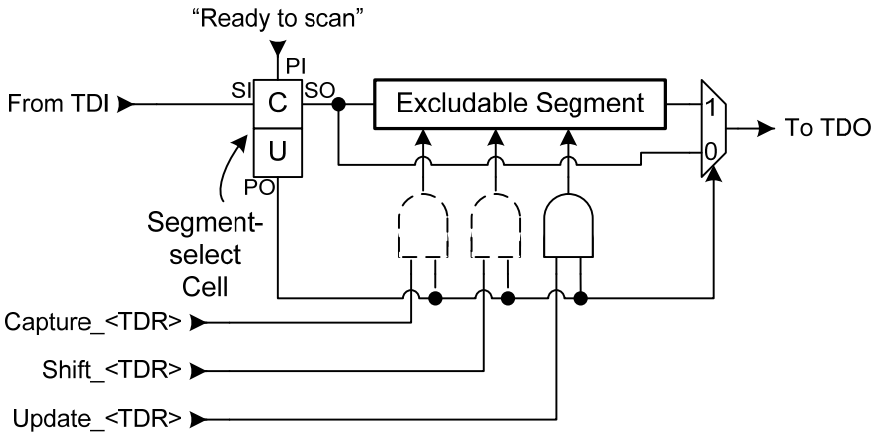


Figure 9-14—Scan control of excludable test data register segments

The excludable segment of the test data register is controlled by a segment-select cell (named a SegSel in Annex B of this standard) with capture, update, and reset capabilities as shown in Figure 9-14. (The clocks and asynchronous resets are not shown for simplicity.) The excludable segment is immediately followed by a switching element, represented in Figure 9-14 with a multiplexer (named a SegMux in Annex B of this standard). When the segment is excluded, at least the Update_<TDR> signal to this segment of the test data register, if used, must be disabled, and the Capture_<TDR> and Shift_<TDR> signals would preferably be disabled, as shown by the dashed lines. Additional fixed (not excludable) or excludable segments may come before or after this segment, and switching element. Note that, except as restricted for some standard registers, the segment selector cell can be in a different TDR or duplicated in a different TDR.

If the excludable segment is a boundary-scan register segment, the mode signals that allow the boundary cells to control the I/O must also be disabled (not illustrated), allowing the system logic to control the I/O associated with the excluded segment. The appropriate mode signal values for an excluded boundary-scan register segment are shown in the mode signal generation tables in Clause 11.

When a logical 0 is scanned into the segment-select cell, or any time the segment-select cell is reset, the segment will be excluded. The length of the test data register with all excludable segments excluded is the default length, and that default length and the lengths of each excludable segment must be documented so that the actual length of the test data register may be determined for any configuration of excludable segments. For example, if the complete test data register is a single excludable segment plus the segment-select cell, then the default length is one. (All TDRs must have at least one scannable element in every configuration.)

Because an excludable segment may not be able to scan under some circumstances, the segment-select cell must capture a signal that indicates whether the segment is ready to scan (logical 1) or not (logical 0). If the segment is always ready to scan, then the segment-select cell always captures a logical 1. In other words, for any segment-select cell in any test data register, if a 1 is captured in that cell, then it may be assumed that the excludable segment may be included without breaking the test data register scan chain.

If the test data register segment requires some action to be taken to make the segment scannable, then a domain-control cell must be provided. It is shown immediately prior to the segment-select cell in the TDR in Figure 9-15, but it could be placed in any nonexcludable portion of this or another TDR. (Boundary-scan and initialization-data registers require the domain-control field instances to be in the same register, although copies may be in other registers.)

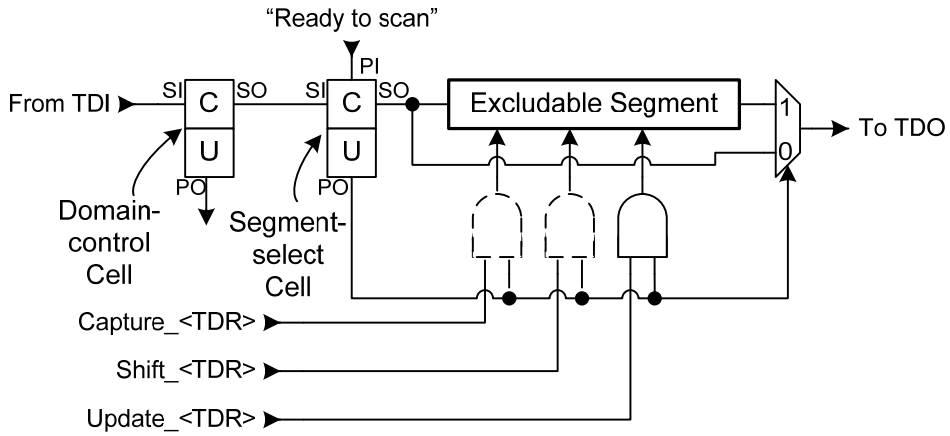


Figure 9-15—Scan control of excludable test data register segments with domain control

If the segment-select cell captures a 0 logic value, indicating that the segment is not ready for scanning, then the domain-control cell must be enabled by scanning a 1 into its parallel output, and the segment-select capture value must be rechecked to verify that the segment is ready. For instance, if a segment is in a power domain that can be powered down, and the power domain is controlled on the component, then a domain-control cell (named a DomCtrl cell in Annex B) must be provided to turn on the power to the domain containing the segment, when required. If multiple TDRs have segments in the same power domain, then there could be multiple domain-control field instances in different TDRs that can turn the domain power on. The "ready to scan" signal from the domain-controller could be captured in multiple segment-select field instances to verify that the power is on and that the various segments are ready to be included for scan.

The parallel output of the segment-select cell is delayed at least one TCK cycle from the latched parallel output. This is to prevent the potential race between update actions and switching the segment between included and excluded by allowing time for the update actions to complete before the segment changes state.

Design-specific TDRs, but not standard TDRs, are allowed to have nested segments, including having segment-select fields within an excludable segment. This raises some special requirements for segment-select fields that are contained within an excludable segment that may be powered down. (Segment-select fields, nested or not but contained in segments that are never powered down independently from the rest of the component, do not have any special requirements.) In keeping with the rules for excludable segments, the segment-select field instance must be reset after being powered up and before the segment it is contained in is included, so that the nested segment it controls is excluded.

The following considerations come into play when designing nested segments for a design-specific TDR. First, any domain-control field instances should be in a nonexcludable segment that is always powered on. Segment-select field instances also placed in a nonexcludable segment that is always powered on provide the most straightforward control, and they can even be used to include both the containing and contained excludable segments with a single scan. For a segment-select cell that is contained in an excludable segment that may be powered down, examples of two ways that they can be properly reset before the containing segment can be included are:

— If the power domain controller or the powered domain provides a signal (such as a local power-up reset), that signal can be used to reset the contained segment select cell. As shown in Figure 9-16, another reset signal such as CHReset* can optionally be combined with the domain POR signal. Separate reset signals are shown for the two segment-select field instances as the type of reset may be different in each case, and CHReset* is shown as an example.

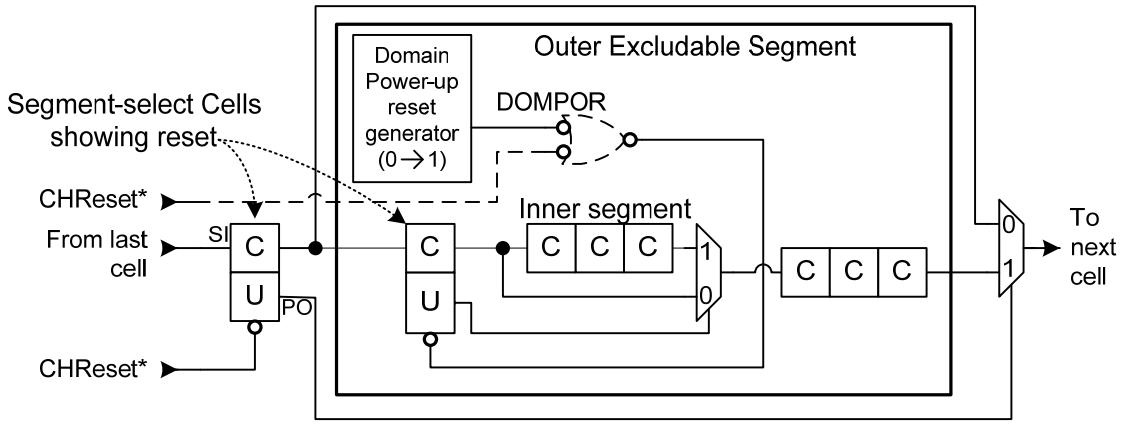


Figure 9-16—Domain POR reset of nested segment-select fields

— The contained segment-select field instance (inner segment-select instance) may be reset by the parallel output of the segment-select instance controlling the containing segment (outer segment-select instance). This outer segment-select instance must be 0 while the segment it controls is being powered-up, and so that state of the cell can be used to reset the inner segment-select instance. Note that this will cause the contained segment to be excluded within its containing segment any time the containing segment is excluded, a behavior that may not be desired. On the other hand, any time the outer segment-select cell is reset, that reset will ripple down to the inner segment-select cell. Figure 9-17 illustrates this option.

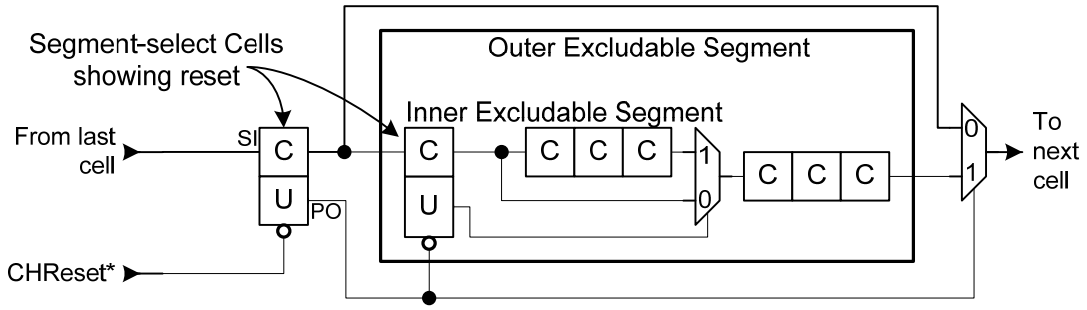


Figure 9-17—Hierarchical reset of nested segment-select fields

If, for whatever reason, none of the above can be done, then the inner segment-select cell should be reset by one of the same means that any other segment-select cell can be reset, and that reset would have to be performed during test after the domain is powered on, and before the outer segment is included. Clearly, this has significant impact on test flow (affecting all components in a scan chain, for example) and should be avoided wherever practical.

Selectable segments

Design-specific TDRs, but not standard TDRs, are also allowed to have selectable segments. The set of selectable segments are essentially in parallel, sharing the same scan-in data, with exactly one of the segments selected as the scan-out for the set. There must be a specified selection field with specified values for selecting each of the public segments in the set, both for scanning and for connection to the scan-out. This is essentially the same structure as the set of test data registers, which are in parallel and exactly one is selected for connection to TDO by the instruction register. Figure 9-18 shows the essential characteristics of a set of selectable segments.

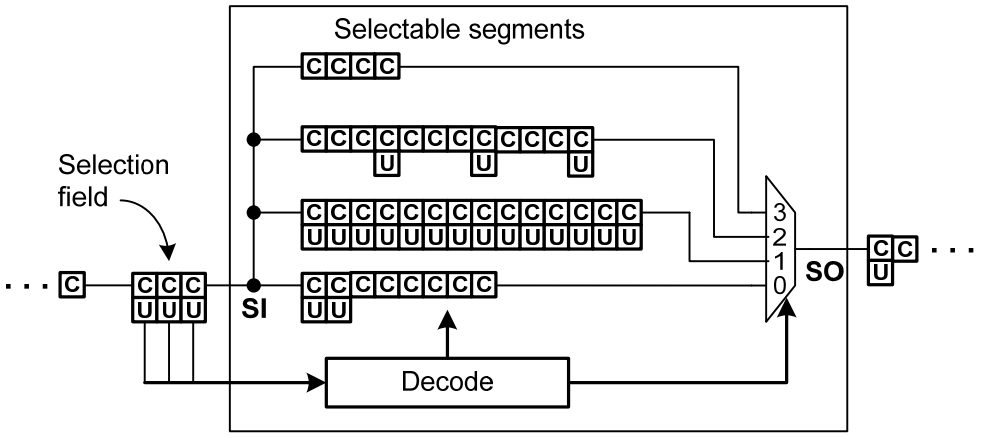
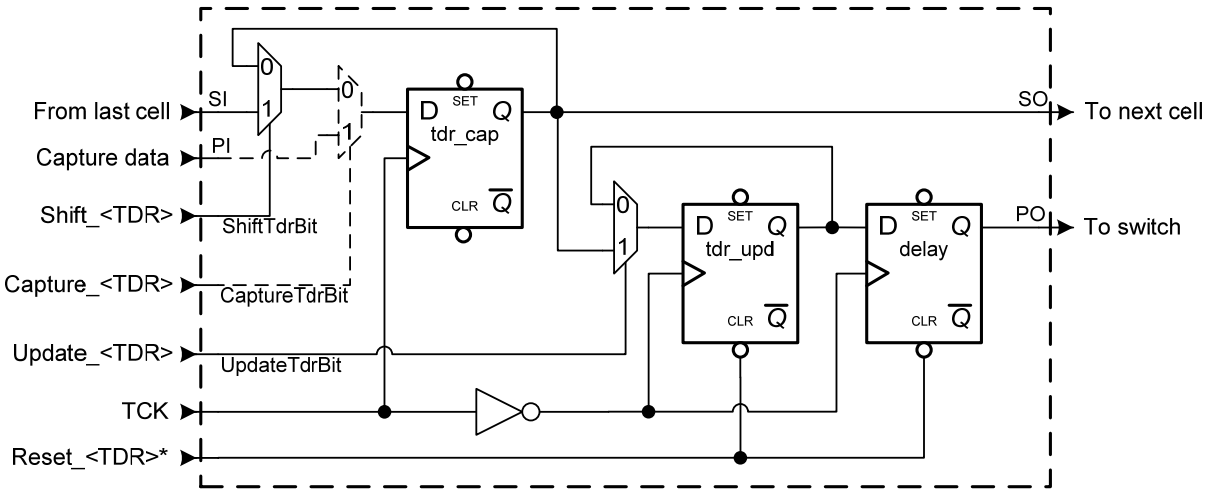


Figure 9-18—Selectable segments and selection field

In Figure 9-18, there are four selectable register segments, each of different length, although they could also be multiple instances of the same segment. The selectable segments start at the point marked SI, where the scan path fans out to each of the segments, and ends at the point marked SO, after the selection circuit (shown as a four-input multiplexer) that connects one segment to the scan chain. The selection field is shown as a contiguous segment immediately preceding the selectable segments, but it could be anywhere in this or any other public TDR, and it could be any defined field, contiguous or not. The output of the selection field is decoded both to select the segment for scan out and to gate the response of the segments to the *Shift-DR, Capture-DR*, and *Update-DR* TAP controller states, as shown by the up arrow in Figure 9-18. There is no requirement that all possible decodes be specified.

Figure 9-19 shows an example segment-select or selection-field cell meeting all of the rules of this clause. For a selection-field cell, the capture logic in Figure 9-19 is optional, as shown with the dashed lines, although it is recommended that it capture the state of the update or delay stage. For a segment-select cell, the capture is required and must capture the "Ready-to-scan" signal.



NOTE—The reset signal in the figure is shown as a Reset_<TDR>* signal. The rules permit any of the test reset signals: (TAP_POR*, TRST*, Reset*, or CHReset*) to be used here, but one of them must be used.

Figure 9-19—Example segment-select or selection-field cell with ungated clocks

10. Bypass register

The bypass register provides a minimum-length serial path for the movement of test data between TDI and TDO. This path can be selected when no other test data register needs to be accessed during a board-level test operation. Use of the bypass register in a component speeds access to test data registers in other components on a board-level test data path.

10.1 Design and operation of the bypass register

10.1.1 Specifications

Rules

a) The bypass register shall consist of a single shift-register stage.
b) When the bypass register is selected for inclusion in the serial path between TDI and TDO by the current instruction, the shift-register stage shall be set to a logic zero on the rising edge of TCK in the *Capture-DR* controller state.
c) The circuitry used to implement the shift-register stage in the bypass register shall not be used to perform any system function (i.e., it shall be a dedicated part of the test logic).
d) The operation of the bypass register shall have no effect on the operation of the on-chip system logic.

10.1.2 Description

The bypass register may be implemented as shown in Figure 10-1.

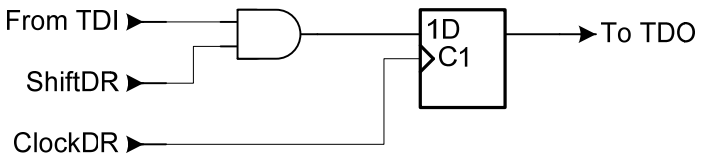


Figure 10-1—Bypass register gated-clock implementation

The provision of this register allows bypassing of segments of the board-level serial test data register that are not required for a specific test. Test access times to the segments of interest are reduced.

As an example, consider a circuit board containing 100 integrated circuits, each of which has 100 bits in its boundary-scan register. The boundary-scan path on the assembled board would include 10 000 shift-register stages if all the segments were connected in series simultaneously. This would give protracted test times, for example, when accessing just one of the integrated circuits on the path.

The ability to bypass segments of the shift-register path under control of the appropriate instruction register allows considerable shortening of the overall path in such circumstances. Continuing the example, 99 of the components could be set to shift only through their bypass register, with the integrated circuit under test having its full boundary-scan register in circuit. This would give a total serial path length of 199 stages—a considerable reduction compared to 10 000.

Rule b) of 10.1.1 is included so that the presence or absence of a device identification register in the test logic can be determined by examination of the serial output data. The bypass register (which is selected in the absence of a device identification register) loads a logic 0 at the start of a scan cycle, whereas a device identification register loads a constant logic 1 into its least significant bit. When the *IDCODE* instruction is loaded into the instruction register, a subsequent data register scan cycle will allow the first bit of data shifted out of each component to be examined—a logic 1 showing that a device identification register is present. This allows blind interrogation of device identification registers by setting the *IDCODE* instruction as outlined in 12.1.

A requirement of the *BYPASS* instruction is that, when it is selected, the on-chip system logic shall continue its normal operation undisturbed. Rule c) of 10.1.1 is included so that this requirement can be met. Note, however, that provided rule d) of 10.1.1 is met, the shift-register stage may be a shared resource used by several of the registers defined by this standard and by any design-specific test data register.

11. Boundary-scan register

The boundary-scan register allows testing of circuitry external to a component, for example, board interconnect or external components that do not conform to this standard. The register also permits the system signals flowing into and out of the system logic to be sampled and examined without causing interference with the normal (nontest) operation of the on-chip system logic. Optionally, additional test functions may be supported—for example, testing of the on-chip system logic.

11.1 Introduction

This clause specifies the design of the boundary-scan register in a component and the operation of the register in response to the various instructions defined by this standard.

Among the registers required by this standard, the boundary-scan register is the most complex. Its complexity lies neither in its shifting function, nor in its architectural placement in parallel with the other required test data registers, nor in its ability to exclude segments, all of which conform to the rules set out in Clause 9. The complexity lies instead in the manner in which the register is connected around the on-chip system logic and in its operation in response to the instructions defined in Clause 8. Design requirements for both connectivity and functional operation vary from cell to cell and are determined both by the type of signal (input to or output from the on-chip system logic) and by the set of instructions to be supported.

The design specifications are presented in three groups:

- *Register design and operation* (11.2 and 11.3). The structure of the boundary-scan register is specified. Also presented are specifications for the operation of the shift-register at the heart of the boundary-scan register and for the parallel output latches or flip-flops that are required for some shift-register stages.
- *Cell provision and operation* (11.4 through 11.8). Rules are presented that specify where boundary-scan register cells must be provided and how they are to be connected between the system pins of a component and the system inputs and outputs of the on-chip system logic. Further rules are presented that define how signals are to be routed through boundary-scan register cells in response to selection of the various instructions defined by this standard.
- *Cell merging* (11.9). Finally, permissible ways in which boundary-scan register cells required by the cell provisioning rules can be merged are specified. Application of these rules will reduce the cost of implementation of a boundary-scan register in certain special cases.

To simplify the presentation of the rules, this clause uses the terminology and approach described in the following clauses.

11.1.1 Approach

To simplify the presentation of the rules in this clause, the boundary-scan register will be described as though it were being added to an already finished design—one that does not conform to this standard. Such a design may be thought of as shown in Figure 11-1.

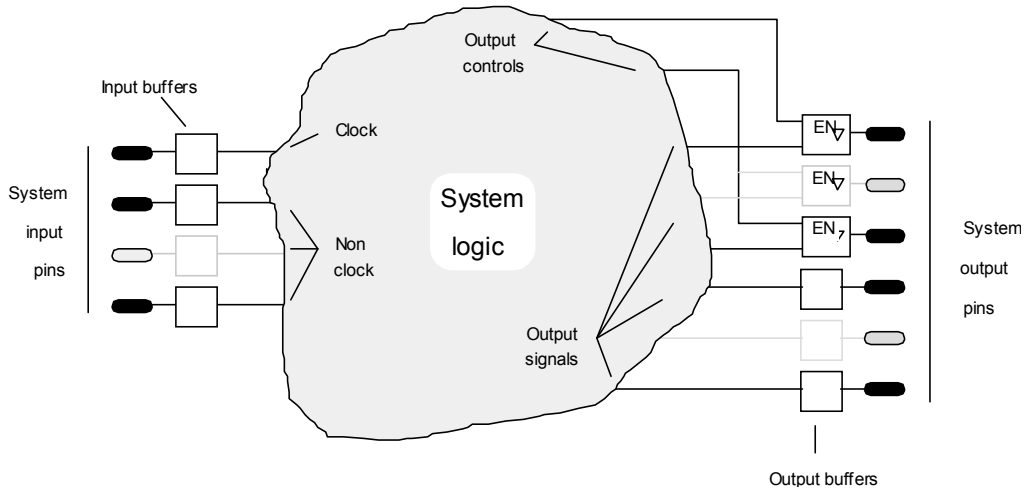


Figure 11-1—Component without boundary scan

Note the following features in Figure 11-1:

— *The system logic*. This is the circuitry that realizes the nontest, digital function of the component.

— *The system pins*. The term "system pin" is used throughout this standard to refer to any system (i.e., nontest) terminal to which an external connection may be made. For packaged components, external connections are made to package pins, typically by means of a soldering process. However, in some cases, an integrated circuit die will be assembled directly onto a substrate without prior packaging. In such cases, the term "system pin" should be interpreted as the point to which the external connection is made, i.e., the bonding pad.

— *The input/output buffers.* The buffers are connected between the system pins and the system logic. Note that output buffers may have control as well as data inputs from the system logic. The signals received at the control inputs determine the manner in which the output buffer operates. For example, Figure 11-1 shows several three-state output buffers at which the enable input (EN) is used to determine whether the output is driven. Other types of output buffers at which control signals may be used to determine different characteristics of the signal driven off-chip are possible. Additional variations include differential drivers and receivers, and other drivers and receivers used in specific communication protocols. Input/output buffers are always assumed to be present, even if not shown in order to simplify an illustration.

— *The inputs and outputs of the system logic*. Two types of input to the on-chip system logic should be distinguished, as follows:

  i) ***Clock inputs***. Transitions at these inputs, from the low to high logic level (or vice versa), are used to indicate when a stored-state device, such as a flip-flop or latch, may perform an operation. In an edge-triggered design, the edges (logic level transitions) received at clock inputs are used to trigger operation of all or part of the on-chip system logic, and steady-state logic values received at these signals have no significance. In a level-sensitive design, clock inputs are used to enable storage devices in the on-chip system logic to load data values. Note that the values loaded into stored-state devices are not determined by the values of clock inputs.

  ii) ***Nonclock inputs***. This group includes all other inputs to the on-chip system logic. Typically, signals applied at these inputs are used to supply data or to select an operation to be performed.

  Outputs from the on-chip system logic drive output buffers (or, as will be discussed later, inputs to mixed analog/digital circuit blocks external to the system logic). It is necessary to distinguish between two types of signal that may be driven to an output buffer, as follows:

iii) ***Output control signals***. In a component without a boundary scan, such as that shown in Figure 11-1, these signals would directly drive "enable" inputs of output buffers and hence determine whether they actively affect the state of the respective connected system pins.

iv) ***Data signals***. In a component without a boundary scan, these signals would drive data inputs to output buffers (or, as will be discussed later, inputs to mixed analog/digital circuit blocks external to the system logic).

NOTE—A single output from the on-chip system logic may drive an output control signal to one output buffer and a data signal to another.

11.1.2 Signal paths to the on-chip system logic

Each signal path into the on-chip system logic is considered to be a fan-out tree with one or more branches. Signals enter the fan-out tree at the trunk (e.g., from an input buffer) and leave through the branches (e.g., at the inputs of the on-chip system logic). For example, Figure 11-2 shows signal paths between one system input pin and several inputs to the on-chip system logic. (In many cases, the fan-out tree will be regarded as being contained within the system logic. In these cases, only a single point-to-point connection is considered to be present between the system input pin and the system logic, which is a connection with only one fan-out branch.)

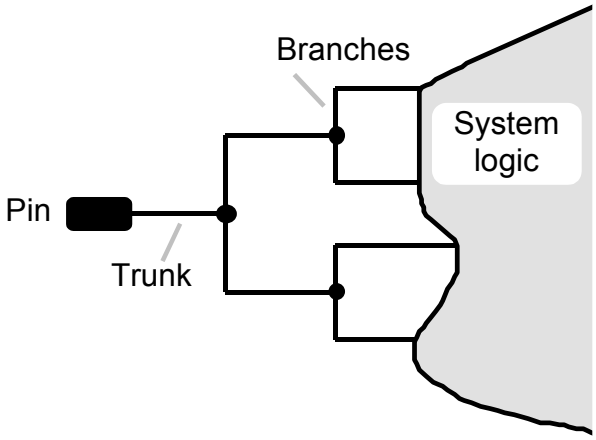


Figure 11-2—Input connections

11.1.3 Boundary-scan register cell

NOTE—The rules contained in this clause describe the boundary-scan register as a logical device, not a particular physical implementation. Furthermore, for clarity of presentation, the example boundary-scan register cell designs presented in this clause show the circuitry to be separate from that used to construct the various other features defined in this standard. However, be aware that the rules of this standard permit parts of the circuitry used to construct boundary-scan register cells—notably the shift-register stages—to be used in the implementation of other features defined by this standard, such as the bypass and device identification registers. Where circuitry is shared between the boundary-scan register and other features defined by this standard, boundary-scan register cells may appear that are more complex than those described here.

Every boundary-scan register cell is considered to have a number of data terminals (at least two) and a number of clock and control inputs appropriate to the style of implementation. Contained within each cell is a single shift-register stage that often is provided with a parallel input and a parallel output (which may be latched). This shift-register stage uses two data connections of the cell as a serial input and a serial output. By way of these connections, the cell is linked to the cells before and after it in the boundary-scan register.

Cells that have three data terminals allow signals entering or leaving the on-chip system logic to be observed, but not controlled. For such a cell, the third data terminal functions as a parallel input to a parallel-in, serial-out shift-register stage. When the boundary-scan register is selected as the serial path between TDI and TDO by an

instruction, the data present at this terminal is loaded into the shift-register stage on the rising edge of TCK in the *Capture-DR* controller state (i.e., as required by the rules of Clause 9). Cells of this type may be described as "observe-only" cells. They will be connected to a signal path entering or leaving the on-chip system logic, as shown in Figure 11-3.

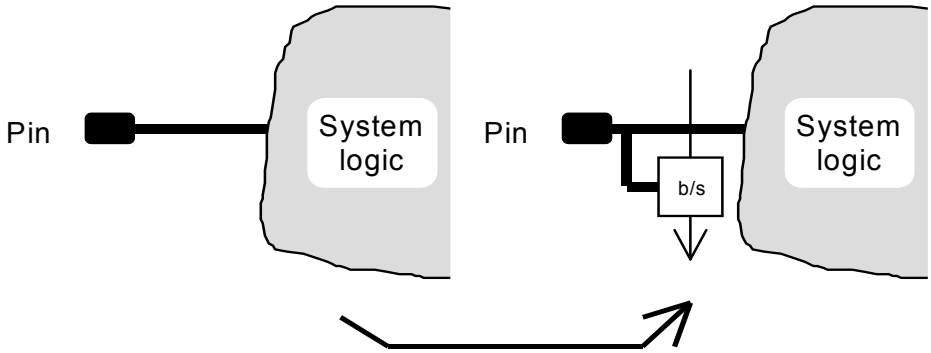


Figure 11-3—Connection of an observe-only boundary-scan register cell

Cells with four or more data terminals are inserted into signal paths entering or leaving the on-chip system logic, as shown for the case of an input in Figure 11-4. Such cells may be described as "control-and-observe" cells. The shift-register stage in a control-and-observe cell can load the value of the signal path into which they are inserted and hence allow observation of that signal. Also, when required, the value held in the shift-register stage can be driven to the wire in place of the normal (nontest) source. In some cases, a constant signal value also may be driven to the wire in place of the normal (nontest) source.

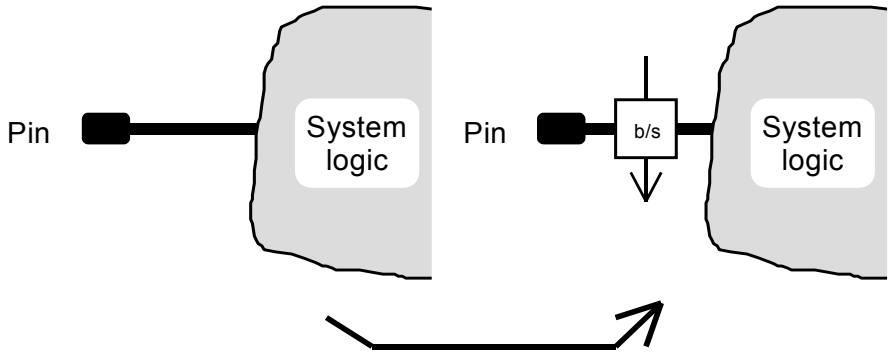


Figure 11-4—Insertion of a control-and-observe boundary-scan register cell

Cells that have only two data terminals are redundant in the sense that they could have been omitted from the design without jeopardizing compliance of the component to this standard. Such cells may be simple shift-register stages. Typically, they will exist in a component design as a result of programming or customization of boundary-scan register cells (see 11.8). These cells do not contribute information to the test process.

The rules presented later in this clause define the manner of cell insertion or addition required at each type of connection into or out of the system logic. These rules also define the manner of insertion or addition of additional observe-only boundary-scan register cells, which may be added at almost any component pin. Such additional cells are redundant in the sense that they could have been omitted from the design without jeopardizing compliance of the component to this standard, and they are described as "redundant observe-only" cells. Given the rules in this clause, there is no such thing as a "redundant" control-and-observe cell.

A conceptual model of a control-and-observe boundary-scan register cell is shown in Figure 11-5. Such cells contain a parallel-in, parallel-out shift-register stage. Signals to be controlled or observed flow into the cell through one or more terminals of the cell (termed the parallel inputs) and out through other terminals (termed the parallel outputs). Logic (typically, one or more multiplexers) within the cell determines routing of the signal from the parallel input(s) of the cell to the parallel output(s) of the cell. The signal may be routed through the PI and PO terminals of the parallel-in, parallel-out shift-register stage or, in normal (nontest) operation of the component, will be driven directly from the parallel input(s) of the cell to the parallel output(s) without any change in signal value.

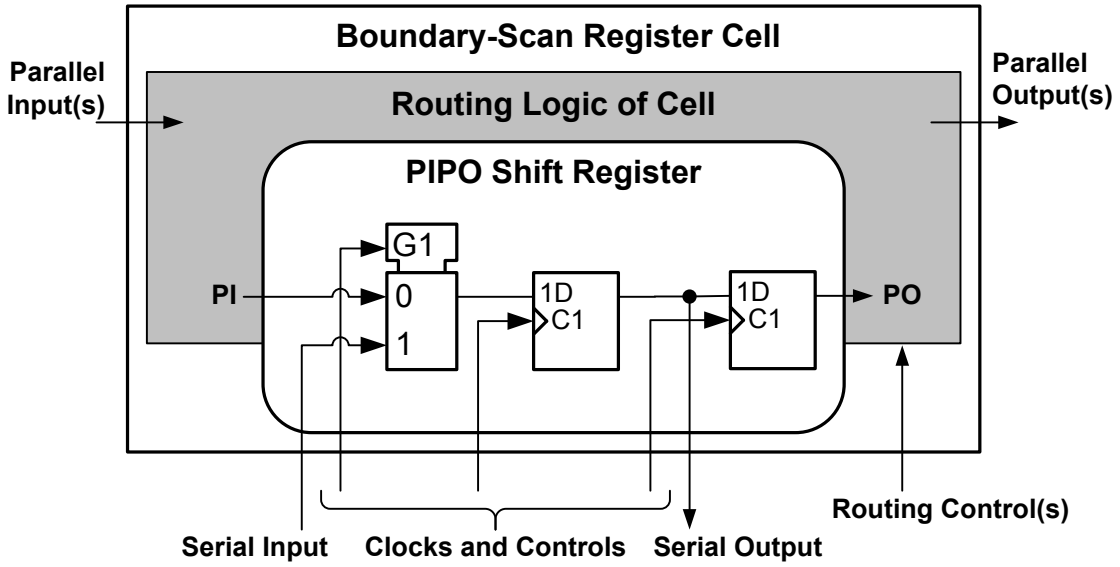


Figure 11-5—Conceptual view of a control-and-observe boundary-scan register cell

11.2 Register design

11.2.1 Specifications

Rules

a) The boundary-scan register shall consist only of boundary-scan register cells as defined in this clause and register control cells as defined in 9.4.

b) Sufficient boundary-scan register cells shall be provided to fully meet the requirements for each connection into or out of the on-chip system logic, as defined later in this clause.

c) Each boundary-scan register cell shall contain a single shift-register stage and shall have a serial input terminal and a serial output terminal by means of which the cell is linked to the cells before and after it in the boundary-scan register or, in the case of the cells at each end of the register, to the remainder of the test logic defined by this standard.

d) A boundary-scan register cell shall have two or more data terminals [including the serial terminals required by rule c) of 11.2.1].

NOTE 1—Each cell will also have several clock and control inputs, the number of which will be determined by the style of implementation.

e) Boundary-scan register cells that have three connected data terminals shall be designed such that:

1) One data terminal is connected by routing logic to a parallel input to the shift-register stage in the cell (referred to as "observe-only" cells).
2) One data terminal is connected by routing logic to the latched parallel output, if provided, or the parallel output of the shift register stage, if not (referred to as "control-only" cells).

f) For boundary-scan register cells that have four or more connected data terminals, data terminals other than those used for serial input and output shall be parallel inputs and parallel outputs of the cell and be connected by routing logic:

1) To each other.
2) To the parallel input to the shift-register stage and to the latched parallel output, if provided, or the parallel output of the shift-register stage.

NOTE 2—Such cells are described as "control-and-observe" cells.

NOTE 3—Often, but not always, shift-register stages will require latched parallel outputs.

g) For a given component, the ordering of cells in the boundary-scan register shall be fixed and shall not vary as a result of any operation of the on-chip system logic.

h) For a given component, the length of the boundary-scan register, or the length of each boundary-scan register segment assembled to form the boundary-scan register, shall be fixed and shall not vary as a result of any operation of the on-chip system logic.

i) In the event that no boundary-scan register cells are required for a component, a register consisting of a single shift-register stage shall be provided.

NOTE 4—This situation will arise when a component contains only test logic as defined or permitted by this standard. Such a component could be described as being dedicated to testing; it will not contribute to the system function of an assembled board.

j) Other than boundary-scan register segments that can be powered down, operation of boundary-scan register cells shall not be interfered with or interrupted by the normal operation of any system function (see 11.3).

NOTE 5—Subject to conformance with rule k) of 11.2.1, circuitry may be shared between the boundary-scan register and another part of the test logic. For example, the shift-register stages also may be used by another test data register.

k) Where a boundary-scan register cell has a latched parallel output, if a system or another test function alters the value of the latched parallel output (i.e., the latched parallel output may be shared), the value that would have existed if the latched parallel output were dedicated shall be restored upon entry into the *EXTEST*, *INTEST*, *CLAMP*, or other instruction requiring the boundary-scan register [see rule b) and rule c) of 11.3.1 and the description of 11.3.2].

NOTE 6—Subject to conformance with rule j) and rule k) of 11.2.1, circuitry may be shared between the boundary-scan register and the normal system logic.

NOTE 7—Rule k) of 11.2.1 applies in situations where the value in the latched parallel output is expected to be held by the test application software. It does not apply when a boundary-scan register segment is excluded and possibly powered down. In this case, it is assumed the values in the latched parallel outputs have not been retained. See rule x) in 9.4.1.

Permissions

l) Rule i) in 11.2.1 may be met by selecting the bypass register whenever this standard requires selection of the boundary-scan register as the path between TDI and TDO.

m) Boundary-scan register cells may be connected in any order.

NOTE 8—See rule g) of 11.2.1.

11.2.2 Description

The boundary-scan register is a mandatory feature of this standard and shall be included in each component that claims conformance to this standard. It consists of a number of cells equal to the number of shift-register stages contained in the register. These cells are positioned around the on-chip system logic of a component as specified later in this clause. They are connected to form a single shift-register-based path that is connected between TDI and TDO in the *Shift-DR* controller state when an appropriate instruction is selected. (With regard to the instructions defined in this standard, the boundary-scan register is defined to be the serial path between TDI and TDO in the *Shift-DR* controller state for the *SAMPLE, PRELOAD*, *EXTEST*, *INTEST,* and, optionally, *RUNBIST* instructions.)

Figure 11-6 illustrates the design of the shift-register portion of the boundary-scan register. Note that not all stages in the figure are provided with latched parallel outputs, and one is an observe-only cell.

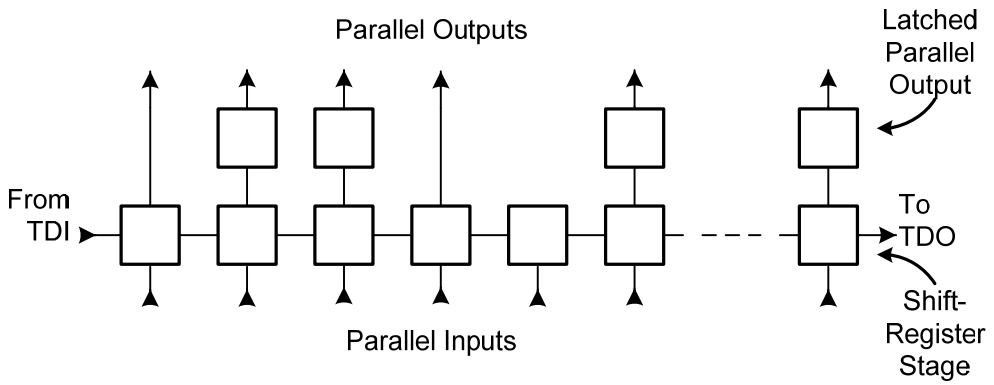


Figure 11-6—Boundary-scan shift-register design

For a given component, the minimum permissible length of the boundary-scan register with all excludable segments, if any, excluded is a function of the number and type of system (nontest) connections into and out of the on-chip system logic. (Typically, these will be off-chip connections.) The rules that determine the minimum set of cells that has to be provided are presented later in this clause. Note, however, that every component claiming to be compliant to this standard is required to have a boundary-scan register that contains at least one shift-register stage.

11.3 Register operation

11.3.1 Specifications

Rules

a) When the boundary-scan register is selected as the serial path between TDI and TDO in the *Shift-DR* controller state, data entered at TDI shall appear without inversion at TDO after the application of a number of paired rising and falling edges at TCK equivalent to the current length of the boundary-scan register.

b) When the *EXTEST*, *INTEST*, or *PRELOAD* instruction is selected, latched parallel outputs of the boundary-scan shift-register shall change state only on the falling edge of TCK in the *Update-DR* controller state; at which time, each shall be set to the state of its corresponding shift-register stage.

c) While any instruction is active that does not select the boundary-scan register for scanning, all latched parallel outputs of the boundary-scan register not in an excluded segment shall retain their state at least until:

   1) The *Test-Logic-Reset* controller state is entered as a result of application of a logic 0 at TRST*.

   2) The first falling edge of TCK occurs in the *Test-Logic-Reset* controller state when that state is entered as a result of signals applied at TCK and TMS if the TMP controller is not provided or is in the *Persistence-Off* state.

NOTE 1—Permission h) of 11.3.1 allows the boundary-scan register to be reset after entry into the *Test-Logic-Reset* controller state; and so in a case in which such implementation details are not known, it should be assumed that the states of the latched parallel outputs of the boundary-scan register are unknown once this state has been entered unless the TMP controller is provided and in the *Persistence-On* state.

NOTE 2—Application of a logic 0 at TRST* will reset both the TAP and TMP controllers. Therefore, boundary-scan register cells can be reset by the TAP *Test-Logic-Reset* state since the TMP controller, if provided, will be in the *Persistence-Off* state.

d) No limit shall be imposed on the number of system output pins that may change state in a single *Update-DR*, *Update-IR*, or *Test-Logic-Reset* controller state as a result of the operation of the test logic; neither shall restrictions be placed on the data patterns that may be driven (e.g., a maximum limit on the number of system logic outputs that may drive a logic 1).

NOTE 3—Designers should consider that in test mode, a boundary-scan test may manipulate several buses in ways that could never occur during normal system operation. This may cause transient or steady-state power consumption to exceed that expected for normal system operation. Be especially aware of this when bread boarding, socketing, or fixturing components compatible with this standard because in these environments, power distribution may be suboptimal.

e) No limit shall be placed on the combinations of logic values that may be shifted into the boundary-scan register.

NOTE 4—During test operations, combinations of signals may be driven either to the on-chip system logic or off-chip that will not arise during normal operation of the component. This is particularly the case where multiple boundary-scan register cells are used to drive signals that normally are driven from a single source [see rule c) and rule d) of 11.5.1 and rule e) of 11.6.1].

Permissions

f) When the *SAMPLE* instruction is selected, latched parallel outputs of the boundary-scan shift-register may either hold their value in any TAP controller state or change state only on the falling edge of TCK in the *Update-DR* controller state; at which time, each shall be set to the state of its corresponding shift-register stage.

g) The delay between the falling edge of TCK and consequent changes at system output pins may be deliberately skewed between system outputs, e.g., because of a need to minimize simultaneous switching at several or all system outputs.

NOTE 5—In the case of the example implementations shown in this standard, this skew could be added by injecting small delays in the *Update-DR* clock path and *Mode-n* control signals to each boundary-scan register cell. Such skew may be required both for boundary-scan register cells that feed data signals and for those that feed output control signals. In the latter case, the added skew will prevent excessive current demand due to simultaneous changes from “disable” to “active” or vice versa.

h) Where boundary-scan register cells are provided with shift-register stages with latched parallel outputs, all, any, or none of these latched parallel outputs may be reset to either logic state (0 or 1):

1) When the *Test-Logic-Reset* controller state is entered as a result of application of a logic 0 at TRST*.
2) On the first falling edge of TCK in the *Test-Logic-Reset* controller state when that state is entered as a result of signals applied at TCK and TMS and if the TMP controller is either not provided or in the *Persistence-Off* state.

NOTE 6—Application of a logic 0 at TRST* will reset both the TAP and TMP controllers. Therefore, boundary-scan register cells can be reset by the TAP *Test-Logic-Reset* state since the TMP controller, if provided, will be in the *Persistence-Off* state.

11.3.2 Description

To meet the requirements of the *SAMPLE* and *PRELOAD* instructions, it has to be possible to move data through the boundary-scan register without interfering with the normal system operation of the component. This may be achieved by making the shift-register stages used by the boundary-scan register a dedicated part of the test logic; that is, they do not perform any system function. Alternatively, the shift-register stages may be a shared resource used by several of the registers defined by this standard and by any design-specific test data register.

In contrast, for the public instructions defined by this standard, the latched parallel output required at some boundary-scan register stages is controlled such that it retains its last state whenever the boundary-scan register is not selected for scanning by the current instruction. This feature allows a set of data values to be shifted into the register and placed onto the latched parallel outputs by use of the *PRELOAD* instruction. As other instructions that do not select the boundary-scan register for shifting are made active, the values in the latched parallel outputs of the boundary-scan register will be retained. Thus, when an instruction is made active that causes the data held in the boundary-scan register to be driven out of the component (e.g., the *EXTEST* or *CLAMP* instruction), the previously loaded data are immediately available for use. (See the discussion in 8.7.)

11.4 General rules regarding cell provision

11.4.1 Specification

Rules

a) One or more boundary-scan register cells shall be provided at each digital system input or output of the on-chip system logic, as detailed in 11.5 and 11.6.

b) For components that contain analog circuitry external to the on-chip system logic (i.e., analog circuitry having off-chip connections as shown in Figure 11-7), the connections between the on-chip analog circuits and the on-chip system logic shall be treated as off-chip connections.

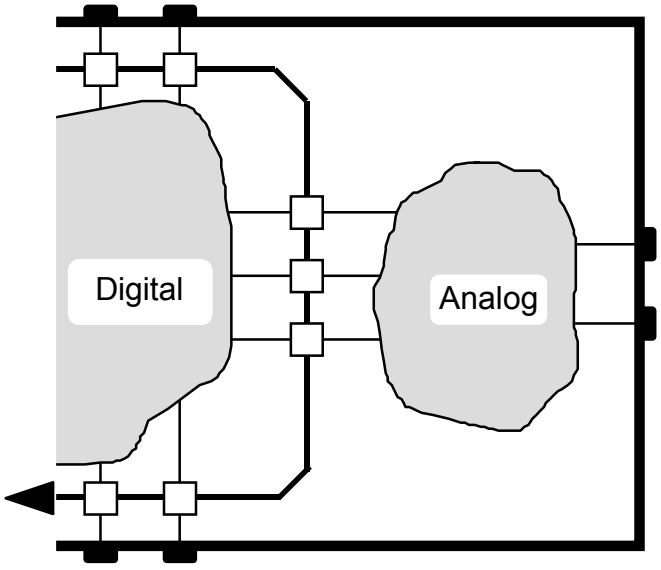


Figure 11-7—Component that contains analog circuitry

NOTE 1—Differential drivers and receivers that operate through detection of the direction of current flow are considered “analog” circuits. Therefore, in such cases, this rule requires a single boundary-scan register cell to be placed between each differential driver or receiver and the on-chip system logic, as illustrated in Figure 11-9. See 11.4.2 for

information on the application of these rules to other types of paired inputs and outputs, e.g., differential signals that operate using conventional logic voltages.

c) Boundary-scan register cells shall not be provided at TAP pins (TCK, TDI, TDO, TMS, and TRST*).

d) The connection of boundary-scan register cells of the control-and-observe type shall be such that if each cell were replaced by a short-circuit connection between its parallel input and parallel output, the normal (nontest) logical operation of the component would not be altered.

NOTE 2—There may, however, be changes in performance.

e) There shall be no logic between any boundary-scan register cell and the system pin to which that cell is connected.

NOTE 3—"Transparent" devices such as buffers and I/O buffers are not considered to be "logic" and may exist outside the boundary-scan register. Inverters may also exist outside the boundary-scan register subject to conformance to rule i) and rule j) of 11.5.1 and rule l) through rule o) of 11.6.1. Devices that perform a logic operation (such as gates, flip-flops, and latches) are considered to be logic devices and are not allowed outside the boundary-scan register.

f) Where the boundary-scan register is segmented [as described in rule c), rule d), and rule h) of 9.2.1], all cells associated with a single port or an associated port pair (including any output control cells) shall appear within a single segment.

g) Where the boundary-scan register is assembled from segments that may be included or excluded, a segment-select cell conforming to the requirements of 9.4.1 shall be provided for each excludable segment.

h) Where an excludable boundary-scan register segment must be conditioned in order to be included, a "control-only" domain-control cell conforming to the requirements of 9.4.1 shall be provided for each such conditioning requirement.

NOTE 4—This is the only defined compliant use of a "control-only" cell in the boundary-scan register. The conditioning requirement is often a power domain that can be powered on or off.

Permissions

i) Where some inputs or outputs are not connected to package pins in a particular packaged configuration of a component, boundary-scan register cells may be provided for the unconnected signals.

j) Redundant observe-only boundary-scan cells may be provided on any package pin other than TCK, TMS, TDI, TDO, and TRST*.

NOTE 5—Redundant observe-only cells are permissible on some nonsystem pins and on nondigital pins as well as pins and signals to and from the system logic in order to enhance fault coverage and diagnostics. See 11.8 for rules concerning these cells. Figure 11-10 illustrates a conceptual schematic of how redundant observe-only cells may be used on differential pins. These cells have the function **OBSERVE_ONLY** as defined by BSDL (see Annex B).

k) Segment-select and domain-control cells controlling excludable boundary-scan register segments may be duplicated in other public test data registers.

NOTE 6—Where more than one domain-control or segment-select cells control a single excludable segment, they would effectively be ORed together so that each one of them has the same effect as the others, and all must be set to zero to exclude the segment or disable the domain.

11.4.2 Description

Figure 11-8 illustrates the placement of required boundary-scan register cells for basic input, output, and bidirectional pin types. The input cells could be control-and-observe (which would support *INTEST*) or observe-only, and the separate output and input cells on the bidirectional output could be replaced by a single cell performing both functions.

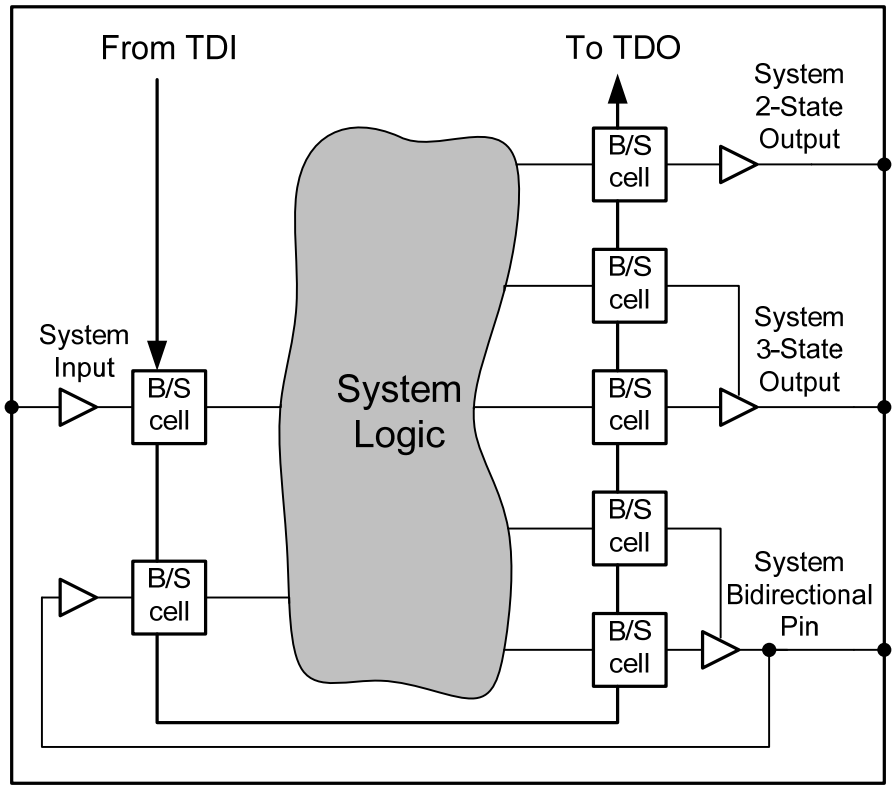


Figure 11-8—Placement of boundary-scan register cells

Boundary-scan register cells are placed such that the state of each digital system pin (including clock pins) can be controlled and/or observed using the boundary-scan register. These cells also may allow the state of the system logic inputs and outputs to be controlled and observed respectively.

If the boundary-scan register is segmented, possibly because portions are contained within predefined blocks of logic, it is commonsense to follow rule f) of 11.4.1, which requires that all cells associated with a single port be contained within the segment. In other words, when integrating multiple boundary-scan register segments, each segment would contain all of the cells needed for compliance with this standard given the I/O that it supports.

Extension of the design of the boundary-scan register to cover cases in which analog circuit blocks are located external to the on-chip system logic, between the logic and the pins, is straightforward. In such components, the signals that form the interface between the purely digital circuitry and the mixed analog-digital circuit block(s) are considered to be equivalent to system pins. Therefore, boundary-scan register cells are provided for connections that flow to or from the mixed analog-digital block(s) [see rule b) of 11.4.1 and Figure 11-7].

The specification of the boundary-scan register also addresses cases in which logic signals are communicated between components by nondigital or nonelectronic means. Examples would be using optical interconnect or capacitive coupling. In these cases, drivers and receivers are considered to be analog circuit blocks and are placed outside the boundary-scan register cells for the relevant logic signals.

Alternatively, readers of this standard may wish to reference the architecture and capabilities defined by the latest version of IEEE Std 1149.4.[7] This standard describes circuits to deliver and measure both noncontinuous (dc) and

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

continuous (ac) signals to and from analog component pins and internal logic via a standardized test interface that is a superset of, and fully compatible with, the test access port defined by this standard. Of further note, the latest version of IEEE Std 1149.6 also makes specific provisions for testing of AC coupled and/or differential-signaling component pins.

The case in which a pair of system pins is used to carry a single logic signal into or out of a component (e.g., at a differential input or output; see Figure 11-9) is slightly more complex and merits further discussion.

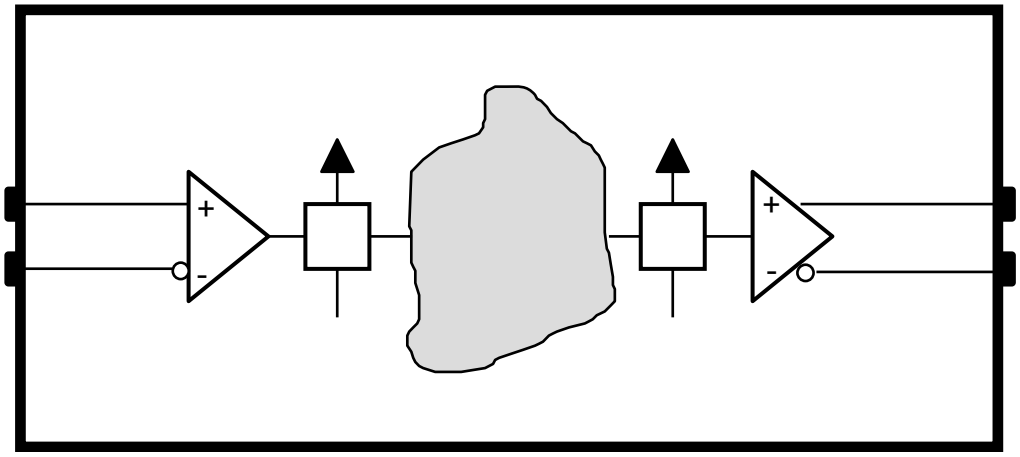


Figure 11-9—Component with differential inputs and outputs

Typically, "paired" I/O will be provided to enhance the performance of connections between components, for example, to enable reliable communication between components in a noisy environment or to reduce skew in high-performance systems. The characteristics that differentiate paired I/O from conventional digital signals are as follows:

— The signals flowing through the pair of pins typically are driven from a single buffer and are always received by a single buffer.
— The signals always should be connected in pairs if the "enhanced" behavior (e.g., noise rejection) of the component-to-component connection is to be maintained.

Two types of paired I/O are commonplace:

a) Those that work by altering or sensing the direction of current flow around the loop formed by the two connections
b) Those that appear in many respects to be "conventional" digital signals, for example, because they use TTL-compatible voltage levels

For current-flow paired I/O signals, the differential input or output buffer should be considered an analog circuit. Therefore, one boundary-scan register cell must be placed between each buffer and the on-chip system logic (Figure 11-9). However, it is recommended, when possible, to add redundant observe-only cells to each pin in order to more fully detect and isolate board -level faults (Figure 11-10).

For paired conventional signals, the location of the boundary-scan register cells at output pins will depend on the precise characteristics of the link. The following is suggested:

— Where it is possible for the output signals to be used independently (losing the enhanced characteristics of the connection but retaining the ability to convey the logic signal), two cells should be provided for each

driver, one for each output pin. This improves the testability of board-level interconnections where one of the output signals from a pair is used to drive a "conventional" input pin on another component.

— Where the signals that constitute the pair cannot be used independently, a single boundary-scan register cell should be provided between the system logic and the output buffer. In this configuration, redundant observe-only cells are recommended on each pin of the pair.

For a paired "conventional" input, a single boundary-scan register cell has to be placed between the buffer and the on-chip system logic, allowing observation of the logic signal transmitted across the pair. In this configuration, redundant observe-only cells are permitted by the rules [see rule r) of 11.5.1 and rule t) of 11.6.1] on each pin of the pair, as shown in Figure 11-10, and their use will improve the testability and diagnosibility of faults during test.

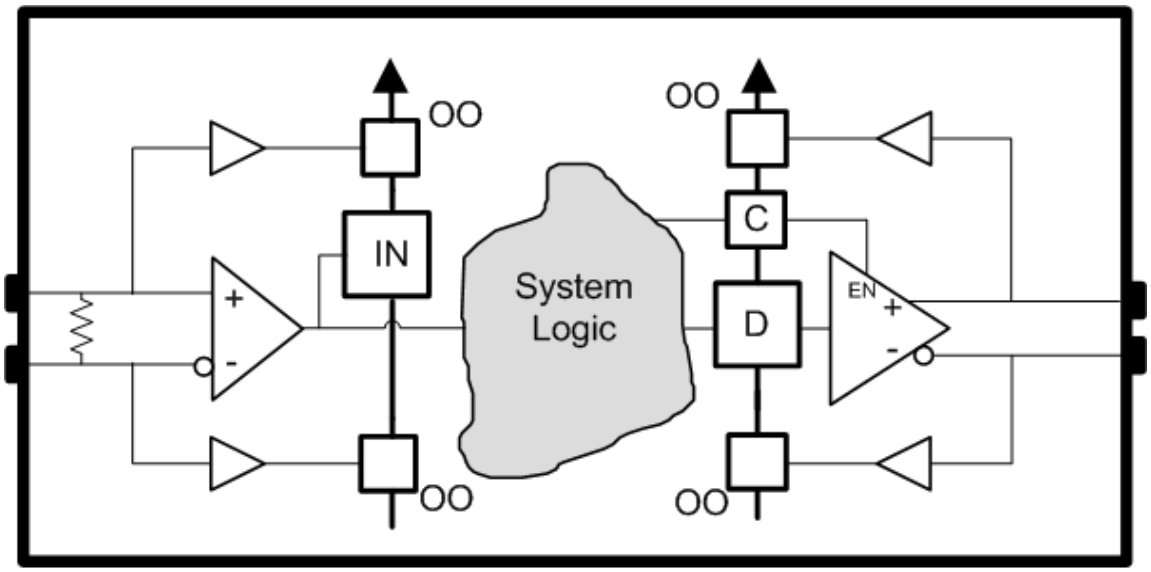


Figure 11-10—Conceptual schematic of redundant observe-only cells on differential pins

Figure 11-10 is a conceptual schematic showing how current-flow differential pins can better support tests for board-level construction faults through the use of redundant observe-only cells on each leg of the differential pair. The schematic is conceptual only, and it is not expected that a designer would simply connect the positive and negative pins to a wire and to a boundary-scan cell. The extra receivers shown connected to the redundant observe-only cell would likely be an integral part of the design of the differential driver or receiver. In the case of a current-flow differential pair [i.e. low-voltage differential signaling (LVDS)], the designer of the driver or receiver would need to consider the receiver output logic value in the presence of any of the possible board defects. Designers familiar with adding LVDS receiver failsafe designs will recognize this requires adding analog circuitry to allow fault identification *per pin* and supporting detection of pins stuck at ground and individually open in addition to the LVDS failsafe modes of both open, shorted and stuck at 1. The additional circuitry, however, needs only to be active during *EXTEST* and for slow speed operation. It does not need to be functional during mission mode operation as *SAMPLE* is no longer required [see permission d) of 11.8.1] on the redundant observe-only cells.

NOTE—Prior to this version of the standard, a loophole in the BSDL language allowed cells with <function> **OBSERVE_ONLY** to be used in the position of the cell labeled IN shown in Figure 11-10. This is no longer allowed; the IN cell must be of BSDL function **INPUT** or **CLOCK** and may be either a control and observe cell or a cell with just an observe capability such as **BC_4**, as shown in the figure.

11.5 Provision and operation of cells at system logic inputs

This subclause provides rules for the provision and operation of boundary-scan register cells at digital inputs to the on-chip system logic. These inputs may be driven by buffers at system input pins or at bidirectional system pins when they operate as system inputs. Alternatively, such digital inputs may be driven by mixed analog/digital circuit blocks located external to the on-chip system logic.

As discussed in 11.1.1, system logic inputs may be either clock or nonclock. Many of the rules presented in 11.5.1 are common to both types of input. However, some rules apply only to one input type. Where this is the case, the rule, recommendation, or permission is labeled *"For clock (non-clock) inputs only.*"

NOTE—This subclause addresses provision of boundary-scan register cells at inputs to the on-chip system logic in cases in which these inputs are driven by system input pins during normal (nontest) operation. The case in which an input to the on-chip system logic is driven by a system bidirectional pin during normal (nontest) operation is discussed in 11.7.

11.5.1 Specifications

Rules

a) Each signal received at a system logic input (e.g., from an input buffer) shall be capable of being observed by at least one boundary-scan register cell (Figure 11-11).

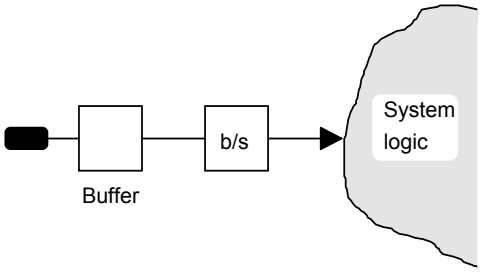


Figure 11-11—Provision of a boundary-scan register cell at a system input

NOTE 1—The cells may be control-and-observe or observe-only. Where at least one control-and-observe cell is provided to observe a single signal received at a system logic input, rule e) of 11.5.1 applies.

NOTE 2—Any additional cells are redundant in the sense that they could be omitted from the design without jeopardizing compliance to this standard [see 11.8 and permission r)]. These cells are called redundant observe-only cells.

NOTE 3—It is permissible for a set of control-and-observe boundary-scan register cells to be distributed in a fan-out network from one system input buffer to multiple inputs to the on-chip system logic normally driven by that buffer (see Figure 11-12). In such cases, rule e) of 11.3.1 applies and the cells are not considered redundant.

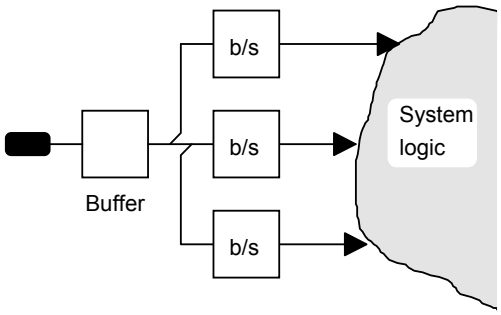


Figure 11-12—Provision of multiple boundary-scan register cells at one input

b) Each boundary-scan register cell that can observe a signal received at a system logic input shall observe precisely one such signal.

c) *For nonclock inputs only*: If the *INTEST* instruction is supported, each nonclock input *I* to the on-chip system logic shall be driven from precisely one boundary-scan register cell, and this cell shall be one of those that observe the system input signal that drives *I* during normal (nontest) operation.

NOTE 4—For clock inputs, provision of a control-and-observe cell is optional [see rule d) and permission l) of 11.5.1].

d) *For clock inputs only:* If the *INTEST* instruction is supported and one or more control-and-observe cells are provided that observe the system clock input signals of a component, each system clock input *I* to the on-chip system logic shall be driven from precisely one control-and-observe cell, and this cell shall be one of those that observe the system clock input that drives *I* during normal (nontest) operation.

NOTE 5—This rule is activated when the option set out in permission 1) of 11.5.1 is exercised.

e) The parallel output of a control-and-observe cell that observes a signal received at a system logic input shall drive only one of the following:

1) One or more system logic inputs
2) An output signal driven to a single system logic output pin or external analog/digital circuit block
3) The signal driven to the output control of one or more output buffers

NOTE 6—It is a consequence of this rule that where a signal from, for instance, a system input pin would normally fan out to drive more than one of the above options, more than one control-and-observe cell is required. Refer to 11.6 for rules relating to output data signals and output controls.

f) Each cell that observes a signal received at a system logic input shall be designed to route signals as shown in Table 11-1.

Table 11-1—Routing of signals in cells at system logic inputs

Instruction	The signal loaded into the shift-register stage of each cell on the rising edge of TCK in the *Capture-DR* controller state is:	The signal driven from the parallel output of control-and-observe cells while the instruction is selected is:
CLAMP, INIT_RUN, INIT_SETUP_CLAMP, CLAMP_HOLD, CLAMP_RELEASE	Not relevant	Not defined[a]
EXTEST	The signal driven to the system logic input from the external source	Not defined[a]
HIGHZ	Not relevant	Not defined[a]
INTEST	Not defined	*For nonclock inputs only:* The parallel output of the shift-register

Instruction	The signal loaded into the shift-register stage of each cell on the rising edge of TCK in the *Capture-DR* controller state is:	The signal driven from the parallel output of control-and-observe cells while the instruction is selected is:
		stage *For clock inputs only:* See rule g) of 11.5.1
PRELOAD	Not defined	The signal received at the connected system pin
RUNBIST	Not defined (not relevant unless boundary-scan register is selected as the serial path between TDI and TDO)	*For nonclock input only:* Not defined[b] *For clock inputs only:* Defined by rule g) of 11.5.1
SAMPLE	*For clock and nonclock inputs only:* The signal driven to the system logic input from the external source *For redundant observe-only cells:* The signal driven to the system logic input from the external source, a constant value signal, or an undefined signal[C]	The signal received at the connected system pin
BYPASS, IDCODE, USERCODE, IC_RESET, INIT_SETUP	Not relevant	The signal received at the connected system pin

[a] See rule h) and permission n) and permission o) of 11.5.1.

[b] See rule h) and rule k) and permission p) and permission q) of 11.5.1.

[C] See rule a) of 11.8.1.

g) *For clock inputs only*: When the *INTEST* or *RUNBIST* instruction is selected, the signal driven to the on-chip system logic shall be one of the following:

1) The signal received at the connected system pin
2) The TCK signal, controlled such that the on-chip system logic changes state only in the *Run-Test/Idle* controller state
3) For *INTEST* only, the parallel output of the shift-register stage.

NOTE 7—Where option g1) of 11.5.1 is selected, the component designer should assume that the signal applied to the clock input pin will be free-running and not externally controllable. Therefore, to meet rule b) of 8.9.1 and rule b) of 8.10.1, circuitry would need to be built into the component to enable only appropriate clock transitions to the on-chip system logic (for example, in the case of the *INTEST* instruction, a "hold" signal may be pulsed internally to provide single stepping in the *Run-Test/Idle* controller state).

NOTE 8—Where a component has more than one clock input pin, the component design should provide correct operation of the *INTEST* and *RUNBIST* instructions, if provided, for all frequency and phase relationships between the clock input signals that are permissible for correct nontest operation of the component.

h) The component shall not be damaged as a result of signals fed to the on-chip system logic when the *CLAMP*, *EXTEST*, *HIGHZ*, *INIT_RUN, INIT_SETUP, INIT_SETUP_CLAMP, CLAMP_HOLD, CLAMP_RELEASE,* or *RUNBIST* instruction is selected.

NOTE 9—This may be achieved by disabling operation of the on-chip system logic or by designing the cell such that a constant logic signal is output when these instructions are selected.

i) The design of a boundary-scan register cell that observes the signal received at a system logic input shall be such that, if a logic value *V* is present at the system input pin at the time when data are loaded from the signal into the shift-register stage of the boundary-scan register cell (in the *Capture-DR* controller state), the value shifted out of the cell through TDO during the immediate subsequent shifting of the boundary-scan register shall be *V*.

NOTE 10—For example, where a logic 0 is applied to a system input pin, a logic 0 has to be observed at TDO after loading of the cell that observes that pin. See Figure 11-13.

NOTE 11—Redundant observe-only cells may capture values other than the signal value received at a system logic input (e.g., a failure indication), and in that case are not necessarily constrained by this rule, but if they do observe the signal value received at a system input, and then this rule applies.

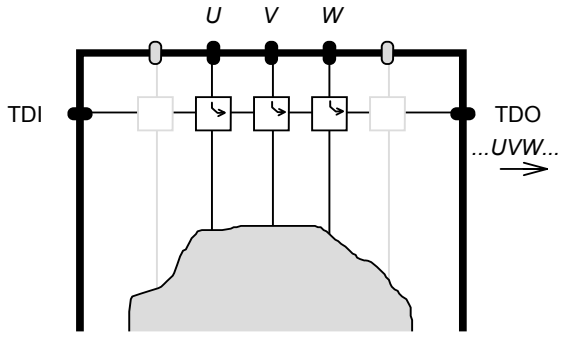


Figure 11-13—Noninversion of data between pin and TDO

j) For each control-and-observe cell that observes a signal received at the system logic from a system input pin *P*, a data value *D* shifted into the cell through TDI and subsequently driven to the on-chip system logic when the *INTEST* instruction is selected shall cause the same result as the application of value *D* at *P* during normal (nontest) operation of the component (Figure 11-14).

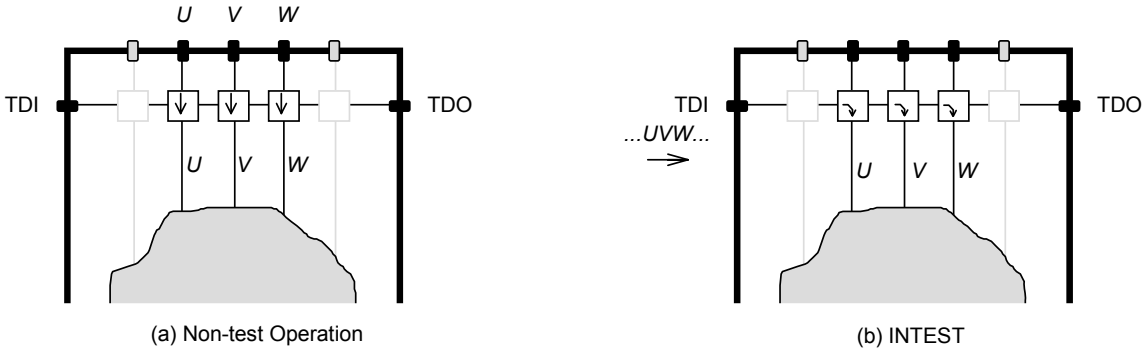


Figure 11-14—Noninversion of data between TDI and the system logic

k) When the *RUNBIST* instruction is provided and selected, the design of boundary-scan register cells that observe signals received at the on-chip system logic shall be such as to prevent interference with self-test execution as a result of data received from external sources (e.g., system input pins).

Permissions

l) *For clock inputs only*: If the *INTEST* instruction is supported, precisely one of the provided boundary-scan register cells that observes the signal received at a clock input to the on-chip system logic also may be able to control that input.

NOTE 12—See rule d) of 11.5.1.

m) The shift-register stage of a control-and-observe cell that drives an input to the on-chip system logic may be provided with a latched parallel output.

NOTE 13—Where the latched parallel output is omitted, transient data values will be driven to the on-chip system logic during shifting of the boundary-scan register when the *INTEST* instruction is selected. Where this would cause unwanted operation of the on-chip system logic, the parallel output of the shift-register stage can be latched such that the data driven to the system logic changes only on completion of shifting (in the *Update-DR* controller state).

NOTE 14—Where latched parallel outputs are provided, rule c) and permission h) of 11.3.1 apply.

n) Control-and-observe cells that drive inputs to the on-chip system logic may be designed such that when the *CLAMP*, *EXTEST,* or *RUNBIST* instruction is selected, the signal driven to the system logic is the parallel output of the shift-register stage.

o) Control-and-observe cells that drive inputs to the on-chip system logic may be designed such that when the *CLAMP*, *EXTEST,* or *HIGHZ* instruction is selected, a constant signal value is driven to the system logic.

p) Cells may be designed to act as generators of test patterns for the on-chip system logic when the *RUNBIST* instruction (or an alternative self-test instruction) is selected.

q) Cells may be controlled such that during the execution of *RUNBIST* or an alternative self-test instruction, data may flow between the system input pins and the on-chip system logic without modification.

NOTE 15—However, the results of executing the *RUNBIST* instruction will be independent of data received at nonclock system input pins [see rule j) of 8.10.1].

r) Input pins or inputs to the on-chip system logic may be observed by one or more redundant observe-only boundary-scan register cells in addition to the cells required by rule a) of 11.5.1.

NOTE 16—Such additional cells, which may be associated with either system or nonsystem input pins, are redundant in the sense that they could be omitted from the design without jeopardizing compliance to this standard (see 11.8 for the definition).

11.5.2 Description

In the example implementations for boundary-scan register cells contained in this clause, the routing of data through each cell is controlled by one or more mode-control signals (labeled Mode or Mode-N). Different mode-control signals are used for cells at input and output pins of the component, and these are derived from the instruction present at the parallel output of the instruction register.

Examples of gated-clock implementations for boundary-scan register cells located at system input pins are given in Figure 11-15 through Figure 11-19. The value of an appropriate <cell name> from the Standard BSDL Package that corresponds to each figure is provided for convenience (see Table B.3 and B.9), offset from the main caption within square brackets [ ]. However, it should be noted that the figures represent neither a preferred nor a required implementation of the named BSDL cell type.

Most figures have “Mode” inputs that perform the signal routing for various instructions and for the TMP controller. The generation of these signals is shown in associated tables, and unique “Mode” signals are given unique names. For convenience, some tables repeat the same “Mode” signal. As with the figures themselves, these are just examples. The precedence in the “Mode” generation table indicates that a row with a lower precedence dominates the conditions of higher precedence rows.

Note that rule e) of 11.4.1 permits the input to an observe-only boundary-scan register cell to be taken from any signal that is transparently driven from the system input via a noninverting path, for example, from a point in a signal distribution tree.

Table 11-2 shows the value of the Mode signal for the cells illustrated in Figure 11-15 and Figure 11-16 for each of the boundary-scan register instructions defined in Clause 8.

NOTE 1—When the *EXTEST* instruction is selected, the cell shown in Figure 11-15 feeds data received at the system input pin to the on-chip system logic. In many cases, the on-chip system logic will be tolerant of such signals, which usually will not be representative of those received during normal (nontest) operation. However, in some cases, it may be necessary to prevent flow-

through of received data, which can be done by adding a logic gate at the output from the cell to the on-chip system logic as shown in Figure 11-17, or the cell shown in Figure 11-19 may be used, which allows the update stage of the cell to control the data presented to the on-chip system logic while the *EXTEST* instruction is selected.

Table 11-2—Mode signal generation for the example cells in Figure 11-15 and Figure 11-16

Precedence	Instruction (Condition)	Mode2
1	(Cell in excluded segment)	0
2	*EXTEST*	0
	INTEST	1
3	(TMP controller *Persistence_on* state)	1
4	*PRELOAD*	0
	SAMPLE	0
	RUNBIST	X
	CLAMP	X
	CLAMP_HOLD	1
	CLAMP_RELEASE	1
	INIT_SETUP_CLAMP	1
	INIT_RUN	1
	Nonboundary instruction	0

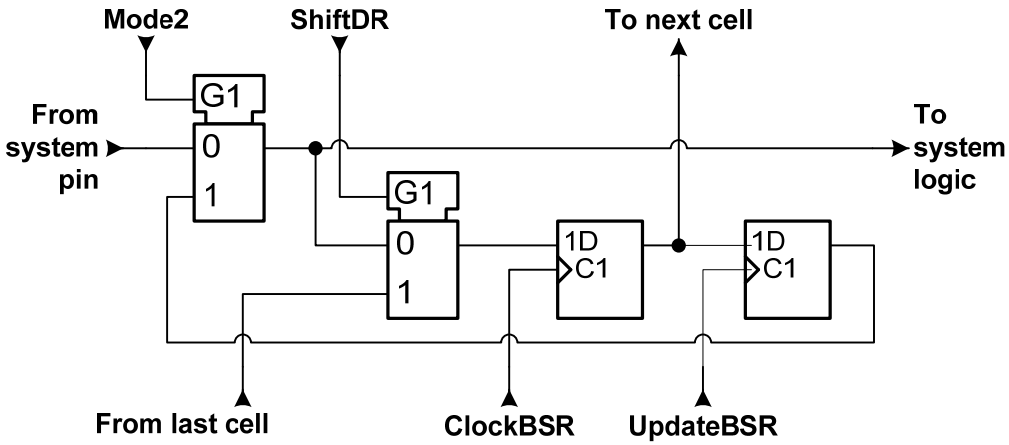


Figure 11-15—Input cell with parallel output register [BC_2]

NOTE 2—See Table 11-2 for mode signal generation.

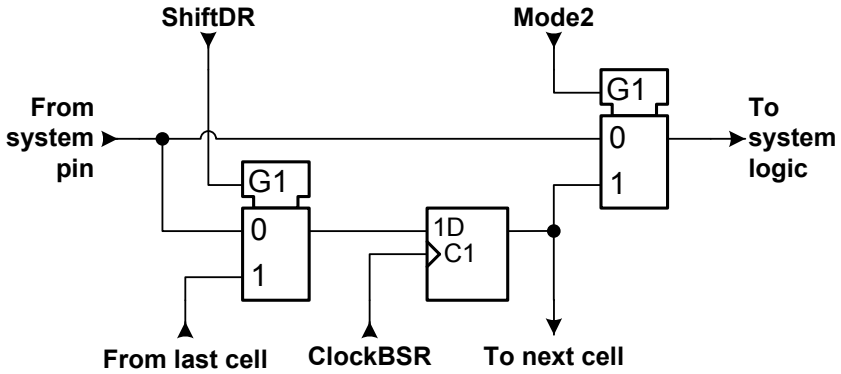


Figure 11-16—Input cell without parallel output register [BC_3]

NOTE 3—See Table 11-2 for mode signal generation.

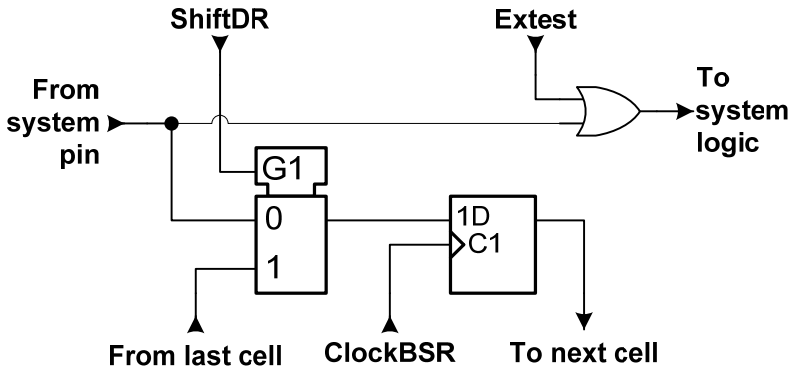


Figure 11-17—Cell that forces the system logic input to 1 during *EXTEST* [BC_4]

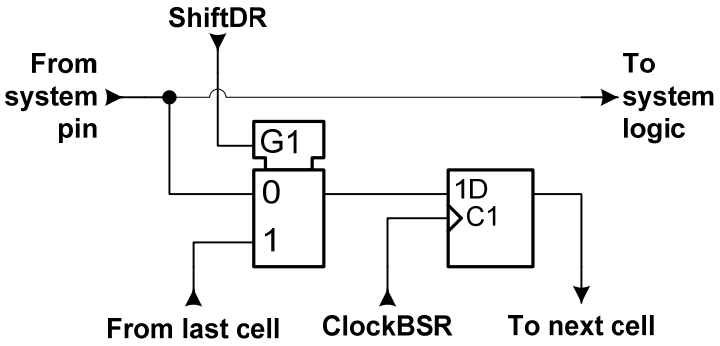


Figure 11-18—Observe-only input cell without control [BC_4]

The circuits in Figure 11-15 and Figure 11-16 allow the on-chip system logic to be driven from the boundary-scan register cell when the *INTEST* instruction is selected in accordance with rule f) of 11.5.1 and, for clock inputs, rule g3) of 11.5.1. The circuit of Figure 11-18 cannot drive signals into the system logic and may be used at a clock input in accordance with rule g1) of 11.5.1. The latter design can be used in circumstances where the delay introduced into the signal path by the multiplexer would cause a design target to be exceeded. (An example would be a high-performance clock pin.)

The circuit in Figure 11-17 implements permission o) of 11.5.1.

The design in Figure 11-15 includes a parallel output register that is updated from the shift-register stage in the *Update-DR* controller state [see permission m) of 11.5.1]. This register is included to prevent the changes at the output of the shift-register stage during shifting from being applied to the on-chip system logic when the *INTEST* instruction is selected (which could cause unwanted operation). Note that the parallel output could alternatively be held in a level-operated latch, enabled by a logic 1 on the *Update-DR* input from the example TAP controller (Figure 6-5 and Figure 6-6).

The design shown in Figure 11-19 can be used for boundary-scan register cells located at both system input and two-state system output pins, although the Mode signal applied to the cell may need to be different in each case. When the cell is used at a system input pin, the Mode signal should be controlled as shown in Table 11-3.

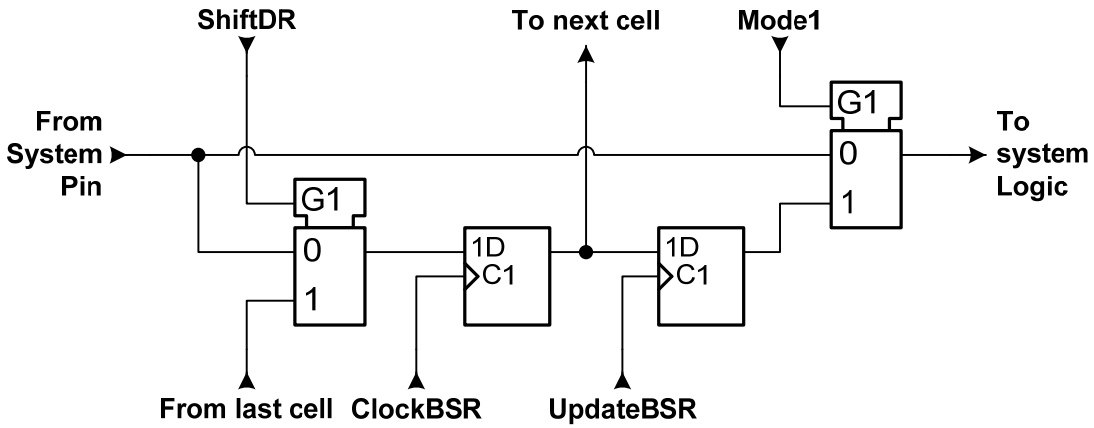


Figure 11-19—Input cell that supports all instructions [BC_1]

NOTE 4—See Table 11-3 for mode signal generation.

Table 11-3—Mode signal generation for the example cell in Figure 11-19

Precedence	Instruction (Condition)	Mode1
1	(Cell in excluded segment)	0
2	*EXTEST*	1
	INTEST	1
3	(TMP controller *Persistence_on* state)	1
4	*PRELOAD*	0
	SAMPLE	0
	RUNBIST	1
	CLAMP	1
	CLAMP_HOLD	1
	CLAMP_RELEASE	1
	INIT_SETUP_CLAMP	1
	INIT_RUN	1
	Nonboundary instruction	0

Note that the sole difference between the rules that apply to nonclock and clock system inputs is that the provision of control-and-observe cells is not mandatory at clock inputs when the *INTEST* instruction is supported. Thus, there is no requirement for circuitry to be inserted into the signal path between a clock input pin and the on-chip system logic.

11.6 Provision and operation of cells at system logic outputs

The rules in this subclause apply to outputs from the on-chip system logic. These outputs may drive data inputs of buffers at system output pins or at bidirectional system pins when they operate as system outputs. They also may drive output activity controls of buffers located at such pins. Alternatively, on-chip system logic outputs may drive on-chip mixed-signal circuit blocks that are not a part of the on-chip system logic.

Many of the rules presented in 11.6.1 apply to both control and data signals output from the on-chip system logic. However, some rules apply only to on-chip system logic outputs that drive data inputs of buffers at system output pins, while others apply only to outputs that drive control inputs at such buffers. In cases in which a rule is intended to have limited application to one or the other use of a system logic output signal, the rule is prefaced by the phrase "*Control (Data) inputs to buffers only:*."

NOTE 1—Inputs to on-chip analog/digital circuit blocks are considered to be equivalent to data inputs to output buffers.

NOTE 2—Where the optional *HIGHZ* instruction is provided, selection of this instruction will place every output pin in an inactive drive state, including pins where there is no system requirement for three-state capability. Where three-state capability will be provided solely to allow implementation of the *HIGHZ* instruction, the pin should be treated as a two-state pin for the purposes of this standard.

NOTE 3—This subclause addresses provision of boundary-scan register cells at outputs from the on-chip system logic in cases where these outputs drive system output pins during normal (nontest) operation. The case in which an output from the on-chip system logic drives a system bidirectional pin during normal (nontest) operation is discussed in 11.7.

11.6.1 Specifications

Rules

a) *Data inputs to buffers only*: For each output of the on-chip system logic that drives the data input of a buffer at a system logic output pin, at least one control-and-observe boundary-scan register cell shall observe only one of the following:

   1) The signal driven from the system logic output (Figure 11-20a)
   2) The signal at the corresponding output pin (Figure 11-20b)

b) *Control inputs to buffers only*: For each output of the on-chip system logic that drives the control input of a buffer at a system logic output pin, at least one control-and-observe boundary-scan register cell shall observe the signal driven from the system logic output (Figure 11-20c).

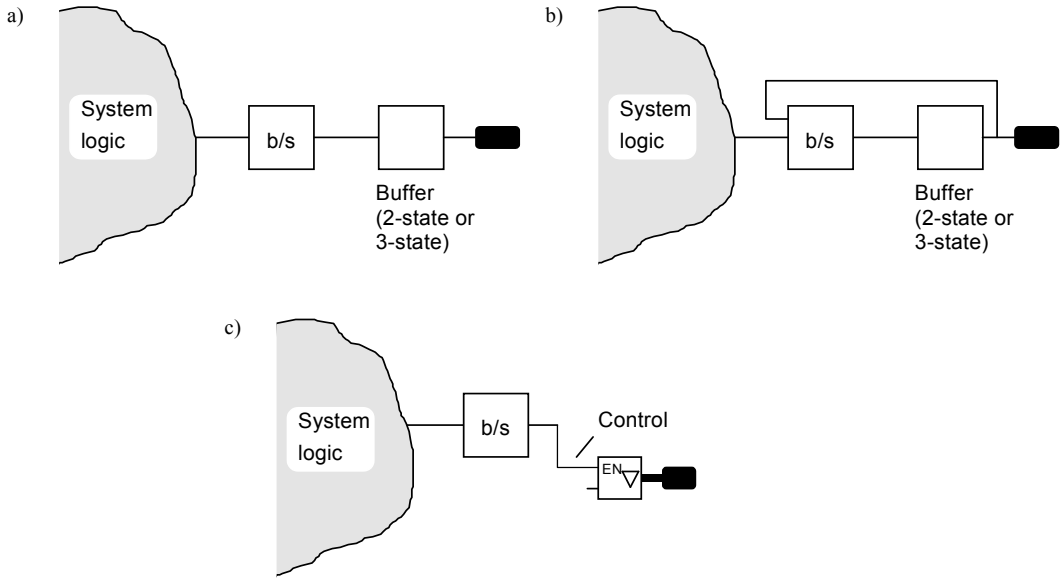


Figure 11-20—Provision of a boundary-scan register cell at a digital system output pin

c) *Data inputs to buffers only*: For a given data input to an output buffer, precisely one of the boundary-scan register cells that satisfy rule a) of this subclause shall be capable of driving that data input (see Figure 11-21).

NOTE 1—Rule a) [rule b)] of this subclause provides that at least one cell will monitor any given data (control) output from the on-chip system logic. There may be more than one such cell. If, for example, an output from the on-chip system logic fans out to several system output pins, rule c) and rule e) require precisely one cell capable of driving the connected pin to be placed in each fan-out branch. Additional redundant observe-only cells may be included in the design [permission t)] that can observe the output from the on-chip system logic or system output pin but that cannot drive any system output pin (see 11.8).

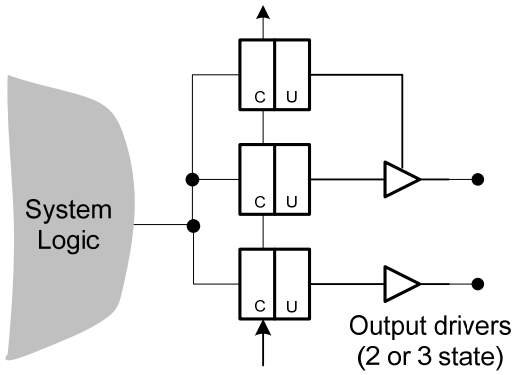


Figure 11-21—Provision of boundary-scan register cells at system logic outputs

d) *Control inputs to buffers only*: For a given control input to an output buffer, precisely one of the boundary-scan register cells that satisfy rule b) of this subclause shall be capable of driving that control input.

NOTE 2—See also rule e) and rule f).

e) The parallel output of a control-and-observe boundary-scan register cell that observes a system output from the on-chip system logic shall drive only one of the following:

1) A single data input to a system output buffer (two-state or three-state, or the output signal for a bidirectional pin)
2) The control inputs to a set of output buffers

NOTE 3—In the latter case, see rule f). Note that where a single output from the on-chip system logic is used both as a control for the output buffers at a group of pins and as the data signal output at one or more other pins, separate boundary-scan register cells are required for the control and noncontrol signal paths, as illustrated in Figure 11-22.

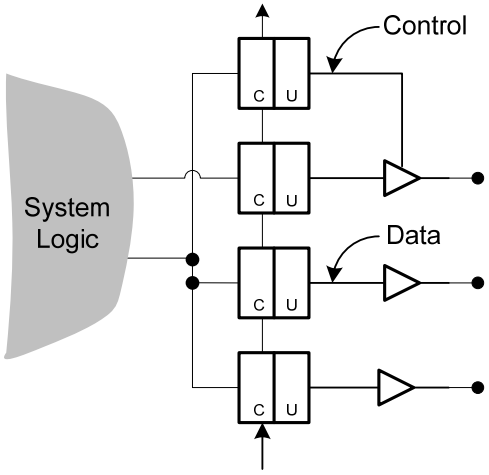


Figure 11-22—Provision of cells when one output Is used both as control and data

f) *Control inputs to buffers only*: Where one boundary-scan register cell drives control inputs to output buffers at several system output pins, a given data value (0 or 1) held in the cell shall cause the same operation at all connected system output pins (e.g., a 0 may be defined to place all connected pins in an inactive drive state).

NOTE 4—See also recommendation q) of this subclause.

g) *Control inputs to buffers only*: In cases where a single control signal is driven to a set of system output pins that includes both three-state and bidirectional pins, in interpreting rule f), enabling the three-state system output(s) shall be considered to be the same as setting bidirectional system pin(s) to output mode; disabling three-state system output(s) shall be considered to be the same as setting bidirectional pin(s) to input mode.

h) Each cell that observes a system logic output shall be designed to route signals as shown in Table 11-4.

i) *Control inputs to buffers only*: For each relevant instruction, the option in the right-hand column of Table 11-4 to drive a value that disables connected output buffers shall be selected either for all cells that drive control inputs of output buffers or for none of these cells.

NOTE 5—For example, if on selection of the *INTEST* instruction one three-state output pin of the component was forced to the high-impedance state, all other three-state output pins also would be forced to the high-impedance state when the *INTEST* instruction was selected.

j) Every control-and-observe boundary-scan register cell that observes an output from the on-chip system logic (data or control) shall be provided with a latched parallel output.

k) When the *EXTEST*, *PRELOAD*, or *INTEST* instruction is selected, the latched parallel outputs of control-and-observe boundary-scan register cells that observe outputs from the on-chip system logic shall latch the data held in the shift-register stage on the falling edge of TCK in the *Update-DR* controller state.

l) When the *CLAMP, HIGHZ,* or *RUNBIST* instruction is selected, the latched parallel outputs of control-and-observe boundary-scan register cells that observe outputs from the on-chip system logic shall retain their state unchanged in all controller states.

m) *Data inputs to buffers only*:

1) If *C* is the control-and-observe boundary-scan register cell that drives data to an output buffer for system output pin *P*
2) If *O* is the output of the on-chip system logic that is observed by *C* and that drives data to *P* during normal (nontest) operation
3) If *C* is the *n*th cell of the boundary-scan register
4) If *V* is the logic signal driven from *P* in normal (nontest) operation (i.e., when the signal driven from *P* is determined by the output from *O*)

When an instruction is selected that requires the output of *O* to be captured into *C* (e.g., the *SAMPLE* instruction), the logic value observed as the *n*th bit output from the boundary-scan register at TDO in the scan operation immediately after capture shall be *V* (Figure 11-23).

NOTE 6—For example, where the system logic would drive a logic 0 through the system output pin, a logic 0 has to be observed at TDO after loading of the cell that observes that pin. See Figure 11-23.

NOTE 7—For outputs where only one logic value is actively driven (e.g., open-collector outputs), receipt of one data value (0 or 1) from the on-chip system logic will cause the output to be inactive. In these cases, the data value observed at TDO will be the value that, when fed to the output buffer from the on-chip system logic, will cause the output to be inactive.

Table 11-4—Routing of signals in cells at system logic outputs

Instruction	The signal loaded into the shift-register stage of each cell on the rising edge of TCK in the *Capture-DR* controller state is:	The signal driven from the parallel output of each control-and-observe cell while the instruction is selected is:
CLAMP, INIT_RUN, INIT_SETUP_CLAMP, CLAMP_HOLD, CLAMP_RELEASE	Not relevant	The latched parallel output of shift-register stage
EXTEST	Not defined	The latched parallel output of shift-register stage
HIGHZ	Not relevant	The value that disables connected output buffers
INTEST	The signal output from the on-chip system logic	The latched parallel output of shift-register stage or the value that disables connected output buffers[a]
PRELOAD	Not defined	The signal output from the on-chip system logic
RUNBIST	Not defined (not relevant unless the boundary-scan register is selected as the serial path between TDI and TDO)	The latched parallel output of shift-register stage or the value that disables connected output buffers[a]
SAMPLE	The signal output from the on-chip system logic or from the connected output pin *For redundant observe-only cells:* The signal output from the on-chip system logic or from the connected output pin, or a constant value signal or an undefined signal[b]	The signal output from the on-chip system logic
BYPASS, IDCODE, USERCODE, INIT_SETUP	Not relevant	The signal output from the on-chip system logic

[a] This option is available only to cells that drive control inputs of output buffers. Where this option is selected for the control input to an output buffer, the data input to that buffer when the instruction is selected may be regarded as "Not defined."

[b] See rule b) of 11.8.1.

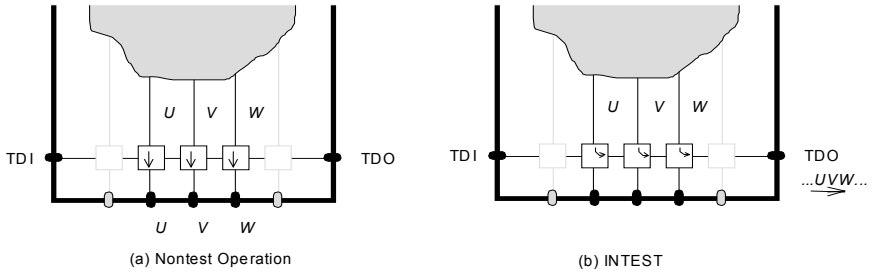


Figure 11-23—Noninversion of data between the system logic and TDO

n) *Data inputs to buffers only*:

1) If *C* is the control-and-observe boundary-scan register cell that drives data to an output buffer for system output pin *P*; and
2) If *C* is the *n*th cell of the boundary-scan register; and
3) If *V* is the value of the *n*th bit of a serial data stream *S* input to the boundary-scan register via TDI; and
4) If the length of *S* is equal to the number of cells in the boundary-scan register

When an instruction is selected that causes the latched parallel output of *C* to determine the value of the data signal driven from *P* (e.g., *EXTEST*) and when the data stream *S* is shifted into the boundary-scan register and, immediately subsequent to the shifting operation, updated to the latched parallel outputs of the boundary-scan register, the value of the data signal output from *P* shall be *V* (Figure 11-24).

NOTE 8—For outputs where only one logic value is actively driven (e.g., open-collector outputs), receipt of one data value (0 or 1) from the on-chip system logic will cause the output to be inactive. In these cases, the data value input at TDI would need to be the value that, when fed to the output buffer from the on-chip system logic, will cause the output to be inactive.

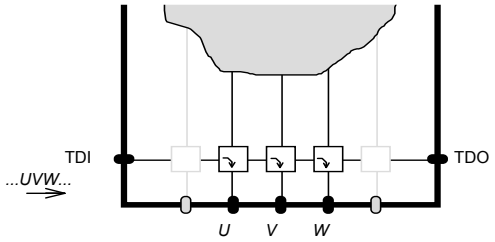


Figure 11-24—Noninversion of data between TDI and a system output pin

o) *Control inputs to buffers only*:

1) If *C* is the control-and-observe boundary-scan register cell that drives the control input of the output buffer for a system output pin *P*
2) If *O* is the output of the on-chip system logic that is observed by *C* and that controls the activity of *P* during normal (nontest) operation
3) If C is the *n*th cell in the boundary-scan register
4) If *V* (or not *V*) is the value of the *n*th bit of S output from the boundary-scan register through TDO immediately following capture of *O* into *C* when *P* would be active (or inactive, respectively) in normal (i.e., nontest) operation
5) If the length of *S* is equal to the number of cells in the boundary-scan register

When an instruction is selected that causes the latched parallel output of *C* to determine the activity of *P* (e.g., *EXTEST*) and when a data pattern *S* containing *V* (or not *V*) as the *n*th bit is shifted onto the latched parallel outputs of the boundary-scan register, *P* shall be active (or inactive, respectively).

NOTE 9—For example, where a logic 0 output from the system logic normally would disable a driven output buffer, a logic 0 should be shifted out through TDO whenever the output would have been disabled. Furthermore, a logic 0 shifted into the cell through TDI should, if driven to the output buffer, cause the output buffer to be disabled. The effect of this rule is similar to that of rule m) and rule n) of this subclause (Figure 11-25).

NOTE 10—Boundary cells associated with output pins may capture values from the output pin; in which case, they must observe rule i) of 11.5.1.

NOTE 11—Redundant observe-only cells associated with output pins may capture the possible fault characteristics of a system logic output or output pin other than the digital signal value.

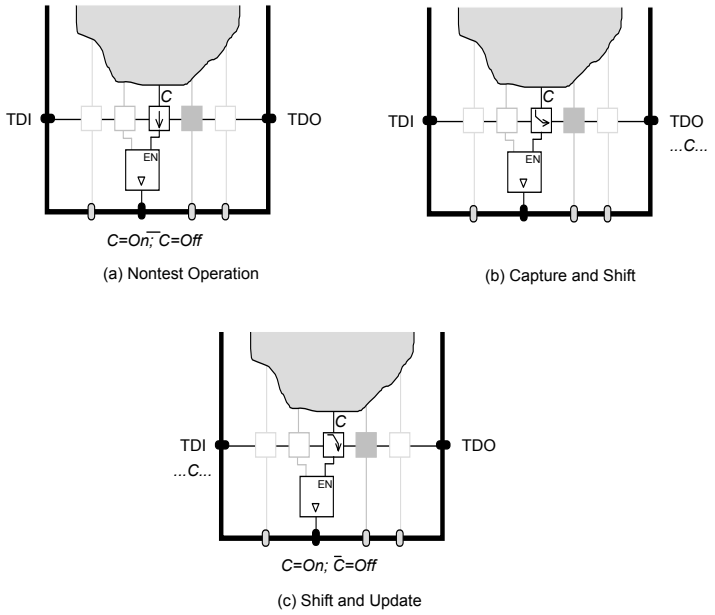


Figure 11-25—Noninversion of control signal values between the system logic and TDO

p) *Control inputs to buffers only*: If the latched parallel output of a boundary-scan register cell that drives a control input to an output buffer is reset in the *Test-Logic-Reset* controller state, it shall be reset to the state that will cause the connected output buffer(s) to be disabled.

NOTE 12—The timing of the reset is specified in rule c) and permission h) of 11.3.1.

Recommendations

q) *Control inputs to buffers only*: The control signal for each functionally distinct group of system output pins (e.g., an address bus or a data bus) should be driven from a distinct boundary-scan register cell dedicated to that purpose, even where the output from the on-chip system logic observed by that cell normally would drive a common control signal to more than one such group of pins.

NOTE 13—This reduces the complexity of the test generation task because during the test, buffers can be enabled independently as required.

r) Where practical, at least one boundary-scan register cell should observe the signal at the corresponding output pin.

NOTE 14—This significantly improves coverage of shorts in board test and may be accomplished by either a self-monitoring boundary cell (see Figure 11-33 and Figure 11-34) or a redundant observe-only cell. Shorts between two output ports on a component are particularly difficult to detect unless the shorted nets are observed at a distance, or if both are three-state capable and one at a time is driving. Analog simulations of shorted drivers can assist in understanding the issues.

Permissions

s) Boundary-scan register cells that observe outputs from the on-chip system logic may be designed to act as a part of a signature computing register for test results when the *RUNBIST* instruction (or alternative self-test instruction) is selected.

t) Outputs from the on-chip system logic or output pins may be observed by one or more redundant observe-only boundary-scan register cells in addition to the control-and-observe cells required by the preceding rules.

NOTE 15—Such additional cells are redundant in the sense that they could be omitted from the design without jeopardizing compliance to this standard (see 11.8 for their definition).

11.6.2 Description

For two-state output pins, where signals only can be at the high or low logic level at any given instant, one boundary-scan register cell is sufficient to allow the state of the pin to be controlled or observed. However, for three-state pins, the capability exists for data to be driven actively or inactively, such that four states are possible. Data from a minimum of two boundary-scan register cells therefore are required to allow the state (signal value plus active/inactive) of a three-state pin to be controlled or observed.

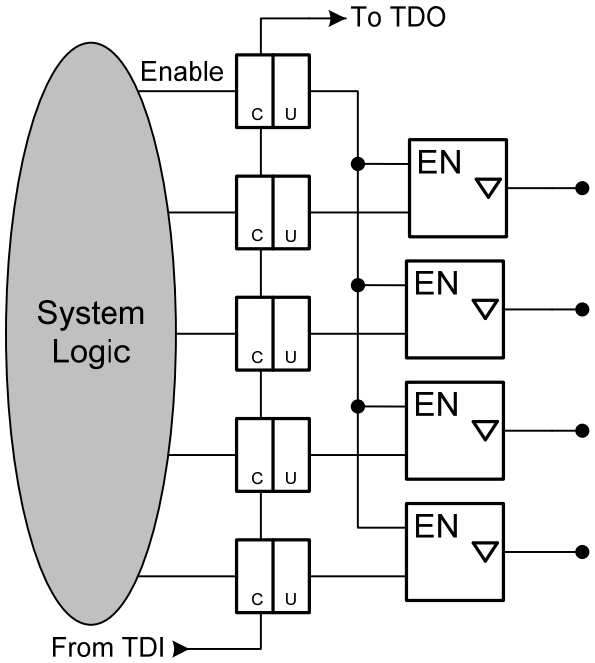


Figure 11-26—Control of multiple three-state outputs from one signal

Although it would appear that the additional cells might significantly increase the overhead needed to implement a boundary scan, it is necessary to provide only one additional cell for each activity control signal generated in the circuit, although a judicious use of a few additional cells is recommended in recommendation q) of 11.6.1 (see, for example, Figure 11-26). Thus, where the activity of many three-state output pins is controlled from a single source, for example in a microprocessor address bus, only one additional cell is required to give the necessary control. Since, given the basic design of the circuit, it would be a design error if such three-state pins were wired together, there is no need for the design of the boundary-scan register to account for this possibility.

The need for the additional cells at output control signals is illustrated in Figure 11-27. This shows a wired junction between three-state outputs from a number of components. To test this junction, a series of tests need to be performed, each of which checks that one of the outputs can drive either a 0 or a 1 to the receiving devices. During each test, the other outputs have to be set to the opposite data value (1 or 0, respectively) with a high-impedance drive. Table 11-5 shows the pair of tests needed to check the operation of one of the outputs connected to the junction.

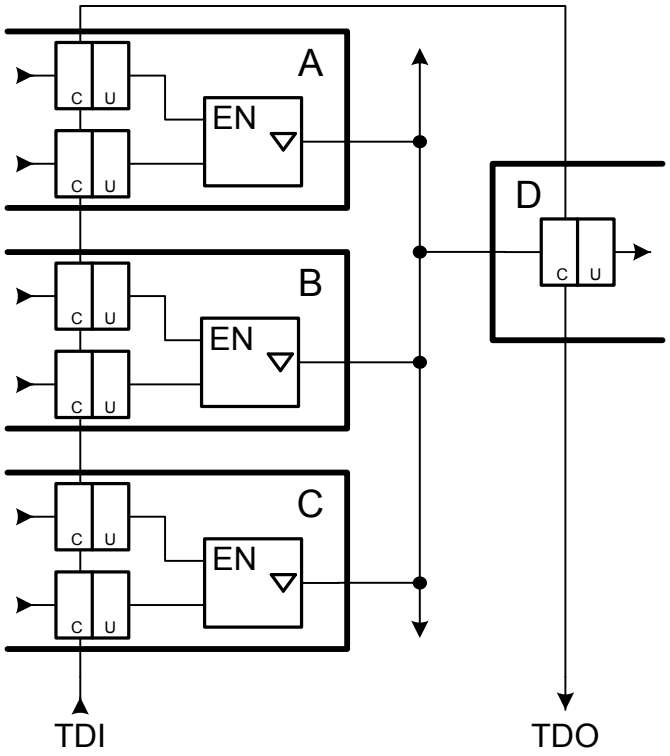


Figure 11-27—Testing board-level bus lines

Table 11-5—Test for driver B

Stimulus applied to the bus from			Result seen at
Component A	**Component B**	**Component C**	**Component D**
1/off	0/on	1/off	0
0/off	1/on	0/off	1

To apply the test, it is necessary to be able to control both the data value at each output and whether the output is enabled. This can be done via the boundary-scan register independently of the on-chip system logic.

For similar reasons, there shall be additional boundary-scan register cells associated with each three-state bidirectional system pin that control the activity of the associated output buffer. As in the case of three-state system pins, these cells may be shared across a bus or between any group of three-state bidirectional system pins that obtain their output control signal from a single source.

Figure 11-28 highlights the following problems that might be encountered when applying tests to logic blocks external to a component by using the boundary-scan register of the component but that are minimized by implementing boundary-scan register cells as defined in this subclause.

— The logic block being tested may contain asynchronous sequential logic that will be set into undesirable states if shifting patterns appear at its inputs.
— The signals applied from the boundary-scan register may feed into clock inputs on the logic block being tested, which again will produce undesirable effects if the logic is not shielded from shifting patterns.

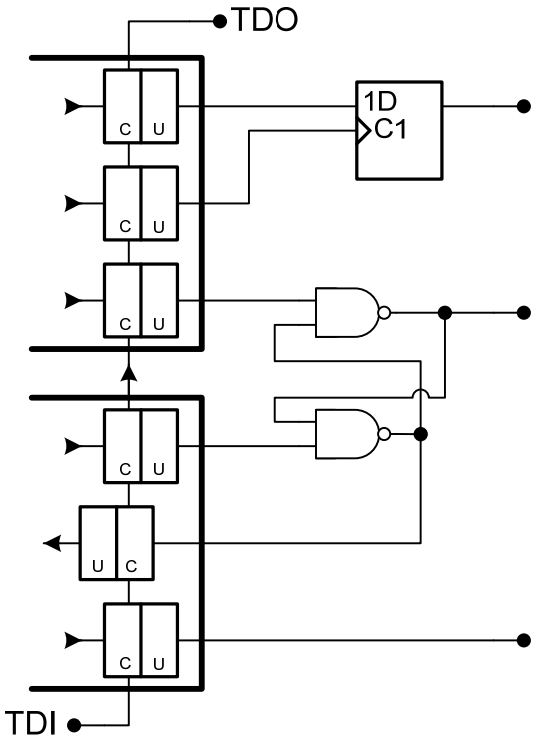


Figure 11-28—Testing external logic via the boundary-scan register

Since in a generally applicable architecture it cannot be guaranteed that such features do not exist in the circuitry under test, the boundary-scan design shall be such that these problems are minimized. A design compatible with this standard does this by requiring a parallel output register or latch in each boundary-scan register cell that can affect the state of an output driver at a system pin. While the *EXTEST* instruction is selected, inclusion of this register or latch allows the data driven from a component to neighboring circuitry to change only on completion of the shifting process.

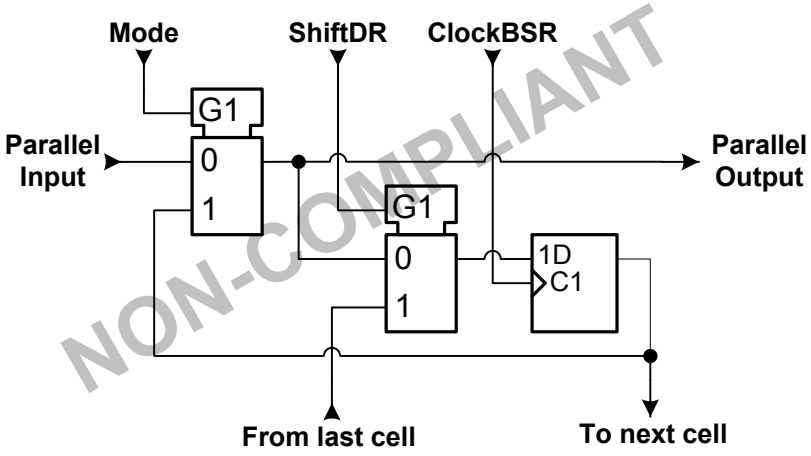


Figure 11-29—Primitive noncompliant output cell design with potential problems

A further potential problem is highlighted by the primitive (noncompliant) boundary-scan register cell design shown in Figure 11-29. During testing of the on-chip system logic (for example, through the *INTEST* or *RUNBIST* instruction), the example cell would allow responses from the system logic to pass through the data-path multiplexer to the shift-register input of the cell. This allows the output response from the on-chip system logic to be loaded into the output boundary-scan register cells and shifted out for inspection. However, a problem arises from the fact that the cell also allows the test response from the on-chip system logic to be output from the host component and hence to be applied to neighboring components on a board assembly.

The application of raw test-response data from one component can have a damaging effect on other components in the circuit if it is received at clock or asynchronous data or control inputs. For example, if built-in self-testing is being performed on the memory controller of Figure 11-30, there is a distinct possibility that one or more test-response patterns from the core logic of the memory controller will cause simultaneous activation of the outputs feeding the component select (CS) inputs of the memory devices. This situation would not occur during normal operation of the complete design, either due to constraints between the logic values applied to the inputs of the memory controller or due to the design of the on-chip system logic. The system logic design would in some way prevent more than one active output from the controller at any time.

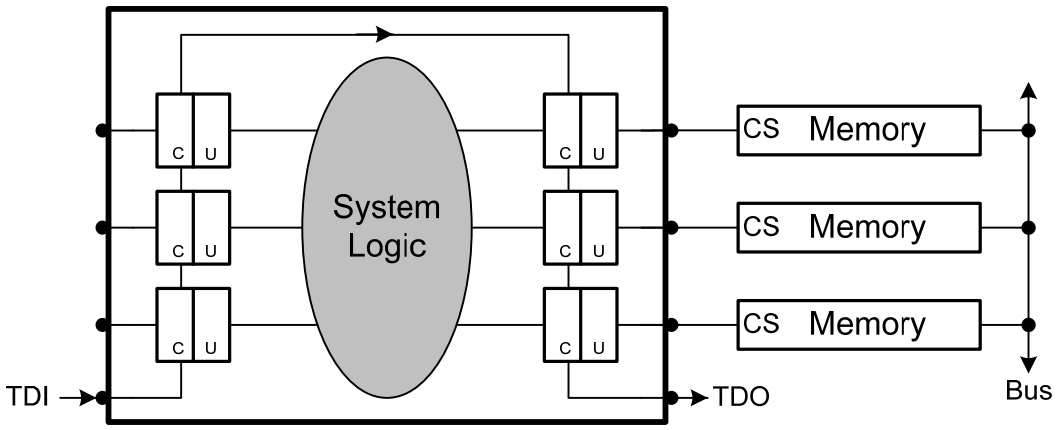


Figure 11-30—Circuit illustrating potential boundary-scan test problem

The duration of an on-chip test is dependent on the type of system logic test performed. For static tests applied using the *INTEST* instruction, these potentially damaging output patterns can remain in effect over the interval between successive occurrences of the *Update-DR* controller state. For instance, in a circuit having a scan path length of 500 bits and a TCK rate of 5 MHz, the approximate interval between closest consecutive *Update-DR* controller states is 100 ms. For large board designs, the period could be sufficiently long to cause damage to drivers in contention on a bus.

One solution is to cause the output buffers of the memory controller that feed the memory CS inputs to be placed in a high impedance state during internal testing of the controller. However, floating inputs can fluctuate between high and low logic levels and are susceptible to induced voltages from adjacent board wiring interconnects. Applying a pull-up resistor on the three-state buffers will solve the bus contention problem in external components with active-low three-state enables, but those with active-high three-state enables are still at risk.

The solution adopted in this standard is to provide boundary-scan register cells to be placed at two-state output pins, which are designed such that specific logic values can be placed at the associated pins while system logic within the component is tested. Figure 11-31 shows a gated-clock design that provides this facility and meets the rules defined in this clause. Table 11-6 shows how the Mode signal for Figure 11-31 is derived for each of the boundary-scan register instructions defined in Clause 8.

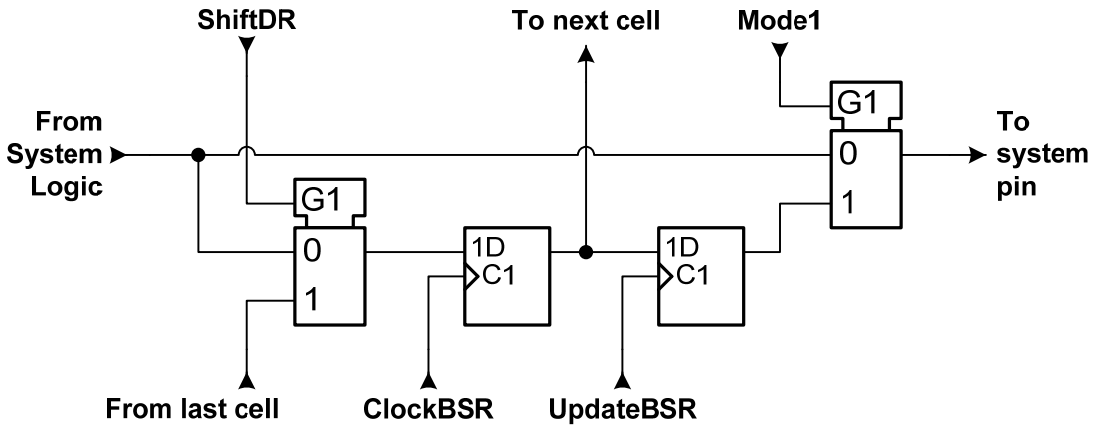


Figure 11-31—Output cell that supports all instructions [BC_1]

NOTE 1—See Table 11-6 for mode signal generation.

Table 11-6—Mode signal generation for the example cells in Figure 11-31, Figure 11-35, Figure 11-37, and Figure 11-47

Precedence	Instruction (Condition)	Mode1
1	(Cell in excluded segment)	0
2	*EXTEST*	1
	INTEST	1
3	(TMP controller *Persistence_on* state)	1
4	*PRELOAD*	0
	SAMPLE	0
	RUNBIST	1
	CLAMP	1
	CLAMP_HOLD	1
	CLAMP_RELEASE	1
	INIT_SETUP_CLAMP	1
	INIT_RUN	1
	Nonboundary instruction	0

Note that the path in Figure 11-31 between the data input from the system logic and the multiplexer that feeds data to the system pin will not be used during execution of either the *EXTEST* or the *INTEST* instruction. In some cases, it may therefore be necessary to use additional test operations at the board level to test the circuitry within a component fully.

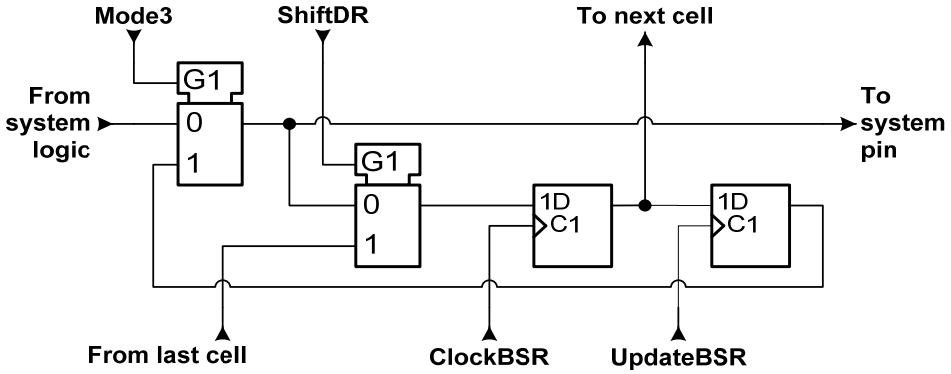


Figure 11-32—Output cell that does not support *INTEST* [BC_2]

NOTE 2—See Table 11-7 for mode signal generation.

The example gated-clock cell design in Figure 11-32 could be used where the *INTEST* instruction is not supported by a component since this design does not permit rule h) of 11.6.1 to be met with respect to the *INTEST* instruction. Note that while the cell meets rule h) of 11.6.1 with respect to the *SAMPLE* instruction without additional provision, if the cell drives off-chip via an output buffer, then the signal value captured using the *SAMPLE* instruction is the one intended to be driven off-chip, not the one actually on the off-chip connection at the time. The latter may be affected by faults on the off-chip connection or, for bus connections, by the combination of drivers active at the time. By ensuring that the signal that should have been driven from the component is sampled at the driving end, while the signal actually driven is sampled at the receiving end, additional diagnostic information is obtained.

Table 11-7 shows how the Mode signal for Figure 11-32 is derived for each of the boundary-scan register instructions supported by the cell design.

Table 11-7—Mode signal generation for the example cells in Figure 11-32, Figure 11-34, and Figure 11-40

Precedence	Instruction (Condition)	Mode3
1	(Cell in excluded segment)	0
2	*EXTEST*	1
3	(TMP controller *Persistence_on* state)	1
4	*PRELOAD*	0
	SAMPLE	0
	RUNBIST	1
	CLAMP	1
	CLAMP_HOLD	1
	CLAMP_RELEASE INIT_SETUP_CLAMP	1
	INIT_RUN	1
	Nonboundary instruction	0
NOTE—Mode3 is the same as Mode1 except that the *INTEST* instruction is not supported.		

On the other hand, it is highly useful that the signal value captured using the *EXTEST* instruction is the one at the corresponding system output pin, according to rule a2) of 11.6.1. Doing so allows a connected system network both to be driven and to be captured at the same pin, thus, allowing such networks to be tested for shorts to others even where there are no other connected boundary-scan device pins. Output boundary cells of this type are termed to be self-monitoring, similar to the test data register cell shown in Figure 9-10, which captures its own output. A gated-clock cell design that implements this option while still capturing the system logic output during *SAMPLE* and *INTEST* is shown in Figure 11-33. An alternative and simpler gated-clock cell design that also implements this option but cannot capture the system logic output and therefore does not support *INTEST* is shown in Figure 11-34.

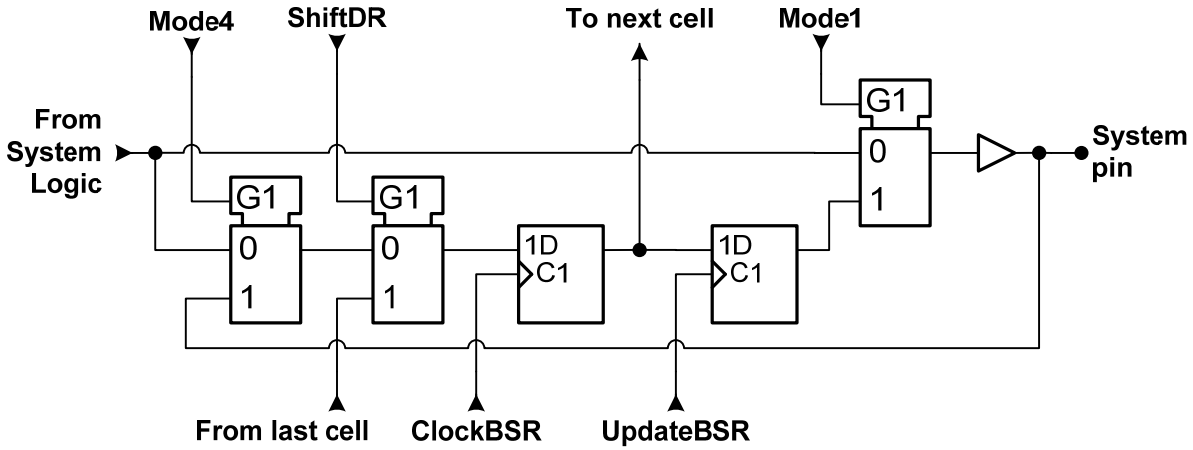


Figure 11-33—Self-monitoring output cell that supports *INTEST* [BC_9]

NOTE 3—See Table 11-8 for mode signal generation.

Table 11-8—Mode signal generation for the example cell in Figure 11-33

Precedence	Instruction (Condition)	Mode4	Mode1
1	(Cell in excluded segment)	0	0
2	*EXTEST*	1	1
	INTEST	0	1
3	(TMP controller *Persistence_on* state)	X	1
4	*PRELOAD*	X	0
	SAMPLE	0	0
	RUNBIST	X	1
	CLAMP	X	1
	CLAMP_HOLD	X	1
	CLAMP_RELEASE INIT_SETUP_CLAMP	X	1
	INIT_RUN	X	1
	Nonboundary instruction	0	0

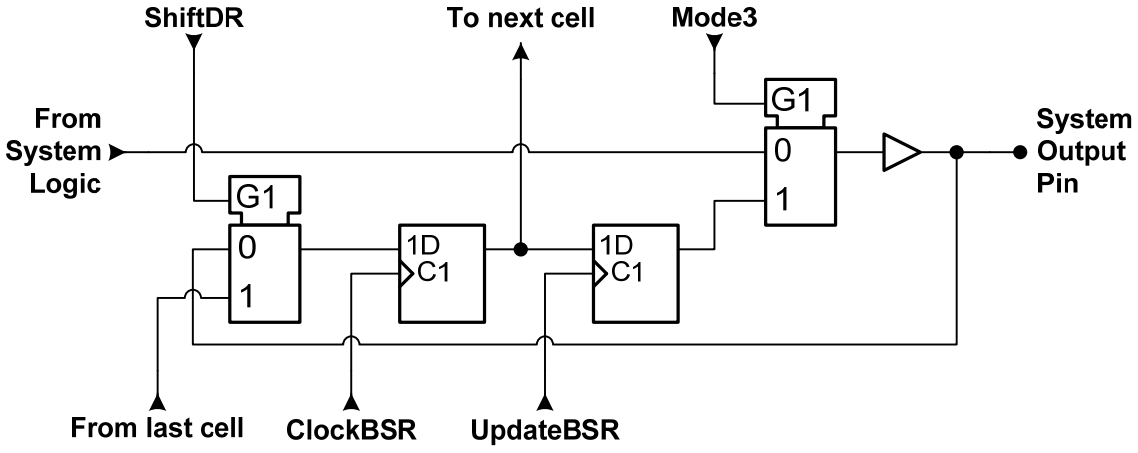


Figure 11-34—Self-monitoring output cell that does not support *INTEST* [BC_10]

NOTE 4—See Table 11-7 for Mode Signal Generation.

Note that the receiver is not shown in the two self-monitoring output cells, but it would be necessary in most cases. Also note that the value captured in the cell should follow the rules for cells at system inputs, and in particular, the "noninversion" rule i) of 11.5.1.

Where a component has three-state system output pins, these pins may feed onto a wired junction at the board level. To test the interconnections forming the wired junction using the *EXTEST* instruction, it shall be possible to drive independently onto the junction from each of the possible driving pins. As was discussed earlier in this clause, to achieve this, it is necessary to be able to control the output control signals fed to the output drivers at three-state or bidirectional system pins.

In addition, it is necessary to minimize the possibility of contention from occurring on board-level interconnections when the on-chip system logic is tested using the *INTEST* or *RUNBIST* instruction. This requirement can be met in either of two ways:

a) The state of a system pin can be fully defined by shifting data into the boundary-scan register.
b) A system pin can be forced into the inactive drive state. This additional option is possible since the board-level circuit design shall necessarily be designed such that components driven from the three-state bus do not erroneously respond to high-impedance conditions during normal system operation. Therefore, the inactive drive state can be safely driven during testing of the system logic within a component.

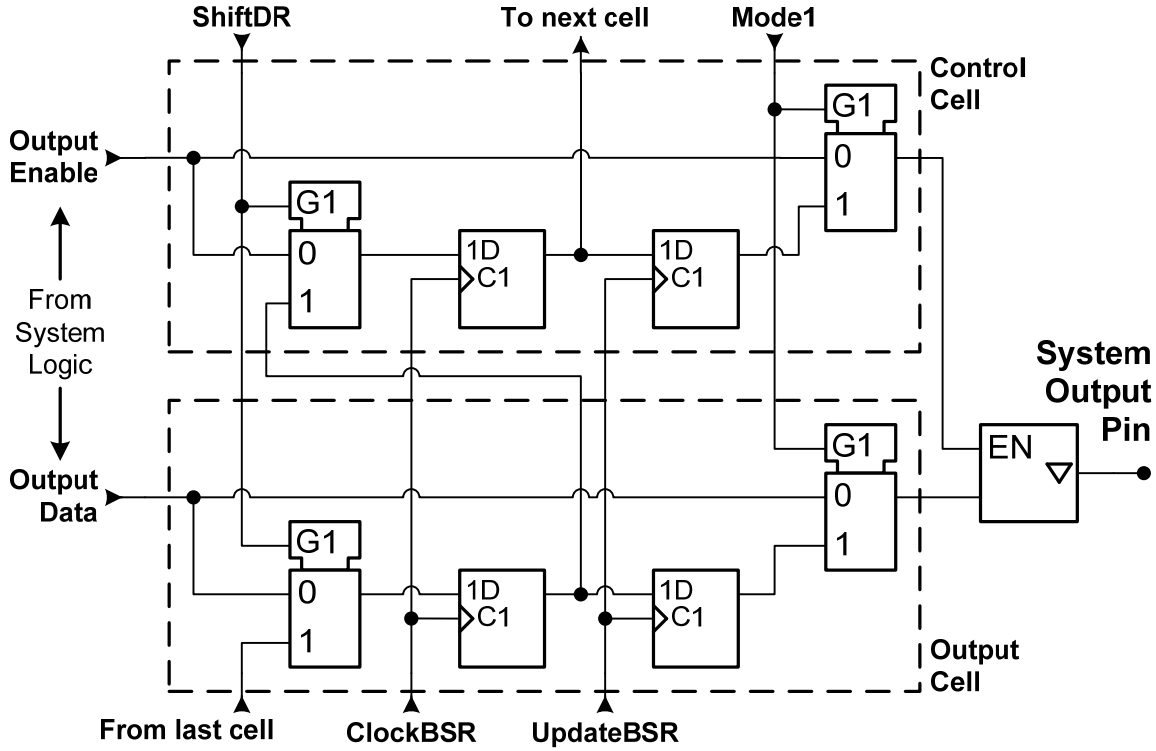


Figure 11-35—Boundary-scan register cells at a three-state output—Example 1 [BC_1, control and data]

NOTE 5—See Table 11-6 for mode signal generation.

The options listed for the *INTEST* and *RUNBIST* instructions in rule h) of 11.6.1 cover these two possibilities. Figure 11-35 and Figure 11-36 give example gated-clock designs for a boundary-scan register cell that could be used at a three-state system output pin. Figure 11-35 implements option a), while Figure 11-36 implements option b). In Figure 11-35, the Mode signal should be controlled as shown in Table 11-6.

In Figure 11-36, the design of the circuitry around the shift-register stages is such that all paths can be tested if both the *EXTEST* and *INTEST* instructions are executed with appropriate data. The Mode_1 and Mode_2 signals should be controlled as shown in Table 11-9.

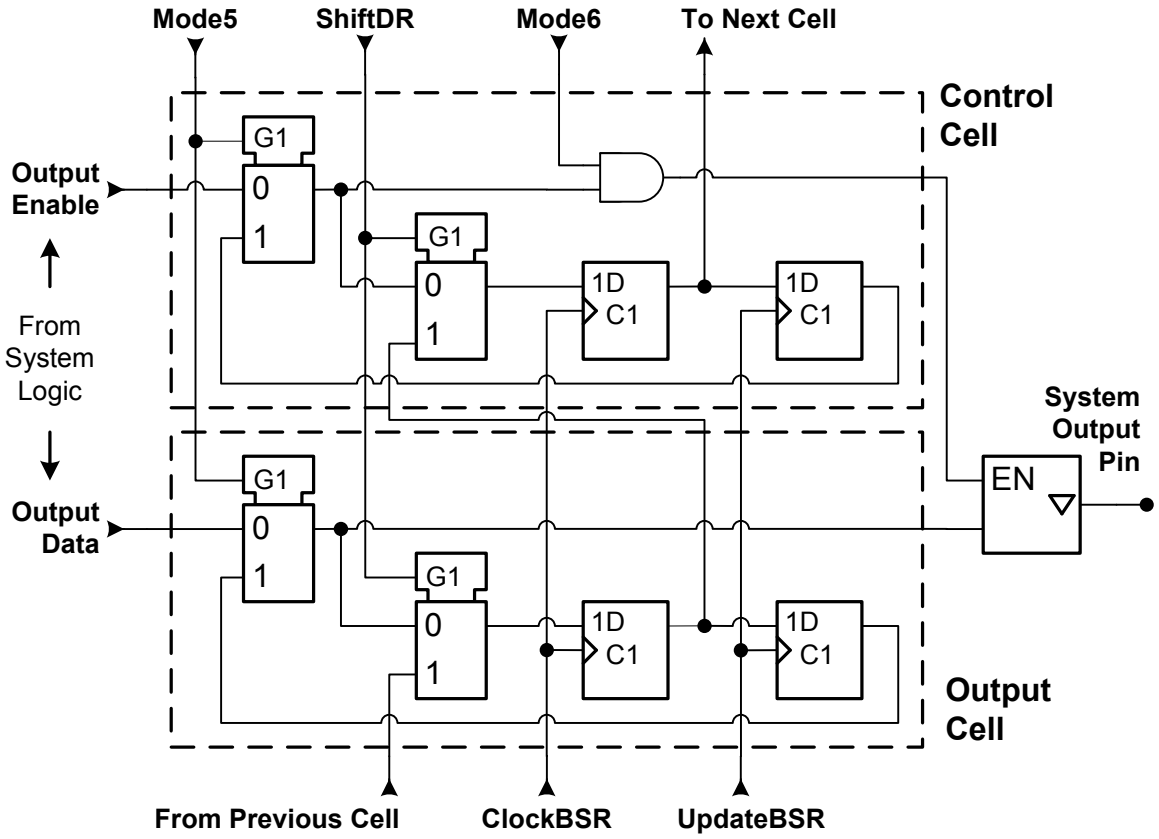


Figure 11-36—Boundary-scan register cells at a three-state output—Example 2 [BC_2, control and data]

NOTE 4—See Table 11-9 for mode signal generation.

Table 11-9—Mode signal generation for the example cell in Figure 11-36

Precedence	Instruction (Condition)	Mode5	Mode6
1	(Cell in excluded segment)	0	1
2	*EXTEST*	1	1
	INTEST	0	0
3	(TMP controller *Persistence_on* state)	1	1
4	*PRELOAD*	0	1
	SAMPLE	0	1
	RUNBIST	0	0
	CLAMP	1	1
	CLAMP_HOLD	1	1
	CLAMP_RELEASE	1	1
	INIT_SETUP_CLAMP	1	1
	INIT_RUN	1	1
	HIGHZ	X	0
	Nonboundary instruction	0	1

11.7 Provision and operation of cells at bidirectional system logic pins

11.7.1 Specifications

Rules

a) Boundary-scan register cells shall be provided at bidirectional system pins such that:
   1) Whenever the pin is functioning as an input pin, all rules are met for cells provided at system input pins and inputs to the on-chip system logic (see 11.5).
   2) Whenever the pin is functioning as an output pin, all rules are met for cells provided at outputs of the on-chip system logic that drive data inputs of system output buffers (see 11.6).
   3) All rules are met for cells provided at outputs of the on-chip system logic that drive control inputs of buffers at system output pins (see 11.6).

   NOTE 1—In cases where the direction of signal flow is determined by an output *O* of the on-chip system logic, a boundary-scan register cell will exist in the signal path between *O* and the system pin. When the *EXTEST*, *CLAMP, CLAMP_HOLD, CLAMP_RELEASE, INIT_RUN*, *INTEST*, or *RUNBIST* instruction is selected, or the TMP controller is in the *Persistence-On* state, the direction of signal flow will be determined by the data held in the latched parallel output of the shift-register stage of the boundary-scan register cell.

b) Whenever two separate boundary-scan register cells are provided at a bidirectional system pin to meet the requirements of rule a1) and rule a2) of 11.7.1:
   1) The cell that meets the requirements of rule a1) of 11.7.1 shall, at all times, meet all rules for cells provided at system input pins and inputs to the on-chip system logic (see 11.5).
   2) The cell that meets the requirements of rule a2) of 11.7.1 shall, at all times, meet all rules for cells provided at outputs of the on-chip system logic that drive data inputs of system output buffers (see 11.6).

   NOTE 2—A structure of two boundary-scan register cells that would meet rule a) of 11.7.1 while failing to meet rule b) of 11.7.1 not only would require more logic than a structure, which meets both rule a) and rule b) of 11.7.1, but also would have less test usefulness.

11.7.2 Description

These requirements represent a merging of those for two-state or three-state output pins with those for system input pins.

Figure 11-37 and Figure 11-38 show gated-clock examples of the provision of boundary-scan register cells at three-state bidirectional pins.

Figure 11-37 allows the state of the pin to be fully controlled while the *INTEST* or *RUNBIST* instruction is selected. The Mode signal shown in Figure 11-37 should be controlled as indicated in Table 11-6. The Reset* or CHReset* signal may be fed to the parallel output register of the control cell in accordance with permission h) of 11.3.1. If the TMP controller is not implemented, then Reset* from the example TAP controller of Figure 6-5 may be used. If the TMP controller is implemented, the CHReset* from the example TMP controller of Figure 16-1 may be used. Reset of the update stage is not required.

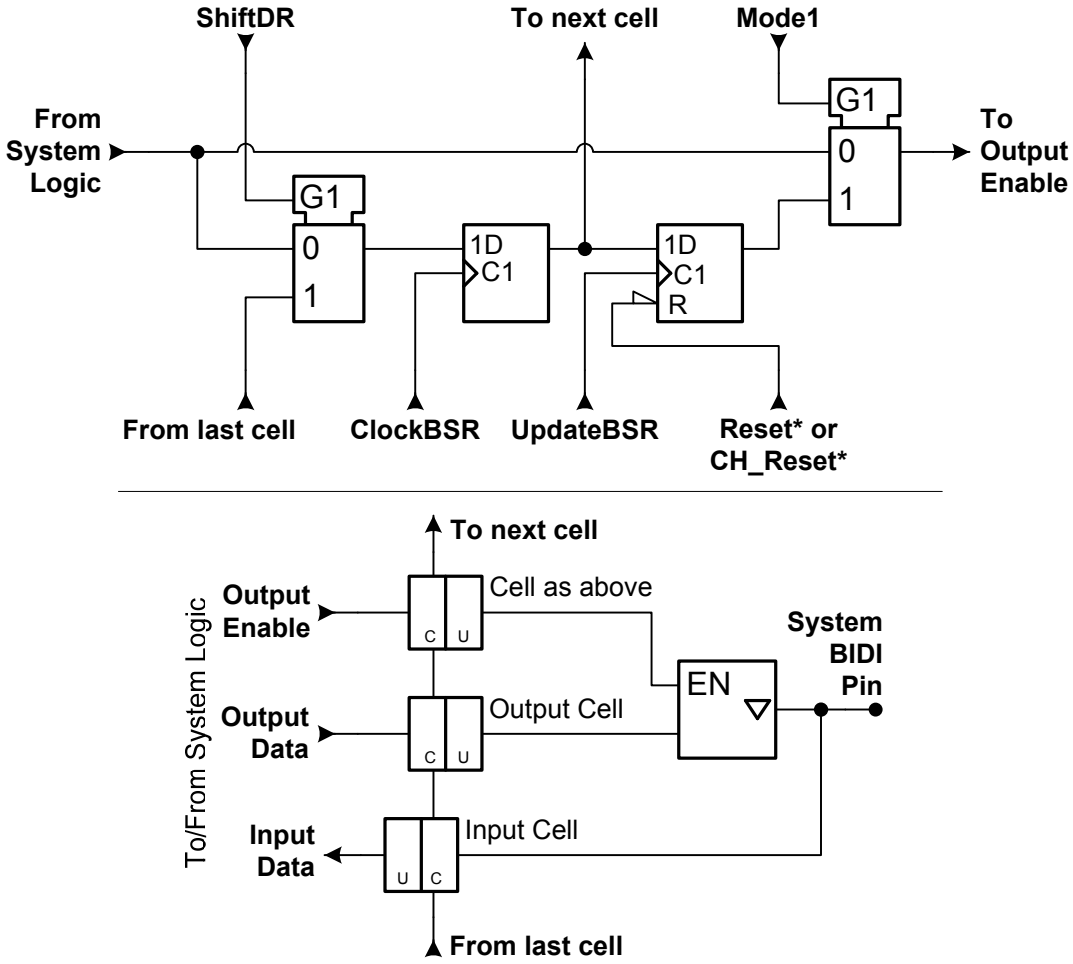


Figure 11-37—Boundary-scan register cells at a bidirectional pin—Example 1 [BC_1, control]

NOTE 1—See Table 11-6 for mode signal generation.

In Figure 11-38, a **BC_2** boundary-scan register cell is used to control and a single **BC_7** boundary-scan register cell is used to observe both output and input data. This cell meets the requirements of 11.6.1 when the control cell puts the pin in the output direction, and the requirements of 11.5.1 when the control cell puts the pin in the input direction. The various control signals used by the cell should be controlled as shown in Table 11-10.

As discussed in connection with Figure 11-36, the design of the circuitry around the shift-register stages in Figure 11-38 permits all circuitry in the cell to be tested if the *EXTEST* and *INTEST* instructions are executed with appropriate data.

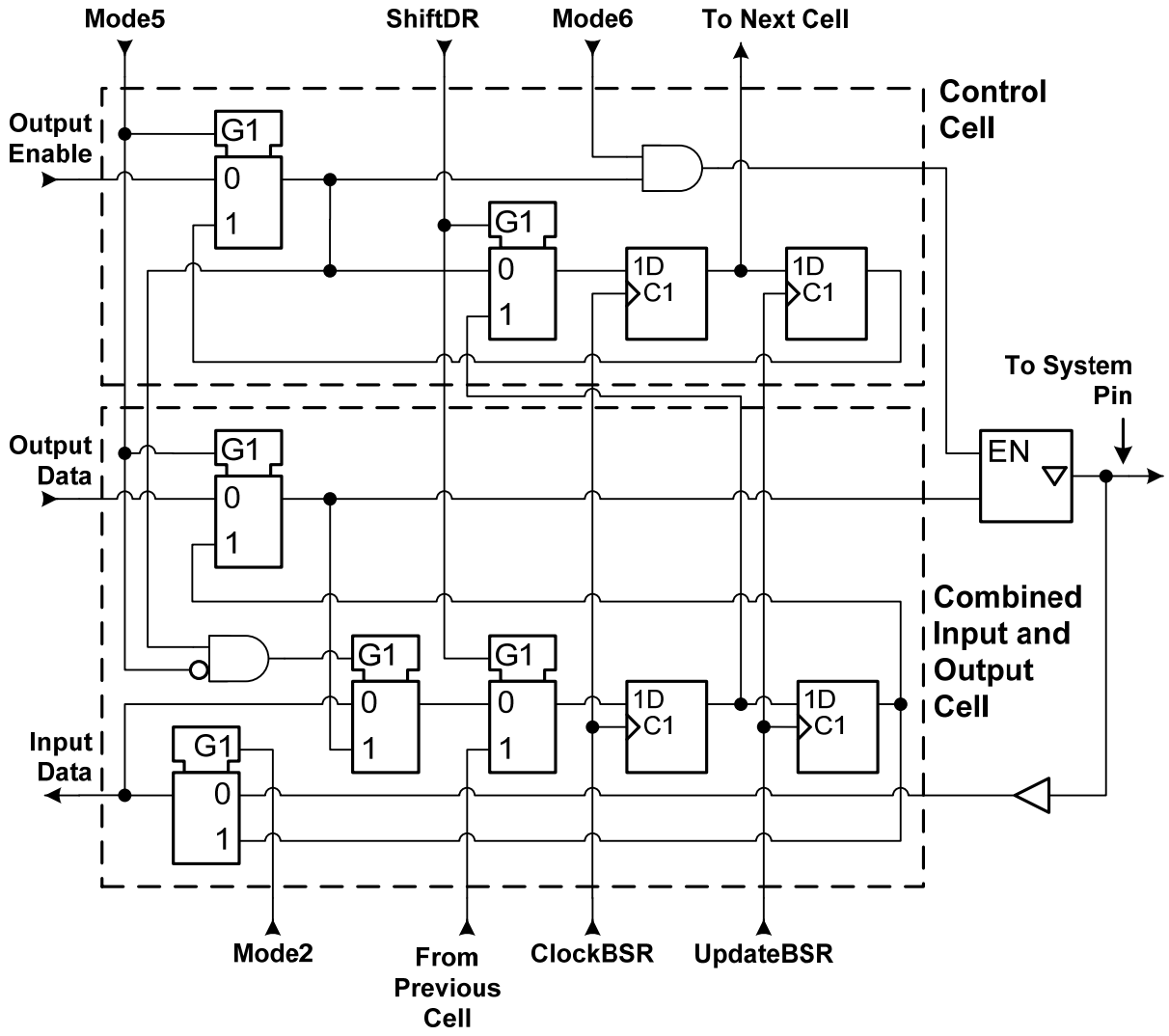


Figure 11-38—Boundary-scan register cells at a bidirectional pin—Example 2 [BC_2 control; BC_7 data]

NOTE 2—See Table 11-10 for mode signal generation.

Table 11-10—Mode signal generation for the example cells in Figure 11-38

Precedence	Instruction (Condition)	Mode5	Mode2	Mode6
1	(Cell in excluded segment)	0	0	1
2	*EXTEST*	1	0	1
	INTEST	0	1	0
3	(TMP controller *Persistence_on* state)	1	1	1
4	*PRELOAD*	0	0	1
	SAMPLE	0	0	1
	RUNBIST	0	X	0
	CLAMP	1	X	1
	CLAMP_HOLD	1	1	1
	CLAMP_RELEASE	1	1	1
	INIT_SETUP_CLAMP	1	1	1
	INIT_RUN	1	1	1
	HIGHZ	X	X	0
	Nonboundary instruction	0	0	1

Figure 11-39 uses a **BC_2** cell for control and a **BC_6** cell to observe data output and input at a bidirectional pin. The various control signals used by the cell should be controlled as shown in Table 11-11.

The **BC_6** cell is no longer supported. It is included for historical reference only. The **BC_6** cell is no longer described by the BSDL Standard Package, and this cell should not be used in any new design. The **BC_6** fails to capture the logic value at the pin in *EXTEST* when the control cell puts the bidirectional pin in output mode. This severely hampers the ability to perform a board-level test, which is the goal of this standard.

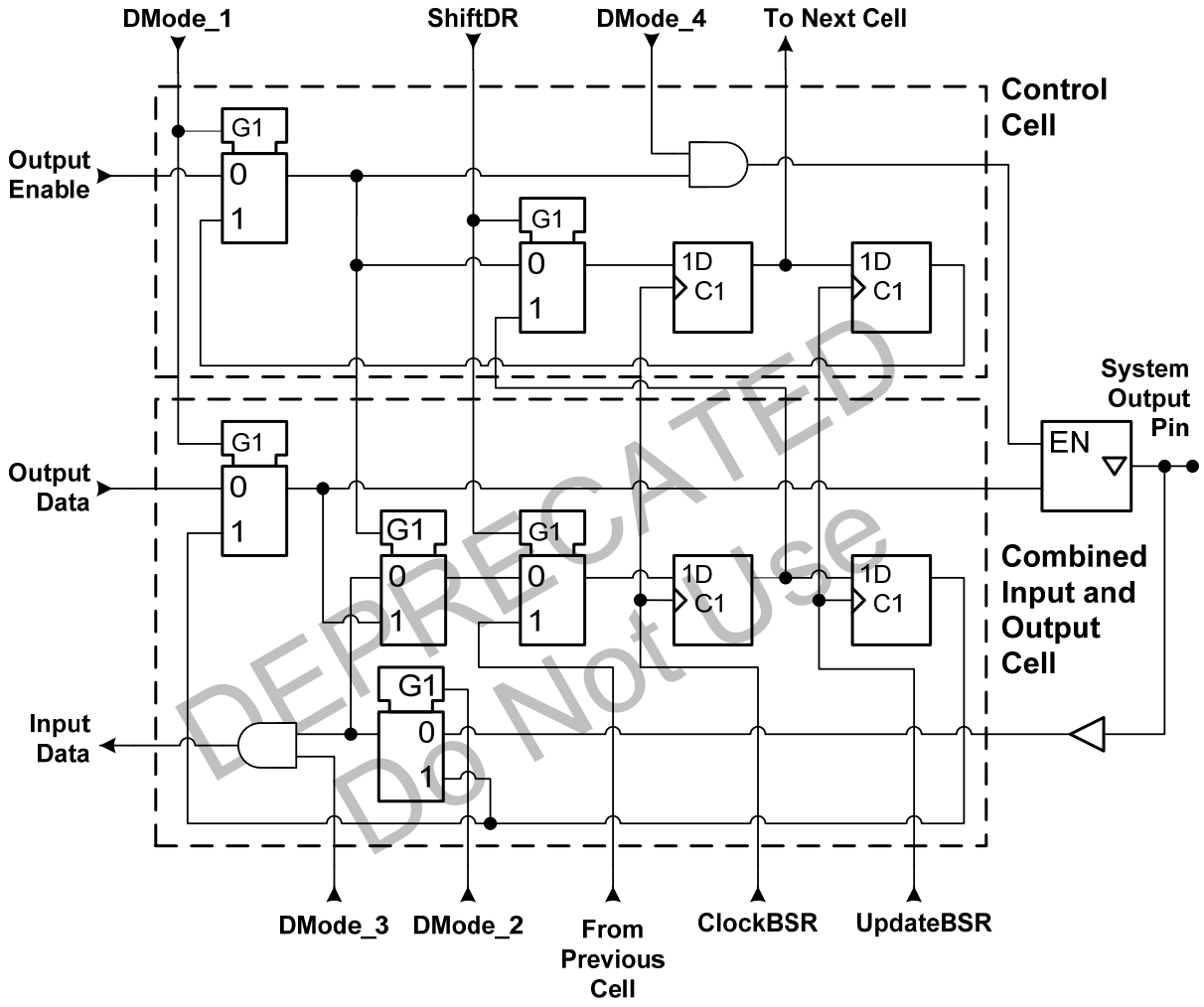


Figure 11-39—Deprecated boundary-scan register cells at a bidirectional pin [BC_2 control; BC_6 data]

NOTE 3—The ***BC_6*** data cell is no longer supported. (See Table 11-11 for mode signal generation.)

Table 11-11—Mode signal generation for the deprecated example cells in Figure 11-39

Instruction	DMode_1	DMode_2	DMode_3	DMode_4
EXTEST		0	0	1
PRELOAD	0	0	1	1
SAMPLE	0	0	1	1
INTEST	0	1	1	0
RUNBIST	X	X	X	0
CLAMP	1	X	0	1
HIGHZ	X	X	0	0

Figure 11-40 shows how boundary-scan register cells may be provided at an open-collector bidirectional pin. Note that, like the cell design of Figure 11-32, this gated-clock cell design does not support the *INTEST* instruction.

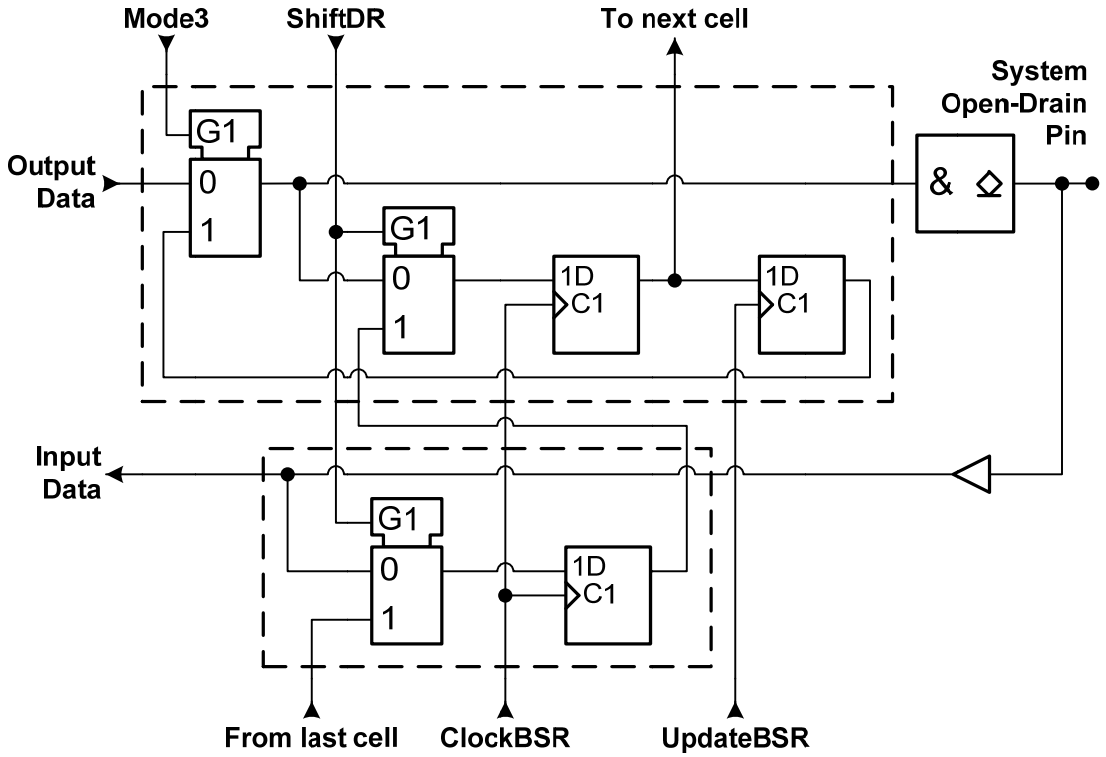


Figure 11-40—Boundary-scan register cells at an open-collector bidirectional pin [BC_4, input; BC_2, output]

NOTE 4—See Table 11-7 for mode signal generation.

Note that in cases where a bidirectional pin is provided using an open-drain or open-collector output driver, the pin direction is defined as output for the one state that is actively driven and as input otherwise. Thus, a separate control cell is not required (i.e., a single bidirectional cell can provide for both data and control), as shown in Figure 11-41.

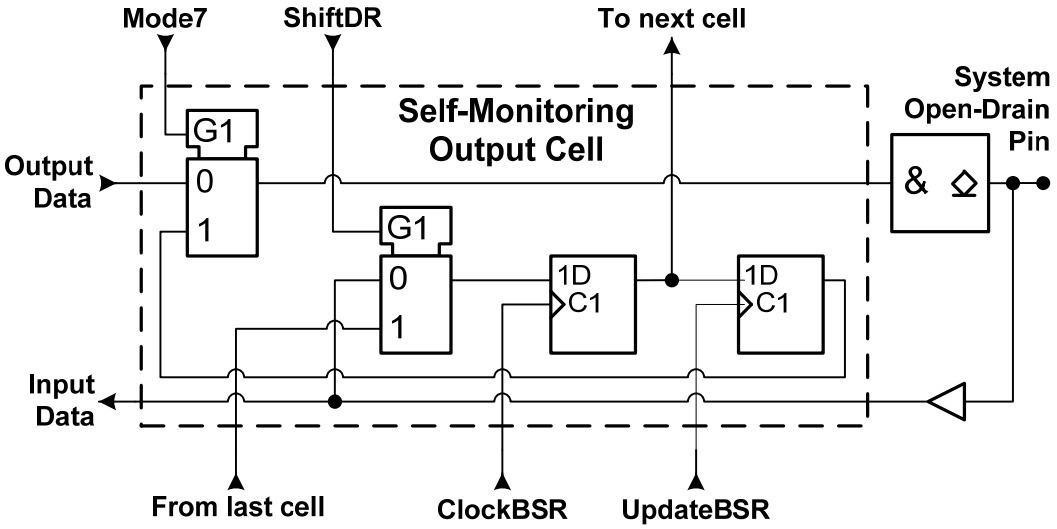


Figure 11-41—Boundary-scan register cell at an open-collector bidirectional pin [BC_8]

NOTE 5—See Table 11-12 for mode signal generation.

Table 11-12—Mode signal generation for the example cells in Figure 11-41 and Figure 11-42

Precedence	Instruction (Condition)	Mode7
1	(Cell in excluded segment)	0
2	*EXTEST*	1
3	(TMP controller *Persistence_on* state)	1
4	*PRELOAD*	0
	SAMPLE	0
	RUNBIST	X
	CLAMP	1
	CLAMP_HOLD	1
	CLAMP_RELEASE	1
	INIT_SETUP_CLAMP	1
	INIT_RUN	1
	Nonboundary instruction	0

Figure 11-42 illustrates an alternative, reduced complexity gated-clock cell for use at a three-state bidirectional pin. This cell is designed such that the signal present at the system pin is always captured. Because of this feature, this cell cannot be used where the *INTEST* instruction is to be provided. Where *INTEST* is to be supported, a cell capable of capturing the output of the system logic and of a design similar to that shown in Figure 11-38 is required.

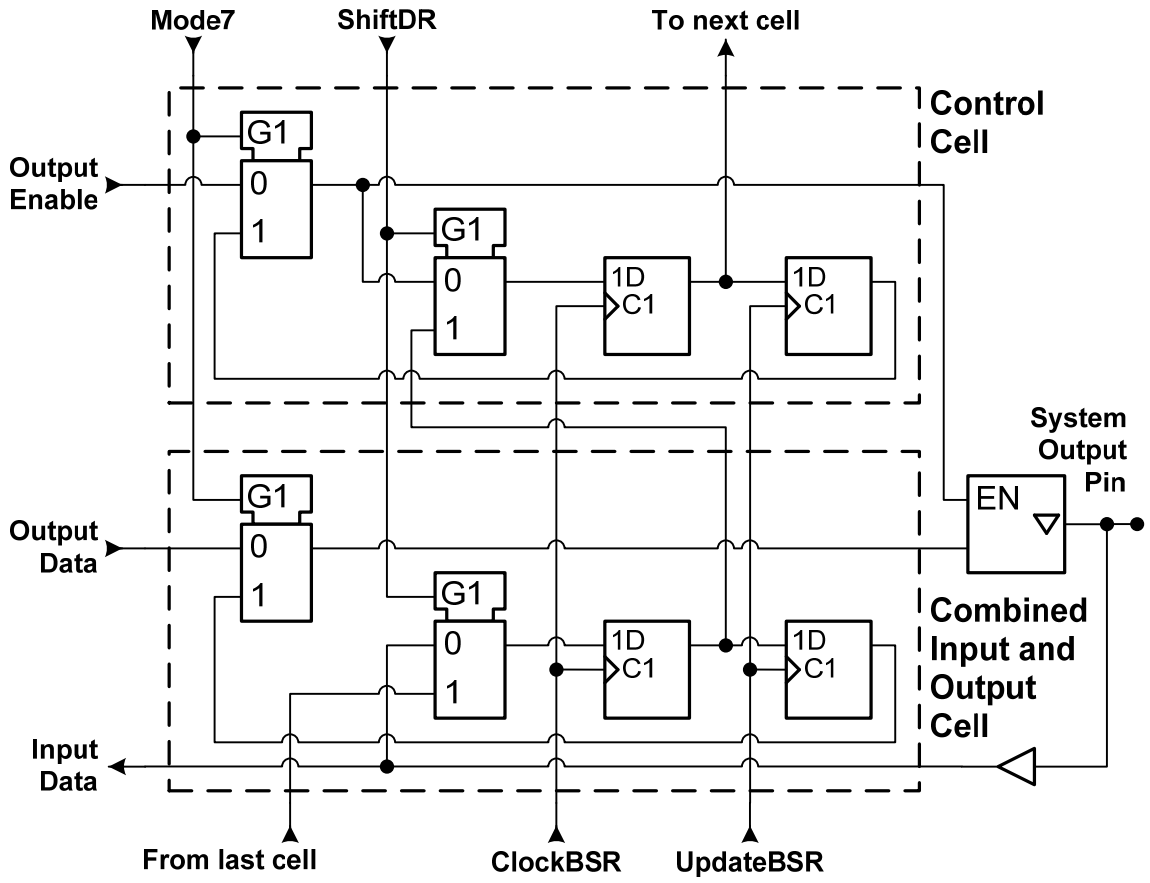


Figure 11-42—Boundary-scan register cells for use at a bidirectional pin where *INTEST* is not provided [BC_2, control; BC_8, data]

NOTE 6—See Table 11-12 for mode signal generation.

11.8 Redundant cells

Redundant boundary-scan register cells are cells that may be omitted from the component without jeopardizing compliance with this standard. They may exist in a component design for a number of reasons. For example:

- They may observe a signal (input or output) that is observed by another required boundary-scan register cell.
- They may observe the individual pins of a differential pair.
- They may observe compliance-enable, analog, power, or other pins where no cell is required. (When observing nondigital pins, they will normally capture a fault condition.)
- They may be parts of boundary-scan register cells designed for bidirectional system pins in cases where the pin has been programmed or otherwise customized to be permanently an input pin or an output pin. For example, a programmable component may be provided with three boundary-scan register cells at each system pin, sufficient to permit each system pin to be programmed as an input pin, two-state or three-state output pin, or bidirectional pin. After programming, certain of these cells may not be logically connected either to a given system pin or to a system logic input or output or both. Alternatively, a vendor of application-specific components may build a boundary-scan register into the basic component design (i.e., the design before the component is "committed") that provides for a fully bidirectional signal at each possible system pin. When the basic component is "committed," these cells will be constrained such that only the required functionality is connected.

11.8.1 Specifications

Rules

a) The contents of a redundant boundary-scan register cell shall not affect the behavior of any other part of the component.

b) Optional redundant boundary-scan register cells with observe-only capability observing the digital value of a component pin or a signal to or from the system logic shall be designed such that the data shifted out through TDO following loading of the shift-register stage in the *Capture-DR* TAP controller state conform to the rules in 11.5.

c) Where an optional special circuit is provided to detect a specific fault condition on a component pin, that circuit and a redundant boundary-scan register cell with observe-only capability observing the output of that circuit shall be designed such that the data shifted out through TDO after the loading of the shift-register stage in the *Capture-DR* TAP controller state are a defined constant in the absence of the fault condition, and the opposite value when the fault is detected.

NOTE 1—The optional fault detection circuit and cell may be placed on any component pin other than the TAP ports. Such fault detection circuits are expected when an optional cell monitors a nondigital pin. There is no restriction on the number of fault detection circuits or cells that may be associated with a single component pin.

Permissions

d) When the *SAMPLE* instruction is selected, redundant cells, which have observe-only capability, may capture a constant value.

NOTE 2—Redundant observe-only cells are the only cells that do not require capturing the logic value at the pin or system logic input/output during *SAMPLE*.

Recommendations

e) The number of redundant boundary-scan register cells included in a component that do not have observe-only capability should be minimized.

f) Redundant cells, which do not have observe-only capability, should be designed such that the data shifted out through TDO after loading of the shift-register stage in the *Capture-DR* controller state are either a constant or the data just previously shifted into the cell.

11.8.2 Description

Some programmable components (e.g., programmable gate arrays or application-specific ICs) offer input/output circuits that can be programmed as input, output, three-state, or bidirectional pins. To permit programming as a three-state or bidirectional pin, two or more boundary-scan register cells would have to be included in each configurable cell to allow access to the data and control signals. However, when the cell is programmed as an input or two-state output pin, only one cell will be required. In some implementations, the cells not associated with the programmed system function of a pin may be logically disconnected from the pin and from the system logic. Under such circumstances, the disconnected cells could no longer be used during testing and would become redundant. Rule h) of 11.2.1 requires that the unused cells remain in the boundary-scan register so that the register has a fixed length regardless of how the component is programmed.

NOTE—In many programmable devices, programmed lack of logical connection(s) may occur only with regard to a boundary-scan register cell and the on-chip system logic. The cells provided for a particular programmable pin may remain logically connectable to that pin during testing, and the bidirectional control cell would then remain functional. The rules of this clause do not prohibit this "excess" functionality at a pin. Indeed, interconnect test generation may actually be easier when all pins on a board-level net *appear* from *outside* the components to be provided with full bidirectional boundary-scan capability.

To minimize the number of redundant cells contained in the boundary-scan register of a component, the register should contain only cells that, in some programmed configuration of the component, can provide access to signals at

the boundary of the on-chip system logic or access to a system pin. For example, other than a segment-select or domain-control cell, a cell that receives its parallel data input from the on-chip system logic and sends its parallel data output into the on-chip system logic should not be included in the boundary-scan register (e.g., as shown in Figure 11-43).

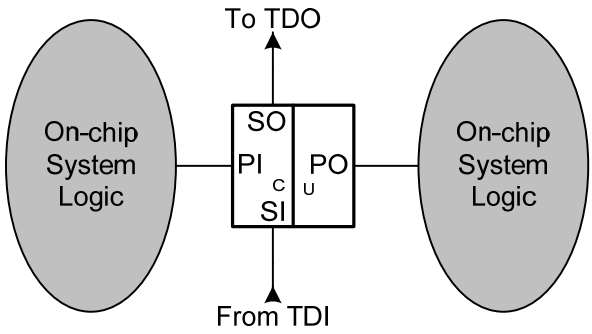


Figure 11-43—Cell that should not be included in the boundary-scan register

Optional redundant observe-only cells observing a system logic pin during *EXTEST* must capture either the same value the nonredundant cell would capture or, where the pin receiver or driver contains circuits designed to detect specific faults, then the fault detection signals. The fault detection signal must provide a known, constant, "good-machine" value (1/0) when no fault is detected, and the opposite value (0/1) when the fault is detected. Such cells provide additional information when detecting and diagnosing defects.

Redundant observe-only cells observing nonsystem logic digital pins (such as compliance-enable pins, see 4.8) during *EXTEST* must capture the same information allowed for redundant observe-only cells on system logic digital pins, even though there is no nonredundant cell on these pins.

Optional redundant observe-only cells observing nonsystem, nondigital pins (such as analog or power pins) require the component designer to provide a circuit between the pin and the cell, which is designed to detect specific faults and output a digital signal. This signal must have the characteristic of a fault detection signal described above with a known, constant value when no fault has been detected. For example, in Figure 11-10, there are special buffer/receiver circuits shown connected between the individual differential input pins and the two redundant observe-only cells. These special buffers/receivers might be window comparators testing that the voltage on the pin is within the valid range defined by the differential protocol. The redundant observe-only cells might then capture a logic 1 when the pins were in the valid range, and a logic 0 when the pins were outside the valid range.

11.9 Special cases

11.9.1 Specifications

Permissions

a) In a case in which a system logic input pin is used *solely* as a source of control or *solely* as a source of data for a system output pin, a single cell may be provided that meets the rules of 11.5.1 (for the input pin) and 11.6.1 (for the output pin).

11.9.2 Description

Where the signal received at a system logic input pin is used solely to provide data or control for a system output pin, it is possible to use a single boundary-scan register cell to meet both sets of requirements. A common example of a situation where this might arise is one in which a system input pin is used solely to provide an output control signal for three-state or bidirectional system pins. In such a case, either:

— Two separate boundary-scan register cells may be included, as shown in Figure 11-44.
— The functions of both cells may be combined into a single cell as shown in Figure 11-45.

In the latter case, the cell should be carefully designed so that it conforms to all the rules for the set of boundary-scan test instructions supported by the component.

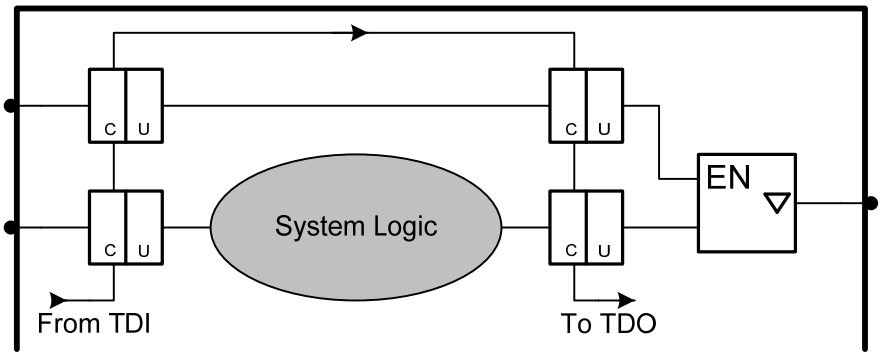


Figure 11-44—Input pins used only to control output pins—Case A

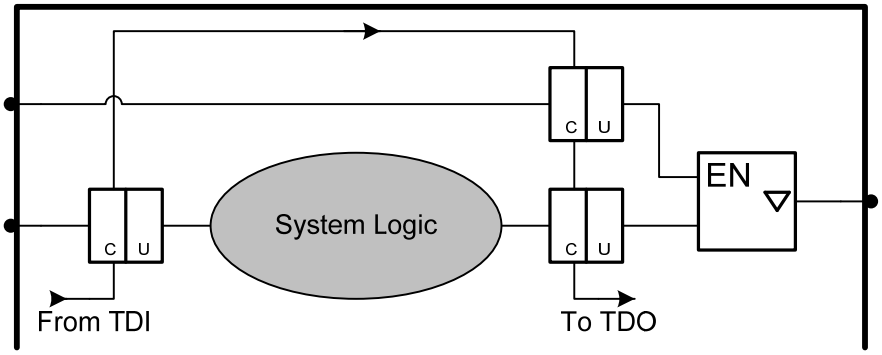


Figure 11-45—Input pins used only to control output pins—Case B

Note that the situation illustrated in Figure 11-46 violates the rules of this standard. In this case, the signal received from the system input pin is used both as an output control and as an input to the on-chip system logic.

In a case in which the signal from a system input pin is used only as a control for the three-state output buffer and the option has been taken to provide a single boundary-scan register cell as shown in Figure 11-19, the top cell in Figure 11-35 has to be modified if recommendation f) in 8.9.1 is to be met. Specifically, the cell has to reload its own state in the *Capture-DR* controller state when the *INTEST* instruction is selected so it does not capture a value dependent on off-chip circuitry. Figure 11-47 shows how this could be achieved. For this gated-clock design, the Mode signal should be controlled as shown in Table 11-6.

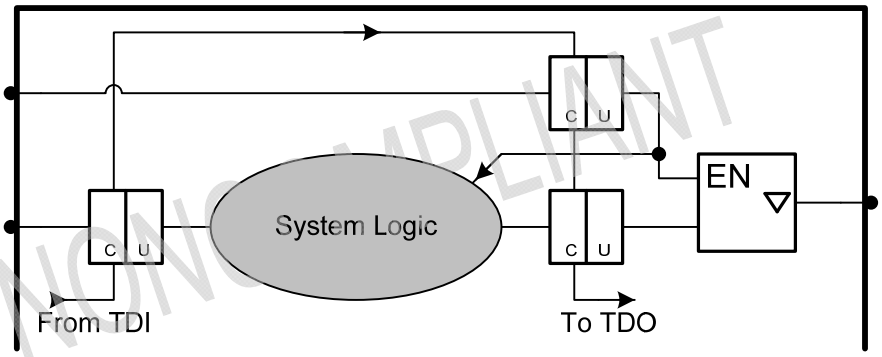


Figure 11-46—Noncompliant use of a single cell for output control and data

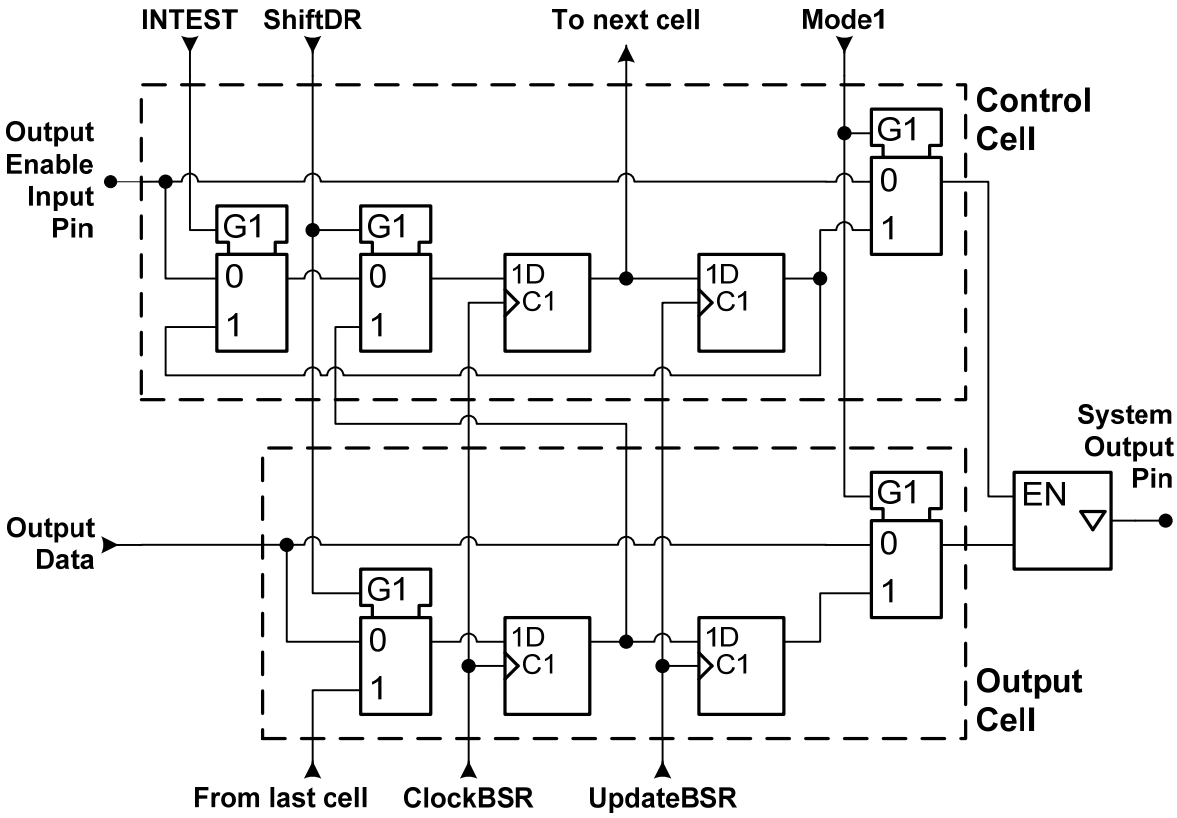


Figure 11-47—Boundary-scan register cells at a three-state pin where output control is from a system pin [BC_5, control; BC_1, data]

NOTE—See Table 11-6 for mode signal generation.

12. Device identification register

This clause defines the design and operation of the optional device identification register. If provided, this register allows the manufacturer, part number, and version of each component to be determined through the TAP, which in turn is used to identify the correct BSDL and correct test patterns for the board. The device identification register allows the test engineer to distinguish the manufacturer(s) of components on a board when multiple sourcing is used, and variations of a component that are acceptable for use on the board. Because the *IDCODE* instruction is loaded when the TAP controller is in the *Test-Logic-Reset* state, the test engineer can blindly interrogate a board design in order to determine the type of each component in each location. The need to do this becomes more apparent if one considers systems that are configurable by the addition of option boards or by programming certain components, etc. This information is also available for factory process monitoring and failure mode analysis of assembled boards.

NOTE—The design requirements contained in this clause apply only when the optional device identification register is included in a component.

12.1 Design and operation of the device identification register

12.1.1 Specifications

Rules

a) The device identification register shall be a shift-register based path that has a parallel input but no parallel output.
b) The circuitry used to implement shift-register stages in the device identification register shall not be used to perform any system function (i.e., it shall be a dedicated part of the test logic).
c) On the rising edge of TCK in the *Capture-DR* controller state, the device identification register shall be set such that subsequent shifting causes an identification code to be presented in serial form at TDO.
d) The component shall contain a vendor-defined identification code, containing four fields (see Figure 12-1), which is accessed when the *IDCODE* instruction is entered.
e) For user-programmable components, the ability shall be provided to permit the user to program a supplementary 32-bit identification code that will be loaded into the device identification register in response to the *USERCODE* instruction.
f) The operation of the device identification register shall have no effect on the operation of the on-chip system logic.

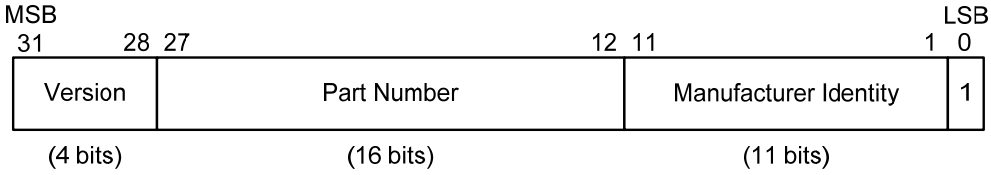


Figure 12-1—Structure of the device identification code

12.1.2 Description

Figure 12-2 shows a design for a gated-clock device identification register cell that satisfies these requirements. The first multiplexer, “USERCODE decode” and “USERCODE bit,” all in broken lines, are only provided for components supporting the *USERCODE* instruction. Otherwise, the “IDCODE bit” is connected to the 0 input of the second multiplexer.

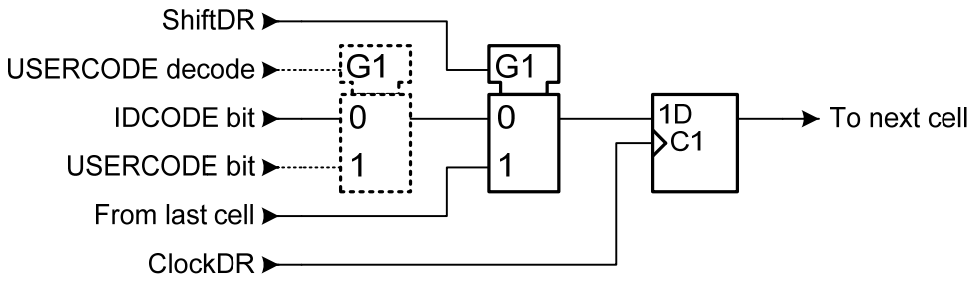


Figure 12-2—Device identification register gated-clock cell design

The board-test user needs to verify whether the component at the location on the board is the one intended to be there. Therefore, the identification code (together with the user code, if provided) must be unique for all components with the same footprint and the same placement of power and TAP pins. This becomes possible when the components are from the same company and perhaps in the same family. To help ensure uniqueness, the identification code has specific fields and coding requirements.

The identification code loaded into the device identification register in response to the *IDCODE* instruction allows the manufacturer, part number, and variant for the component to be read out in a serial binary form. In situations where blind interrogation of a product is necessary, this information allows the components on the board to be identified and paired with their documentation (BSDL) and thereby to the instruction set, boundary-scan register description, and other details for each component. The boundary-scan register description includes the provisioning of boundary cells at input and output pins, and the location of cells that control three-state or bidirectional pins. This information is invaluable in ensuring that contention between drivers at the board level is minimized, as discussed in 11.6. (It is assumed that the components in a product will be selected from a limited set.)

For programmable components, however, the configuration of pins as inputs, outputs, etc., may be determined by programming rather than by the basic design of the component. In such cases, therefore, a supplementary identification code is required to verify that the component at the location on the board has been properly programmed and matches the documentation used to generate the test patterns. This supplementary code is user programmable and accessed through the device identification register in response to the *USERCODE* instruction.

NOTE—The supplementary identification code is required only in cases where the component cannot be reprogrammed through the test logic defined by this standard. In cases where such reprogramming is possible, the ATE or master device controlling the operation of the component can program it to the correct state at the start of the test sequence.

Since the bypass register (which is selected in the absence of a device identification register by the instruction loaded in the *Test-Logic-Reset* controller state) loads a logic 0 at the start of a scan cycle, and a device identification register will load a constant logic 1 into its least significant bit, examination of the first bit of data shifted out of a component during a test data scan sequence immediately after exit from the *Test-Logic-Reset* controller state will show whether a device identification register is included in the design.

A requirement of the *IDCODE* and *USERCODE* instructions is that when they are used, the on-chip system logic shall continue its normal operation undisturbed. Rule b) of 12.1.1 is included so that this requirement can be met. Note, however, provided rule f) of 12.1.1 is met, the shift-register stages may be shared resources used by several of the registers defined by this standard and also by any design-specific test data register.

12.2 Manufacturer identity code

12.2.1 Specifications

Rules

a) The 11-bit manufacturer identity code shall be a compressed form of the code specified by EIA/JEP106 generated as follows:
   1) Manufacturer identity code bits 7-1. The seven least significant bits of this field are derived from the last byte of the EIA/JEP106 code by discarding the parity bit.
   2) Manufacturer identity code bits 11-8. The four most significant bits of this field provide a binary count of the number of bytes in the EIA/JEP106 code that contain the continuation character (hex 7F). Where the number of continuation characters exceeds 15, these four bits contain the modulo-16 count of the number of continuation characters.

b) The manufacturer code 00001111111 shall not be used in components that are otherwise compatible with this standard.

c) The manufacturer identity code shall reflect the company that owns and supports the component at the time it is first put into production by that company, whether that company is the sole, prime, or second source of the component. (See 18.2 for a description of prime and second sources.)

NOTE 1—When manufacturing is carried out by a party contracted by the component owner, the component owner's manufacturer identity code is used.

Recommendations

d) If the original component owner subsequently transfers component ownership to a new company with a different manufacturer identity code, the component may retain the former owner's manufacturer identity code until such a time, if ever, that the new owner modifies the design.

NOTE 2—Nothing in this clause would require a new owner to modify the design only to change the manufacturer identity code.

12.2.2 Description

The manufacturer identity code identifies the owner of the component rather than the company that actually manufacturers the component. If a company contracts component manufacturing to a third party, the manufacturer identity code is that of the owner of the component rather than of the second party manufacturer. For example, the manufacturer identity code for an application-specific integrated circuit (ASIC) would reflect the owner of the device rather than the manufacturer.

The manufacturer identity encoding utilizes a listing of manufacturer identification codes specified by EIA/JEP106 as administered by Electronic Industries Association/Joint Electron Device Council (EIA/JEDEC).

The EIA/JEP106 code is formed from a variable number of eight-bit bytes. Each byte contains seven data bits and an odd parity bit (the most significant bit). Bytes other than the last contain continuation characters (hex 7F), while the last contains 127 different codes that, together with a knowledge of the number of preceding continuation code bytes, allow the manufacturer's identity to be determined.

The compressed form of the EIA/JEP106 code used within the device identification register limits the number of bits needed in the device identification register to contain the manufacturer identity code and allows the length of the code to be standardized. The length of the compressed code is fixed at 11 bits (see 12.1), which allows for 2032 different manufacturer codes. (Note that 16 codes are unused since these correspond to the hex 7F code in the seven least significant bits—the EIA/JEP106 continuation character.)

One of the unused codes (00001111111) should be treated as noncompliant for components compatible with this standard. By shifting a dummy device identification code containing this manufacturer identity code from the bus master (ATE, board-level controller, etc.) into the board-level serial path set up by moving directly from the *Test-Logic-Reset* controller state into scanning of the test data registers, it is possible to detect the end of the identity code sequence.

When test data register scanning is entered in this way, the serial path at the board level includes:

— The device identification registers of components that provide them.
— The bypass registers of components that do not include a device identification register.

As discussed in 12.1, the fact that identification codes begin with a logic 1 whereas the bypass registers load a logic 0 allows the identification codes in the serial stream read out of the board to be detected. By feeding in the dummy identification code at the board's serial input and checking the serial output for the invalid manufacturer identity code 00001111111, it is possible to locate the end of the identification code sequence for a board containing an unknown number of components.

12.3 Part-number code

12.3.1 Specifications

Rules

a) The part-number code shall consist of 16 bits located in bits 27 through 12 of the register.
b) When two components from the same owner have different test or system functions and are offered in the same package with the TAP pins in the same location, they shall have different part-number codes.

Recommendations

c) When a hard selection mechanism (such as internal fuses) is provided to enable or disable certain test or system functions of a component after component manufacturing, the same mechanism should also select a separate part-number code for each selectable set of enabled or disabled functions.

Permissions

d) When variations of a component are produced or sorted to meet different performance specifications, such variations may share the same part-number code.

NOTE—This could result from speed binning, or even by a fabrication process that does not include all of the mask layers of the device.

12.3.2 Description

The part-number code identifies a component with unique test and system functions.

The "footprint" of a device is defined as the device package geometric pattern that facilitates the mechanical and electrical connection to the board. Two different devices with packages that have the same geometric pattern can be said to be mechanically footprint compatible. If, in addition, the TAP and power pins are located in the same position, then the two devices can be said to be "footprint-compatible" with respect to this standard. For footprint compatible devices, board manufacturing processes need to determine whether the expected device type is present on the board. The *IDCODE* instruction is intended to facilitate that test, but as a practical matter, it cannot serve as a guarantee that the correct device type is present.

NOTE—Identical die may be mounted in multiple incompatible footprint packages and sold as different part-numbered devices. Such devices would exhibit the same device identification register code but would not operate compatibly if loaded at the wrong location on a board. The common BSDL for such devices accounts for this by documenting the packaging variants (see B.8.7).

A device variant that differs by the system frequency, or one of the component voltage specifications, will likely result in the component owner assigning a new device part number for data specification purposes. It is also likely that such changes would result in a "footprint compatible" device. Such changes should be reflected in a different device identification part-number code, if possible, so that software can detect which device is present on the board. In many cases, changing the device identification code for such a change in specifications is not reasonably possible.

The part-number code is used to verify the type of the component inserted in a particular location on an assembled product. The use of a 16-bit value for this code gives an acceptably low chance that an incorrect component inserted in the location will return a correct part-number code.

Part-number codes could, for example, be generated from the textual part-number code using a data compaction scheme. The device vendor is constrained only by the above rules in generating part-number codes, and the primary goal is uniqueness.

12.4 Version code

12.4.1 Specifications

Rules

a) The version code shall consist of 4 bits located in bits 31-28 of the register.
b) The version code shall be changed if there are changes to the test logic defined in this standard and the part-number code is not changed.

Recommendations

c) The version code should be changed for all significant system logic changes.

NOTE—The word "significant" constitutes an understanding between the device producer and the consumer as to what system logic changes are relevant to the performance of the device (and the board) as seen by the board consumer. The board manufacturer is obligated to provide an expected set of board performance specifications to the end user. If a new version of the device could compromise this obligation, then the device supplier should signal this change with a version change. This allows the board manufacturer to verify that the correct version is being used.

d) Version code values should start with all 0 and be incremented in binary or gray-code order.

Permissions

e) The version code initially released to customers may be other than all 0 if there were versions not publicly released.

12.4.2 Description

The version code is used to distinguish device variants that do not result in a part-number code change, providing an additional means to determine whether the expected device is the actual device located on a board.

Initially, a new device or a variant of the device with a new part-number code will typically have an all 0 version code. The version code is then incremented in a binary (or other counting scheme such as gray-code) fashion for each new version code.

13. Electronic chip identification (ECID) register

An electronic chip identification (ECID) is a value unique to the individual component, within all components of the same type. It therefore complements the Device ID and User codes allowing identification of individual components. This may be used to retain historical data about the manufacturing, test, usage, etc. for the life of the component, which in turn may allow better tracking and improvement of the overall process.

13.1 Design and operation of the ECID register

13.1.1 Specifications

Rules

a) The ECID register shall be a shift-register based path that has a parallel input.
b) On the rising edge of TCK in the *Capture-DR* controller state, the ECID register or field of the ECID register representing the unique code shall be set such that subsequent shifting causes to be presented in serial form at TDO:
    1) An electronic chip identification code.
    2) A value of all 1 to indicate that the electronic chip identification code is not available.
c) The operation of the ECID register shall have no effect on the operation of the on-chip system logic.

Permissions

d) The ECID register may have additional design-specific fields defined.

NOTE—Design-specific register fields could be used to provide a "Ready" bit, fixed bits for TDR identification, or other design-specific purposes. If specific fields are not defined for this register, then the entire register is assumed to return the unique code, with codes all 1 reserved to indicate "not ready."

13.1.2 Description

The ECID register is a test data register without any special requirements other than that the capture capability is required. There is no required use for the parallel output from this register, but nothing to prevent its use either. Figure 12-2 shows a design for a gated-clock test data register cell, and Figure 9-8 shows a design for an ungated-clock test data register cell, both of which satisfy the requirements.

The electronic chip identification code loaded into the ECID register in response to the *ECIDCODE* instruction allows a unique identifier for each copy of the component to be read in a serial binary form. This information allows the manufacturing, test, and use history of individual components to be tracked for future reference.

A requirement of the *ECIDCODE* instruction is that when it is used, the on-chip system logic shall continue its normal operation undisturbed. Note, however, that the ECID shift-register stages may be shared resources used by several of the registers defined by this standard and also by design-specific test data registers.

14. Initialization data register

This clause defines the design and operation of the optional initialization data register, which is selected for shifting by the *INIT_SETUP* and *INIT_SETUP_CLAMP* instructions (see 8.18). The initialization data register provides a means to provide component initialization data.

14.1 Design and operation of the initialization data register

When a simple power-up (explicit internal POR or TRST* signal) is inadequate for initializing a complex component for board test, then the initialization data register can be used to provide the information required to complete the initialization prior to the start of board or other testing. This is intended for programmable I/O in particular, but it can be used for other characteristics of the component as well. Optionally, if some of the critical initialization parameters are supplied through input pins, then this register can monitor those pins so that their value is verified prior to starting interconnection tests.

The initialization required will often vary from board to board, or from one use on a board to another use on the same board. It cannot be defined directly in the BSDL since the BSDL documents the component design, not its use. BSDL now supports the naming of fields within a register and provides named constants (mnemonics) for use with those fields. A new language (PDL, see Annex C) allows specification of the values to be loaded into, or expected from, specific uses of a component in a specific environment. These new capabilities should be used to initialize complex components for test. They are not intended to initialize the component for system operation.

To prevent the effects of initialization from changing in unanticipated ways, the data shifted into this register are expected to persist until new data are shifted in.

14.1.1 Specifications

Rules

a) The initialization data register shall consist of one or more stages conforming to the rules of Clause 9.
b) The values shifted into the initialization data register in the *Shift-DR* state and subsequently driven out the parallel outputs of the initialization data register shall not be modified by system operation or the *Test-Logic-Reset* TAP controller state if the TMP controller is in the *Persistence-On* state.
c) Where the initialization data register is assembled from segments that may be included or excluded, a segment-select cell conforming to the requirements of 9.4.1 shall be provided for each excludable segment.
d) Where an excludable initialization data register segment must be conditioned in order to be included, a domain-control cell conforming to the requirements of 9.4.1 shall be provided for each such conditioning requirement.

Recommendations

e) The fields of the initialization data register and any associated values for those fields should be defined in BSDL to support use of component initialization for test.
f) Where critical initialization parameters are provided from a source external to the component through some set of pins, the value supplied at the pins should be captured without inversion into the initialization data register in the *Capture-DR* TAP controller state so that the values may be verified after shifting the register contents out and prior to the execution of *EXTEST* or other test mode instructions.

Permissions

g) If the optional TMP controller is either not provided, or is in the *Persistence-Off* state, the initialization data register contents may be altered either by system logic while the TAP controller is in the *Test-Logic-Reset* state or by the *Test-Logic-Reset* state itself.

h) When the initialization data register is selected, the values loaded into the initialization data shift register stage during the *Capture-DR* state may be selected by the component designer.

i) Segment-select and domain-control cells controlling excludable initialization register segments may be duplicated in other public Test Data Registers.

NOTE—Where more than one domain-control or segment-select cells control a single excludable segment, they would effectively be ORed together so that each one of them has the same effect as the others, and all must be set to zero to exclude the segment or disable the domain.

14.1.2 Description

The initialization data register is optional, but if provided, then the *INIT_SETUP* and *INIT_SETUP_CLAMP* instructions must also be provided (see 8.18).

The initialization data register may be very long, depending on the complexity of the programmable input and output circuits. To help ensure that the initialization is not altered unexpectedly, it is preferred that the initialization data register be a dedicated register. However, if the component designer determines that this does not need to be a dedicated test register; the register may be shared with system logic (when the TAP is in the *Test-Logic-Reset* state) so long as it remains dedicated to maintaining the initialization of the component, and in particular the component programmable I/O, as long as the test logic is active. This is defined in the combination of rule b) and permission g).

In addition, when the TMP controller is present and in the *Persistence-On* state, the values held in the register should not be changed, including by the *Test-Logic-Reset* TAP controller state. This will allow the supplied data to persist across other instructions and the *Test-Logic-Reset* state, so the *INIT_SETUP* or *INIT_SETUP_CLAMP* instructions (see 8.18) do not need to be re-run unless a change in the parameters is required for a subsequent test.

This standard now defines, in Annex B, BSDL mechanisms for documenting and naming the fields of test data registers, and of providing named constants for those fields. Such documentation of the initialization data register is strongly recommended and provides the board designer or test engineer the information he or she will require to define the setup data for each use of a component on each board. This includes the ability to define named constants expected to be captured in the *Capture-DR* TAP controller state as well as named constants to be written to the register.

In addition, this standard now defines, in Annex C, a new file type called Procedural Description Language (PDL) that is used to document how the various fields of the initialization data register are to be written and read for each use of the component on each board where it is used. A separate file for each instance is needed because the data required for specific uses will vary by usage, and so they cannot be specified once by the component designer.

In most applications, this register is intended to be a write-only register, and all parameters needed for initialization to be provided only through this register. However, there are designs where some initialization parameters are supplied through the input pins. These need to be monitored to ensure that the initialization received the correct values.

When the parameters supplied through such pins are not critical either to the operation of the test or to preventing damage to the components on the board, the pins may be monitored in *EXTEST* using input or redundant observe-only cells. This has the advantage of supplying multiple observations of the pin to help detect shorts to other signals, for instance.

However, when the parameters supplied through the pins are critical to successful initialization, or must be set correctly before the start of *EXTEST* to help prevent component damage, the signals from these pins should be captured directly (without interference by a boundary-scan register cell) in the initialization data register so they can be verified to be at the expected value for the board (specified in PDL) prior to the start of *EXTEST*. This is intended to detect when a board defect could cause incorrect initialization and possible component damage or simply an invalid test.

The tester can compare the value captured in the initialization data register to an expected value for the specific board, and then stop the process if the value returned does not match. Note that the initialization data register cells can only observe; they cannot control the input signals from these pins. These pins may also be monitored in the boundary-scan register as described above to detect other problems (such as shorts to other signals) during test.

There may be times when these input pins, providing critical parameters to initialization, are captured in the initialization data register. The test software (or test engineer) will determine the value expected from the PDL. If the tester detects an incorrect value, then the test could be aborted with an appropriate error message. If the tester can control the pins (by some external interface), the expected value will indicate how to drive them as well.

In addition, for other initialization data register bits that do not capture input data, they can be designed to capture a fixed bit pattern that can also be compared to an expected value to verify that the correct register is being accessed by this critical instruction.

15. Initialization status register

This clause defines the design and operation of the optional initialization status register. If provided, this register must be used with the *INIT_RUN* instruction (see 8.19). The initialization status register allows the status of an initialization procedure to be determined through the TAP.

15.1 Design and operation of the initialization status register

15.1.1 Specifications

Rules

a) The initialization status register shall be two or more shift-register stages with parallel input but no parallel output.
b) The circuitry used to implement shift-register stages in the initialization status register shall not be used to perform any system function (i.e., it shall be a dedicated part of the test logic).
c) When the initialization status register is selected, the current status of the initialization process shall be loaded into the register on the rising edge of TCK in the *Capture-DR* TAP controller state.
d) Bit 0 of the initialization status register shall capture, if such information is provided, a logical 1 during execution of the initialization process (“Busy”) and a logical 0 (“Done”) otherwise; and if the initialization process is designed without a completion indication, this bit shall always capture a logical 0 and the time required for initialization shall be documented in PDL.
e) Bit 1 of the initialization status register shall capture, if such information is provided, a logical 1 after successful completion of the initialization process (“Pass”) and a logical 0 otherwise (“Fail”); if the initialization process is designed without a pass/fail indication, this bit shall always capture a logical 1.

Recommendations

f) Optional higher order bits of the initialization status register should capture additional information about the initialization process to permit reasonable diagnosis of failure to initialize.

15.1.2 Description

The initialization status register is optional, but if present, it implies the existence of the *INIT_RUN* instruction (see 8.19). If provided, it is treated as a “read-only” register; any data written to it is ignored.

Other than the two low-order bits, the component designer is free to define what status will be captured, and he or she is encouraged to define and document as much as possible about possible causes of failure to initialize. Such failures can be extremely difficult to diagnose and can result in significant added rework expense when they are not diagnosable. This standard requires at least two status bits, one indicating that the initialization process is done (1) or still in-progress (0), and a second bit indicating that the process was successful. This second bit would typically be treated as undefined until the “done” bit is set. If the “in-progress” bit is still set at the end of the specified duration, or if the “done” bit is set and the “success” bit is reset when the initialization status register is read, the initialization process is assumed to have failed and the component is not ready for test.

Such encoding permits the initialization status register to be repeatedly read during the initialization process to determine when the process has completed. Such polling could abbreviate the time spent in board test initializing the board, especially if the component initialization process is highly variable in the amount of time it takes to execute. If an expected successful completion encoding is not documented, then the component cannot be polled for completion.

It may not always be possible to determine whether the initialization process has actually completed. In those cases, the initialization process is assumed to have successfully completed upon expiration of the specified duration.

16. TMP status register

16.1 Design and operation of the TMP status register

This clause defines the design and operation of the TMP status register, which is part of the optional test mode persistence controller. This register must be selected for scan by the *CLAMP_HOLD, CLAMP_RELEASE,* and *TMP_STATUS* instructions (see 8.20). The TMP status register provides both a means of observing the TMP controller state and a means to enable a reset of the test mode persistence controller (see 6.2 and 8.20).

16.1.1 Specifications

Rules

a) If and only if the TMP controller (described in 6.2) is provided per permission d) of 5.1.1, then the TMP status register shall be provided.
b) The TMP status register shall consist of two shift-register stages, a TMP-status bit closest to TDI and a bypass-escape bit closest to TDO.
c) The circuitry used to implement the shift-register stage in the TMP status register shall not be used to perform any system function (i.e., it shall be a dedicated part of the test logic).
d) The operation of the TMP status register shall have no effect on the operation of the on-chip system logic.
e) A logic value of 1 in the bypass-escape bit of the TMP status register shall enable the reset of the test mode persistence controller to the *Persistence-Off* state in the *Update-IR* TAP controller state when making *BYPASS* the active instruction, and a logic value of 0 shall disable such reset.
f) The bypass-escape bit of the TMP status register shall be set to a logic 1 by the same means used to initialize the test mode persistence controller (see 6.2.3).
g) The TMP-status bit of the TMP status register shall capture a 1 when the TMP controller is in the *Persistence-On* state and a 0 when it is in the *Persistence-Off* state, and the bypass-escape bit shall capture its update register, if any, or not change state in the *Capture-DR* TAP controller state.

NOTE—This implies that the bypass-escape bit of the TMP status register will return its current state when scanned out.

16.1.2 Description

The TMP status register, as well as the *CLAMP_HOLD, CLAMP_RELEASE,* and *TMP_STATUS* instructions (see 8.20), are required if the test mode persistence controller is provided. The TMP status register is a two bit register, the first bit (closest to TDI) captures the current state of the TMP controller (TMP-status bit) and the second bit (closest to TDO) sets and returns a value, which enables or disables an escape from the TMP controller *Persistence-On* state (bypass-escape bit).

When the bypass-escape bit of this register has been set to 1, loading the *BYPASS* instruction (through *Shift-IR* and *Update-IR*) will cause the TMP controller to change state to *Persistence-Off*, as shown in Figure 6-9 and Figure 6-10. This allows the system to recover in the event that the configuration of the scan chain is unknown or corrupted since a lengthy *Shift-IR* of all 1s into the scan chain followed by *Update-IR* will then clear any *Persistence-On* state that had been previously set.

The TMP-status bit is read-only (does not have a primary output, so does not control any test or mission mode logic) and captures the current state of the TMP controller.

Figure 16-1 shows a possible implementation of the TMP status register that meets the requirements of this clause. The connection of the “TMP_state” input signal and the “Bypass_Escape” output signal are to the same signals as in Figure 6-10. Figure 6-8 shows the generation of the TAP_POR* signal from the TRST* or on-component power-up generation circuit.

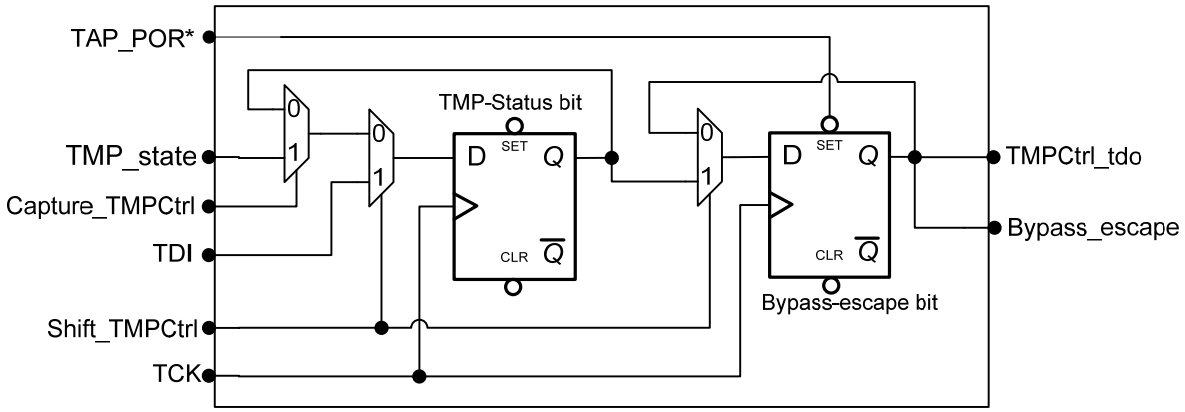


Figure 16-1—Example TMP status register (nongated clocks)

17. Reset selection register

This register and its associated *IC_RESET* instruction (see 8.2.1) allow control of system reset functions through the TAP, including blocking undesired resets to the system logic during testing.

17.1 Design and operation of the reset selection register

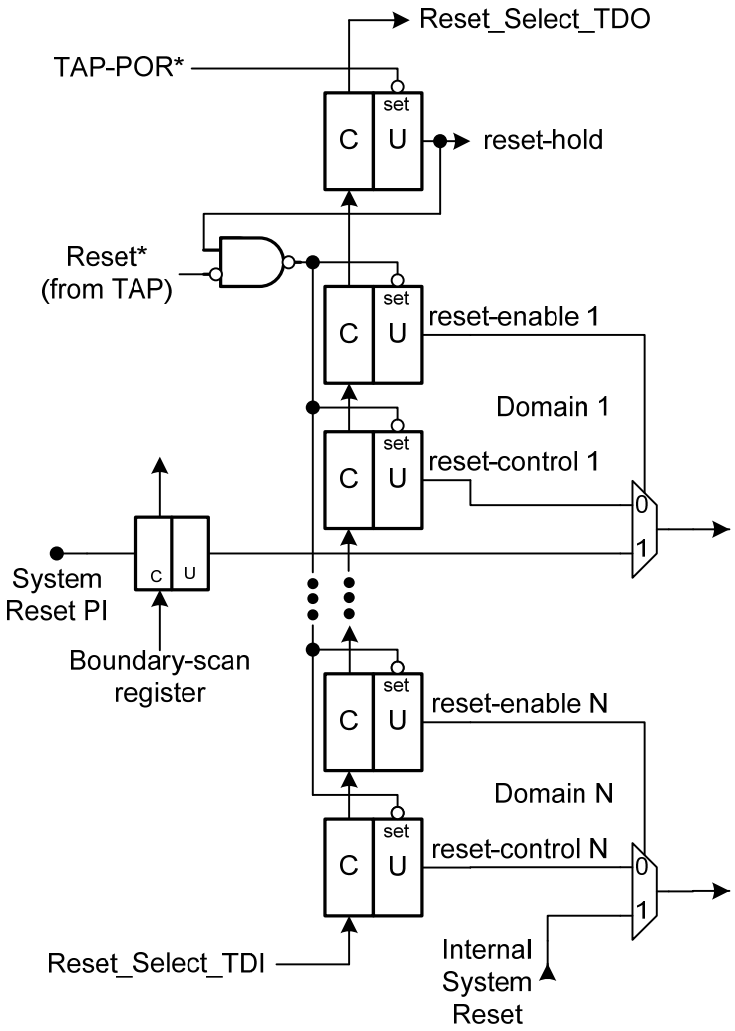


Figure 17-1—Reset selection register overview

The reset selection register may be described, as shown in Figure 17-1, in terms of the capture/shift register (the capture capability is not shown here), the update register, and reset select logic. A more detailed example is shown in Figure 17-2. The reset selection register is also divided into fields and bits within the fields as described in the rules. The labeling in Figure 17-1 illustrates the terminology of the rules. Figure 6-8 shows the generation of the TAP_POR* signal and Figure 6-5 the generation of the Reset* signal. The source of the system reset can be either a primary input or an internal source, and one of each is shown.

17.1.1 Specifications

Rules

a) The reset selection register shall consist of at least a single cell to control its own reset plus one or more pairs (two-bits) of cells and associated reset select logic, one such pair for each reset signal to the system logic to be controlled,

b) Each cell shall consist of a shift-register stage and an update register stage.

c) The least-significant bit (closest to TDO) of the reset selection register shall be a reset-hold bit.

d) The logic value of the reset-hold update register stage shall be initialized at power-up to a logic 1 by the same means used to initialize the TAP controller. (See 6.1.3 and Figure 6-8.)

e) When the reset-hold update register stage is set to a logic 1, the reset selection update register shall be initialized to the values enabling the functional reset source to control all of the system reset signals during the *Test-Logic-Reset* TAP controller state.

NOTE 1—The "functional reset source" is normally a reset pin, but it could be an internally generated reset, as might happen in an SOC, going from one part of the logic to another. The "system reset signal" is the input to the system logic.

f) When the reset-hold update register stage is set to a logic 0, the logic values of the reset selection update register shall not change during the *Test-Logic-Reset* TAP controller state.

g) The most significant bit (closest to TDI) of the two-bit pairs shall be a reset-control bit.

h) The least significant bit (closest to TDO) of each two-bit pair shall be a reset-enable bit, selecting control of the system reset signal between the functional reset source and the reset-control bit.

i) A logic 1 in a reset-enable update register stage shall select reset operation from the functional reset source for that system reset signal, and conversely, a logic 0 in a reset-enable update register stage shall select reset operation from the associated reset-control update register stage of the reset selection register.

j) A logic 0 in a reset-control update register stage shall assert the system reset signal when enabled by the associated reset-enable update stage.

NOTE 2—These rules do not constrain the internal logic value scanned into the reset selection scan register during *Shift-DR* nor loaded into the update register during *Update-DR* states of the TAP controller. They do require that when TDI is held at a logic 1, possibly a result of a problem in the scan chain causing the TDI pull-up to take control, the test logic does not interfere with system operation (the reset-hold bit and all reset-enable and reset-control bits are de-asserted by being set to 1).

k) To observe as well as control the reset signals to the system logic, the reset signal(s) to the system logic shall be captured in the reset selection register reset-control cell(s).

l) As the relationships between pairs of reset-control and reset-enable cells and individual functional reset sources and system reset signals to the system logic are not defined by these rules, the reset selection register shall be documented in BSDL using the **REGISTER_FIELDS** or **REGISTER_ASSEMBLY** attributes and, if the functional reset source is an input signal pin, the **REGISTER_ ASSOCIATION** attribute.

m) The reset selection register shall not control TRST* or any other test logic reset signal, including any power-up reset to the test logic.

n) The reset selection register shall be dedicated test logic.

Permissions

o) The component designer may choose a logic value to be captured by the reset selection register in the *Capture-DR* TAP controller state, and this choice may be documented in a BSDL register access description for the reset selection register.

17.1.2 Description

The reset selection register is used to control and possibly observe various system reset functions through the TAP. The reset selection register may also be used to block the effects of either reset pins or other functional reset sources that may not be under test control during testing. This can prevent inadvertently resetting system logic during test.

The rules define a single reset-enable and reset-control bit pair associated with each system reset signal to the system logic. Where a single reset pin or other functional reset source fans out to different parts of the system logic, there may be multiple reset-enable and reset-control pairs associated with a single reset pin or other functional reset source, allowing greater granularity when controlling the system resets from the TAP than is practical through the pins.

To match the common practice of reset signals being negative-active (0 is the "on" state, and 1 is the "off" state), this entire register is negative-active. In other words, after a reset, the state of all update register bits will be 1. This is the opposite of all other TDRs defined in this standard. We continue to use the term "reset" to indicate that the update registers are set to their "off" state, in this case, 1.

A reset-hold bit is provided to control whether the logic values of the reset selection register are to be reset when entering the *Test-Logic-Reset* TAP controller state. Upon power-up, the reset-hold bit is forced to the logic value allowing the *Test-Logic-Reset* state to reset the rest of the reset selection update register, using the same means as the TAP controller. Otherwise, the reset-hold bit is enabled by explicitly shifting in a logic 0 (at TDI) value to the least significant bit of the reset selection register to prevent the *Test-Logic-Reset* state from resetting the reset-enable and control bits of the register, and is disabled by shifting in a logic 1 (at TDI) value to allow the *Test-Logic-Reset* state to reset the enable and control bits of the register.

A logic 1 value (at TDI) shifted into all bits of the reset selection register will, after those values are transferred to the update register stages, disable all register selectable options and allow the normal reset pin (or other internal functional reset source) to control the system reset signals. As many reset functions may be controlled and possibly observed as desired.

The *IC_RESET* instruction initiates the selected reset functions when a logic 0 value (at TDI) is shifted into a reset-enable bit of a control pair and the TAP controller passes through the *Update-DR* state. The reset-enable bit selects control either from the functional reset source or from the associated reset-control bit of the pair. If a reset-control bit is selected, it may then assert and de-assert on-chip system reset. Multiple loadings of the register can set or clear reset signals and can cause reset functions to be performed in parallel or in sequence. At a minimum, two loads of the register will typically be used to first initiate the reset and then to remove the reset signal. The reset process in the system logic itself may take more or less time than the amount of time the reset signal is enabled.

In addition to controlling the system reset signals to the system logic (RS_System_Reset* in Figure 17-2), it is valuable to be able to observe them, especially when the functional reset source is internal. The recommended practice of capturing the reset signal to the system logic allows the test software (including a PDL program) to verify the state of the system reset pins or other internal functional reset sources when control is not enabled, and to verify that the reset selection register is being properly set.

The reset selection register and *IC_RESET* are not intended to be automatically used by test software during testing; there are just too many ways that this capability could be used. The required documentation of the register bits, and PDL procedures provided by the component designer for specific reset sequences, possibly in conjunction with running other on-chip tests, will allow test engineers to use this capability to enhance the test process. One common use might be to reset the system logic after interconnection tests that will normally leave the system logic in an indeterminate state.

Nothing in these rules prevents the designer from using the reset selection register to control other similar signals affecting the system logic. Given how this register can adversely affect the operation of the system logic, the documentation of all fields of this register should be thorough, if public, or documented as reserved if private.

Documenting a register field for each pair of register bits controlling a reset signal to the system logic would be a good practice, as would, where appropriate, the association between the field and the reset input pin.

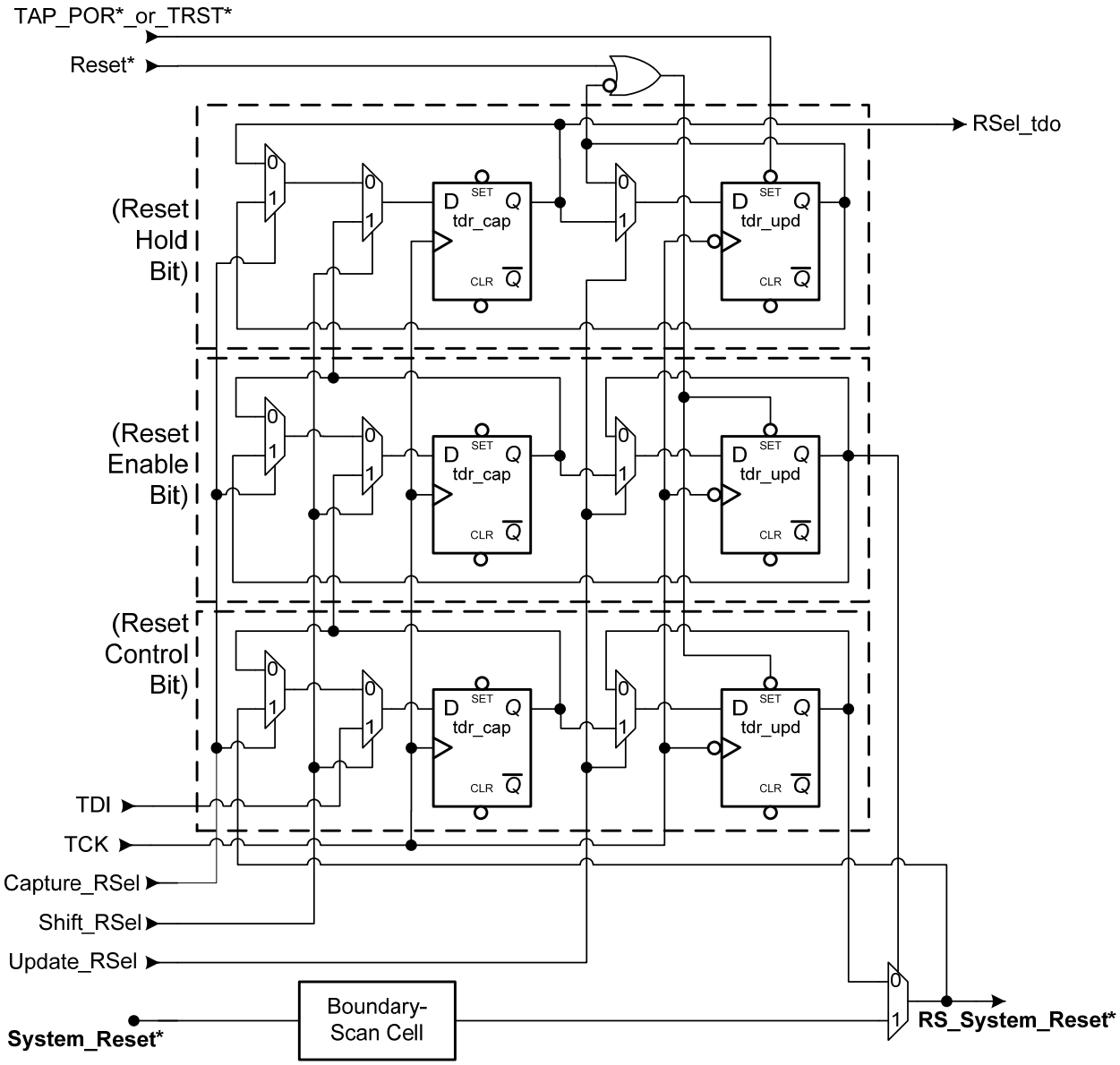


Figure 17-2—Minimal reset selection register example

Figure 17-2 is an example circuit of a minimal (controls and observes only one system reset input) reset selection register with no capture capability beyond that recommended in the specifications. As always, this circuit is just an example to illustrate how the intent of the rules might be implemented. In this example, all shift and update flops use the recommended ungated TCK and data-wrap-back rather than gated TCK to maintain their value, although a gated TCK implementation could be used instead. The reset-enable update register output switches between the functional reset source (in this case an input pin) and the reset-control update register. The reset-hold register controls whether the rest of the update registers will be reset (to the all 1 state) by the *Test-Logic-Reset* TAP controller state. The reset-hold register itself is reset by the same signal used to reset the TAP controller (see Figure 6-8).

Component designers should be aware of two aspects of register-select design implementation. First, both the rules and this design example assume that all of the functional reset sources are negative active signals, so all of the register bits are also negative active signals. When the reset-enable cell update register stage is set to its default high, then the reset control bit is not in control. When the reset-control cell update register stage is set to its default high, it is assumed to be inactive, even if selected by the reset-enable cell update register stage. Even the reset-hold cell update register stage, when set to its default high state, allows the Reset* signal to reset the enable and control bits. Designers should be careful to handle both negative-active and positive-active functional reset sources correctly.

Second, there is the possibility of glitches on the output that could create problems for an asynchronous reset. The output multiplexer sourcing RS_System_Reset* shown in Figure 17-2 (or equivalent logic) should be "hazard-free," although that is not sufficient. There is also an inherent race condition to the data and select inputs of that output multiplexer if the reset-enable and reset-control cell update register stage pair are both toggled with a single scan. Under each combination of functional reset source state and whether reset-enable or reset-control reaches the multiplexer first, one of the four possible transition pairs can produce a glitch on the output as shown in Table 17-1.

For this reason, the reset-enable bit(s) could be asserted in one scan to take control away from the functional reset source(s) prior to asserting and de-asserting the reset-control bit(s) in subsequent scans. Obviously, this could lead to up to three scans to complete asserting and de-asserting a system reset function through the TAP.

Table 17-1—Logic hazards of dual transitions of reset-enable and reset-control pairs

System_Reset*	reset-enable*	reset-control*	RS_System_Reset* (reset_enable faster than reset_control)	RS_System_Reset* (reset_enable slower than reset_control)
1	1➡0	1➡0	1➡0	1➡0
1	1➡0	0➡1	1➡0➡1	1
1	0➡1	1➡0	1	1➡0➡1
1	0➡1	0➡1	0➡1	0➡1
0	1➡0	1➡0	0➡1➡0	0
0	1➡0	0➡1	0➡1	0➡1
0	0➡1	1➡0	1➡0	1➡0
0	0➡1	0➡1	0	0➡1➡0

18. Conformance and documentation requirements

18.1 Claiming conformance to this standard

The level of conformance to this standard can vary according to the range of test operations supported.

18.1.1 Specifications

Rules

a) Components that claim conformance to this standard shall comply with all relevant rules in the Specifications Clauses of this standard.

b) When it is claimed that a component conforms to this standard, the claim shall clearly identify the subset of the public instructions defined in this standard that is supported, as listed in Table 18-1 and defined in 8.2.

Table 18-1—Public instructions

Instruction	Status
BYPASS	Mandatory
CLAMP	Optional (Recommended)
CLAMP_HOLD, CLAMP_RELEASE, TMP_STATUS	Optional as a group
EXTEST	Mandatory
HIGHZ	Optional (Recommended)
IC_RESET	Optional
IDCODE	Optional
ECIDCODE	Optional
INIT_SETUP, INIT_SETUP_CLAMP	Optional as a group
INIT_RUN	Optional
INTEST	Optional
PRELOAD	Mandatory
RUNBIST	Optional
SAMPLE	Mandatory
USERCODE	Optional

Recommendations

c) It is recommended that components support either the *CLAMP* or the *HIGHZ* instruction or both.

NOTE—The importance of *CLAMP* and *HIGHZ* is emphasized in this version of the standard. ICs may be included in scan chains with ICs that support the optional initialization instructions. ICs without support for *CLAMP* and *HIGHZ* must be in *EXTEST* during the initialization process of the ICs requiring initialization and presenting a longer *SHIFT-DR* operation during access to the initialization status register of those ICs.

Permissions

d) ASIC vendors may claim conformance to this standard by illustrating an interconnection of cells that, if built, would produce a component that meets the requirements of this standard.

18.1.2 Description

The minimum requirement for conformance to this standard is set to help ensure that the user of an integrated circuit can perform two basic functions using the test logic: examine the operation of a prototype system and test assembled products for assembly-induced defects during manufacturing.

18.2 Prime and second source components

18.2.1 Specifications

Rules

a) With the sole exception of the device identification code, the digital operation of publicly accessible test logic for second source components shall be identical to that for the prime source component in response to all public instructions.

18.2.2 Description

It is essential that both the system and the test logic of prime and second-source components operate in the same manner in the component purchaser's environment. This helps ensure that test programs created for a printed circuit board containing multiply sourced components produce consistent results regardless of the source of individual components. However, differences in analog characteristics of the I/O that do not change the digital behavior of the component are not considered to violate this rule. Differences in analog characteristics of the I/O that would prevent the second-source component from interoperating with components compatible with that of the prime-source components would violate this rule since the digital operation is affected.

This standard only sets requirements for the test logic. There are many functional and physical characteristics (such as the I/O pin locations) that would have to be met for a part to be considered a second source, and those are not addressed here.

The only exceptions to this requirement are the optional device identification register and any test logic that is accessed only in response to private instructions. In the former case, the identification code shall vary to identify the source of the particular component, its part number, and its revision (see Clause 12). In the latter case, test logic that is not publicly accessible is not intended for use other than by the component vendor; therefore, this test logic should not be operated by a board-level test program.

18.3 Documentation requirements

18.3.1 Specifications

Rules

a) For any component that claims conformance to this standard, the operation of all public test logic shall be fully documented.
b) The following information, required by the component purchaser for use in test development and other activities, shall be supplied by the component manufacturer using the BSDL language described in Annex B, the PDL language described in Annex C where required, or in a published specification, as appropriate.
    1) *Instruction register.* The following information pertaining to the instruction register is required:
        i) Its length.
        ii) The pattern of fixed values loaded into the register during the *Capture-IR* controller state.
        iii) The significance of each design-specific data bit presented at a parallel input, where provided.
    2) *Instructions.* For each public instruction offered by a component, the following information is required:
        i) The binary code(s) for the instruction.
        ii) A list of test data registers placed in a test mode of operation by the instruction.
        iii) The name of the serial test data register path enabled to shift data by the instruction.
        iv) A definition of any data values that shall be written into test data registers before selection of the instruction and the order in which these values shall be loaded.

v) The effect of the instruction. Any system pins whose drivers become inactive as a result of loading the instruction should be clearly identified.

vi) A definition of the test data registers that will hold the result of applying a test and of how they are to be examined.

vii) A description of the method of performing the test and of how data inputs and their corresponding data outputs are to be computed. In particular, PDL (see Annex C) procedures are required for the *INIT_SETUP, INIT_SETUP_CLAMP,* and *INIT_RUN* instructions, and for the *ECIDCODE* instruction when actions are required to retrieve the ECID value.

viii) If private instructions are utilized in a component, the vendor shall clearly identify any instruction binary codes that, if selected, would cause hazardous operation of the component.

3) *Self-test operation.* For each instruction that causes operation of a self-test function, the following information is required in addition to that listed under rule b2) of this subclause:

i) The minimum duration (e.g., a number of cycles of TCK) required to complete the test.

ii) A definition of the test data registers whose states are altered during execution of the test.

iii) A definition of the results of executing the self-test on a fault-free component.

iv) An estimate of the percentage (e.g., to the nearest 5%) of the single stuck-at faults in the component's circuitry that will be detected by the self-test function *or* a description of the operation of the self-test function and the circuitry exercised.

4) *Initialization operation.* For each instruction that causes operation of an initialization function, the following information is required in addition to that listed under rule b2) of this subclause:

i) A definition of fields and possible values of those fields for the test data registers whose states are altered during execution of the initialization instructions.

ii) The maximum duration (e.g., a number of cycles of TCK or absolute time) required to complete any initialization sequential process.

iii) A definition of successful completion of the initialization on a fault-free component.

5) *Test data registers.* For each test data register available for public use and access in a component, the following information is required:

i) The name of the register, used for reference in other parts of the data sheet.

ii) The purpose of the register.

iii) The length with all excludable segments excluded, and the length of each excludable segment.

iv) The beginning and end of each excludable segment, and the association of domain-control and segment-select cells with each excludable segment.

v) A full description of the operating modes of the register.

vi) The result of setting each bit at the parallel output of the register.

vii) The significance of each bit loaded from the parallel input of the register.

6) *Boundary-scan register.* The following information is required in addition to that listed under rule b5) of this subclause:

i) The correspondence between boundary-scan register bits and system pins, system direction controls, or system output enables.

ii) Whether each pin is an input, a two-state output, a three-state output, or a bidirectional pin.

iii) For each boundary-scan register cell at an input pin, whether the cell can apply tests to the on-chip system logic.

iv) For each boundary-scan register cell associated with an output or direction control signal, a list of the pins controlled by the cell and the value that shall be loaded into the cell to place the driver at each pin in an inactive state or will be observed using the *SAMPLE*, *PRELOAD*, or *INTEST* instructions when the on-chip system logic causes the driver to be inactive.

v) The method by which single-step operation is to be achieved while the *INTEST* instruction is selected if this instruction is supported.

vi) The method of providing clocks to the on-chip system logic while the *RUNBIST* instruction is selected, if this instruction is supported.

vii) For each redundant cell, whether the cell returns either the value shifted in or a constant after loading of the cell in the *Capture-DR* controller state.

7) *Device-identification register.* Where a device identification register is included in a component, the following information is required in addition to that listed under rule b5) of this subclause:

i) The value of the manufacturer's identification code.

ii) The value of the part number code.

iii) The value of the version code.

iv) The method of programming the value of the supplementary identification code, where required.

8) *Performance.* The performance of the test logic should be fully defined, including the following information:

i) The maximum acceptable TCK clock frequency.

ii) A full set of timing parameters for the test logic.

iii) The logic switching thresholds for TAP input and output pins.

iv) The load presented by the TCK, TMS, TDI, and TRST* pins.

v) The drive capability of the TDO output pin.

vi) The extent to which the TDO driver may be overdriven when active (e.g., using an in-circuit test system).

vii) Whether TCK may be stopped in the logic 1 state.

9) *Compliance-enable inputs*. If a component has compliance-enable inputs as defined in 4.8.1, then the following documentation shall be provided:

i) A complete list of these inputs labeled as compliance-enable inputs.

ii) A complete list of those logic patterns that, when applied at the compliance-enable inputs, will enable compliance to this standard.

iii) A clear indication of any patterns that, if applied to the compliance-enable inputs, would cause hazardous operation of the component.

c) A verified BSDL description of the component shall be supplied by the component manufacturer (see Annex B).

NOTE—This rule mandates correct and accurate BSDL documentation to claim a component is conformant to this standard. The component manufacturer must take reasonable steps to validate that the BSDL matches the silicon through simulation or physical compliance testing.

18.3.2 Description

Figure 18-1 and Figure 18-2 show how setup and hold timing parameters and propagation delays should be measured relative to the test clock TCK and a reference voltage V_{ref}. Note that such timing parameters are required for TMS, TDI, and TDO and for system pins that can be driven from the test logic (e.g., the system data input setup time for the boundary-scan register before the rising edge of TCK in the *Capture-DR* controller state).

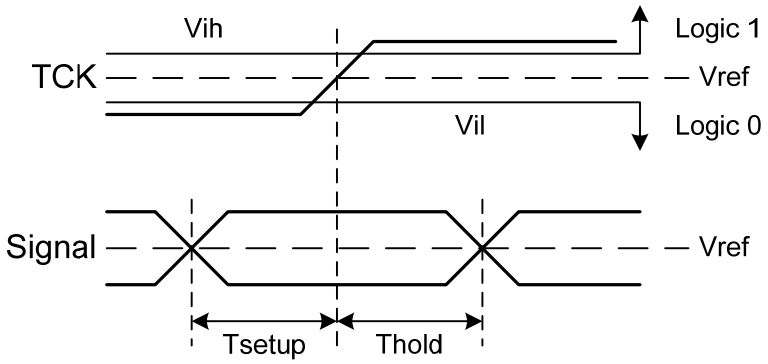


Figure 18-1—Measuring setup and hold timing

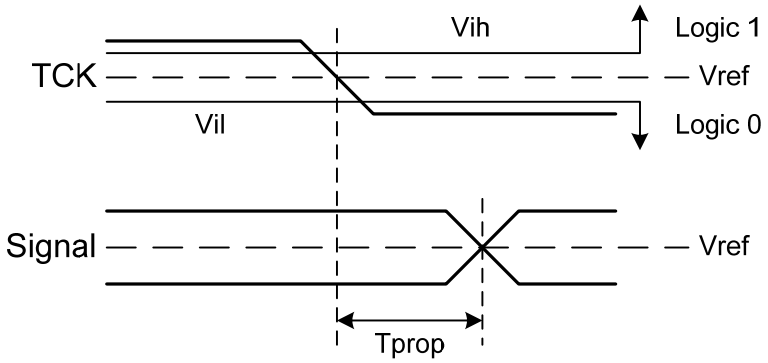


Figure 18-2—Measuring propagation delay

Annex A

(informative)

Example implementation using level-sensitive design techniques

This annex has been deleted as of the 2013 standard. It has served its purpose, and the industry has moved to a position where many implementation details are now handled by tools that can provide either LSSD or Mux-scan solutions, as desired.

Annex B

(normative)

Boundary Scan Description Language (BSDL)

This annex defines a machine-readable language that allows rigorous description of testability features in components that comply with IEEE Std 1149.1. The language is called the Boundary-Scan Description Language (BSDL). BSDL documentation is a mandatory requirement of this standard; components cannot claim conformance without valid BSDL documentation. It is based on the syntax and grammar of VHSIC Hardware Description Language (VHDL) (IEEE Std 1076) but includes constructs that meet the objectives of the board test that are not anticipated by VHDL. BSDL was never intended to be synthesized or simulated with VHDL-based tools. It deviates from VHDL in several ways as the purpose and mission of BSDL is different than VHDL.

B.1 General information

B.1.1 Document outline

In B.2 to B.4, the purpose of BSDL, its scope, and its relationship to VHDL is defined. In B.5 to B.7, the general characteristics of the language are described. In B.8, the Entity description describes the overall structure of a BSDL description. In B.8.2 to B.8.25 detailed descriptions of each mandatory and optional section of a BSDL description are provided. They are documented in the order they should appear. The Standard BSDL Package is described in B.9, and B.10 describes a design-specific BSDL package supplied by the component designer, both based on VHDL packages. Some special cases are provided in B.11 for purposes of illustration and indicates how such components can be specified in BSDL. A typical BSDL description is shown in B.11.1, while B.12 briefly documents the 1990 version of BSDL (see also B.1.3). The 1994 version of BSDL is documented in B.13 (the first version approved by the IEEE Standards Board and published by the IEEE), while B.14 documents the 2001 version of BSDL.

In B.8.2 to B.8.25, the detailed descriptions of each mandatory and optional section of a BSDL description are organized in the following way:

— Short introduction
— Specifications, including Syntax, Rules, Recommendations (if any), and Permissions (if any)
— Description
— Examples

The Specifications subclause is normative. The other subclauses are descriptive.

Commonly used syntactic elements are defined in B.6.2.

B.1.2 Conventions

— Examples are printed in `Courier New` font.
— See B.6 for conventions relating to lexical items and syntax.
— For clarity, all reserved words, predefined words, and punctuation are shown in text (not examples) in **bold Helvetica** font within this document. VHDL reserved and predefined words used in BSDL will be shown in **lowercase** letters, and BSDL reserved words will be shown in **UPPERCASE** letters. (BSDL itself is case-insensitive; this convention is adopted for clarity.)

B.1.3 BSDL history

The development of BSDL started soon after the first promulgation of this standard in 1990. When, out of common self-interest, an industry-wide group of companies implementing tools to support this standard realized that a single

language for describing the boundary-scan implementations in components would be of benefit, many tools were built using early draft specifications of the BSDL language.

Developments made both to this standard and to the BSDL definition since the language was first proposed in 1990 have resulted in the obsolescence of some of the constructs from the first draft versions of the language. Some constructs were rendered unnecessary as a result of the standardization of the *CLAMP* and *HIGHZ* instructions in 1993, while others were found to duplicate information provided elsewhere in a BSDL description and thus were removed as redundant.

Because a significant number of BSDL descriptions have been written based on the 1990 draft version of the language and because these descriptions are likely to remain in circulation for some time, implementers of tools based on BSDL may wish to design them to read both the BSDL language defined in this annex and the earlier 1990 version. Information on how to do this is contained in B.12 through B.14.

The first version of BSDL issued by the IEEE was the 1994 version. As this standard has been revised, the definition of BSDL has been modified to correspond. Implementers of tools based on BSDL should design them to accommodate all versions of the BSDL language defined in this annex and the earlier versions. Information on how to do this is contained in B.12.3.

This update of the standard introduces a number of new documentation capabilities in BSDL. See "Changes introduced by this revision" in the front matter of this standard for a quick overview.

B.2 Purpose of BSDL

BSDL provides a human-readable and machine-readable means of representing some parts of the documentation specified in 18.3. The scope of the language is defined in B.3.

The goal of the language is to facilitate communication among companies, individuals, and tools that need to exchange information on the design of test logic that complies with this standard. For example:

- A vendor of a component that supports this standard supplies a BSDL description to purchasers.
- A vendor of intellectual property (IP) intended for use in a component that supports this standard supplies a BSDL User Package to purchasers.
- Automated test-generation tools may use a library of BSDL descriptions to allow generation of a test for a particular loaded board.
- The test logic defined by this standard could be synthesized with the support of a BSDL description.

BSDL describes "finished" design, not "work-in-progress." For example, when a bare die conforming to this standard is produced, a fully compliant BSDL description must be provided for it. A die may be inserted into one or more types of component packages as well, with each variation described in BSDL (see B.8.7). BSDL for partially synthesized test logic is considered "work-in-progress," is not necessarily compliant with this annex, and in most cases, should not be transmitted beyond the synthesis environment. Similarly, BSDL packages supplied by an IP provider to describe the "finished" IP design must be compliant with the appropriate subset of BSDL defined for use in user packages.

B.3 Scope of BSDL

BSDL is not a general-purpose hardware description language—it is intended solely as a means of describing key aspects of the implementation of this standard within a particular component. A BSDL description is not itself a simulation model. Examples of features that are and are not described using BSDL are listed in Table B-1.

Table B-1—Scope of BSDL

Features described by BSDL	Features that cannot or need not be described
Length and structure of the boundary-scan register and lengths of other TDRs	TAP-controller state diagram
Availability of the optional TRST* pin	Bypass register
Physical locations of the TAP pins	Length of the device-identification register
Instruction binary codes	Operation of standard and design-specific instructions
Provision of optional *INTEST, RUNBIST, CLAMP, IDCODE, USERCODE, ECIDCODE, HIGHZ, INIT_SETUP, INIT_SETUP_CLAMP, INIT_RUN,* and *IC_RESET* standard instructions and design-specific instructions	Provision of *BYPASS*, *SAMPLE*, *PRELOAD*, and *EXTEST* instructions
Device-identification code	Operation of standard and design-specific TDRs

Note that the language describes only features of the test logic that can vary from component to component, depending on the choice of the component designer. Features that are completely specified by this standard (without option) are not required to be described in BSDL but may be described if component designers so wish.

Furthermore, BSDL does not have a general means for providing for the specification of logic levels, timing parameters, power requirements, and similar factors. These data do not affect the logical behavior of an implementation and most likely are already described in other parts of the specification of any given component.

B.4 Relationship of BSDL to VHDL

BSDL is based on VHDL (IEEE Std 1076).

For BSDL conforming to the 2013 version of this standard, or later, BSDL cannot be processed by VHDL tools due to some extensions (such as the new <pin type> keywords **LINKAGE_IN** or **POWER_POS**) that are incompatible with the VHDL standard.

For BSDL conforming to versions of this standard predating 2013, if BSDL is to be processed by VHDL tools, the user must be prepared to modify a BSDL description to account for implementation dependencies in VHDL-based tools. No way has been found to avoid this small amount of effort without introducing further undesirable complications. Specifically, the <standard use statement> (see B.8.4) and the <use statement> (see B.8.5) may require editing because of tool and file system dependencies. The syntax of the statements as defined is compliant with VHDL; however, an additional prefix (identifying a library in which the Standard BSDL Package will be found) will need to be added for most VHDL tools. A syntax lacking such a prefix has been chosen to force an error in such an application rather than risk unpredictable and confusing errors due to inclusion of an inappropriate prefix.

NOTE—In the event of an error or omission in this annex regarding VHDL syntax, other than deliberate exceptions, IEEE Std 1076 takes precedence.

B.4.1 Specifications

BSDL does not employ all the syntactic elements of VHDL, only those required to meet the scope of BSDL. In addition, BSDL imposes additional requirements on the syntax and content of certain character strings, that is, sequences of characters between quotation marks (e.g., "**EXTEST**"). A VHDL parser will not check the information in these strings while there are several string syntaxes defined within BSDL. For cases in which a feature could be described in several ways within VHDL, a restricted set of ways has been selected and defined explicitly as the standard practice for BSDL. This restriction simplifies the application of the VHDL subset for BSDL, particularly for tools that are required only to read or generate BSDL (i.e., tools that have no requirement to read or write the full VHDL language).

Rules

a) The following VHDL statements, and no others, shall be employed in BSDL:

attribute	**generic**	**subtype**
constant	**package**	**type**
end	**package body**	**use**
entity	**port**	

NOTE—This is not a complete list of VHDL keywords used in BSDL, only the VHDL statements. In some cases, only a subset of a particular VHDL language element syntax is used in BSDL. Descriptions of the lexical elements and statement syntax for each syntax element are contained in B.5 through B.8 and B.10. Historical information on such elements is found in B.12.

b) A BSDL parser shall check that the information in BSDL-defined strings is appropriate for the relevant parameters or attributes for which such strings are values.

Permissions

c) A BSDL parser may implement just the subset of VHDL needed to implement the syntax listed in this annex.

B.5 Lexical elements of BSDL

The lexical elements of BSDL are a subset of those of VHDL as defined in IEEE Std 1076-. The following subclauses enumerate the lexical elements needed to understand the BSDL language definition.

B.5.1 Character set

The language is not case sensitive: that is, for example, the character **a** is considered identical to the character **A**. Therefore, the following names are identical:

```
FRED        Fred        fred
```

B.5.1.1 Specifications

Rules

a) Except for specific tokens that allow expanded character sets, only the following characters shall be permitted within the language, including within <*name* string> tokens where most of BSDL is defined:
   1) Letters: The uppercase and lowercase 26 letters of the Roman alphabet: **A** to **Z** and **a** to **z.** These shall be represented in the syntax by the token <letters>.

      NOTE 1—This list is smaller than that of VHDL.

   2) Digits: **0** to **9.** These shall be represented in the syntax by the token <digit>.
   3) Special characters: **" & ' ( ) [ ] * , - + .: ; < = > _** These shall be collectively represented in the syntax by the token <special characters>.

      NOTE 2—This list is different than that of VHDL.

   4) Whitespace: The space character and the VHDL format effector “horizontal tabulation” shall be used within a line as logical separators.

5) New line: The VHDL format effectors called “vertical tabulation,” “carriage return,” “line feed,” and “form feed” shall be used to start a new line and as logical separators. These shall be represented in the syntax by the token <newline>.
6) End-of File: An end-of-file may only occur where a <newline> may be expected, and it acts as a final logical separator.

NOTE 3—Tokens that allow expanded character sets include <mnemonic identifier> and <information tag>.

NOTE 4—Logical separators have two purposes. They are used to eliminate lexical ambiguity by separating lexical tokens such as reserved words and/or identifiers. For example, the reserved word **entity** must be separated from the component name identifier that immediately follows it rather than being run together with it. Logical separators and whitespace may also be used in combination to create visually appealing layouts.

B.5.2 BSDL reserved words

B.5.2.1 Specifications

Rules

a) The identifiers listed in this subclause shall be BSDL reserved words and shall have a fixed significance in the language; these identifiers shall not be used for any purpose in a BSDL description, other than as specified in the syntax or as part of a comment or other unparsed element.
b) A reserved word shall not be used as an identifier.
c) Identifiers **BC_0** to **BC_99** shall be boundary-scan register cell identifiers used in the Standard BSDL Package and Standard BSDL Package Body; names **BC_0** through **BC_10** are used today, while **BC_11** through **BC_99** shall be reserved for future use.
d) All identifiers that start with **STD_1149_** shall be reserved.

AT_PINS
BC_0 to **BC_99**
BIDIR
BIDIR_IN
BIDIR_OUT
BOTH
BOUNDARY
BOUNDARY_LENGTH
BOUNDARY_REGISTER
BOUNDARY_SEGMENT
BROADCASTFIELD
BROADCASTVALUES
BSCAN_INST
BSDL_EXTENSION
BYPASS
CAP
CAP_DATA
CAPTURES
CELL_DATA
CELL_INFO
CELL_TYPE
CHRESET
CLAMP
CLAMP_HOLD
CLAMP_RELEASE
CLOCK
CLOCK_INFO
CLOCK_LEVEL
COMPLIANCE_PATTERNS
COMPONENT_CONFORMANCE
CONTROL
CONTROLR
DEFAULT
DELAYPO
DESIGN_WARNING
DEVICE_ID
DIFFERENTIAL_CURRENT
DIFFERENTIAL_VOLTAGE
DOMAIN
DOMAIN_EXTERNAL
DOMCTRL
DOMPOR
ECID
ECIDCODE
EXPECT_DATA
EXPECT0
EXPECT1
EXTEST
HIERRESET
HIGHZ
IC_RESET
ID_BITS
ID_STRING
IDCODE
IDCODE_REGISTER
INIT_DATA

INIT_RUN
INIT_SETUP
INIT_SETUP_CLAMP
INIT_STATUS
INPUT
INSTRUCTION_CAPTURE
INSTRUCTION_LENGTH
INSTRUCTION_OPCODE
INSTRUCTION_PRIVATE
INTERNAL
INTEST
INTEST_EXECUTION
KEEPER
LINKAGE_INOUT
LINKAGE_BUFFER
LINKAGE_IN
LINKAGE_OUT
LINKAGE_MECHANICAL
LOW
MON
NOPI
NOPO
NORETAIN
NOUPD
OBSERVE_ONLY
OBSERVING
ONE
ONE_HOT
OPEN
OPEN0
OPEN1
OPENX
OUTPUT2
OUTPUT3
PHYSICAL_PIN_MAP
PI
PIN_MAP
PIN_MAP_STRING
PO
PORRESET
PORT_GROUPING
POWER_0
POWER_POS
POWER_NEG
POWER_PORT_ASSOCIATION
PRELOAD
PULL0
PULL1
PULSE0
PULSE1
REGISTER_ACCESS
REGISTER_ASSEMBLY
REGISTER_ASSOCIATION
REGISTER_CONSTRAINTS
REGISTER_FIELDS
REGISTER_MNEMONICS
RESET_SELECT
RESETVAL
RUNBIST
RUNBIST_EXECUTION
SAFE
SAMPLE
SEGMENT
SEGMUX
SEGSEL
SEGSTART
SELECTMUX
SELECTFIELD
SELECTVALUES
SHARED
STD_1149_*
TAP_SCAN_CLOCK
TAP_SCAN_IN
TAP_SCAN_MODE
TAP_SCAN_OUT
TAP_SCAN_RESET
TAPRESET
TIE1
TIE0
TMP_STATUS
TRSTRESET
UPD
USER
USERCODE
USERCODE_REGISTER
VREF_IN
VREF_OUT
WAIT_DURATION
WEAK0
WEAK1
X
Z
ZERO

NOTE—In this list of reserved words, the entry **STD_1149_*** is to be interpreted to mean all names that start with **STD_1149_**, for example, **STD_1149_1_2013**.

B.5.3 VHDL reserved and predefined words

The reserved words shown in the following list are the BSDL subset of VHDL. As BSDL can no longer be parsed by a VHDL compiler, other VHDL reserved words are no longer reserved in BSDL.

B.5.3.1 Specifications

Rules

a) The identifiers listed as follows shall be called VHDL (IEEE Std 1076) reserved and predefined words and shall have a fixed significance in the BSDL language.
b) These identifiers shall not be used for any purpose in a BSDL description other than as defined in BSDL syntax, or as part of a comment or other unparsed element.
c) A reserved word shall not be used as an explicitly declared identifier.

all	**entity**	**positive**
array	**generic**	**range**
attribute	**in**	**record**
bit	**inout**	**signal**
bit_vector	**is**	**string**
body	**of**	**subtype**
buffer	**others**	**to**
constant	**out**	**true**
downto	**package**	**type**
end	**port**	**use**

B.5.4 Identifiers

Identifiers are user-supplied names. BSDL supports three types of identifiers. The most common, and used almost universally, is the "VHDL identifier," which follows the rules of VHDL. For mnemonic names, a much less restrictive "mnemonic identifier" is defined to allow more expressive and meaningful names for the constant values that may be assigned to registers and register fields. To support register field names, which include the full logical hierarchy in the field name, the "prefix identifier" is defined to allow for the range of naming conventions of various languages and tools. This is only used in the various ways of defining an <extended field name> (see B.8.19.1).

B.5.4.1 Specifications

Rules

a) A VHDL identifier shall be represented in the syntax by the token <VHDL identifier> and shall be any string chosen as a name for an item and conforming to the following:
   1) Identifiers shall start with a letter and may contain letters, digits, or, within restrictions, the underscore character.

      NOTE 1—For example, the following are valid identifiers:

      ```
      BSDL
      IEEE_STD_1149_1
      ```

   2) There shall be no upper limit to the number of characters in an identifier.
   3) The underscore character (_) shall not be allowed as the last character in an identifier.

      NOTE 2—This rule is derived from VHDL. Example:

      ```
      IEEE_STD_1149_  -- This is not a valid VHDL identifier.
      ```

   4) Adjacent underscore characters (__) shall not be allowed.

      NOTE 3—This rule is derived from VHDL. Example:

      ```
      IEEE_STD__1149  -- This is not a valid VHDL identifier.
      ```

b) A mnemonic identifier shall be represented in the syntax by the token <mnemonic identifier> and shall be any combination of letters, digits, and special characters conforming to the following:

1) Mnemonic identifiers shall include at least one alphabetic character.
2) Mnemonic identifiers shall start with any valid character.
3) Mnemonic identifiers shall not resolve to:
   i) A <real>, <integer>, <binary pattern>, <hex pattern>, <decimal pattern>, or <pattern> value.
   ii) A single character U.
   iii) A constraint expression operator as defined in Table B-5.
   iv) A BSDL comment; that is, it shall not start with or contain a double dash.
4) Mnemonic identifiers shall not include special characters other than at sign (**@**), asterisk (*), underscore (_), minus sign (-), plus sign (**+**), vertical bar (|), percent sign (%), tilde (~), and period (.).

NOTE 4—A valid VHDL identifier is also a valid mnemonic identifier.

NOTE 5—Examples:

```
12.5E6   -- Not a compliant mnemonic identifier (resolves to BSDL <real>)
12.5exp6 -- A compliant mnemonic identifier
12.5--E6 -- Not a compliant mnemonic identifier (contains double dash)
12.5__E6 -- A compliant mnemonic identifier
1010Xx   -- Not a compliant mnemonic identifier (resolves to <pattern>)
_1010Xx_ -- A compliant mnemonic identifier
```

c) A prefix identifier shall be represented in the syntax by the token <prefix identifier> and shall be any combination of letters, digits, and the underscore character not starting with a digit.

NOTE 6—A prefix identifier essentially follows the rules of Verilog identifiers except for case sensitivity. Verilog identifiers are case sensitive, where Fred and FRED are two different identifiers in Verilog but the same identifier in VHDL and BSDL. Given the use of prefix identifiers in BSDL, this is not expected to be an issue.

NOTE 7—A valid VHDL identifier is also a valid prefix identifier.

NOTE 8—Examples:

My_Reg – A compliant prefix and VHDL identifier
12_5exp6 – Not a compliant prefix identifier (starts with a digit)
_12_5__E6 – A compliant prefix identifier (note use of initial and double underscores)
_0x10107f_ – A compliant prefix identifier

B.5.5 Numeric literals

"Numeric literals" are commonly used definitions for syntax items representing a numerical value

B.5.5.1 Specifications

Rules

a) An integer shall be represented in the syntax by the token <integer> and shall be an unsigned decimal number consisting only of the digits 0 through 9.

NOTE 1—An integer may start with any number of leading zeros, which is different from a <decimal pattern> defined below.

b) A real number shall be represented in the syntax by the token <real> and shall be of the form <integer>**.<**integer> or <integer>**.<**integer>**E<**integer>, all written contiguously without spaces, underscores, or format effectors; the **E** is case insensitive.

NOTE 2—**1E3** is not real because it does not contain a decimal point. It is not an integer because it does include a letter. The number **20.0E6** is real, as is the equivalent **20000000.0**.

c) The syntactical token <pattern> shall be a contiguous sequence of one or more **0**, **1**, **X** characters containing no spaces or format effectors; the **X** is case insensitive.

NOTE 3—For example, 001x00 and XX010X are compliant. However, 100 X00 is not compliant because of the embedded space.

d) The syntactical token <32-bit pattern> shall be a <pattern> with exactly 32 characters in its character sequence.

e) The syntactical token <binary pattern> shall be a contiguous string of characters starting with **0b** or **0B** followed by one of the set [**01xX**] followed by zero or more of the characters in the set [**01xX_**] and containing no spaces or format effectors.

f) The syntactical token <hex pattern> shall be a contiguous string of characters starting with **0x** or **0X** followed by one of the set [**0-9a-fA-FxX**] followed by zero or more of the characters in the set [**0-9a-fA-FxX_**] and containing no spaces or format effectors.

g) The syntactical token <decimal pattern> shall be an unsigned contiguous string of characters of the set [**0-9**]; multicharacter values shall not start with 0, shall contain no spaces or format effectors, shall have a value less than $2^{32} - 1$, and shall always match any binary field large enough to hold the most significant 1 bit of the binary equivalent value of the decimal pattern.

B.5.5.2 Description

A pattern generally represents a logical (unsigned binary) value. A low state for each bit is denoted by **0**, a high state is denoted by **1**, and a don't-care value shall be denoted by **X** or **x**. When comparing two values, an "X" will never cause a miscompare for that bit or bits. When setting a value, an "X" will leave the corresponding bit or bits unchanged.

Lexical ambiguity exists in certain situations and is resolved by context. For example, a <pattern> that starts with an **X** can be differentiated from a <VHDL identifier> by context derived from the syntax. Similarly, a <pattern> that does not include an **X** can be differentiated from an integer such as **100** (one hundred), again by context derived from the syntax.

B.5.6 Strings

B.5.6.1 Specifications

Rules

a) A string shall be represented in the syntax by the token <string> or <*name* string> and shall be defined as a sequence of zero or more characters from the language character set (see B.5.1.1) enclosed between quotation marks.

b) A quotation mark character shall not be allowed within a string in BSDL.

NOTE 1—For example:

```
"Mary had a little lamb"   -- Allowed
"Fred said ""HELP"""       -- Not allowed
"Fred said 'HELP'"         -- Allowed
```

c) A string shall fit on one line since it is a lexical element.

NOTE 2—Therefore, the only compliant VHDL format effector in a string literal is horizontal tabulation.

d) The concatenation operator **&** shall be used to concatenate strings.

NOTE 3—For example:

```
"Mary had a little lamb. " &
"Its fleece was white as snow."
```

is a single string, identical to:

```
"Mary had a little lamb. Its fleece was white as snow."
```

e) BSDL shall not permit replacement of the quotation mark with any other character.

NOTE 4—Many character encoding schemes support open and close quotation marks (“ ”) as well as the generic quotation mark ("). These may all be treated as identical for this definition of a String.

B.5.6.2 Description

Strings are arbitrary sequences of characters selected from the allowed character set (see B.5.1.1) enclosed in quotation marks. Strings are used extensively in BSDL, and the content of strings contains most of the description of the test logic. Such descriptive strings have syntactical tokens of the form <*name* string>, where *name* is replaced by a specific name, and the internal structure of the named string is further syntactically defined. These named strings must still conform to the rules in B.5.6.1. Examples include <conformance string> (see B.8.6.1) or <cell table string> (see B.8.14.1).

B.5.7 Information tag

An information tag is a lexical element that provides textual information that may be retained by the tools for later use.

B.5.7.1 Specifications

Rules

a) An <information tag> shall be defined as a sequence of zero or more characters including all alpha-numeric characters plus special characters except as noted in rule b), and shall be enclosed in chevrons (< >).
b) The right chevron mark character (>), quotation mark ("), double dash (--), and <newline> characters shall not appear within an information tag.

B.5.7.2 Description

An information tag is the left chevron (<), followed by any sequence of characters not including a right chevron (>), quote (“), double-dash (--), or <newline>, and terminated by a right chevron. Since an <information tag> can only appear within a <string>, the quote, and newline characters (vertical tabulation, carriage return, line feed, form feed) are not permitted within the <information tag>. Information tags always appear within BSDL strings, which may be concatenated substrings split across multiple lines. While a VHDL comment may not start within a string, the double-dash is still prohibited within an information tag.

The information tag is intended as a comment that can be processed and retained by tools. It is used, for instance, to assist in the selection of appropriate mnemonic values, and for error messages in register constraints. Because a quote is a BSDL string terminator, no quote can be conveyed as text within an information tag.

```
"... <This is a valid information tag within a string.> ..."

"... <   The characters between the chevrons are information. > ..."

"... <The information tag above has leading and trailing spaces.> ..."

"... <This one contains -- which is illegal.> ..."

"... <This is a valid information tag " &  -- valid VHDL comment
      "within a string that is split across several lines " &
      "with concatenation. The information tag data " &
      "does not include the quotes or newlines.> ..."

"... <Tags can include 'non-alpha' characters like ?]*:;!0-9&%$.> ..."

"... <This is NOT a valid information tag (or string)
                because of the embedded newline.> ..."
```

B.5.8 Comments

B.5.8.1 Specifications

Rules

a) Text between double dash (`--`) characters and the end of a line shall be treated as a comment.
b) Comment text shall be allowed to contain any special character allowed by VHDL, in addition to those given in rule a) of B.5.1.1.
c) The entire comment, from the first dash up to but not including the <newline> defining the end of the line, shall be ignored by the parser.
d) Comments shall appear only where a <newline> is allowed.

NOTE—For example, consider the following:

```
"This is all" &    -- An example of a string split by a comment
" a single string"
"This is not       -- A non-compliant string split by a comment
 a single string"  -- A string may not have an embedded <newline>
```

B.6 Syntax definition

B.6.1 BNF conventions

The syntax of BSDL is presented in a modified Extended Backus-Naur Form (BNF) as follows:

a) Any item enclosed in chevrons (i.e., between the character “<” and the character “>”) is the name of a syntax token that will be defined in this annex, generally within the same syntax description where it is used.
b) Tokens defined in other syntactical descriptions in this annex are underlined and are links to the location of their definition.
c) Items enclosed between braces (i.e., between the character “{” and the character “}”) can either be omitted or included one or more times.
d) Items enclosed between square brackets (i.e., between the character “[” and the character “]”) can be either omitted or included only one time.

e) Text (terminal tokens) shown in **bold Helvetica** type shall be included exactly as it is presented in this annex, other than case.

f) Where there is a choice of tokens, the choices are separated by a vertical bar ("**|**").

g) The symbol "**::=**" is read as "is defined as."

h) Whitespace (spaces, tabulation, carriage returns, etc.) is used in these BNF descriptions to provide enhanced readability and is not part of the syntax. However, whitespace needed for resolving lexical ambiguity (logical separation) is required as described in B.5.1.

i) The use of parenthesis to group items is not used in this description.

j) To minimize ambiguity between the BNF notation and the special characters required in an input stream, the following syntax tokens will be used in place of special characters allowed in the parsed input stream:

1) <left bracket> and <right bracket> shall be the bracket characters "**[ ]**".
2) <left paren> and <right paren> shall be the parenthesis characters "**( )**".
3) <left chevron> and <right chevron> shall be the chevron characters "**< >**".
4) <left brace> and <right brace> shall be the curly brace characters "**{ }**".
5) <asterisk> shall be the character "*****".
6) <ampersand> shall be the character "**&**".
7) <semicolon> shall be the character "**;**".
8) <colon> shall be the character "**:**".
9) <comma> shall be the character "**,**".
10) <period> shall be the character "**.**".
11) <quote> shall be the character "**"**".
12) <minus sign> shall be the character "**-**".
13) <colon-equal> shall be the VHDL variable assignment operator character pair "**:=**".

B.6.2 Commonly used syntactic elements

Two commonly used syntax elements are a <port ID> and <instruction name>. A <port ID> identifies a component *signal* that may be used to interface to external signals at device I/O pins. An <instruction name> is chosen from a list of instructions defined by this standard or may be an added instruction uniquely named by the vendor of the component.

B.6.2.1 Specifications

Syntax

<port ID>::= <port name> | <subscripted port name>
<port name>::= <VHDL identifier>
<subscripted port name>::= <port name> <left paren> <subscript> <right paren>
<subscript>::= <integer>

<instruction name>::= **BYPASS** | **CLAMP** | **EXTEST** | **HIGHZ** | **IDCODE** |
INTEST | **PRELOAD** | **RUNBIST** | **SAMPLE** | **USERCODE** |
ECIDCODE | **CLAMP_HOLD** | **CLAMP_RELEASE** | **TMP_STATUS** |
IC_RESET | **INIT_SETUP** | **INIT_SETUP_CLAMP** | **INIT_RUN** | <VHDL identifier>

NOTE—In earlier editions of this standard, the BSDL reserved word **PRELOAD** did not exist and **SAMPLE** was used as an abbreviated name when the *SAMPLE* and *PRELOAD* instructions were merged [see permission h) of 8.1.1 and rule f) of B.8.11.1].

Rules

a) A <port ID> shall identify a single-bit component signal that is used to interface to external signals.

b) A <port name> shall be included in the <logical port description> statement (see B.8.3).

c) For a <subscripted port name>, the <port name> shall be defined as a **bit_vector** (see B.8.3) and the value of the <subscript> shall be within the <range> specified for that <port name>.

d) For a <port name> that is not a <subscripted port name>, the <port name> shall be defined as **bit** (see B.8.3).

e) A <port ID> of length one defined as a <subscripted port name> shall appear with the subscript everywhere the identifier appears in the BSDL.

f) An <instruction name> shall be the name of an instruction defined in this standard or a design-specific instruction name, and <instruction name> shall represent the instruction of the same name.

g) Where the value of <conformance identification> is **STD_1149_1_2001, STD_1149_1_1993,** or **STD_1149_1_1990**, the <instruction name> shall not be any instruction defined in a later version of this standard.

B.7 Components of a BSDL description

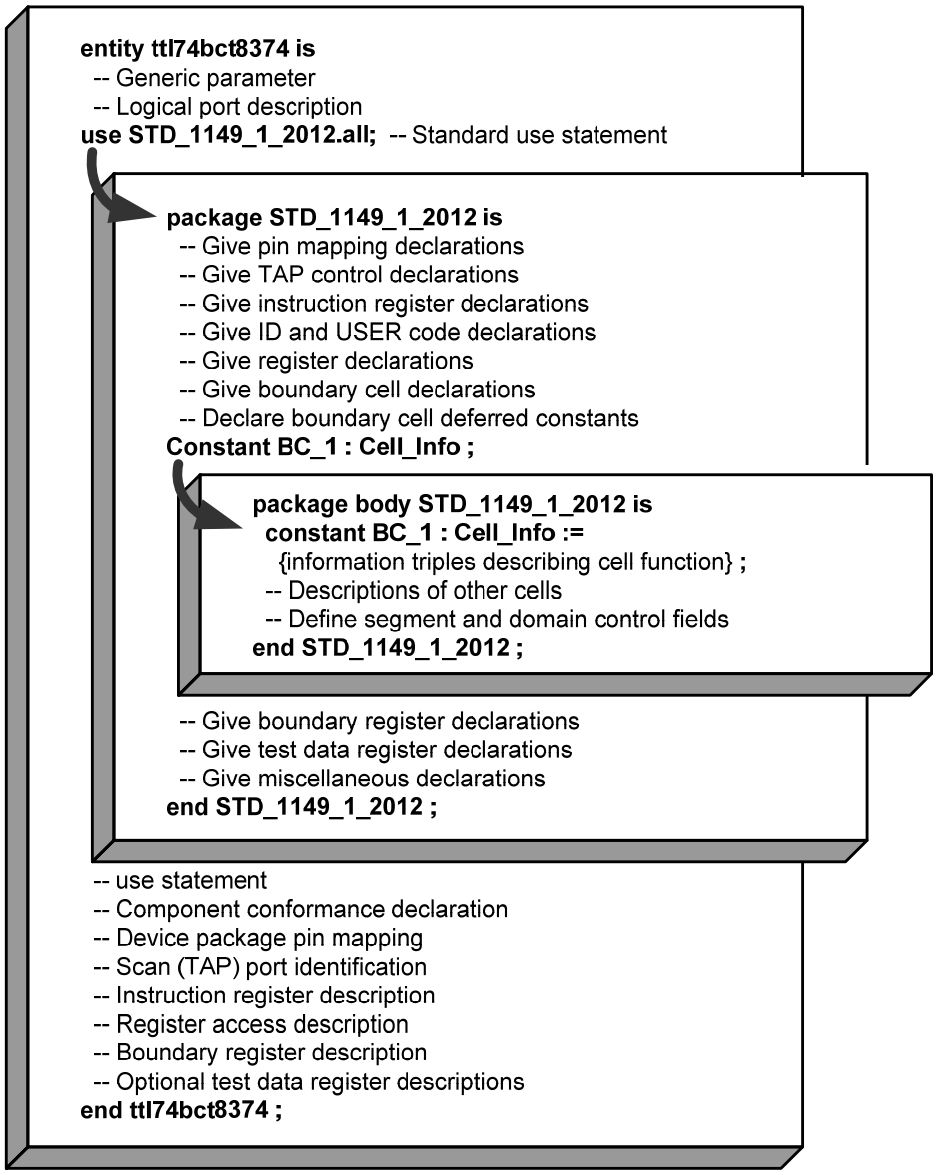


Figure B-1—Components of a BSDL description

A BSDL description is composed as shown in part in Figure B-1. The following rules and permissions define the segments of a BSDL description. (Not all possible BSDL elements are shown in the figure, just the common ones.)

B.7.1 Specifications

Rules

a) *The entity description:* An entity description shall be written for each component and shall specify component-specific parameters of the test.

b) *The Standard BSDL Package and Standard BSDL Package Body*: These shall contain three types of information:

1) The Standard BSDL Package shall give a definition of BSDL statements in terms of VHDL constructs.
2) The Standard BSDL Package Body shall give definitions of commonly used types of boundary-scan register cells.
3) The Standard Package Body shall give definitions of **Register_Mnemonics** and **Register_Fields** used for defining excludable segments and domain control.

Permissions

c) *User-specified BSDL packages and BSDL package bodies:* Users may provide BSDL packages and package bodies that define:
   1) Boundary-scan register cell designs specific to any group of components.
   2) Design-specific test data register descriptions.
d) *The standard BSDL Package Body* may contain test data register descriptions for registers defined in this standard.

B.7.2 Description

Typically, the Standard BSDL Package and Standard BSDL Package Body would reside with system-accompanying software that utilizes BSDL descriptions. Individual BSDL descriptions are not burdened with having to include all the elements contained in the Standard BSDL Package and Standard BSDL Package Body.

NOTE—The Standard BSDL Package and Standard BSDL Package Body are listed in B.9. They are read-only and would normally be included with a BSDL-compliant tool and supplied by the tool supplier. They are not typically supplied along with BSDL files describing a component.

The cell definitions provided in the Standard BSDL Package Body could have been given within the Standard BSDL Package in a full VHDL implementation. The advantage of a package body is that the information it contains can be updated without causing the need for recompilation of all entities that reference the package. If a package is modified, recompilation is necessary. The package with package body structure is a standard practice of BSDL.

User-defined packages may also be provided for a variety of purposes. A vendor of application-specific ICs (ASICs), for example, could provide a user-specified BSDL package and package body to describe the particular boundary-scan register cell designs offered. Standard and design-specific test data registers or register segments may be defined in user-specified BSDL package bodies. Global BSDL extensions also could be provided in user-specified BSDL package bodies (see B.10 and B.8.24).

B.8 Entity description

The entity description and supporting BSDL packages make up a BSDL model of the component and are, in effect, the electronic data sheet for its test logic. It contains statements through which parameters that may vary from one component to another are defined, as discussed in B.3.

B.8.1 Overall syntax of the entity description

B.8.1.1 Specifications

Syntax

<BSDL description>::=
entity <component name> **is**
<generic parameter> (see B.8.2)
<logical port description> (see B.8.3)
<standard use statement> (see B.8.4)
{<use statement>} (see B.8.5)

<component conformance statement> (see B.8.6)
<device package pin mappings> (see B.8.7)
[<grouped port identification>] (see B.8.8)
<scan port identification> (see B.8.9)
[<compliance-enable description>] (see B.8.10)
<instruction register description> (see B.8.11)
[<optional register description>] (see B.8.12)
[<register access description>] (see B.8.13)
<boundary-scan register description> (see B.8.14)
[<runbist description>] (see B.8.15)
[<intest description>] (see B.8.16)
[<system clock description>] (see B.8.17)
{<register mnemonics description>} (see B.8.18)
{<register fields description>} (see B.8.19)
{<register assembly description>} (see B.8.21)
{<register constraints description>} (see B.8.22)
{<register association description>} (see B.8.23)
{<power port association description>} (see B.8.23)
{<BSDL extensions>} (see B.8.24)
[<design warning>] (see B.8.25)
end <component name> <semicolon>

<component name>::= <VHDL identifier>

NOTE 1—While VHDL permits some elements within an entity description to be in an arbitrary order, the fixed ordering above is required for BSDL. This ordering is defined to ease the development of tools that are not themselves required to be fully VHDL compliant.

Rules

a) The <component name> shall identify a particular integrated circuit.
b) Any <component name> referenced in any attribute statement shall be the same as the component name declared in the entity description.

Recommendations

c) The <component name> should contain a string unique to the component owner and a string unique among the components produced by the owner to maximize the probability that it is distinct from the names of all other components that may be used together on a board or in a system.

NOTE 2—Multiple copies of the same component may exist on a board but are given different reference designators.

B.8.2 Generic parameter statement

The <generic parameter> statement facilitates selection between multiple component packaging options that are described within the BSDL description (see B.8.7). Each such option defines a mapping between the physical package pins of the component and the <port ID> elements of the component (see B.6.2).

The component package option relevant to a particular use of a component is identified each time a given BSDL description is referenced.

B.8.2.1 Specifications

Syntax

<generic parameter>::= <generic default> | <generic no default>
<generic default>::= **generic** <left paren> **PHYSICAL_PIN_MAP** <colon>
string <right paren> <semicolon>
<generic no default>::= **generic** <left paren> **PHYSICAL_PIN_MAP** <colon>
string <colon-equal> <default device package type> <right paren> <semicolon>

<default device package type>::= <quote> <pin mapping name> <quote>

Rules

a) If no <default device package type> is specified in the <generic parameter> statement of a BSDL description, a <pin mapping name> (see B.8.7) shall be specified when the BSDL description is processed.
b) The <default device package type> specified in the <generic parameter> statement shall match a <pin mapping name> appearing in some <pin mapping> (see B.8.7).
c) A <pin mapping name> specified when a given BSDL description is processed shall match a <pin mapping name> appearing in some <pin mapping> (see B.8.7) of that BSDL description.

B.8.2.2 Description

In the first alternative <generic parameter> statement syntax, a <pin mapping name> (see B.8.7) that identifies a component package option is supplied with the BSDL description when it is referenced. In the second alternative syntax, a value is given for the <default device package type> that is used as the default in the case in which no <pin mapping name> is supplied with the BSDL description when it is referenced. The <default device package type> is a quoted name of a <pin mapping name> used to specify the pin mapping for the component (see B.8.7).

B.8.2.3 Examples

```
generic (PHYSICAL_PIN_MAP: string);
```
or
```
generic (PHYSICAL_PIN_MAP: string := "DW");
```

NOTE—It is recommended that the component package name from the data sheet of a given component be used as the relevant <pin mapping name>, for example, SSOP_56, PQFP_84, or PGA_18x18.

B.8.3 Logical port description statement

The BSDL port description is a specialization of the VHDL port list. It is used to assign meaningful symbolic names to the pins of a component. These symbolic names, which are referenced in subsequent statements in the description, allow the majority of such statements to be independent of a renumbering or other reorganization of the pins of the component.

B.8.3.1 Specifications

Syntax

<logical port description>::= **port** <left paren> <pin spec>
{ <semicolon> <pin spec> } <right paren> <semicolon>
<pin spec>::= <identifier list> <colon> <pin type> <port dimension>
<identifier list>::= <port name> { <comma> <port name>}

<pin type>::= **in** | **out** | **buffer** | **inout** | **LINKAGE_INOUT** | **LINKAGE_BUFFER** | **LINKAGE_IN** | **LINKAGE_OUT** | **LINKAGE_MECHANICAL** | **POWER_0** | **POWER_POS** | **POWER_NEG** | **VREF_IN** | **VREF_OUT**
<port dimension>::= **bit** | <bit vector spec>
<bit vector spec> ::= **bit_vector** <left paren> <range> <right paren>
<range>::= <up range> | <down range>
<up range> ::= <integer1> **to** <integer2>
<down range> ::= <integer2> **downto** <integer1>
<integer1>::= <integer>
<integer2>::= <integer>

Rules

a) A <range> shall have the value of <integer1> less than or equal to the value of <integer2>.
b) Each <port name> appearing in an <identifier list> of a <logical port description> with a <port dimension> of bit shall occur only once within the <logical port description> statement.
c) Each <port name> appearing more than once in an <identifier list> of a <logical port description>:
   1) Shall have a <port dimension> of **bit_vector**.
   2) The <range> of each appearance shall not overlap with other appearances.
   3) Shall have no gaps in the total sequence of all such appearances.
   4) The <range> of each appearance shall have the same direction (**to** or **downto**).

Permissions

d) Each <port name> appearing more than once in an <identifier list> of a <logical port description> may have a different <pin type> for each subrange listed in the <identifier list> and can occur in any order in the input.

B.8.3.2 Description

The definitions of the possible values of <pin type> are given in Table B-2, and <port name> is defined in B.6.2.

Table B-2—Pin types

Value	Meaning
in	An input-only port that is associated with at least one boundary scan cell unless explicitly exempted in this standard.
out	An output-only port that may be connected to an external bus wire driven by multiple drivers (e.g., a three-state or open-drain output) and that is associated with at least one boundary scan cell.
buffer	A two-state, output-only port where either state is always actively driven (e.g., cannot be connected to an external bus wire) and that is associated with at least one boundary scan cell. See NOTE 1.
inout	A bidirectional port that is associated with at least one boundary scan cell. A bidirectional port is one that is both a system input and a system output.
LINKAGE_OUT	A nonboundary scan analog port designed to source and/or sink current and that has a disable method (not defined or documented in this standard). Normally, the value on this port would be variable. See NOTE 2 and NOTE 3.
LINKAGE_IN	A nonboundary scan analog port that is not designed to source or sink current. See NOTE 2 and NOTE 3.
LINKAGE_INOUT	A nonboundary scan analog bidirectional. See NOTE 2 and NOTE 3.
LINKAGE_BUFFER	A nonboundary scan analog port designed to source and/or sink current and that does not have a disable method. See NOTE 2 and NOTE 3.

Value	Meaning
LINKAGE_MECHANICAL	A nonelectrical port used for positioning, heat sinks, or other nonelectrical use. There is generally no connection to the silicon. See NOTE 2 and NOTE 3.
VREF_IN	A nonboundary scan input reference voltage port, which would normally be a constant value and would have to be driven for the component to function correctly. See NOTE 2 and NOTE 3.
VREF_OUT	A nonboundary scan output reference voltage port. See NOTE 2 and NOTE 3.
POWER_0	A nonboundary scan zero-volt power supply port. These are ports that are normally associated with GROUND. Keyword GROUND or GND is not used here to leave these words for signal names. See NOTE 2 and NOTE 3.
POWER_POS	A nonboundary scan power supply port that receives a constant potential with respect to **POWER_0** that is greater than zero volts. See NOTE 2 and NOTE 3.
POWER_NEG	A nonboundary scan power supply port that receives a constant potential with respect to **POWER_0** that is less than zero volts. See NOTE 2 and NOTE 3.

NOTE 1—Where a two-state output port has a three-state mode only to meet the requirements of the *HIGHZ* instruction, it is still described as a buffer.

NOTE 2—As of the 2013 version of this standard, nondigital ports are no longer allowed to be grouped into the one category of "linkage." This standard now requires the type of linkage port present and classifies reference voltage, power, and ground ports. The improvement in handling of nondigital ports enables board-level ATPG tools to make better decisions about the topology when these ports are defined.

NOTE 3—All pin types described as "nonboundary scan" are not "system pins" subject to the provisioning rules in Clause 11. They can be observed, however, by redundant observe-only boundary cells. See 11.8.

The <port dimension> defines the number of signals that constitute a port. If the <port dimension> is **bit**, one <port name> corresponds to that one signal. If the <port dimension> is **bit_vector**, one <port name> corresponds to a collection of *n* signals that are individually referenced by subscripting the port name, i.e., by a <subscripted port name> (see B.6.2). A <port name> of dimension **bit_vector** may appear more than once in an <identifier list>, but the total range, now specified in pieces, must comprise a contiguous range. Breaking a multibit port into multiple items in an <identifier list> allows a different <pin type> for portions of the total range, such as a multiuse bus that can drive a 12-bit address or drive or receive an 8-bit data value. The high 4 bits could be <pin type> **out**, while the lower 8 bits could be <pin type> **inout**.

NOTE 4—The signal types **bit** and **bit_vector** are signal types known to VHDL and are the only signal types permitted in BSDL. Also, while VHDL allows multiple, possibly broken (i.e., noncontiguous) ranges to be specified for a **bit_vector**, BSDL syntax allows only a single unbroken range, although that range may be split into multiple items in the <identifier list>.

Figure B-2 shows example use of the nonboundary-scan pin types. In the 2001 and earlier versions of this standard all power, ground, and analog ports were grouped as the type **linkage**. This version of the standard requires the IC vendor to classify the ports with more granularity. The purpose is to aid board ATPG tools in improving fault coverage, cross-checking input data such as netlists, and enabling more detailed and accurate fault coverage reports. When the boundary-scan cells of IC2 are bidirectional or self-monitoring outputs, nets A, B, C, and D are automatically tested for shorts because the ATPG tool can read that the ports they are connected to on IC1 are input ports (**LINKAGE_IN**) without requiring additional modeling or manual entry from the board test engineer. (In this case, the designer of IC2 defined his outputs as digital, but the designer of IC1 defined her inputs as analog, which may indicate a high-speed circuit.)

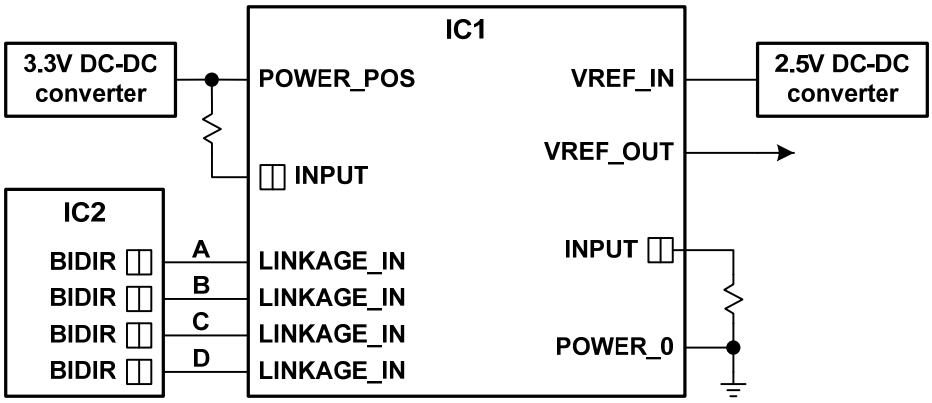


Figure B-2—Example use of nonboundary-scan port-types

Similarly, the pull-up and pull-down on the two **INPUT** ports of IC1 can be identified because the power and ground nets on the board are now made explicit by the **POWER_POS** and **POWER_0** pin types and not all grouped as **linkage**. An expected value can be predicted with high confidence (not all cases) on the **INPUT** port boundary-scan cell (shown in Figure B-2 with the small capture-update box on the two input ports) without the user supplying information about the nature of the nets. If the user does enter information about which nets are power nets, or the ATPG tool uses heuristics to determine the power and ground nets, the **POWER_POS** and **POWER_0** keywords enable the cross-checking of the correctness of these data.

Finally, the new pin types enable better classification of fault coverage from the ATPG process. Nets that are power inputs, for instance, can be placed in a unique class of "potentially untested nets," thereby informing the user to seek additional test methods to maximize the board fault coverage.

Note that on IC1, only the two pins marked **INPUT** are required to have boundary-scan cells but that redundant observe-only cells may be used to observe the other linkage, vref, and power pins. See 11.8.

B.8.3.3 Example

```
port( TDI, TMS, TCK: in bit;
      TDO: out bit;
      IN1, IN2: in bit;
      OUT1: out bit;
      OUT2: buffer bit;
      AOUT : LINKAGE_BUFFER bit;                    -- analog signal out
      OUT3: out bit_vector (1 to 8);
      OUT4: out bit_vector (4 downto 1);
      SERDES1: out bit;
      BIDIR1, BIDIR2, BIDIR3: inout bit;
      SERDES_REF :  VREF_IN bit;
      ANALOG_POWER :  POWER_NEG bit;
      IO_VCC  :   POWER_POS bit;
      GND :  POWER_0 bit;
      CORE :  POWER_POS bit );
```

B.8.4 Standard use statement

The <standard use statement> identifies one Standard BSDL Package in which attributes, types, constants, and other elements are defined, to be referenced elsewhere in the BSDL description.

Note that what, up to the 2001 version of this standard, were known as the Standard VHDL Package and the Standard VHDL Package Body are currently known as the Standard BSDL Package and Standard BSDL Package Body, respectively. This change was made in recognition of the fact that, as of the 2013 version of this standard, the syntax and semantics of BSDL are no longer a subset and standard practice of VHDL.

B.8.4.1 Specifications

Syntax

<standard use statement>::= **use** <standard package name> <period> **all** <semicolon>
<standard package name>::= **STD_1149_1_1990** | **STD_1149_1_1994** |
STD_1149_1_2001 | **STD_1149_1_2013**

Rules

a) The <standard package name> shall be the name of the package inside the file and the name of the file, minus any directory path or extensions, for the Standard BSDL Package and Package Body that contains the information to be included, and the suffix **.all** shall indicate that all declarations within the BSDL package are to be used.
b) The Standard BSDL Package and Standard BSDL Package Body content shall be used when processing a BSDL description.
c) The <standard use statement> also shall be used for version control (see B.8.4.4).
d) The <standard use statement> shall be provided exactly once in each BSDL entity (see B.8.1) and in both the package and package body sections of each user-supplied package (see B.10.1).

B.8.4.2 Description

Additional values for <standard package name> may be assigned with future revisions of this annex as the language evolves. For toolmakers desiring to support the original draft version of BSDL, the value of <standard package name> would be **STD_1149_1_1990** (see B.12.1) for that draft version.

NOTE 1—The construct of the <standard use statement> may not be syntactically complete for use by a given VHDL analyzer. This is because a library or default working area has not been specified. In such a case, a complete statement is "use work.**STD_1149_1_2013.all**;" in which the prefix "work." tells a VHDL analyzer to find the Standard BSDL Package in the current work area. The field "work" could be replaced by an arbitrary library name such as "BSCAN." telling a VHDL analyzer where in its system of libraries to find the Standard BSDL Package. Since there is no standardization of library structures from one VHDL environment to another, some editing of BSDL files to specify the location of Standard BSDL Packages is generally unavoidable if using VHDL parsers to parse BSDL. The specification used in this subclause may cause an error in a VHDL analyzer, forcing the user to edit the BSDL file for the correct location of the Standard BSDL Package information.

The <standard package name> indicates:

— An instruction to tools to read a standard package that contains BSDL syntax definitions.
— The version of the BSDL language (shown by the year number embedded in the identifier) that was used when creating the BSDL.

As a general rule, future enhancements to the BSDL language will be designed as extensions to previous versions of the language. Therefore, the version information contained in the <standard package name> may be used by BSDL-specific tools to indicate which tool versions will be able to process the information in the input BSDL. For example, if the input BSDL records the version as "_1994", tools that can process any language variant "**_1994**" or later will be able to process the input BSDL correctly.

NOTE 2—Versions of the BSDL language should not be confused with the version of this standard to which a given IC may conform. The <component conformance statement> identifies the version of the standard (see B.8.6).

Within a specific system processing BSDL descriptions, the Standard BSDL Package may be a file somewhere in the file system of the host computer. The **.all** suffix is meaningful to VHDL and is not part of a file name. While VHDL permits the use of a wider range of suffixes, **.all** is the only suffix permitted in BSDL.

The content of the Standard BSDL Package is the current definition of the BSDL language and is not intended to be modified by users.

The <standard use statement> appears in every BSDL description before any <use statement> (see B.8.5) so that tools that are fully VHDL compliant can locate information relevant to all components that conform to this standard. (Tools that are limited to the BSDL language always use this information.)

If converting a BSDL originally written using the **STD_1149_1_2001** or earlier version into a BSDL using the **STD_1149_1_2013** version, the syntax and rules of the 2013 version will apply. At least, the following changes would be required:

- Conversion of all LINKAGE ports in the logical port description to the appropriate new types (see B.8.3).
- Addition of appropriate pin types for unconnected pins in the device package pin map (see B.8.7).
- Ensuring that none of the new standard instruction or TDR names are used (see B.5.2).
- Addition of <input spec> to all input ports in the boundary register description (see B.8.14.3.8).
- Changing nonredundant occurrences of the boundary cell function **OBSERVE_ONLY** to **INPUT** [see rule u) of B.8.14.1].

While the new, optional, architectural features, instructions, standard TDRs, and so on cannot be used to describe a component originally designed to conform to the 2001 version of this standard, the new register description attributes may be used to document TDRs in components conforming to earlier versions of this standard, including TDRs with instructions that may have been considered private before, which in turn would enable use of PDL (see Annex C) to document procedures for testing such components.

B.8.4.3 Examples

```
use STD_1149_1_2013.all;  -- Today
      or
use STD_1149_1_2022.all;  -- Sometime in the future
```

The contents of the **STD_1149_1_2013** Standard BSDL Package and its associated Standard BSDL Package Body are listed in B.9. (The contents of the **STD_1149_1_1990** Standard VHDL Package and its associated Standard VHDL Package Body are listed in B.12.1. The contents of the **STD_1149_1_1994** Standard VHDL Package and its associated Standard VHDL Package Body are listed in B.13.1. The contents of the **STD_1149_1_2001** Standard VHDL Package and its associated Standard VHDL Package Body are listed in B.14.1. Due to deviations from VHDL introduced in this version of the standard, the name of the standard Package and Package Body was changed from VHDL to BSDL.)

B.8.4.4 Version control

At the time this standard was approved, there were multiple versions of BSDL descriptions: the 1990 version (see B.12), the 1994 version (approved by the IEEE Standards Board and published by the IEEE in 1994; see B.13), the 2001 version (approved by the IEEE Standards Board and published by the IEEE in 2001; see B.14), and later versions described here. The <standard use statement> indicates whether the BSDL description has been written using the provisional syntax published in the *Proceedings of the International Test Conference* in 1990 or according to this annex. This additional application of the <standard use statement> is intended to provide backward compatibility to all BSDL descriptions already written and can be used in a similar manner for BSDL descriptions based on future revisions of this annex. It is intended that a parser handling a form of BSDL described in a version

of this annex *approved by the IEEE Standards Board and published by the IEEE* handles all forms of BSDL described in previous versions of this annex *approved by the IEEE Standards Board and published by the IEEE*.

In the 1990 version of BSDL, the following syntactic elements are not supported:

<component conformance statement>
<grouped port identification>
<compliance-enable description>
<runbist description>
<intest description>
<BSDL extensions>

Also, in the 1990 version, cell types **BC_0** and **BC_7** as originally identified and defined by the 1994 version need to be specified in a user-supplied BSDL package if they are to be referenced. In the 1990 and 1994 versions, cell types **BC_8**, **BC_9**, and **BC_10,** which are identified and defined in the 2001 and later versions, need to be specified in a user-supplied BSDL package if they are to be referenced. In the 2013 version, cell type **BC_6** is no longer supported and would need to be specified in a user-supplied BSDL package if it is referenced.

In the 1990 and 1994 versions, the identifiers **KEEPER** and **PRELOAD** were not BSDL reserved words. **KEEPER** and **PRELOAD** become BSDL reserved words as of the 2001 version.

In the 1990, 1994, and 2001 versions, the identifiers **LINKAGE_IN**, **LINKAGE_OUT**, **LINKAGE_INOUT**, **LINKAGE_BUFFER**, **LINKAGE_MECHANICAL**, **VREF_IN**, **VREF_OUT**, **POWER_0**, **POWER_POS**, **POWER_NEG**, **TIE0**, **TIE1**, and **OPEN**, as well as new defined instruction names **ECIDCODE**, **INIT_SETUP**, **INIT_SETUP_CLAMP**, **INIT_RUN**, **CLAMP_HOLD**, **CLAMP_RELEASE**, **TMP_STATUS**, and **IC_RESET** and attribute names **BOUNDARY_SEGMENT**, **ASSEMBLED_BOUNDARY_LENGTH**, **SYSCLOCK_ REQUIREMENTS**, **REGISTER_MNEMONICS**, **REGISTER_FIELDS**, **REGISTER_ASSEMBLY**, **REGISTER_CONSTRAINTS, REGISTER_ASSOCIATION**, and **POWER_PORT_ASSOCIATION** were not BSDL reserved words. They become reserved words as of the 2013 version.

Software that processes BSDL descriptions needs to have access to all Standard VHDL or BSDL Packages related to BSDL. See B.9 and B.12 through B.14 for the ones that are currently defined. Obviously**,** a BSDL parser would issue warning or error messages when it encounters unrecognized text.

NOTE—Specific additional attributes may be added via BSDL Extensions (see B.8.24).

B.8.5 Use statement

The optional <use statement> identifies a BSDL package in which specific attributes or constants are defined so that they can be referenced elsewhere in a BSDL description.

B.8.5.1 Specifications

Syntax

<use statement>::= **use** <user package name> <period> **all** <semicolon>
<user package name>::= <VHDL identifier>

Rules

a) Information in a user-specified BSDL package and package body named in a <use statement> shall be used when processing a BSDL description (see B.10).
b) Moved to B.10 in this version of the standard.

c) The <user package name> in the <use statement> shall be the name of the package inside the file and the name of the file not including any directory path or extensions for the BSDL package that contains the information to be included, and the suffix **.all** shall indicate that all declarations within the BSDL package are to be used.

B.8.5.2 Description

The **.all** suffix is meaningful to VHDL and is not part of a file name. While VHDL permits the use of a wider range of suffixes, **.all** is the only suffix permitted in BSDL. See B.10.1 for the content of a user-supplied package.

B.8.5.3 Example

```
use Private_Package.all;    -- Identifies the proprietary BSDL
                            -- package of a user, named Private_Package
```

NOTE—The construct of the <use statement> may not be syntactically complete for use by a given VHDL analyzer. This is because a library or default working area has not been specified. In such a case, a complete statement is "use work.Private_Package.all;" in which the prefix "work." tells a VHDL analyzer to find the user-supplied BSDL package in the current work area. The field "work." could be replaced by an arbitrary library name such as "BSCAN." telling a VHDL analyzer where in its system of libraries to find the user-supplied BSDL package. Since there is no standardization of library structures from one VHDL environment to another, some editing of BSDL files to specify the location of a user-supplied BSDL package is generally unavoidable. This specification may cause an error in a VHDL analyzer, forcing the user to edit the BSDL for the correct location of the user-supplied BSDL package information.

In full VHDL, two or more use statements may be given that reference VHDL packages that contain different definitions of the same item, such as a Register Field definition. Later, the ambiguity of which package is being referenced can be removed by qualifying each reference in a specified manner. This facility is not supported in BSDL for boundary cell definitions, but it is supported for the register descriptions. See the definition and use of <package hierarchy> in B.8.20 and B.8.21.

B.8.6 Component conformance statement

The <component conformance statement> identifies the edition of this standard to which the testability circuitry of a physical component conforms.

It is possible for a component designed in 1990 to be described by the version of BSDL defined by this annex, but this cannot imply that the component conforms to the rules of IEEE Std 1149.1-2013. For example, IEEE Std 1149.1-1990 allowed (by omission of rules) two drivers controlled by a single control cell to be disabled by opposing values loaded into that cell. This is explicitly forbidden starting in IEEE Std 1149.1a-1993. The <component conformance statement> allows tools to account for changes in the rules that may occur in past and future editions of this standard.

B.8.6.1 Specifications

Syntax

<component conformance statement>::= **attribute COMPONENT_CONFORMANCE of**
<component name> <colon> **entity is** <conformance string> <semicolon>
<conformance string>::= <quote> <conformance identification> <quote>
<conformance identification>::= **STD_1149_1_1990** | **STD_1149_1_1993** |
STD_1149_1_2001 | **STD_1149_1_2013**

Rules

a) The reserved words:
   1) **STD_1149_1_1990** shall refer to IEEE Std 1149.1-1990.
   2) **STD_1149_1_1993** shall refer to IEEE Std 1149.1a-1993.
   3) **STD_1149_1_2001** shall refer to IEEE Std 1149.1-2001.
   4) **STD_1149_1_2013** shall refer to IEEE Std 1149.1-2013.

   NOTE—Subsequent editions of this annex may add new values to the <conformance identification> element.

b) When the <conformance identification> is for an earlier version (as indicated by the year number in the version name) of this standard than the <standard package name>, the BSDL shall only document standard and design-specific features of the test logic as defined in the earlier version of this standard.
c) When the <conformance identification> is for an earlier version of this standard than the <standard package name>, the BSDL shall not use any reserved words defined in versions later than the version specified in the <conformance identification>.

B.8.6.2 Description

Some semantic checks described in this annex are influenced by the value that appears in the <conformance identification> element [see, for example, semantic check in rule p) of B.8.14.1].

A component designed in compliance with the 2001 version of this standard will also comply with the 2013 version. One exception would be the use of the **BC_6** boundary cell, which is no longer defined in the 2013 version of the Standard Package and would require a user BSDL package defining the cell in order to maintain compliance. Thus, if it is desired to document such a component using the 2013 version of BSDL, it is reasonable, although not required, to set the component conformance to 2013 as well.

B.8.6.3 Example

```
attribute COMPONENT_CONFORMANCE of My_Old_IC:entity is
      "STD_1149_1_1990";      -- 1990 component described
                              -- in this annex
```

B.8.7 Device package pin mappings

The mapping of logical signals onto the physical pins of a particular component package is defined through use of a <device package pin mappings> attribute statement and an associated BSDL string.

B.8.7.1 Specifications

Syntax

<device package pin mappings>::= <pin map statement> <pin mappings>
<pin map statement>::= **attribute PIN_MAP of** <component name> <colon> **entity is**
PHYSICAL_PIN_MAP <semicolon>
<pin mappings>::= <pin mapping> { <pin mapping> }
<pin mapping>::= **constant** <pin mapping name> <colon> **PIN_MAP_STRING :=**
<map string> <semicolon>
<pin mapping name>::= <VHDL identifier>
<map string>::= <quote> <port map> { <comma> <port map> } <quote>
<port map>::= <port name> <colon> <pin or list>
<pin or list>::= <pin desc> | <pin list>
<pin list> ::= <left paren> <pin desc> { <comma> <pin desc>} <right paren>

<pin desc> ::= <pin ID> | **OPEN** | **TIE0** | **TIE1**
<pin ID>::= <VHDL identifier> | <integer>

Rules

a) Within a given <pin mapping>, each <pin ID> shall appear only once.
b) All ports in the <logical port description> of a given BSDL description shall be referenced in each <pin mapping> of that description, and vice versa.
c) The <port map> for a <port name> defined as **bit** in the <logical port description> shall be a single <pin desc>.
d) The <port map> for a <port name> defined as a **bit_vector** in the <logical port description> shall be a <pin list> with an ordered list of <pin desc> elements, each associated with one bit of the <port name> in the order of the bits in the <range> of the <port name>.
e) Each <pin mapping name> shall be unique within a <pin mappings>.
f) Any <port name> element in a <port mapping> element of a given BSDL description shall appear in an <identifier list> element of the <logical port description> statement of the description.
g) No subscripting of <port name> shall be allowed; only the base name of a port shall appear if the port was described to be a **bit_vector**.
h) A <pin desc> of **OPEN** shall imply that the <port ID> is not electrically connected to a package pin, and the associated boundary-scan register cells shall capture the same value as if the <port ID> were unconnected on the board.
i) A <pin desc> of **TIE0** or **TIE1** shall imply that the <port ID> is not electrically connected to a package pin but is forced to the specified value either on the integrated circuit or in the package, and the associated boundary-scan register cells shall capture the specified tie value.
j) A <pin desc> of **TIE0** or **TIE1** shall only appear on a <port name> associated with a <pin type> of **in, LINKAGE_IN, POWER_0**, **POWER_POS,** or **VREF_IN** in the <logical port description>.
k) No <pin desc> shall have a value of **OPEN**, **TIE0**, or **TIE1** if the <portID> is physically connected to a package pin.
l) No <pin desc> shall have a value of **OPEN**, **TIE0**, or **TIE1** if the <portID> :
    1) Appears in a <TCK stmt>, <TDI stmt>, <TMS stmt>, or <TDO stmt> (see B.8.9).
    2) Appears in a <grouped port identification> (see B.8.8), unless both the <representative port> and the <associated port> have a <pin desc> of **OPEN**, or one has a <pin desc> of **TIE0** and the other has a <pin desc> of **TIE1.**
m) No <pin desc> shall have a value of **OPEN**, **TIE0**, or **TIE1** if the <portID> appears in a <TRST stmt> (see B.8.9), unless an on-chip POR circuit (see 6.1.3) is provided; in which case, only the values of **OPEN** or **TIE1** shall be allowed.
n) If the <port ID> appears in the <compliance port list> (see B.8.10) or is otherwise required for initialization or proper operation of test or system logic, then that <port ID> shall not have a <pin desc> of **OPEN**, and if the <pin desc> is **TIE0**, or **TIE1**, the implied values shall be a compliant or enabling value.

Permissions

o) A <pin desc> for a single <portID> may have one of the values **OPEN**, **TIE0**, or **TIE1** in all <pin mapping> statements.

NOTE—This allows a reduced pin version of a product to be introduced without the full package version.

B.8.7.2 Examples

```
attribute PIN_MAP of ttl74bct8374: entity is PHYSICAL_PIN_MAP;
constant DW:PIN_MAP_STRING:=
      "CLK:1, Q:(2,3,4,5,7,8,9,10), " &
```

```
      "D:(23,22,21,20,19,17,16,15), " &
      "GND1:6, VCC1:18, GND2:open, VCC2:open, OC_NEG:24, " &
      "TDO:11, TMS:12, TCK:13, TDI:14";
constant FK:PIN_MAP_STRING:=
      "CLK:9, Q:(10,11,12,13,16,17,18,19), " &
      "D:(6,5,4,3,2,1,26,25), " &
      "GND1:15, VCC1:8, GND2:14, VCC2:22, OC_NEG:7, " &
      "TDO:20, TMS:21, TCK:23, TDI:24";
constant SM:PIN_MAP_STRING:=
      "CLK:9, Q:(open,open,open,open,16,17,18,19), " &
      "D:(tie0,tie0,tie0,tie0,2,1,26,25), " &
      "GND1:15, VCC1:8, GND2:open, VCC2:open, OC_NEG:7, " &
      "TDO:20, TMS:21, TCK:23, TDI:24";
constant DIE_BOND:PIN_MAP_STRING:=
      "CLK:Pad01, " &
      "Q:(Pad02,Pad03,Pad04,Pad05,Pad06,Pad07,Pad08,Pad09), " &
      "D:(Pad10,Pad11,Pad12,Pad13,Pad14,Pad15,Pad16,Pad17), " &
      "GND1:Pad18, VCC1:Pad19, GND2:Pad20, VCC2:Pad21, " &
      "OC_NEG:Pad22, " &
      "TDO:Pad23, TMS:Pad24, TCK:Pad25, TDI:Pad26";
```

NOTE—Revisions of this standard after IEEE Std 1149.1-2001 support naming no-connect signals with a pin labeled as **OPEN**, **TIE0**, or **TIE1**. Package variation SM illustrates using **TIE0** and **OPEN** to support a smaller physical package where only the four pads representing the upper bits of D and Q and a single set of power and ground pins are bonded out.

B.8.7.3 Description

Attribute **PIN_MAP** is a string that is set to the value of the parameter **PHYSICAL_PIN_MAP**, which is defined by the generic statement. VHDL constants are then declared, one for each packaging variation. In the example description, four packaging variations exist: `DW`, `FK`, `SM`, and `DIE_BOND`.

The constants identify component packages that are typified for BSDL purposes by the mapping between logical port names and the physical pins of a component. A BSDL parser looks for the constant with a name matching the value of **PIN_MAP**. The standard practice for BSDL mandates that the type of the constant is **PIN_MAP_STRING**.

The following is an example of a <map string>:

```
"CLK:1,Q:(2,3,4,5,7,8,9,10), D:(23,22,21,20,19,17,16,15), " &
"GND:6, VCC:18, OC_NEG:24, TDO:11, TMS:12, TCK:13, TDI:14"
```

Notice that this is the concatenation of two smaller strings. A BSDL parser reads the contents of the string. It matches signal names, such as `CLK`, with the names in the port definition.

For a given <port map>, the <pin list> identifies the physical pin (or set of physical pins) associated with the port called <port name>. A <pin desc> is either a <pin ID>, a physical pin, or one of the **OPEN**, **TIE0**, or **TIE1** keywords indicating the port name is not connected to a package pin in this particular pin map. A <pin ID> may be a number or an alphanumeric identifier because some component packages, such as pin-grid arrays (PGAs), use coordinate identifiers, such as A07 or H13. Note, however, that names such as 7A and 13H are non-compliant since they are not valid VHDL identifiers. Obviously, **OPEN**, **TIE0**, and **TIE1** are reserved words and cannot be used as a <pin ID>.

If signals such as that having <port name> `Q` in the example pin map are identified as **bit_vector** in the <logical port description>, there must be a one-to-one mapping between the members of that **bit_vector** and the members of the <pin list> associated with `Q`.

The ordering of items in the <pin list> is significant, and the ordering provides correlation between a given <subscripted port name> and its associated <pin desc>.

For example, if `Q` were defined to have members `"1 to 8"`, the physical pin mapped onto port `Q(1)` in the example (for the `DW` component package) would be pin `2`, and `Q(2)` would be pin `3`, and so on. If `Q` were defined to have members `"8 downto 1"`, `Q(1)` would be pin `10` and `Q(2)` would be pin `9`, and so on. Nonbonded pads indicated by **OPEN**, **TIE0**, or **TIE1** keywords also provide position place holders in providing the correlation between a given <subscripted port name> and its associated keyword.

Of the four mappings shown in the example, the first three are for packaging variations of a finished IC and the fourth shows a die-bond mapping for a finished bare die that might be used in a "Multi-Chip Module."

There may be different numbers of linkage and power ports among packaging variants for a component. In the given example, the `FK` package had two more power/ground pins than the `DW` package. All ports `GND1`, `GND2`, `VCC1`, and `VCC2` must have appeared in the port definition. It is required that all physical linkage and power pins be included in a <pin mapping>.

Figure B-3 illustrates the intent of the three values for unconnected pins.

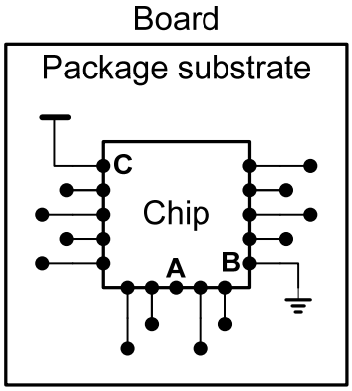


Figure B-3—Example of unconnected pin types

In Figure B-3, there are three ports with <port ID> of "A," "B," and "C," respectively. <port ID> "A" has no electrical connection to the package substrate, so it would have the keyword **OPEN** in the package pin map and the boundary-scan register cells associated with this <port ID> would capture exactly the same thing as they would capture if there were a connection to the board, but there was no electrical connection on the board. <port ID> "B" is electrically connected to the package substrate, but it is tied to ground in the package and not electrically connected to a package pin to the board. It would be coded as **TIE0** in the package pin map, and associated boundary-scan register cells would observe a 0 on the port. Similarly, <port ID> "C" is electrically connected to the package substrate where it is tied to Vdd in the package and is not connected to a package pin to the board. It would be coded as **TIE1** in the package pin map, and associated boundary-scan register cells would observe a 1 on the port. More complex electrical connections in the package cannot be represented in the package pin map.

B.8.8 Grouped port identification

The optional <grouped port identification> is used to identify system I/O signals that have the special characteristic of using more than one pin to carry a bit of data and for which the allowed states of the pins are restricted. Typically, these are differential pairs of signals operating in either the voltage or the current domain. The syntax shown in B.8.8.1 allows for future expansion if it is later determined that the description of signal groups containing more than two signals is important.

In the case of differential pairs where one signal is always the complement of the other (a state restriction), there is a "Plus" pin (i.e., representative port) and a "Minus" pin (i.e., associated port). This standard states that a differential pair may need to be treated as an analog circuit with a single boundary-scan register cell providing or receiving data (see Figure 11-7 and Figure 11-9). However, due to the prevalence of differential signaling and the fact that digital data are indeed being transmitted, it is desirable to accomplish boundary-scan interconnect testing on each pin of a differential signal pair. The grouped port identification is used to describe such a situation to test generation software.

Rule i) of 11.5.1 and rule n) of 11.6.1 mandate that the data provided by or captured by a boundary-scan register cell have the same polarity as the data bit transmitted by the associated I/O pin. Clearly, the "Minus" signal cannot do this since it always transmits the complement of the "Plus" signal. The "Plus" signal can be identified to software so that it can be directly associated with the required boundary-scan register cell. Then the "Minus" signal is linked to the "Plus" signal to indicate that a pairing exists and that the "Minus" signal is *not* associated with a required cell in the boundary-scan register. (The "Minus" signal can, however, be associated with a redundant observe-only cell in the boundary-scan register. See 11.8.)

Ports have a <grouped port identification> when the state restrictions apply during boundary-scan operation, e.g., during *EXTEST*. If the ports are restricted during normal system operation but *not* during boundary-scan operation, then <grouped port identification> will lead software to produce incomplete results, e.g., an inadequate board interconnect test. Therefore, such ports should not be listed in the **PORT_GROUPING** attribute.

B.8.8.1 Specifications

Syntax

<grouped port identification> ::= **attribute PORT_GROUPING of**
<component name> <colon> **entity is** <group table string> <semicolon>
<group table string>::= <quote> <group table> <quote>
<group table>::= <twin group entry> { <comma> <twin group entry> }
<twin group entry>::= <twin group type> <left paren> <twin group list> <right paren>
<twin group type>::= **DIFFERENTIAL_VOLTAGE** | **DIFFERENTIAL_CURRENT**
<twin group list>::= <twin group> { <comma> <twin group> }
<twin group>::= <left paren> <representative port> <comma> <associated port> <right paren>
<representative port>::= <port ID>
<associated port>::= <port ID>

Rules

a) Any <port ID> used in a <grouped port identification> shall have been declared in the <logical port description> statement with a <pin type> of **in**, **out**, **buffer**, or **inout**.

b) Removed in this version of the standard; see B.6.2.

c) Any <port ID> appearing as a <representative port> shall also appear as a <port ID> in a <cell spec> in the subsequent <boundary-scan register description> (see B.8.14).

d) Any <port ID> appearing as an <associated port> *shall not* appear as a <port ID> in a <cell spec> in the subsequent <boundary-scan register description> (see B.8.14) unless the <function> of the <cell entry> is **OBSERVE_ONLY**.

NOTE—This is an exception to semantic check s) of B.8.14.1.

e) The two ports listed in a <twin group> shall have the same pin type (see Table B-2).

f) Removed in this version of the standard; see B.6.2.

g) The <representative port> pin shall be the positive ("Plus") pin of a differential pair, and the <associated port> shall be the negative ("Minus") pin of a differential pair.

B.8.8.2 Description

Software must determine how to handle **DIFFERENTIAL_CURRENT** signals that are directly accessible to tester resources. The **DIFFERENTIAL_VOLTAGE** signals typically would be physically similar to other logic signals being tested. These two keywords help to inform software as to the physical nature of the way data are transmitted.

B.8.8.3 Examples

This example includes a <boundary-scan register description> (see B.8.14) for purposes of illustrating certain semantic relationships.

```
entity diff is
generic(PHYSICAL_PIN_MAP:string:= "Pack");

port (CLK:in bit;
      D_Pos:in bit_vector(1 to 4);
      D_Neg:in bit_vector(1 to 4);
      Q_Pos:out bit_vector(1 to 4);
      Q_Neg:out bit_vector(1 to 4);
      GND : POWER_0 bit;
      VCC : POWER_POS bit;
      OC_NEG : in bit;
      TDO : out bit; TMS, TDI, TCK, TRST : in bit);

-- Get IEEE Std 1149.1-2013 attributes/definitions

use STD_1149_1_2013.all;

attribute COMPONENT_CONFORMANCE of diff : entity is "STD_1149_1_2013";

attribute PIN_MAP of diff : entity is PHYSICAL_PIN_MAP;

constant PACK:PIN_MAP_STRING:="CLK:1, " &
      "Q_Pos:(2,3,4,5), " &
      "Q_Neg:(7,8,9,10), " &
      "D_Pos:(23,22,21,20), " &
      "D_Neg:(19,17,16,15), " &
      "GND:6, VCC:18, OC_NEG:24, " &
      "TDO:11, TMS:12, TCK:13, TDI:14";

attribute PORT_GROUPING of diff : entity is
      "Differential_Voltage ((Q_Pos(1),Q_Neg(1)),"& -- Voltage signals
            "(Q_Pos(2), Q_Neg(2)),"&
            "(Q_Pos(3), Q_Neg(3)),"&
            "(Q_Pos(4), Q_Neg(4))),"&
      "Differential_Current ((D_Pos(1),D_Neg(1)),"& -- Current signals
            "(D_Pos(2), D_Neg(2)),"&
            "(D_Pos(3), D_Neg(3)),"&
            "(D_Pos(4), D_Neg(4)))";
      (... some BSDL deleted for brevity ...)

attribute BOUNDARY_REGISTER of diff : entity is
-- num cell port     function safe [input/ccell disval rslt]
   "9 (BC_1, CLK,      input,   X,   OPENX)," &
   "8 (BC_1, OC_NEG,   input,   X,   OPEN1)," & -- Merged input/control
   "8 (BC_1, *,        control, 1)," &          -- Merged input/control
```

```
    "7 (BC_1, D_Pos(1), input,   X,   PULL0)," &
    "6 (BC_1, D_Pos(2), input,   X,   PULL0)," &
    "5 (BC_1, D_Pos(3), input,   X,   PULL0)," &
    "4 (BC_1, D_Pos(4), input,   X,   PULL0)," &
    "3 (BC_1, Q_Pos(1), output3, X,   8,       1, PULL0),"& -- Also bussable
    "2 (BC_1, Q_Pos(2), output3, X,   8,       1, PULL0),"&
    "1 (BC_1, Q_Pos(3), output3, X,   8,       1, PULL0),"&
    "0 (BC_1, Q_Pos(4), output3, X,   8,       1, PULL0)";

end diff;
```

Software processing this BSDL description example is able to identify four pairs of voltage-differential pins and four pairs of current-differential pins. Each differential signal is separated into positive and negative pins, with the positive pins associated with cells in the boundary-scan register.

B.8.9 Scan port identification

The scan port identification statements identify the TAP of the component.

B.8.9.1 Specifications

Syntax

<scan port identification>::= <scan port stmt> { <scan port stmt> }
<scan port stmt>::= <TCK stmt> | <TDI stmt> | <TMS stmt> | <TDO stmt> | <TRST stmt>
<TCK stmt>::= **attribute TAP_SCAN_CLOCK of** <port ID> <colon> **signal is**
<left paren> <clock record> <right paren> <semicolon>
<TDI stmt>::= **attribute TAP_SCAN_IN of** <port ID> <colon> **signal is true** <semicolon>
<TMS stmt>::= **attribute TAP_SCAN_MODE of** <port ID> <colon> **signal is true** <semicolon>
<TDO stmt>::= **attribute TAP_SCAN_OUT of** <port ID> <colon> **signal is true** <semicolon>
<TRST stmt>::= **attribute TAP_SCAN_RESET of** <port ID> <colon> **signal is true** <semicolon>
<clock record>::= <real> <comma> <halt state value>
<halt state value>::= **LOW** | **BOTH**

Rules

a) The <port ID> occurring in a <TDO stmt> shall have a <pin type> of **out** in the <logical port description> statement.
b) The <port ID> occurring in a <TCK stmt>, a <TDI stmt>, a <TMS stmt>, or a <TRST stmt> shall have a <pin type> of **in** in the <logical port description> statement.
c) No <port ID> in the <scan port identification> shall later appear in the <boundary-scan register description> (see B.8.14).

NOTE 1—This is an exception to semantic check s) of B.8.14.1.

d) A given <port ID> shall occur at most once in the <scan port identification>.
e) No value of a <port ID> element in the <scan port identification> shall have appeared as a <representative port> or as an <associated port> in a <twin group> (see B.8.8).
f) The <TCK stmt>, <TMS stmt>, <TDI stmt>, and <TDO stmt> shall appear exactly once in a <scan port identification>.
g) The <TRST stmt> shall either not appear, or shall appear exactly once in the <scan port identification>.

NOTE 2—These rules assure that exactly four or five statements, in any order, are found in the <scan port identification>.

B.8.9.2 Description

The statements identify specific logical signals of the component as being signals of the TAP. A <clock record> is a pair consisting of:

— A real number that gives the maximum operating frequency for TCK in hertz. In the following example, 20.0 MHz is specified as the maximum operating frequency.
— A VHDL type that has one of two values—**LOW** and **BOTH**—which specifies the state(s) in which the TCK signal may be stopped without causing loss of data held in the test logic. **BOTH** indicates that the clock can be stopped in either state. Components that allow TCK to be stopped only in the high state do not conform to this standard.

Examples

```
attribute TAP_SCAN_IN of TDI : signal is true;
attribute TAP_SCAN_OUT of TDO : signal is true;
attribute TAP_SCAN_MODE of TMS : signal is true;
attribute TAP_SCAN_RESET of TRST : signal is true;
attribute TAP_SCAN_CLOCK of TCK : signal is (20.0e6, BOTH);
```

Signal names `TDI`, `TDO`, `TMS`, `TRST`, and `TCK` are those that were used in the <logical port description> in the above example (see B.8.8.3). The names used in the above examples are those defined in this standard; however, arbitrary names could have been used.

NOTE—The function of each signal is specified by the attribute name rather than by the value of a <port ID> element in the <scan port identification>.

B.8.10 Compliance-enable description

This portion of a BSDL description appears in the description of a component if the optional compliance-enable feature described by this standard (see 4.8) has been implemented in that component. Otherwise, improper operation of the part may occur during an automatically generated test.

B.8.10.1 Specifications

Syntax

<compliance-enable description> ::= **attribute COMPLIANCE_PATTERNS of**
<component name> <colon> **entity is** <compliance pattern string> <semicolon>
<compliance pattern string>::= <quote> <left paren> <compliance port list> <right paren>
<left paren> <pattern list> <right paren> <quote>
<compliance port list>::= <port ID> { <comma> <port ID> }
<pattern list>::= <pattern> { <comma> <pattern> }

NOTE 1—A number of <pattern> elements may be specified, reflecting the fact that there may be multiple combinations of bits that enable compliance. For convenience, a <pattern> may contain X bits to reduce the size of a <pattern list>. As an example, the <pattern list> 1111X, 0XXX0 specifies 10 unique bit patterns that enable compliance.

Rules

a) The <port ID> elements in a <compliance port list> shall be positionally associated with the bits in a <pattern>.

NOTE 2—Thus, the first appearing <port ID> is associated with the first (leftmost) bit in <pattern> and so on.

b) The number of <port ID> elements in the <compliance port list> shall be equal to the number of bits in each <pattern> within the <pattern list>.

c) No <port ID> value shall occur more than once in the <compliance port list>.

d) No <port ID> value in the <compliance port list> shall also appear as a <port ID> value in the <scan port identification> (see B.8.9).

e) Each <port ID> in the <compliance port list> shall have a <pin type> value in the <logical port description> of **in**.

f) A <port ID> value in the <compliance port list> shall not appear in the <boundary-scan register description> (see B.8.14) unless the <function> of the <cell entry> associated with the <port ID> is **OBSERVE_ONLY.**

NOTE 3—This is an exception to semantic check s) of B.8.14.1.

g) Removed in this version of the standard; see B.6.2.

h) No <port ID> value in the <compliance port list> shall appear as a <port ID> in the <grouped port identification> (see B.8.8).

Permissions

i) A <port ID> value in the <compliance port list> may appear in the <boundary-scan register description> (see B.8.14) if the <function> of the <cell entry> associated with the <port ID> is **OBSERVE_ONLY.**

NOTE 4—This is an exception to semantic check s) of B.8.14.1.

B.8.10.2 Description

The following example is given for a component that implements an LSSD test mode as well as a standard boundary scan. When the LSSD test mode is used, the test logic defined by this standard becomes configured as a part of a scan path such that LSSD techniques can be used to verify the operation of the complete component. Thus, the LSSD test mode takes priority over the IEEE 1149.1 test mode, and the control pins are compliance-enable pins.

B.8.10.3 Examples

```
attribute COMPLIANCE_PATTERNS of LSSD_IC: entity is
      "(LSSD_A, LSSD_B, LSSD_P, LSSD_C1, LSSD_C2) (00011)";
```

Software that generates tests using test facilities defined by this standard must assure that any compliance-enable conditions are first set up before exercising the TAP of the affected IC. Any compliance-enable pattern (provided as a <pattern> within the <pattern list>) must be held constant for the duration of all boundary-scan testing.

B.8.11 Instruction register description

The next segment of the BSDL description concerns the component-dependent characteristics of the instruction register. The details specified to characterize the implementation of the instruction register in a particular component are:

— *Length.* The instruction register will be at least two bits long. Its length is not otherwise limited.

— *Instructions.* The register is required to support certain instructions. A designer may add any or all of the optional instructions defined by this standard and/or any number of design-specific instructions. Also, this standard provides for private instructions, which may be marked as such to warn applications not to use them in order to prevent unsafe or undocumented behavior.

— *Instruction binary codes (opcodes).* The *BYPASS* instruction is decoded from a bit pattern fixed by this standard [see rule b) of 8.4.1]. Bit patterns for other instructions are be specified by the test logic designer. Each instruction may be decoded from several bit patterns.

— *Instruction capture.* On passing through the *Capture-IR* controller state, the instruction register will load data from its parallel input. In some register stages, certain fixed values are required, but in other register stages, design-dependent values may be loaded.

BSDL provides a means of describing these characteristics and takes advantage of opportunities for semantic checks, thus, verifying that the component is in compliance with this standard (that is, it has implemented the required instruction binary codes properly).

The characteristics of the instruction register that are specified using BSDL are its length, the opcodes, the pattern captured in the *Capture-IR* controller state, and whether any given instruction is public or private.

B.8.11.1 Specifications

Syntax

<instruction register description>::=
<instruction length stmt>
<instruction opcode stmt>
<instruction capture stmt>
[<instruction private stmt>]

<instruction length stmt>::= **attribute INSTRUCTION_LENGTH of** <component name>
<colon> **entity is** <integer> <semicolon>
<instruction opcode stmt>::= **attribute INSTRUCTION_OPCODE of** <component name>
<colon> **entity is** <opcode table string> <semicolon>
<instruction capture stmt>::= **attribute INSTRUCTION_CAPTURE of** <component name>
<colon> **entity is** <pattern list string> <semicolon>
<instruction private stmt>::= **attribute INSTRUCTION_PRIVATE of** <component name>
<colon> **entity is** <instruction list string> <semicolon>
<opcode table string>::= <quote> <opcode description> { <comma> <opcode description> } <quote>
<opcode description>::= <instruction name> <left paren> <opcode list> <right paren>
<opcode list>::= <opcode> { <comma> <opcode> }
<opcode> ::= <pattern>
<pattern list string>::= <quote> <opcode list> <quote>
<instruction list string>::= <quote> <instruction list> <quote>
<instruction list>::= <instruction name> { <comma> <instruction name> }

Rules

a) The integer value of the attribute **INSTRUCTION_LENGTH** shall be greater than or equal to 2, and this value shall be interpreted as the length of the instruction register.

b) All <opcode> elements shall have length equal to that of the instruction register.

c) The <opcode> element having a value of all 1s shall decode to *BYPASS* and shall not be defined explicitly for any other instruction.

d) Where the value of <conformance identification> is not **STD_1149_1_1990 or STD_1149_1_1993**, an <opcode> element for *EXTEST* shall be defined, and the <opcode description> in which it is defined shall have **EXTEST** as the value of its <instruction name> element.

NOTE 1—Where a device conforms to the 2001 edition or later of this standard, as indicated by a <conformance identification> value of, the all 0s opcode is not mandated for *EXTEST*. Therefore, an opcode bit pattern for *EXTEST* is required to be explicitly defined. Furthermore, if the all 0s opcode is not otherwise assigned, it decodes to *BYPASS*.

e) Where the value of <conformance identification> is **STD_1149_1_1990** or **STD_1149_1_1993**, the <opcode> element made up of all 0s shall decode to *EXTEST* and shall not be defined explicitly for any other instruction.

NOTE 2—Where a device conforms to earlier editions of this standard, as indicated by a <conformance identification> value of **STD_1149_1_1990** or **STD_1149_1_1993**, the all 0s opcode is mandated for *EXTEST* and so it is explicitly defined as such or presumed by implication.

f) Where the value of <conformance identification> is not **STD_1149_1_1990** or **STD_1149_1_1993**, an opcode for *SAMPLE* shall be defined, and the <opcode description> in which it is defined shall have **SAMPLE** as the value of its <instruction name> element.

g) Where the value of <conformance identification> is not **STD_1149_1_1990** or **STD_1149_1_1993**, an opcode for *PRELOAD* shall be defined, and the <opcode description> in which it is defined shall have **PRELOAD** as the value of its <instruction name> element.

h) Where the value of <conformance identification> is **STD_1149_1_1990** or **STD_1149_1_1993**, an opcode for *SAMPLE/PRELOAD* shall be defined, as follows:

   1) At least one <opcode description> shall have **SAMPLE** as the value of its <instruction name> element.

   2) Any <opcode description> that has **PRELOAD** as the value of its <instruction name> element shall contain only such values for <pattern> as have been defined for any <opcode description> that fulfills the requirement for rule h1).

NOTE 3—Where a device conforms to earlier editions of this standard, as indicated by a <conformance identification> value of **STD_1149_1_1990** or **STD_1149_1_1993**, the functions of *SAMPLE* and *PRELOAD* were required to be implemented in a merged fashion, *SAMPLE/PRELOAD*. Therefore, every opcode that is defined for *SAMPLE* is implicitly defined for *PRELOAD*. Furthermore, where an opcode is defined for *PRELOAD*, it must have a binary value that matches one defined for *SAMPLE*.

i) Where permission h) of 8.1.1 is met, the length of the selected registers shall be identical for each of the instructions that share the same opcode.

j) Where permission h) of 8.1.1 is not met, opcodes containing **X** bits shall not be ambiguous (i.e., decodable as two or more different instructions); that is, if there are two <opcode description> elements and two <pattern> elements such that an **X** appears in each of the two <opcode description> elements, the two <pattern> elements shall differ in some character position in which *neither* pattern contains the character **X**.

k) The <pattern> value in the <instruction capture stmt> shall have a length equal to that of the instruction register, and the two least significant bits of this <pattern> shall be **01**.

l) Only design-specific instructions shall be defined as private.

m) Any <instruction name> appearing in an <instruction list> shall appear only once in the <instruction list> and shall also appear in the <opcode table string>.

n) Where the value of <conformance identification> is not **STD_1149_1_1990**, **STD_1149_1_1993**, or **STD_1149_1_2001**, and the <opcode table string> contains an instruction name of **CLAMP_HOLD**, **CLAMP_RELEASE**, or **TMP_STATUS**, the <opcode table string> shall contain all three instruction names.

o) Where the value of <conformance identification> is not **STD_1149_1_1990**, **STD_1149_1_1993**, or **STD_1149_1_2001**, and the <opcode table string> contains an instruction name of **INIT_SETUP** or **INIT_SETUP_CLAMP**, the <opcode table string> shall contain both instruction names.

B.8.11.2 Description

To summarize, the purpose of each attribute is as follows:

INSTRUCTION_LENGTH: The <instruction length stmt> defines the length of the instruction register and, hence, the number of bits that each opcode pattern must contain in subsequent statements of the <instruction register description>.

INSTRUCTION_OPCODE: The <instruction opcode stmt> is a BSDL string containing instruction identifiers and their associated bit patterns. The rightmost bit in the pattern is that closest to TDO (i.e., that shifted in first). Each <opcode description> is such a pair. This standard mandates the existence of *BYPASS*, *SAMPLE*,

PRELOAD, and *EXTEST* instructions, with a mandatory bit pattern for *BYPASS*. Decoding of bit patterns that are not explicitly listed must default to the *BYPASS* instruction [see rule d) of 8.1.1].

INSTRUCTION_CAPTURE: The <instruction capture stmt> specifies the bit pattern that is loaded into the instruction register when the TAP controller passes through the *Capture-IR* state. This bit pattern is shifted out whenever a new instruction is shifted in. This standard mandates that the two least significant bits must be 01. The remainder of this bit pattern is design specific, and the presence of X bits indicates bits that are not deterministic.

INSTRUCTION_PRIVATE: The optional <instruction private stmt> identifies instructions that are private and potentially unsafe for use by other than the manufacturer of the component. By definition, the effects of these instructions are undefined to the general public; their use should be avoided. Note that failure to follow warnings about private instructions can result in damage to the component, circuit board, or system.

B.8.11.3 Examples

```
attribute INSTRUCTION_LENGTH of My_IC:    -- Must be first
            entity is 4;
attribute INSTRUCTION_OPCODE of My_IC:    -- Must be second
            entity is
      "EXTEST (0011), " &
      "EXTEST (1011), " &
      "BYPASS (1111), " &
      "SAMPLE (0001, 1000), " &
      "PRELOAD(1001, 1000)," &
      "HIGHZ  (0101), " &
      "SECRET (1010) ";
attribute INSTRUCTION_CAPTURE of My_IC:   -- Must be third
            entity is "0001";
attribute INSTRUCTION_PRIVATE of My_IC:   -- Optional
            entity is "Secret";
```

NOTE 1—In the above example, *BYPASS* was shown to be decoded from 1111. Because this is the mandatory pattern specified by this standard, its expression is redundant and not required. In addition to any explicitly assigned patterns, all unassigned patterns will also be decoded as *BYPASS* [see rule d) of 8.1.1].

NOTE 2—For devices designed to conform to editions of this standard before IEEE Std 1149.1-2001, *EXTEST* had a mandatory pattern of all zeros (for example, “0000”) and so, where the value of <conformance identification> in a given BSDL description is **STD_1149_1_1990** or **STD_1149_1_1993**, the expression of this mandatory decode is redundant and not required. For devices designed to conform to all later editions of this standard, *EXTEST* has no mandatory bit pattern and so, where the value of <conformance identification> in a given BSDL description is not **STD_1149_1_1990 or STD_1149_1_1993**, **EXTEST** and its bit pattern must occur in the <instruction opcode stmt>.

NOTE 3—Also, notice that **EXTEST** is given on two lines with two decodes. This shows that multiple lines may be used for each instruction if needed (for instance, when this attribute is written by a computer program).

NOTE 4—As would be required in the case that the <conformance identification> were **STD_1149_1_2001, STD_1149_1_2013**, or later, the above example illustrates the specification of a pattern for **PRELOAD**. It also illustrates options for *SAMPLE* and *PRELOAD* instructions both where the same pattern is used for both and where unique patterns are used for each. It should be further noted that where a pattern for *SAMPLE* is the same as a pattern for *PRELOAD*, all rules for both instructions must be met. This combination is equivalent to the *SAMPLE/PRELOAD* instruction in previous editions of this standard.

B.8.12 Optional device register description

This clause defines which components are documented by the BSDL by specifying one or more bit pattern values returned in response to selection of the optional device identification register instructions—*IDCODE* and, if

implemented, *USERCODE*. If the value returned matches a value documented by these attributes, then the BSDL describes that component.

B.8.12.1 Specifications

Syntax

<optional register description> ::= <optional register stmt> [<optional register stmt>]
<optional register stmt> ::= <idcode statement> | <usercode statement>
<idcode statement>::= **attribute IDCODE_REGISTER of** <component name>
<colon> **entity is** <quote> <32-bit pattern list> <quote> <semicolon>
<usercode statement>::= **attribute USERCODE_REGISTER of** <component name>
<colon> **entity is** <quote> <32-bit pattern list> <quote> <semicolon>
<32-bit pattern list> ::= <32-bit pattern> { <comma> <32-bit pattern> }

Rules

a) If a device identification register is specified by inclusion of an <idcode statement>, the least significant bit in any <32-bit pattern> element within the <idcode statement> shall be **1**.
b) A single <idcode statement> shall appear in a BSDL description if and only if **IDCODE** appears as the value of an <instruction name> element in an <opcode description> of the <instruction opcode stmt> (see B.8.11).
c) A single <usercode statement> shall appear in a BSDL description if and only if both **IDCODE** and **USERCODE** appear as the value of <instruction name> elements in an <opcode description> of the <instruction opcode stmt> (see B.8.11).
d) A bit pattern in the manufacturer code shall conform to the specifications in 12.2.1.

Recommendations

e) When an <idcode statement> lists more than one <32-bit pattern>, the first <32-bit pattern> in the list should be the value of the most recent change and should not include the character **X**.

Permissions

f) An <idcode statement> may list every device ID code for which the BSDL is applicable.
g) A <usercode statement> may list every device user code for which the BSDL is applicable.

NOTE—Such lists create a one-to-many relationship between the BSDL and the components it describes. There is no implication that a single component may return multiple ID code values.

B.8.12.2 Description

Only one **IDCODE_REGISTER** attribute and one **USERCODE_REGISTER** attribute may appear in a single BSDL, although order is not important. The coding of **IDCODE_REGISTER** values is defined in 12.2.1, and not repeated here. If **USERCODE_REGISTER** appears, then **IDCODE_REGISTER** must also appear, as the **USERCODE_REGISTER** defines multiple ways of programming a single type of programmable component.

It may be desirable to include in a single BSDL a list of all of the device identification register and usercode register values for which a particular BSDL is valid. The alternative is having multiple BSDL files that differ only by the value of the device identification register and/or usercode register.

For a new device, the manufacturer provides a new BSDL for the device. For later versions of the device, if the previous BSDL is otherwise applicable, the manufacturer may modify the existing BSDL to include a new device identification register pattern, or may issue a new BSDL. In some cases, an X in the version or part-number codes

will be sufficient to specify all allowable codes. In other cases, a comma-separated list of values will be required to describe all applicable device identification register patterns for which the BSDL is applicable.

B.8.12.3 Examples

```
attribute IDCODE_REGISTER of My_IC: entity is
      "0011" &                  -- Version
      "1111000011110000" &      -- Part number
      "00001010100" &           -- Identity of the manufacturer
      "1";                      -- Required by IEEE STD 1149.1-1990
```

In the above example, only one device ID is listed in the BSDL.

```
attribute IDCODE_REGISTER of My_IC: entity is

     -- latest version
      "0100" &                  -- Version code 0100
      "1111000011110000" &      -- Part number
      "00001010100" &           -- Identity of the manufacturer
      "1," &                    -- Required by IEEE STD 1149.1

     -- older versions
      "001X" &                  -- Version codes 0011 and 0010
      "1111000011110000" &      -- Part number
      "00001010100" &           -- Identity of the manufacturer
      "1";                      -- Required by IEEE STD 1149.1
```

In the above example, devices with device ID versions 0100, 0011, and 0010 are compatible with the BSDL, and version 0100 is the most recent.

```
attribute USERCODE_REGISTER of My_IC: entity is
      "10XX" & "0011" & "1100" & "1111" & -- Start 1st 32-bit pattern
      "0000" & "0000" & "0000" & "1111,"& -- End 1st 32-bit pattern

      "111X" & "0011" & "1001" & "1000" & -- Start 2nd 32-bit pattern
      "0000" & "0100" & "1001" & "1000";  -- End 2nd 32-bit pattern
```

In the above example, two user codes are described. In all of these examples, concatenation is used to delimit fields within the codes.

The <idcode statement> and <usercode statement> define one or more specific components documented by the BSDL. When the codes are read from components on a board, the value read is compared to the values supplied by these attributes to help verify that the correct BSDL is being used for that component on the board. This allows a single BSDL to document more than one component, and all components documented by the BSDL may be specified using either multiple patterns or **X** bits within a pattern.

For the IDcode register, it is also possible that a single component type may be sourced from different manufacturers with different manufacturer fields or that a component may be released in different versions from the same manufacturer, all without needing a different BSDL.

An **X** can also be used to mask subfields within a code that are not important for testing purposes; in this case, **X** specifies a “don’t care” position in the pattern.

B.8.13 Register access description

All instructions place a test data register between TDI and TDO. Design-specific instructions may access test data registers mandated by this standard or design-specific registers. This standard allows a designer to place additional test data registers, referenced by design-specific instructions, in the component.

It is important for test development software to know of the existence and length of all public test data registers and the names of their associated instructions.

B.8.13.1 Specifications

Syntax

<register access description> ::= **attribute REGISTER_ACCESS of**
<component name> <colon> **entity is** <register access string> <semicolon>
<register access string>::= <quote> <register association> { <comma> <register association> } <quote>
<register association>::= <register> <left paren> <instruction capture list> <right paren>
<instruction capture list>::= <instruction capture> { <comma> <instruction capture> }
<instruction capture>::= <instruction name> [**CAPTURES** <pattern>]
<register>::= <std fixed register> | <std var register> | <design specific register>
<std fixed register>::= **BOUNDARY** | **BYPASS** | **DEVICE_ID** | **TMP_STATUS**
<std var register>::= <std var reg name> [<left bracket> <reg length> <right bracket>]
<std var reg name>::= **ECID** | **INIT_DATA** | **INIT_STATUS** | **RESET_SELECT**
<design specific register>::= <VHDL identifier> [<left bracket> <reg length> <right bracket>]
<reg length>::= <integer> | <asterisk>

Rules

a) The association of the *BYPASS*, *CLAMP*, *EXTEST*, *HIGHZ*, *IDCODE*, *INTEST*, *PRELOAD*, *SAMPLE*, *USERCODE, CLAMP_HOLD, CLAMP_RELEASE,* and *TMP_STATUS* instructions with registers is mandated by this standard, and those registers either have lengths that are fixed and mandated by this standard or lengths that are defined elsewhere in BSDL. Descriptions of these assignments (a <register association>) are redundant and not needed in BSDL, but if such descriptions are given:
   1) They shall be checked against the mandatory assignment specified in this standard, and an error is issued if they are not correct.
   2) They shall not have a "**CAPTURES** <pattern>" element in their description. These capture data are specified either in this standard or elsewhere in BSDL.
   3) They shall not have a register length specification in their description.

b) A <register association> shall be given for the *ECIDCODE, IC_RESET, INIT_SETUP, INIT_SETUP_CLAMP,* and *INIT_RUN* instructions, and:
   1) They shall be checked against the mandatory register assignment specified in this standard, and an error is issued if they are not correct.
   2) They shall not have a "**CAPTURES** <pattern>" element in their description. These capture data are specified either in this standard or elsewhere in BSDL.
   3) They shall have a register length specification in their description.

   NOTE 1—The association of registers with the *ECIDCODE, IC_RESET, INIT_SETUP, INIT_SETUP_CLAMP,* and *INIT_RUN* instructions is mandated by this standard, but the length is not mandated.

c) Any public instruction (an instruction whose name appears in the <instruction opcode stmt> but does not appear in the <instruction private stmt> [see B.8.11]) not listed in rule a) or rule b) in this subclause shall have an associated test data register defined.

d) The <reg length> of each <std var register> or <design specific register> shall be specified on the first appearance in a <register association>, and shall be greater than 0 or the deferred value of asterisk (*);

furthermore, if the <register> appears in more than one <register association>, subsequent appearances shall either not define the length again or define it identically.

e) All instructions, the names of which appear as the value of an <instruction name> element in any <instruction capture> element of the <register access string>, shall appear as an <instruction name> element in an <opcode description> element in the <instruction opcode stmt> (see B.8.11).

f) Any <instruction name> element shall appear in only one <instruction capture list> within the <register access string>.

g) The <pattern> value in an <instruction capture> element shall contain the same number of bits as the <register> in the same <register association>.

h) When a design-specific <register> is defined by either a **REGISTER_FIELDS** or a **REGISTER_ASSEMBLY** attribute, the <reg length> for that register in the **REGISTER_ACCESS** attribute shall have a deferred value of <asterisk> (*), and a "**CAPTURES** <pattern>" element shall not be included.

NOTE 2—Both the length and the capture value, if any, are defined in the **REGISTER_FIELDS** and **REGISTER_ASSEMBLY** attributes. See B.8.19, B.8.20, and B.8.21.

i) When a design-specific <register> is implemented as a variable-length register, the <reg length> for that register in the **REGISTER_ACCESS** attribute shall be either the length after the TAP controller is reset, as specified in 6.1.3.1, or a deferred value of <asterisk> (*).

NOTE 3—The reset of the TAP controller by either a power-up signal or the TRST* TAP pin will force both the Reset* and CHReset* signals as well.

B.8.13.2 Examples

```
attribute REGISTER_ACCESS of ttl74bct8374: entity is
      "BOUNDARY (READBN, READBT, CELLTST), " &
      "BYPASS (TOPHIP, SETBYP, RUNT, TRIBYP), " &
      "MEMBIST[*] (MBIST)," &
      "BCR[2] (SCANCN, SCANCT CAPTURES 0X)";
```

B.8.13.3 Description

In this example, `READBN`, `READBT`, `CELLTST`, `TOPHIP`, `SETBYP`, `RUNT`, `TRIBYP`, `MBIST`, `SCANCN`, and `SCANCT` must have been defined in the <instruction opcode stmt> (see B.8.11). The first three instructions select the boundary-scan register, while the next four instructions select the bypass register. The `MBIST` instruction selects the design-specific `MEMBIST` register for which the length and possible capture value are deferred until defined by either a **REGISTER_FIELDS** or a **REGISTER_ASSEMBLY** attribute. The last two instructions (`SCANCN` and `SCANCT`) select a two-bit design-specific register called `BCR`. The `SCANCT` instruction also shows a capture value `0X` that will be loaded into `BCR` when passing through the *Capture-DR* controller state. No capture value is specified for the `SCANCN` instruction.

By identifying the association between instructions and test data registers, the length of the test data register scan sequence can be determined for a given instruction. The lengths and associations to standard instructions of the mandatory boundary-scan register, bypass register, and instruction register, as well as the optional device identification register, are fixed by the standard or known from other BSDL statements. Instructions defined by this standard that select predefined registers with a fixed length are not required to be listed in the **REGISTER_ACCESS** description. Each instruction must have an associated test data register except for private instructions (see B.8.11), for which the identification of the selected register is optional.

The specification of the length of a register may be deferred by using the asterisk character (*) in place of a numeric length. In that case, the register must be defined in either a **REGISTER_FIELDS** or a **REGISTER_ASSEMBLY** attribute (see B.8.19 and B.8.21, respectively).

As of the 2013 version of this standard, variable-length registers are permitted. If such registers are associated with a public instruction, then the default length after a reset must be defined.

Note that this standard allows user instructions to reference several registers at once if they are concatenated [see permission h) of 9.2.1]. In BSDL, the logical register resulting from such a concatenation is treated as if it were a separate register with a distinct name and length.

B.8.14 Boundary-scan register description

The boundary-scan register description may be defined either of two ways. It can consist of a single list of boundary-scan register cells numbered 0 to RegLength-1 (where the total number of cells in the boundary-scan register is RegLength), or it can consist of several boundary-scan register segment descriptions, each a list of boundary-scan register cells numbered 0 to SegLength-1 (where the total number of cells in the segment is SegLength). The cells may be listed in any order, but all must be defined. Cell 0 is closest to TDO.

The boundary-scan register cells can vary in design and purpose. Clause 11 shows many example cell designs, but many others are possible under the rules of this standard.

When there are several boundary-scan register segments defined, a **REGISTER_ASSEMBLY** statement (see B.8.19) is used to define the construction of the full boundary-scan register, including segment-select or domain-control cells (see 9.4), which are placed before the segment they control.

The list of cells is the same regardless of whether the cell description is part of a full boundary-scan register description or a boundary-scan register segment description.

The characteristics of each cell design used in a component are specified before the cells can be referenced in the <boundary-scan register description>. For the example cell designs included in this standard, cell descriptions are contained in the Standard BSDL Package **STD_1149_1_2013** (see B.9). Cells defined in this BSDL package are referenced through a simple set of names listed in Table B-3.

Table B-3—List of cells defined in the Standard BSDL Package and relevant figure numbers

Name	Figures[a]	Comments
BC_0	Special cell	Degenerate form[b]
BC_1	Figure 11-19, Figure 11-31, Figure 11-35c, Figure 11-35d, Figure 11-37c, Figure 11-47d	Design usable for many functions
BC_2	Figure 11-15, Figure 11-32, Figure 11-36c, Figure 11-36d, Figure 11-38c, Figure 11-39c, Figure 11-40(output), Figure 11-42c	*INTEST* unsupported on Output2
BC_3	Figure 11-16	Input or Internal only
BC_4	Figure 11-17, Figure 11-18, Figure 11-40(input)	Input, Observe_Only, Clock, or Internal only
BC_5	Figure 11-47c	Combined Input/Control
BC_7	Figure 11-38d	Bidirectional
BC_8	Figure 11-41, Figure 11-42d	Simpler bidirectional, lacking *INTEST* support; captures the signal at the corresponding pin even while operating in output mode
BC_9	Figure 11-33	Output that captures the signal at the corresponding pin for *EXTEST* and captures the signal driven from the system logic for *INTEST* and *SAMPLE*
BC_10	Figure 11-34	Simpler output, lacking *INTEST* support; always captures the signal at the corresponding pin instead of the signal driven from the system logic

[a] The suffix "c" is used to denote a control cell shown in a cited figure. The suffix "d" denotes a data cell.

[b] **BC_0** is a cell that captures the value specified by the rules of this standard and that captures a don't-care value whenever this standard allows options. It can be used whenever there is uncertainty about the exact behavior of a compliant cell.

The method of describing cells other than those depicted in this standard is described in B.10. When such cell designs are to be used, their descriptions must be given in a user-supplied BSDL package and BSDL package body.

Several rules must be observed when combining cells to create a boundary-scan register conformant to this standard. Adherence to some rules can be checked during processing of the BSDL description of a component. For example, some cell designs may be used only on a component input. Some will not support the *INTEST* instruction—this is allowable if **INTEST** does not appear in the <instruction opcode stmt> (see B.8.11). Some cells require the aid of another to control three-state enables or the direction of signal flow.

A very general cell design from this standard (see Figure 11-19 and Figure 11-31) is shown in Figure B-4. Figure B-5 shows a symbolic representation of the same cell design.

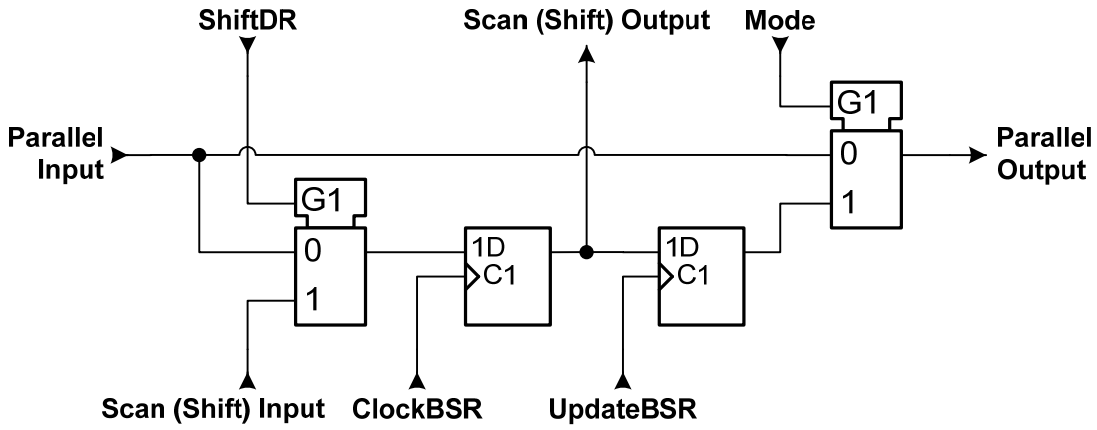


Figure B-4—Cell design corresponding to Figure 11-19 and Figure 11-31

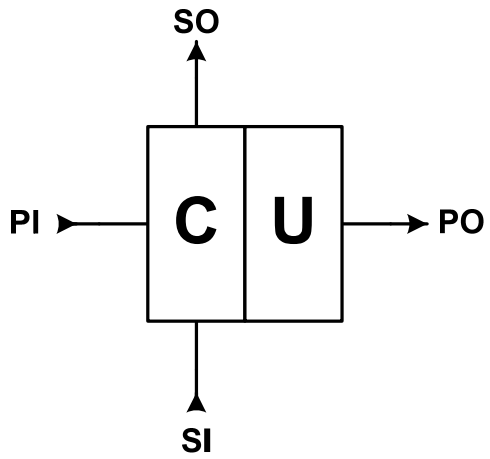


Figure B-5—Symbolic representation of a boundary-scan register cell

The design in Figure B-4 consists of a parallel input (PI), a parallel output (PO), a multiplexer controlled by a mode signal, and two flip-flops. The mode signal is a function of the current instruction. A serial input (SI) and a serial output (SO) form the shift path through the cell. The mode signal is a logic 0 or 1 that tells a cell what test function to perform (see Table 11-3). Note that the symbolic representation does not include:

— The multiplexer controlled by signal *Shift-DR*

— The mode signal and multiplexer
— The clock signals, *Clock-DR*, and *Update-DR*

These parts of the cell design do not need to be considered in BSDL because the control and operation of boundary-scan register cells are fully defined by this standard. Thus, Figure B-5 is a full representation of the cell design shown in Figure B-4 for the purposes of BSDL. The parallel input and output are shown in the figure, and they are connected to various places depending on the application. The two flip-flops are labeled C (for Capture) and U (for Update) to represent their uses. The C flip-flop captures data from the system data input of the cell in the *Capture-DR* controller state and lies on the shift register path. The U flip-flop loads data from the C flip-flop in the *Update-DR* controller state. The shift path is shown because many such cells will be linked together in a shift chain that makes up the boundary-scan register. The shift path links only the C flip-flops.

One cell design, shown in Figure 11-18 in this standard, is a cell with observe capability and no control capability. It has a symbol without a U flip-flop (Figure B-6). This cell can be used at a system input pin, and it has the advantage of a lower propagation delay in some implementations, or as a redundant observer on a system output pin. However, it does not support the optional *INTEST* instruction at a nonclock input.

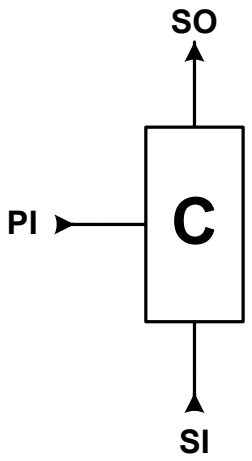


Figure B-6—Symbolic representation of a boundary-scan register cell without an update stage

B.8.14.1 Specifications

Syntax

<boundary-scan register description> ::= <fixed boundary stmts> | <segment boundary stmts>
<fixed boundary stmts> ::= <boundary length stmt> <boundary register stmt>
<segment boundary stmts> ::= <assembled boundary length stmt> <boundary register segments>

<boundary length stmt> ::= **attribute BOUNDARY_LENGTH of**
<component name> <colon> **entity is** <register length> <semicolon>
<register length> ::= <integer>

<boundary register stmt> ::= **attribute BOUNDARY_REGISTER** of
<component name> <colon> **entity is** <cell table string> <semicolon>
<cell table string> ::= <quote> <cell table> <quote>

<assembled boundary length stmt> ::= **attribute ASSEMBLED_BOUNDARY_LENGTH of**
<component name> <colon> **entity is**
<left paren> <reset length> <comma> <register length> <right paren> <semicolon>
<reset length> ::= <integer>

<boundary register segments> ::= <boundary register segment> { <boundary register segment> }
<boundary register segment> ::= **attribute BOUNDARY_SEGMENT** of
<component name> <colon> **entity is** <boundary segment string> <semicolon>
<boundary segment string>::= <quote> <boundary segment list>
{ <comma> <boundary segment list> } <quote>
<boundary segment list>::= <boundary segment name>
<left bracket> <boundary segment length> <right bracket>
<left paren> <cell table> <right paren>
<boundary segment name> **::=** <VHDL identifier>
<boundary segment length> ::= <integer>

<cell table> ::= <cell entry> { <comma> <cell entry> }
<cell entry> ::= <cell number> <left paren> <cell info> <right paren>
<cell number> ::= <integer>
<cell info> ::= <cell spec> [<comma> <input or disable spec>]
<cell spec> ::= <cell name> <comma> <port ID or null> <comma> <function> <comma> <safe bit>
<cell name> ::= <VHDL identifier>
<port ID or null> ::= <port ID> | <asterisk>
<function> ::= **INPUT** | **OUTPUT2** | **OUTPUT3** | **CONTROL** |
CONTROLR | **INTERNAL** | **CLOCK** | **BIDIR** | **OBSERVE_ONLY**
<safe bit>::= **0** | **1** | **X**
<input or disable spec> ::= <input spec> | <disable spec>
<input spec> :: = **EXTERN0** | **EXTERN1** | **PULL0** | **PULL1** | **OPEN0** | **OPEN1** | **KEEPER** |
OPENX | **EXPECT1** | **EXPECT0**
<disable spec>::= <ccell> <comma> <disable value> <comma> <disable result>
<ccell>::= <integer>
<disable value>::= **0** | **1**
<disable result>::= **WEAK0** | **WEAK1** | **PULL0** | **PULL1** | **OPEN0** | **OPEN1** | **KEEPER** | **Z**

Rules

a) The value of the integers <register length>, <boundary segment length>, and <reset length> shall be greater than zero.

b) For every <cell entry> element, the <cell number> element and the optional <ccell> element of the <disable spec> element shall have a value in the range from 0 to <register length> minus 1 for a <cell table> in a <cell table string>, or <boundary segment length> minus 1 for a <cell table> in a <boundary segment list>.

c) Every <integer> with a value from 0 to <register length> minus 1 or <boundary segment length> minus 1, as applicable, shall appear as a <cell number> in some <cell entry> of the <cell table>.

d) Only a pair of merged cells (see B.11.2.3) shall correspond to two <cell entry> elements containing identical <cell number> elements in the <cell table>; moreover:

1) The only possible mergers shall be of cells with <function> equal to **INPUT** and cells with <function> equal to **OUTPUT2**, **OUTPUT3**, **CONTROL**, or **CONTROLR**.
2) The value of the <cell name> element in both <cell entry> elements shall be equal.
3) The <safe bit> values for these two cells shall be identical unless one value is **X**.
4) The <data source> values of the <capture descriptor> values (see B.10.1) for these two cells shall be identical for all supported instructions.

e) Every <cell name> appearing in the <cell table> shall be the name of a cell described in either the Standard BSDL Package or a user-supplied BSDL package.

f) While <cell entry> elements may be listed in the <cell table> in any order of <cell number>, the order of scan of boundary-scan cells shall be in numerical order of the <cell number>, with the <cell number> value of 0 being closest to TDO.

g) Removed in this version of the standard; see B.6.2.

h) A <port ID or null> shall have the value asterisk (*****) when, in the same <cell spec> element, the <function> element has the value **CONTROL**, **CONTROLR**, or **INTERNAL**.

i) Any <cell entry> element containing a <function> element equal to **INPUT**, **CONTROL**, **CONTROLR**, **INTERNAL**, **OBSERVE_ONLY**, or **CLOCK** shall not also contain a <disable spec> element, and any <cell entry> element containing a <function> equal to **CONTROL**, **CONTROLR**, **INTERNAL**, **OUTPUT2**, **OUTPUT3**, or **BIDIR** shall not also contain an <input spec> element.

j) Any <cell entry> element containing a <function> element equal to **OUTPUT3** or **BIDIR** also shall contain a <disable spec> element, and any <cell entry> element containing a <function> element equal to **INPUT** or **CLOCK** shall also contain an <input spec> element.

NOTE 1—It is possible for two or more <cell entry> elements with <function> **INPUT** (or **CLOCK**) to have the same <port ID>. These would normally have the same <input spec> value, but there may be situations such as two mutually exclusive I/O macros, which connect to the same pin, where the entries could have different <input spec> values.

k) Any <cell entry> element containing a <function> element equal to **OUTPUT2** and containing a <disable spec> element shall satisfy the following conditions:

1) The value of <cell number> shall equal the value of <ccell>. (This implies that an **OUTPUT2** cell may control itself.)

2) The value of the <disable value> shall be equivalent to the (weak) logical value of the <disable result>; that is, when the <disable value> is **0**, the <disable result> shall be **WEAK0** or **PULL0**, and when the <disable value> is **1**, the <disable result> shall be **WEAK1** or **PULL1**.

l) Any <cell entry> element containing a <function> element equal to **BIDIR** and a <ccell> element value equal to the value of the <cell number> element (implying a bidirectional cell that controls itself) shall have the value of the <disable value> equivalent to the (weak) logical value of the <disable result> element; that is, when the <disable value> is **0**, the <disable result> shall be **WEAK0** or **PULL0**, and when the <disable value> is **1**, the <disable result> shall be **WEAK1** or **PULL1**.

m) For any <cell entry> element containing a <function> element equal to **CONTROL** or **CONTROLR**, the value of the <cell number> element of that <cell entry> shall appear as the value of the <ccell> element of the <disable spec> element of some other <cell entry> elements.

n) For any <cell entry> containing a <function> element equal to **CONTROL** or **CONTROLR**, the value of the <safe bit> element of that <cell entry> shall be equal to the value of the <disable value> element of the <disable spec> element of the other <cell entry> elements that satisfy semantic check m) of this subclause (i.e., the controlled cells).

o) The <ccell> element of a <disable spec> element shall have only the values permitted under the conditions of semantic check k) through semantic check m) of this subclause.

p) When the value of <conformance identification> (see B.8.6) is not **STD_1149_1_1990**, and two distinct <disable spec> elements in the <cell table> have <ccell> elements with a common value, the values of the <disable value> elements of these two <disable spec> elements shall also be equal.

NOTE 2—Starting with the 1993 revision, a single control cell is not allowed to enable some drivers while simultaneously disabling others.

q) Removed in this version of the standard.

r) Removed in this version of the standard.

s) Excepting those elements explicitly mentioned in the following list, all <port ID> elements having a <pin type> value of **in**, **out**, **buffer**, or **inout** in the <logical port description> (see B.8.3) shall appear as <port ID> elements in the <boundary register stmt> or in the assembled boundary-scan register with all excludable segments included; specifically exempted from this check are any <port ID> elements satisfying any of the following conditions:

1) Rule d) of B.8.8.1 (grouped ports)

2) Rule c) of B.8.9.1 (scan port identification)

3) Rule f) of B.8.10.1 (compliance-enable description)

NOTE 3—This semantic check means all nonexempt system pins must be associated with cell(s) in the boundary-scan register. This semantic check also means that all <scan port identification> pins must not be associated with cells in the boundary-scan register. <Grouped ports> and <compliance-enable descriptions> may appear in the boundary-scan register description.

NOTE 4—Semantic check cc) and semantic check s) of this subclause state which <port ID> elements in the <logical port description> must appear in the <boundary register stmt>, and vice versa. The next semantic checks state the properties that must exist for <function> elements within <cell entry> elements.

t) Moved to Permissions in this version of the standard [see permission bb) in this subclause].

u) For any <port ID> element that is not an <associated port> or an element of a <compliance port list> or a <port ID> element in a <scan port stmt> appearing in one or more <cell entry> element of the <boundary register stmt>, when the <pin type> in the <pin spec> of that <port ID> is:

1) **in**, the <function> of at least one <cell entry> shall be **INPUT** or **CLOCK** unless the <port ID> is a <representative port>; in that case there shall be one and only one <cell entry> with a <function> **INPUT** or **CLOCK**.

NOTE 5—The cell of a <representative port> described with <function> **INPUT** or **CLOCK** is the cell on the single ended output of a differential receiver. Additional cells on a <representative port> are described with <function> **OBSERVE_ONLY**.

2) **out**, the <function> of exactly one <cell entry> shall be **OUTPUT2** or **OUTPUT3**; furthermore, when the <function> is **OUTPUT2**, the <cell entry> shall have a <disable spec> according to semantic check k) of this subclause.

3) **buffer**, the <function> of exactly one <cell entry> shall be **OUTPUT2**, and the <cell entry> shall not contain a <disable spec>**.**

4) **inout**, the <function> of the <cell entry> shall be **BIDIR**, **OUTPUT2**, **OUTPUT3**, or **INPUT**; furthermore, if the <function> value is **BIDIR**, no other <cell entry> containing the same <port ID> shall have the <function> **BIDIR**, **OUTPUT2**, or **OUTPUT3**; furthermore, if the <function> of the <cell entry> is **OUTPUT2** or **OUTPUT3**, no other <cell entry> containing the same <port ID> shall exist with the <function> value of **BIDIR**, **OUTPUT2**, or **OUTPUT3**, *and* at least one other <cell entry> containing the same <port ID> but a different <cell number> shall exist with the <function> value of **INPUT**; furthermore, if the <function> of the <cell entry> is **INPUT,** one other <cell entry> containing the same <port ID> but a different <cell number> shall exist with the <function> value of **OUTPUT2** or **OUTPUT3.**

NOTE 6—Additional **OBSERVE_ONLY** cells may monitor the state of any I/O pin other than TAP scan pins.

v) The <function> in a <cell entry> shall be an existing <cell context> (see B.10.1) within the <capture descriptor> of the cell named by <cell name>.

w) If **INTEST** occurs as the value of an <instruction name> element in an <opcode description> element of the <instruction opcode stmt>, then, for each <port ID> element satisfying semantic check s) in this subclause, a <cell entry> shall exist that references that <port ID> and that possesses *INTEST* support capability.

NOTE 7—For this semantic check, a given <cell entry> does not possess *INTEST* support capability unless the <capture descriptor list> (see B.10.1) of the cell design named by the <cell name> element meets one of the following conditions:

— For a <function> element value of **INPUT**, **CLOCK**, **OUTPUT2**, **OUTPUT3**, **CONTROL**, or **CONTROLR**, a <capture descriptor> element contains a <cell context> element value that matches the <function> element value *and* has a <capture instruction> element value of **INTEST**.

— For a <function> element value of **BIDIR**, one <capture descriptor> element contains a <cell context> element value of **BIDIR_IN** *and* another <capture descriptor> element contains a <cell context> element value of **BIDIR_OUT**, *and* both such <capture descriptor> elements have a <capture instruction> element value of **INTEST**.

NOTE 8—For this semantic check, a given <cell entry> does not possess *INTEST* support capability if its <function> element value is **OBSERVE_ONLY**. An **OBSERVE_ONLY** cell cannot provide *INTEST* support capability.

x) The <disable result> values of **OPEN0** and **OPEN1** shall only be used in a <cell entry> element containing a <function> element equal to **BIDIR** and a <ccell> element value *not* equal to the value of the <cell number> element (the bidirectional cell does not control itself) and shall describe the behavior of the input path associated with the **BIDIR** when the <disable result> of the driver would otherwise be **Z**.

y) When <segment boundary stmts> are used to define boundary-scan register segments, each <boundary segment name>:
   1) Shall be unique within the BSDL.
   2) There shall be a <register assembly description> (see B.8.19) defining the **BOUNDARY** register.
   3) Every <boundary register segment> shall appear exactly once as a <boundary instance> in the <register assembly description>.

z) When the boundary-scan register is assembled from segments, then in the <assembled boundary length stmt>, the <reset length> shall be the minimum length of the register with all excludable segments excluded, and the <register length> shall be the length with all excludable segments included.

NOTE 9—The <register length> documents the total number of cells in the boundary register, not the maximum achievable length, which could be less due to mutually exclusive excludable segments.

aa) When the value of <conformance identification> is **STD_1149_1_2001, STD_1149_1_1993** or **STD_1149_1_1990**, the attributes **BOUNDARY_SEGMENT** and **ASSEMBLED_BOUNDARY_LENGTH** shall not appear in the BSDL.

Permissions

bb) Except TAP pins, all <port ID> elements having a <pin type> value of **in**, **out**, **buffer**, or **inout** in the <logical port description> (see B.8.3) may appear as <port ID> elements with a <function> value of **OBSERVE_ONLY** in the <boundary register stmt> or in the assembled boundary-scan register with all excludable segments included.

NOTE 10—A <port ID> required to appear in the <boundary register statement> by rule s) in this subclause may additionally appear as allowed by this permission.

cc) If a <port ID> has a <pin type> value beginning with the word **LINKAGE_** (except **LINKAGE_MECHANICAL**), **POWER_**, or **VREF_**, the given <port ID> may appear in the <boundary register stmt> in a <cell entry> only with a <function> value of **OBSERVE_ONLY** and an <input spec> value of either **EXPECT0** or **EXPECT1**. (See B.8.14.3.8 for the definition of **EXPECT0** and **EXPECT1**.)

B.8.14.2 Examples

Example 1

The syntax of BSDL requires a boundary-scan register description, and it further requires that the description be in one of two forms. It may be a flat, fixed-length register as shown in the following example:

```
attribute BOUNDARY_LENGTH of ttl74bct8374: entity is 18;
```

The <boundary length stmt> defines the number (LENGTH) of cells in the boundary-scan register. This number must match the number of <cell entry> elements in the <boundary register stmt>, which describes the structure of the boundary-scan register. Some cells may require two lines of description (see B.11.2.3).

```
attribute BOUNDARY_REGISTER of ttl74bct8374: entity is
--
--    num   cell  port/*  function safe [ccell disval rslt]
--
      "17  (BC_1, CLK,    input,   X, PULL1)," &
      "16  (BC_1, OC_NEG, input,   X, OPEN1)," &
      "16  (BC_1, *,      control, 1)," &
      . . .
      . . .
      "3   (BC_1, Q(5),   output3, X,   16,    1,    PULL1)," &
      "2   (BC_1, Q(6),   output3, X,   16,    1,    PULL1)," &
      "1   (BC_1, Q(7),   output3, X,   16,    1,    PULL1)," &
      "0   (BC_1, Q(8),   output3, X,   16,    1,    PULL1)";
```

Example 2

The second form of description is a segmented register, possibly with excludable segments, as shown in this example:

```
Attribute ASSEMBLED_BOUNDARY_LENGTH of Chip_2013 : entity is (41,47);
```

The first number in this attribute represents the number of cells in the boundary-scan register with all excludable segments excluded, as would be the case after a reset. This is the minimum possible length. The second number is the same as described in the first example; the total number of cells in the boundary-scan register with all excludable segments included. This is the maximum possible length.

```
Attribute BOUNDARY_SEGMENT of Chip_2013 : entity is
   "north [11] ("&
   -- num   cell  port      function safe [ccell disval rslt]
      "10  (BC_1, *,        controlr,   1 ), "&
       "9  (BC_1, N_D(0),   input,      X, PULL0), "&
       "8  (BC_1, N_D(1),   input,      X, PULL0), "&
       "7  (BC_1, N_Q(1),   output3,    X,  10,   1,   PULL0 ), "&
      ...
       "0  (BC_1, N_Q(8),   output3,    X,  10,   1,   PULL0 ) ), "&
   "south [11] ("&
   -- num   cell  port      function safe [ccell disval rslt]
      "10  (BC_1, *,        controlr,   1 ), "&
       "9  (BC_1, S_D(0),   input,      X, PULL0), "&
       "8  (BC_1, S_D(1),   input,      X, PULL0), "&
       "7  (BC_1, S_Q(1),   output3,    X,  10,   1,   PULL0 ), "&
      ...
       "0  (BC_1, S_Q(8),   output3,    X,  10,   1,   PULL0 ) ), "&
   "west [11] ("&
   -- num   cell  port      function safe [ccell disval rslt]
      "10  (BC_1, *,        controlr,   1 ), "&
       "9  (BC_1, W_D(0),   input,      X, PULL0), "&
       "8  (BC_1, W_D(1),   input,      X, PULL0), "&
       "7  (BC_1, W_Q(1),   output3,    X,  10,   1,   PULL0 ), "&
      ...
       "0  (BC_1, W_Q(8),   output3,    X,  10,   1,   PULL0 ) )";
```

```
Attribute BOUNDARY_SEGMENT of Chip_2013 : entity is
   "east1 [6] ("&
   -- num   cell  port    function safe [ccell disval rslt]
       "5  (BC_1, *,      controlr,   1 ), "&
       "4  (BC_1, E_D(0),   input,      X, PULL0), "&
      ...
       "0  (BC_1, E_Q(4),   output3,    X,  5,   1,   PULL0 ) ), "&
   "east2 [6] ("&
   -- num   cell  port    function safe [ccell disval rslt]
       "5  (BC_1, *,      controlr,   1 ), "&
       "4  (BC_1, E_D(1),   input,      X, PULL0), "&
      ...
       "0  (BC_1, E_Q(8),   output3,    X,  5,   1,   PULL0 ) )";
```

See B.8.21 for the **REGISTER_ASSEMBLY** attribute that puts these segments together to form the boundary-scan register. Suffice it for this example to say that only the "east2" segment is excludable, so the minimum length of the boundary-scan register is 11 + 11 + 11 + 6 + 2 + 0 = 41, and the maximum length is 11 + 11 + 11 + 6 + 2 + 6 = 47. The "2" in these additions account for the domain-control and segment-select cells, defined in the Standard BSDL Package Body in B.9, and which must be included in the Boundary-Scan register to control the excludable segment.

B.8.14.3 Description

By processing the <boundary-scan register description>, it is possible for software to check that every nonlinkage, nonpower, non-TAP controller, non-compliance-enable, nongrouped port name in the port statement has been named by a <port ID> of the <boundary-scan register description>. Missing <port ID> values (other than the linkage, vref, and power type ports, TAP controller and compliance-enable ports, and grouped ports) identify digital system signals lacking corresponding cells in the boundary-scan register, which indicates a noncompliant device or an error in entering the BSDL description.

See B.11 for more information given by example for describing the boundary-scan register.

The <boundary register description> contains a <cell table> within which there is a list of elements (<cell entry>), each with two fields. The <cell entry> elements may be listed in any order, but all are listed:

- The <cell number> element must be in the range from 0 to LENGTH-1, where LENGTH is the length of the boundary-scan register (<register length>) or the length of the boundary-scan register segment (<boundary segment length>).
- The <cell info> contains a list of four, five, or seven elements contained within parentheses. (In the above example, the elements are labeled—`cell`, `port`, `function`, `safe`, `ccell`, `disval`, and `rslt`—as indicated by the commented header.)

All <cell entry> elements include values for the first four elements of the second field. Only <cell entry> elements observing inputs have one additional element, which defines how the input behaves when undriven (<input spec>). Only <cell entry> elements for cells that drive system outputs that can be set to an inactive drive state (e.g., open-collector or three-state outputs) have the three additional elements of the second field, which specify how the output may be disabled (<disable spec>). If the <function> element is **OUTPUT3** or **BIDIR**, the three elements of the <disable spec> must be defined. If the <function> element is **BIDIR**, the action of placing the relevant driver in the inactive drive state is taken as equivalent to setting the cell to operate as a receiver. If the <function> element is **OUTPUT2**, the last three elements may or may not be defined, depending on whether the described driver is an asymmetrical driver, e.g., open-collector (VHDL <pin type> equal to out) or capable of actively driving both states (VHDL <pin type> equal to buffer), respectively.

The <cell spec>, <input spec>, and <disable spec> elements are defined in the following subclauses.

B.8.14.3.1 <cell name> element

This identifies the cell design used. It must match a cell described in the Standard BSDL Package or in a user-supplied BSDL package.

B.8.14.3.2 <port ID or null> element

This element identifies the system input or output connected to a given cell. Any name supplied for this element must match one specified in the <logical port description>. A cell serving as an output control or internal cell has an asterisk (*) supplied for this element. Either a <port name> element with the corresponding <port dimension> previously described as **bit** or a <subscripted port name> must be supplied as the value of a <port ID or null> element.

B.8.14.3.3 <function> element

This element defines the primary function of the relevant cell. Table B-4 lists the possible values of the <function> element.

Table B-4—Function element values and meanings

Value	Meaning	Example figure in this standard[a]
INPUT	A cell observing a system logic input, which may have either control-and-observe capability (required to support *INTEST*) or observe-only capability	Figure 11-19 Figure 11-17 Figure 11-18
CLOCK	Cell at a clock input	Figure 11-18
OUTPUT2	A cell that drives a two-state (either symmetric or asymmetric) output	Figure 11-31
OUTPUT3	A cell that drives data to a three-state output	Figure 11-35d
CONTROL	A cell that controls a three-state enable or direction control	Figure 11-35c
CONTROLR	A control cell that is forced to its disable state in the *Test-Logic-Reset* controller state	Figure 11-37c
INTERNAL	Cell not associated with a device signal pin that captures constants 1, 0, or X	—
BIDIR	A reversible cell for a bidirectional pin	Figure 11-38d
OBSERVE_ONLY	Additional cell observing any device signal or pin	Figure 11-18

[a] The suffix "c" is used to denote a control cell shown in a cited figure. The suffix "d" denotes a data cell.

Notice that many of the cell designs of this standard are somewhat general, meaning they can be used in more than one context. For example, the <function> element in a description of the cell depicted in Figure 11-31 can have as its value **INPUT**, **OUTPUT2**, **OUTPUT3**, **CONTROL**, or **INTERNAL**. The value of the <function> element has important implications in describing a given cell (see B.10.1).

An **INPUT** function indicates that the cell has observe capability (control-and-observe capability if *INTEST* is implemented in the device) and is connected to a system logic input pin. This pin must have a <pin type> value of **in** or **inout** only.

A **CLOCK** function indicates a cell with observe capabilities that is connected to a system logic input clock pin that allows support of *INTEST* and *RUNBIST* [see rule g1) of 11.5.1]. This pin must have a <pin type> of **in.**

An **OUTPUT2** function indicates that the cell (which must have control-and-observe capability) provides data for a two-state (symmetric or asymmetric) driver and is connected to a <port ID> with a <pin type> value of **out**, **buffer**, or **inout**.

An **OUTPUT3** function indicates that the cell (which must have control-and-observe capability) provides data for a three-state driver and is connected to a <port ID> with a <pin type> value of **out** or **inout**.

A **CONTROL** function indicates that the cell (which must have control-and-observe capability) provides output enable control and/or direction control to one or more output drivers or bidirectional pins. A **CONTROL** cell must not be referenced to a <port ID> in the <cell spec> element; see B.11.2.3 for details on system input pins that are used to control system output drivers.

A **CONTROLR** function is identical to a **CONTROL** function with the exception that it also has the capability to be reset or cleared to the **safe** value when the TAP passes through the *Test-Logic-Rese*t state and the TMP controller (if provided) is in the *Persistence-Off* state (see Figure 11-37).

A **BIDIR** function indicates a control-and-observe cell that is connected to a <port ID> with <pin type> **inout**.

An **INTERNAL** function indicates that the cell is either a placeholder cell that has gone unused because of the programming of user-programmable logic or a cell that sits at the interface between digital and analog portions of the core circuitry of a component [see rule b) of 11.4.1]. An **INTERNAL** cell must not reference a <port ID>.

An **OBSERVE_ONLY** function indicates that the cell does not have an update stage and is an additional cell that can monitor any kind of system pin not exempted by semantic checks [rule c) of B.8.9.1 (scan port identification)].

Clause 11 effectively classifies <port ID> signals as input, clock, two-state output, three-state output, and bidirectional pins. For a system pin classified as:

— An input pin, a cell with a <function> of **INPUT** is required. Additional cells with <function> **INPUT** or **OBSERVE_ONLY** may be connected to the input pin. If *INTEST* is a supported instruction, there must be at least one cell with control capability and <function> of **INPUT** connected to the input pin.
— A clock pin, there are two cases:
  i) At least one cell must have a <function> of **CLOCK**. Additional cells with <function> **OBSERVE_ONLY** also may be connected to the pin. This case is used where external clocking must be supplied to support *INTEST* or *RUNBIST*.
  ii) At least one cell must have a <function> of **INPUT**. Additional cells with <function> **INPUT** (see Figure 11-12) or **OBSERVE_ONLY** also may be connected to the clock pin. This case is used where clocking must be supplied by shifting to support *INTEST* [see rule g3) of 11.5.1].
— A two-state output pin, one cell must have a <function> of **OUTPUT2**. Additional cells with <function> **OBSERVE_ONLY** also may be connected to the pin or the output of the system logic.
— A three-state output pin, one cell must have a <function> of **OUTPUT3**. Additional cells with <function> **OBSERVE_ONLY** also may be connected to the pin or the outputs of the system logic.
— A bidirectional pin, there are two cases:
  i) A single cell with <function> **BIDIR** is attached to the pin. Additional cells with <function> **OBSERVE_ONLY** also may be connected to the pin or outputs of the system logic.
  ii) A two-cell structure is used to create bidirectionality; one of these cells must have a <function> of either **OUTPUT2** or **OUTPUT3**, and the other cell must have a <function> of **INPUT**. Additional cells with <function> of **INPUT** or **OBSERVE_ONLY** also may be connected to the pin or the outputs of the system logic.

Clause 11 classifies system logic (different from system pin) input signals and output signals as input, clock, output data, and output control. For a system logic signal classified as:

— Input or clock, there are two cases:
  i) Cell provisions are the same as noted for system input or clock pins.
  ii) For system logic receiving data from analog circuitry, cells with <function> **INTERNAL** are connected, but they must not be referenced to a system pin in the <cell spec> element.

— Output data or output control, there are two cases:
  i) Cell provisions must be the same as noted for system pins above; additional cells with <function> **INTERNAL** also may be connected, but they must not be referenced to a system pin in the <cell spec> element.
  ii) For system logic providing data to analog circuitry, cells with <function> **INTERNAL** must be connected, but they must not be referenced to a system pin in the <cell spec> element.

Clause 11 also specifies that cells may exist that are connected neither to system pins nor to system logic due to the programming of programmable system logic (see 11.8). Such cells must have <function> values of **INTERNAL**.

B.8.14.3.4 <safe bit> element

This element supplies a value that should be loaded into the shift/capture stage (and the update stage if it exists) of a given cell when board-level test generation software might otherwise choose a value randomly.

The <safe bit> value is not intended to force software to use particular values for cells. Rather, it provides values for cells where software would otherwise choose a 0 or 1 at random. An **X** signifies that the value does not matter and that test generation software may assign either a 1 or a 0 in a case where there is no value that the algorithm requires.

For control cells, the <safe bit> value must be that which turns off the associated drivers. Other examples where the <safe bit> value might be defined as **0** or **1** (rather than **X**) are:

— The value that an output should have during *INTEST* that minimizes driver current.
— A preferred value to present to on-chip logic at a component input during *EXTEST*.
— The value that should be presented to unconnected <portID> (i.e., identified by an **OPEN**, **TIE0**, or **TIE1** in the applicable pin map) ports.

NOTE—See B.8.14.3.8 for the definition of <input spec>. The location of the description was moved to the end of this clause to preserve the numbering in this clause.

B.8.14.3.5 <ccell> element

For the relevant <port ID>, this element identifies the <cell number> of the control cell that can disable the output.

B.8.14.3.6 <disable value> element

This element gives the value that must be scanned into the control cell identified by the previous <ccell> element to disable the port named by the relevant <port ID>.

B.8.14.3.7 <disable result> element

For a <cell spec> with a <function> of **OUTPUT2**, **OUTPUT3**, or **BIDIR**, the <port ID> element is the name of a signal driven out of the component. If the driver of that signal can be disabled, the value of the <disable result> element within the same <cell info> specifies the condition of the signal when the driver is disabled. The permissible values are:

— A weak 0 internal pull down (**PULL0**)
— A weak 1 internal pull up (**PULL1**)
— A weak 0 external pull down (**WEAK0**)
— A weak 1 external pull up (**WEAK1**)
— A weak state memory of the last strongly driven logic state (**KEEPER**)
— A received value of 0 when the external net is undriven by any source, weak or strong (**OPEN0**)

— A received value of 1 when the external net is undriven by any source, weak or strong (**OPEN1**)
— A high impedance state (**Z**)

The values **WEAK1** and **WEAK0** would be used for asymmetrical drivers, such as TTL open-collector or ECL open-emitter outputs, when a pull-down or a pull-up is required external to the component.

The values **PULL0** and **PULL1** are used to describe symmetrical drivers and bidirectional drivers that have a high impedance state and an internal weak pull-down or pull-up. Fault coverage is improved when the bidirectional driver has a weak pull-down on the input and is described as **PULL0** rather than as Z. The Z high-impedance state description should be avoided with bidirectional drivers and drivers with a high-impedance mode that float to a predetermined logic state. The values **PULL0** and **PULL1** are also used for asymmetrical drivers, such as TTL open-collector or ECL open-emitter outputs or bidirectional outputs when a pull-down or a pull-up is internal to the component.

The value **KEEPER** would be used for drivers that maintain a weakly driven memory (H, L) of the strongly driven state (1, 0) last seen on the board network to which such a driver is connected.

NOTE 1—It must be emphasized that bus keepers generally do not retain a reliable logic state useful as part of a logic implementation. Indeed, any glitch or system noise on a "kept" bus may upset the state of any connected keepers. Drivers with bus keepers can be thought of as types that disable to a high-impedance state that always stays out of the forbidden voltage zone between defined low and high logic values. Just as a high-impedance state conveys no information useful to test and diagnosis, neither does a kept state. Implementers of board or system application software will likely choose the same treatment of the <disable result> values **KEEPER** and **Z**.

NOTE 2—The keeper feature is an important parametric option in the design of a device's drivers. IC vendors using such drivers would want to verify their action on an IC tester. By giving the ability for BSDL to denote the existence of such drivers, IC test software can automatically set up tests (at the logical level) for these features that are similar to hysteresis measurements. Of course, the analog parameters of a keeper, like other analog information, is not described in BSDL.

The values **OPEN0** and **OPEN1** are used to describe the behavior of the receiver of a boundary cell with the function of **BIDIR** when the driver and the external net are undriven by any source (the net has a resolved state of Z). This is identical to the same behavior described for these keywords in B.8.14.3.8.

B.8.14.3.8 <input spec> element

For a <cell info> with a <function> **INPUT** or **CLOCK**, the <port ID> element is the name of a signal received into the component. The <input spec> is required and the value specifies the behavior of the receiving circuits when that signal is not driven. For example, the source of the signal on the board may be disabled, the connection of the pin to the board may be open due to a defect, or the pin may simply not be connected on the board, or the pin map for the package may mark the pin as **OPEN**.

For a <cell spec> with a function of **OBSERVE_ONLY**, the <port ID> element is the name of the port being observed and the following apply: The <input spec> is prohibited for digital signal pins also observed by a nonredundant **INPUT** or **CLOCK** cell unless the observe-only cell is observing a fault condition. The <input spec> is required for any pin that does not have a required cell on it. For nondigital pins, and for digital pins where the redundant observe-only cell is observing a fault condition, the value of <input spec> must be either **EXPECT1** or **EXPECT0**, indicating the value captured when a fault is not detected ("good machine" value).

The permissible values are:

— A weak 0 internal pull down (**PULL0**)
— A weak 1 internal pull up (**PULL1**)
— A received value of 0 when the signal is undriven (**OPEN0**)
— A received value of 1 when the signal is undriven (**OPEN1**)
— A 0 external pull or tie down (**EXTERN0**) for a signal not connected to a driving source

— A 1 external pull or tie up (**EXTERN1**) for a signal not connected to a driving source
— A "kept" state memory of the last strongly driven logic state (**KEEPER**)
— An indeterminate value when the signal is undriven (**OPENX**)
— A fault detector no-fault condition captured as a 1 (**EXPECT1**)
— A fault detector no-fault condition captured as a 0 (**EXPECT0**)

The values **PULL0** and **PULL1** are used to describe receivers that have an internal weak pull-down or pull-up circuit, either passive or active, which is relatively high impedance, but will source or sink sufficient current or voltage to the board net to establish a valid logic value in the absence of a strong driver or a conflicting weak source.

The values **OPEN1** and **OPEN0** are used to describe receivers that will produce a known output to the system logic and boundary-scan register cell when the associated pin is not connected or completely undriven. It does not imply the sourcing or sinking of any significant current or voltage to the board net, if any, and therefore cannot affect the value seen on the net by other inputs. An example appears in the LVDS differential standard where the receiver is required to identify an unconnected input pair and an output of 1 in that case.

The values **EXTERN0** and **EXTERN1** are used to describe receivers that require board-level termination if they are not connected to a driver source. They could cause excessive currents or other anomalous behavior if not driven or terminated (usually due to the input floating to the V-threshold). As long as the input signal has a driver source, this designation can be ignored. However, in board test, the driver source may not be present and this designation will provide a warning to the board test engineers that termination is required for reliable operation.

The value **KEEPER** would be used for receivers that maintain a weakly (relatively high impedance) driven memory of the state last seen on the board network to which such a driver is connected. Typically, a keeper will source or sink sufficient current or voltage to the board net to establish a valid logic value in the absence of a strong driver or a conflicting weak source. As with output pins, this designation provides no information of any practical use for defect detection or diagnosis, but it does prevent anomalous behavior due to an undriven or unterminated input. The cautionary notes about keepers in the previous clause apply here as well.

The value **OPENX** is used to describe receivers that do not fit into any of the other designations. They may produce an unpredictable output when the associated pin is not connected to any driving source, strong or weak. As with the **Z** designation for output pins, this designation provides no information of any practical use for defect detection or diagnosis.

The values **EXPECT1** and **EXPECT0** are only used with an optional **OBSERVE_ONLY** cell that observes the output of a fault detection circuit. Such fault detection circuits are required in order to observe nondigital pins, but they may also be used with digital pins. The value implied by the keyword is the expected value for a defect-free circuit.

Figure B-7 illustrates an input with an explicit pull-up, which biases the input pin when it is open or undriven. This type of input would use the **PULL1** keyword. The insert (dashed) box shows a pull-down alternative to the pull-up that would be described by the **PULL0** keyword**.**

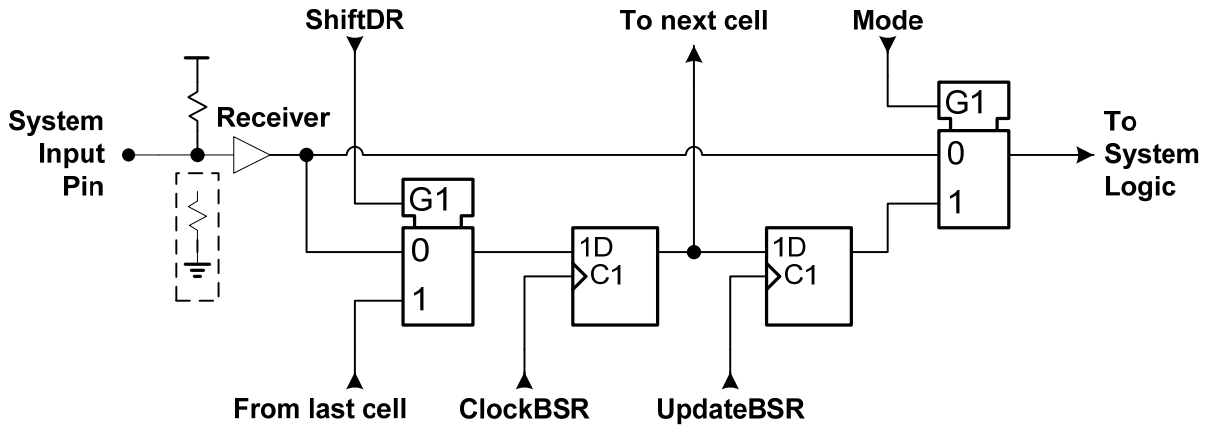


Figure B-7—Cell on an input, which pulls to a logic 1

Cell 0 in the example BSDL boundary-scan register description below could describe an input cell similar to Figure B-7. Cells 1, 2, and 3 all are associated with a single bidirectional port called MyBidi. Cell 1 in the example inherits its undriven behavior (**PULL0**, in this case) from the **OUTPUT3** cell, cell 2, driving the same port:

```
--num cell port/*  function     safe [input/ccell disval rslt]
" 0  (BC_1, MyInput,  input,        X,   PULL1),            "&
" 1  (BC_1, MyBidi,   input,        X,   PULL0),            "&
" 2  (BC_1, MyBidi,   output3,      0,      3,        0,  PULL0), "&
" 3  (BC_1, *,        control,      0),                     "&
```

The <input spec> provides valuable information to the ATPG process and the board test engineer. Figure B-8 illustrates some of the test coverage improvements achieved by the use of <input spec> when the IC is on a board. The function of each pin is shown with the <input spec>, where appropriate, in parenthesis.

The upper left input is tied to 2.5 V power through an external pull-up resistor. Without an <input spec> of **OPEN0**, the presence of an open on this pin cannot be determined. When the input pin is defined by the IC vendor as **OPEN0**, an ATPG tool can predict a different capture value when the pin is open than when it is properly connected.

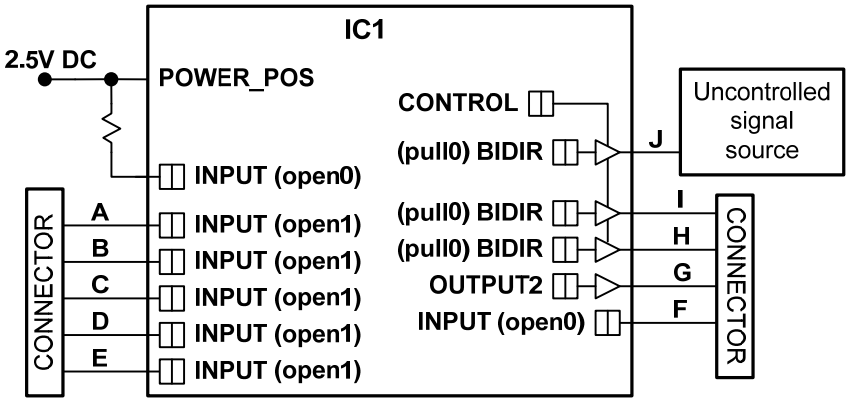


Figure B-8—Illustration of use of <input spec> for an IC

The other five inputs on the left side are connected to a passive connector by nets A through E. Without an <input spec> on these pins, the test engineer is forced to populate or drive the connector in order to prevent undriven inputs from introducing noise in an IEEE 1149.1 assisted on-chip test or a board-level IEEE 1149.1 assisted at-speed test such as SERDES PRBS. With the <input spec>, the test engineer knows that the undriven inputs are not floating and hence additional actions to prevent noise during at-speed tests are not needed.

Nets F–I depict an input (F), a two-state output (G), and two bidirectional pins (H and I) connected to another connector. The three bidirectional pins H, I, and J may have been designed to be bidirectional for test, but inputs in mission mode. Such design usually improves test coverage on a board. However, the last bidirectional pin (J) is connected to a signal source that cannot be controlled during test. All three pins are shown connected to a single control cell, so the uncontrolled signal source on pin J prevents driving signals on any of the three pins. This illustrates how potential board-level conflicts can limit tests of IEEE 1149.1 bidirection outputs during the ATPG process. With the <input spec>, ATPG can correctly set expected data for the input on nets F, H, and I to be a logic zero; hence, any shorts that exist from net G (or any other driven net that may be present) to F, H, I, or J can be detected at the inputs, even if H and I are not driven.

Component designers are encouraged to design ICs with predictable inputs rather than with an <input spec> of **OPENX**, or a bidirectional with a <disable result> of **OPENX** or **Z,** whenever possible.

Figure B-9 shows a portion of a component showing three possible fault detection circuits and their associated **OBSERVE_ONLY** cells.

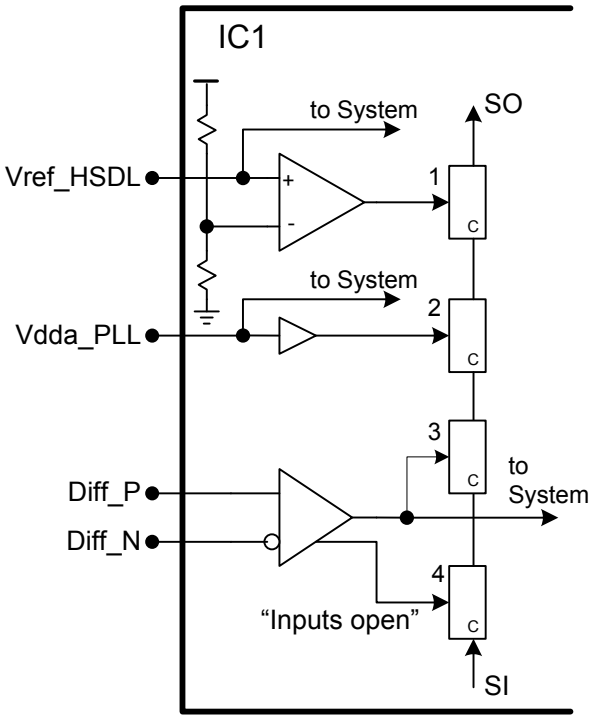


Figure B-9—Illustration of use of fault detection boundary cells for an IC

Cells 1, 2, and 4 are **OBSERVE_ONLY** cells. Cells 1 and 2 have an <input spec> value of **EXPECT1**. (Cell 3 is the required **INPUT** cell for the mission mode output of the differential receiver.)

Input "Vref_HSDL" is <pin type> of **VREF_IN**, and a comparator circuit verifies that the signal is above a threshold relative to the Vdd of the I/O. As long as this reference voltage is present, the comparator outputs a 1, which is captured in cell 1. Cell 1 has an <input spec> value of **EXPECT1**. A window comparator could also have been provided to determine whether the reference was within a valid range.

Cell 2 uses a normal digital receiver to monitor the "Vdda_PLL" input of <pin type> **POWER_POS**. The VDDA_PLL supply is essentially the same as the digital Vdd, so it will appear to be a valid logic 1. Cell 2 has an <input spec> value of **EXPECT1**.

The differential receiver is designed to the LVDS protocol, and as part of that standard, the receiver is capable of detecting when the inputs are open. In that situation, the output of the receiver is forced to 0 and, hence, would have an <input spec> of **OPEN0** on cell 3. However, this is an ambiguous fault indication for diagnosis of a failing interconnect test. In this case, the "Inputs Open" signal (1 means the inputs are open) has been brought out of the receiver and is captured in cell 4, which has an <input spec> of **EXPECT0**.

These are just a few simple examples to illustrate possible ways of providing some test coverage of nondigital pins. They certainly are not definitive. To complete the example, the cells of Figure B-9 could be defined as shown in the following part of a boundary-scan register description:

```
...
--num cell  port/*  function     safe   [input]
" 1  (BC_4, Vref_HSDL, observe_only, X, EXPECT1), "& -- Vref above minimum
" 2  (BC_4, Vdda_PLL,  observe_only, X, EXPECT1), "& -- Power present
" 3  (BC_1, Diff_P,    input,        0, OPEN0),  "&
" 4  (BC_4, Diff_P,    observe_only, X, EXPECT0), "& -- Diff pair connected
...
```

This example can be further expanded by using the **REGISTER_MNEMONICS** (see B.8.18) and **REGISTER_FIELDS** (see B.8.19) attributes to provide machine-readable information to the test engineer. When desired, the value captured in boundary cells 1, 2, and 4 can cause the appropriate message from the mnemonic <information tag> to be presented to the test engineer. If there were multiple pins with the same type of fault detection circuits, such as multiple differential pairs, all could be listed in the one "DiffPins" field and use the same mnemonic.

```
attribute REGISTER_MNEMONICS of ROO_Example : entity is
   "Vref_Range ( "&
      " Pass (1) <Vref above minimum>, "&
      " Fail (0) <Vref below minimum or open>  "&
      " ), "&
   "PLL_Power ( "&
      " Pass (1) <PLL Vdd present>, "&
      " Fail (0) <PLL Vdd absent or open>  "&
      " ), "&
   "Diff_open ( "&
      " Open      (1) <Differential inputs open/undriven>, "&
      " Connected (0) <Differential inputs driven>  "&
      " )" ;

attribute REGISTER_FIELDS of ROO_Example : entity is
   Boundary [289] ( "&
      "(Vref     [1] IS (1 to 1) captures(Vref_Range(-))), "&
      "(PLLpwr   [1] IS (2 to 2) captures(PLL_Power(-))), "&
      "(DiffPins [1] IS (4 to 4) captures(Diff_open(-))), "&
      ...
```

B.8.15 RUNBIST description

The goal of this portion of a BSDL description is to provide support for the *RUNBIST* instruction as specified within this standard. The intent is to describe *only* those aspects of the *RUNBIST* instruction that this standard specifies. In some cases, this may not completely define the built-in self-test (BIST) operational environment. In such cases, additional information must be supplied externally.

Note that the following features of a BIST implementation are *not* supported explicitly by BSDL:

— Timing-related information (beyond active clock and number of clock cycles)
— Frequency and phase relationship(s) of clock(s)

B.8.15.1 Specifications

Syntax

<runbist description> ::= **attribute RUNBIST_EXECUTION of** <component name>
<colon> **entity is** <quote> <runbist spec> <quote> <semicolon>
<runbist spec> ::= <wait spec> <comma> <pin state> <comma> <signature spec>
<wait spec> ::= **WAIT_DURATION** <left paren> <duration spec> <right paren>
<duration spec> ::= <clock cycles list> | <time and clocks>
<time and clocks> ::= <time> [<comma> <clock cycles list>]
<clock cycles list> ::= <clock cycles> { <comma> <clock cycles> }
<time> ::= <real>
<clock cycles> ::= <port ID> <integer>
<pin state> ::= **OBSERVING** <condition> **AT_PINS**
<condition> ::= **HIGHZ** | **BOUNDARY**
<signature spec> ::= **EXPECT_DATA** <det pattern>
<det pattern> ::= <bit> { <bit> }
<bit> ::= **0** | **1**

Rules

a) A <det pattern> shall be a contiguous sequence of one or more **0** and **1** characters containing no spaces or format effectors.

NOTE 1—For example, **001100** and **110101** are compliant. However, **100 X00** is not compliant because of the embedded space and the **X** character.

b) Time shall be specified in seconds (via the value of the <time> element), where:
   1) Both time and clock cycles are specified, they shall be interpreted as the maximum of the time specified or the time required to apply the required number of clock cycles.
   2) More than one clock is specified, the duration shall be the time required for all of the clock inputs to receive the specified number of clock cycles.

c) The number of bits in the value of the <det pattern> element of the <signature spec> element shall be equal to the length of the register whose name appears in the <register> element of that <register association> element of the <register access description> in which **RUNBIST** appears as the value of an <instruction name> element:
   1) If the value of the associated <register> element is **BOUNDARY**, the register length shall be specified by the value of the <integer> element of the <boundary length stmt>.
   2) If the value of the associated <register> element is *not* **BOUNDARY**, the register length shall be specified by the explicitly defined value of the <integer> element in that same <register> element.

d) Any value of <port ID> in the <wait spec> statement shall:
   1) Appear as the value of <port ID> in the <TCK stmt> of the BSDL description (see B.8.9).
   2) Appear as the value of <port ID> in a <cell spec> of the <boundary register stmt> in which the <function> element has the value **CLOCK** (see B.8.14).

e) If the <runbist description> statement occurs in a BSDL description, **RUNBIST** shall be the value of some <instruction name> element in the <opcode table> of the <instruction opcode stmt>.

f) Values of <time> and <clock cycles> shall be greater than 0.

g) A given <port ID> shall not appear more than once in the <runbist description> element.

NOTE 2—The existence of **RUNBIST** in the **INSTRUCTION_OPCODE** table does not require <runbist description> to be specified in a BSDL description.

B.8.15.2 Examples

Example 1

```
attribute RUNBIST_EXECUTION of BIST_IC1: entity is
      "Wait_Duration (1.0e-3)," &
      "Observing HIGHZ At_Pins," &
      "Expect_Data 010101";
```

In this example, the value of <time> in the <wait spec> is specified at 1 ms, which is the minimum duration the device needs to stay in the *Run-Test/Idle* controller state. Also, note that the output pins are forced to high impedance, which implies that there is no need to initialize the update latches of the boundary-scan register.

Example 2

```
attribute RUNBIST_EXECUTION of BIST_IC2: entity is
      "Wait_Duration (TCK 23000)," &
      "Observing HIGHZ At_Pins," &
      "Expect_Data 010101";
```

In this example, the device needs to wait in the *Run-Test/Idle* controller state for the duration sufficient for the application of 23 000 clock cycles at TCK.

Example 3

```
attribute RUNBIST_EXECUTION of BIST_IC3: entity is
      "Wait_Duration (1.0e-3, TCK 23000)," &
      "Observing HIGHZ At_Pins," &
      "Expect_Data 010101";
```

In this example, <wait spec> is to be interpreted as 1 ms. or the time required for TCK to receive 23 000 cycles, whichever is greater.

Example 4

```
attribute RUNBIST_EXECUTION of BIST_IC4: entity is
      "Wait_Duration (CLK 100000, SYSCK 24000)," &
      "Observing BOUNDARY At_Pins," &
      "Expect_Data 010101";
```

In this example, <wait spec> is to be interpreted as the time required for CLK and SYSCK to receive 100 000 and 24 000 clock cycles, respectively. Also note that the boundary-scan register is visible at the pins, indicating that it needs to be initialized before the execution of *RUNBIST*.

B.8.16 INTEST description

The goal of this portion of a BSDL description is to describe:

- How test patterns are to be applied to the component when the *INTEST* instruction is selected (i.e., the source of clock pulses for the component and the time for which the test logic must remain in the *Run-Test/Idle* controller state to permit execution of each applied test)
- The external behavior of the component while the *INTEST* instruction is selected

Note that the test patterns themselves are not specified and are assumed to be provided by an alternative method not specified in this annex. For *INTEST*, the duration is not the duration for the entire test (as is the case of *RUNBIST,*) but the time required for the application of a single vector. With the application of each vector via the boundary-scan register, this standard permits the device to execute a single step of the operation that may require several clock cycles to complete. Otherwise, the interpretation of <pin spec> is identical to that in **RUNBIST_EXECUTION**. The syntax of the <wait spec> and <pin spec> elements is given in B.8.15.

B.8.16.1 Specifications

Syntax

<intest description>::= **attribute INTEST_EXECUTION of** <component name>
<colon> **entity is** <quote> <intest spec> <quote> <semicolon>
<intest spec>::= <wait spec> <comma> <pin state>

Rules

a) Any value of <port ID> in the <wait spec> statement shall:
   1) Appear as the value of <port ID> in the <TCK stmt> of the BSDL description (see B.8.9).
   2) Appear as the value of <port ID> in a <cell spec> of the <boundary register stmt> in which the <function> element has the value **CLOCK** (see B.8.14).

b) If the <intest description> statement occurs in a BSDL description, **INTEST** shall be the value of some <instruction name> element in the <opcode table> of the <instruction opcode stmt>.

c) Values of <time> and <clock cycles> in the <wait spec> shall be greater than 0.

d) A given value of <port ID> shall not appear more than once in the <intest description> element.

NOTE—The existence of **INTEST** in the **INSTRUCTION_OPCODE** table does not require <intest description> to be specified in a BSDL description.

B.8.16.2 Examples

Example 1

```
attribute INTEST_EXECUTION of IC1: entity is
      "Wait_Duration (1.0e-3)," &
      "Observing HIGHZ At_Pins";
```

In this example, the value of <time> in the <wait spec> is specified as 1 ms, which is the minimum duration the device needs to stay in the *Run-Test/Idle* controller state. Also, note that the output pins are forced to high impedance.

Example 2

```
attribute INTEST_EXECUTION of IC2: entity is
      "Wait_Duration (TCK 250)," &
      "Observing HIGHZ At_Pins";
```

In this example, the device needs to wait in the *Run-Test/Idle* controller state for a duration sufficient for the application of 250 clock cycles of TCK to permit the device to complete one step of operation.

Example 3

```
attribute INTEST_EXECUTION of IC3: entity is
      "Wait_Duration (CLK 100, SYSCK 200)," &
      "Observing BOUNDARY At_Pins";
```

In this example, <wait spec> is to be interpreted as the time required for CLK and SYSCK to receive 100 and 200 clock cycles, respectively. Also note that the state of the pins is controlled by the data held in the boundary-scan register.

B.8.17 System clock requirements attribute

The **SYSCLOCK_REQUIREMENTS** attribute is used to describe the use of system clocks for various instructions, and to define the required minimum and maximum frequency. These definitions allow test engineers to know what resources will be needed for specific tests.

B.8.17.1 Specifications

Syntax

<system clock description> ::= **attribute SYSCLOCK_REQUIREMENTS**
of <entity target> **is** <system clock description string> <semicolon>

<system clock description string>::= <quote> <system clock requirement>
{ <comma> <system clock requirement> } <quote>
<system clock requirement>::= <left paren> <port ID> <comma> <min freq> <comma> <max freq>
<comma> <clocked instructions> <right paren>
<min freq> ::= <real>
<max freq> ::= <real>
<clocked instructions> ::= <clocked instruction> { <comma> <clocked instruction> }
<clocked instruction> ::= **RUNBIST** | **INTEST** | **INIT_SETUP** | **INIT_SETUP_CLAMP** | **INIT_RUN** |
ECIDCODE | **IC_RESET** | <VHDL identifier>

Rules

a) The **SYSCLOCK_REQUIREMENTS** attribute shall be provided if any system clock input (other than TCK) is required for execution of an instruction.
b) The <port ID> shall have a <pin type> of **in**, **inout**, **LINKAGE_IN**, or **LINKAGE_INOUT** in the <logical port description>.
c) If the <port ID> has a <pin type> of **in**, it shall also have a <function> of **CLOCK** in the associated <cell entry> of the boundary-scan register description.
d) If the <port ID> appears in a <grouped port identification>, it shall be a <representative port>.
e) If the <port ID> is defined as a bit vector in the <logical port description>, then the <port ID> shall include an index that lies within the defined <range>.
f) The <max freq> value shall be greater than or equal to the <min freq> value.
g) If <clocked instruction> is a <VHDL identifier> and not the name of an instruction defined in this standard, it shall be an <instruction name> in the <instruction opcode statement> (see B.8.11).
h) Where the value of <conformance identification> is **STD_1149_1_2001**, **STD_1149_1_1993**, or **STD_1149_1_1990**, the <clocked instructions> list shall include only **RUNBIST** and **INTEST** instructions.

B.8.17.2 Description

A complex IC may have multiple system clock inputs, and different ones may be needed for different tests. The documentation of sequential processes, which would use these clocks, is now done with PDL (see Annex C), so a BSDL attribute is used to specifically describe clocks, which may be used in PDL, and to associate those clocks with the instructions that will require them. Where a system clock is used in PDL, it must reference a clock defined in the **SYSCLOCK_REQUIREMENTS** attribute.

Note that, of the instructions defined in this standard, the *BYPASS, PRELOAD, SAMPLE, EXTEST, CLAMP, CLAMP_HOLD, CLAMP_RELEASE, TMP_STATUS, IDCODE, USERCODE,* and *HIGHZ* instructions do not allow the use of system clocks. The standard instructions *RUNBIST, INTEST, INIT_SETUP, INIT_SETUP_CLAMP, INIT_RUN, ECIDCODE,* and *IC_RESET* do allow, but do not require, the use of system clocks. Design-specific instructions can use system clocks as required and documented. This attribute is required only if an instruction defined in the component uses a system clock for execution.

The minimum and maximum frequency (at the component pin) is documented for the use of the test engineer.

B.8.17.3 Examples

```
Attribute SYSCLOCK_REQUIREMENTS of MyChip : entity IS
  "(F125MHz_in, 120.0e6, 130.0e6, INIT_RUN, ECIDCODE), "&
  "(SerDesClk,  198.5e6, 201.5e6, SerDes_Loopback, SerDes_BER) ";
```

B.8.18 Register mnemonics description

Register mnemonics provide meaningful text names (and text descriptions) for values that may be loaded into a TDR or a part of a TDR. While not required as a part of the documentation of the test logic on an IC, they can provide important improvements by moving the tables in component specification documents, which list all the binary values used in a TDR and their effects, to the machine- and human-readable BSDL description of the test logic. This is expected to aid those that have to set up test values long after the IC designers have moved on, and reduce errors and test development time.

In addition, these values may be supplied by an IP provider in a BSDL Package Body format to their IC design customers. These Packages may be used directly, aiding the IC designer in building up the documentation of the IC and preserving the intent of the IP provider. Note that multiple occurrences of <register mnemonic description> are allowed that support multiple sources with no requirement to merge them.

Finally, tool providers may take advantage of this information to provide a more productive environment for development of test for IC and higher level assemblies.

B.8.18.1 Specifications

Syntax

<register mnemonics description>::= **attribute REGISTER_MNEMONICS of**
    <target> **is** <register mnemonics string> <semicolon>
<target>::= <entity target> | <package target>
<entity target>::= <component name> <colon> **entity**
<package target>::= <user package name> <colon> **package**
<register mnemonics string>::=
    <quote> <mnemonic definition> { <comma> <mnemonic definition> } <quote>
<mnemonic definition>::= <mnemonic group name> <left paren> <mnemonic list> <right paren>
<mnemonic group name>::= <VHDL identifier>
<mnemonic list>::= <mnemonic assignment> { <comma> <mnemonic assignment> }
<mnemonic assignment>::= <mnemonic identifier>

<left paren> <pattern specification> <right paren> [<information tag>]
<pattern specification>::= <binary pattern> | <hex pattern> | <decimal pattern> | **others**

Rules

a) When the <target> is an <entity target>, the <component name> shall match the <component name> of the containing <BSDL description> (see B.8.1.1).
b) When the <target> is a <package target>, the <user package name> shall match the <user package name> of the containing <user package> (see B.10.1).
c) All <mnemonic group name> elements that appear in a <mnemonic definition> shall be unique within the containing package or entity.
d) All <mnemonic identifier> elements that appear in a <mnemonic list> shall be unique within that list.
e) Within a <mnemonic list>, there shall be at least one <mnemonic assignment> containing a <binary pattern>, <hex pattern>, or <decimal pattern>.
f) Every <binary pattern>, <hex pattern>, or <decimal pattern> (enumerated or implied by X characters in a value) that appears in a <mnemonic list> shall have binary equivalent values that are unique within that list; that is, every pair of such binary equivalent values shall differ in at least one bit position where both contain only 0 or 1.

   NOTE 1—A <mnemonic list> does not need to contain every possible value, either explicitly or implicitly.

   NOTE 2—None of the multiple patterns implied by X characters in a pattern value may be included elsewhere in the <mnemonic list>.

g) The **others** keyword shall only appear in the last <mnemonic assignment> element of a <mnemonic list>, when it appears at all.
h) The **others** keyword shall represent all patterns that have not appeared or been implied in any <mnemonic assignment> element of a <mnemonic list>.

   NOTE 3—The **others** keyword can be used even if all possible patterns have already appeared or been implied. This usage is consistent with VHDL.

i) The **others** keyword shall be interpreted as assigning a <mnemonic identifier> and possibly an optional <information tag> to those values of a <mnemonic definition> that are not used for assigning values to a register field.

B.8.18.2 Description

Mnemonics are grouped based on the register or register field they are intended to be used with. Within each group, each mnemonic name is associated with a unique binary, hexadecimal, or decimal pattern. Binary and hexadecimal patterns are allowed to include the X value.

Each mnemonic group has a name that follows the rules of VHDL identifiers, and each value has a name that is allowed to be much more free form and descriptive. In fact, the names of mnemonic values (<mnemonic identifier> in the syntax) may contain any valid characters other than PDL or Tcl formatting characters [see rule b4) of B.5.4.1]. A mnemonic therefore may contain periods or start with a number, unlike VHDL identifiers. Mnemonic identifier names must be unique within a group, but they need not be between groups. Mnemonic group names, on the other hand, need to be unique within a BSDL or a Package. If the same mnemonic group name is used in two different Packages, or in a BSDL and a Package used in that BSDL, then the Package name can be prefixed to the mnemonic group name to differentiate them. See <mnemonic association> in B.8.20.1.

Multiple mnemonic groups may be defined, and a given mnemonic group may be associated with more than one register or register field.

The information tag (enclosed by chevrons) is descriptive and mostly free-form, and it is intended to provide additional information to assist users in picking the correct value to write to or read from a TDR during test. See B.5.7 for more information.

B.8.18.3 Examples

Example 1

This example is a **REGISTER_MNEMONICS** attribute as found in a BSDL:

```
attribute REGISTER_MNEMONICS of INIT_Example : entity is
   "SerDes_Protocol ( "&
      "   off  (0) <Powered down>, "&
      " Resvd1 (1) <Reserved for future use>, "&
      "   SATA (2) <Serial Advanced Technology Attachment>, "&
      "   SRIO (3) <Serial RapidIO>, "&
      " Resvd2 (4) <Reserved for future use>, "&
      "   XAUI (5) <10 Gbps Attachment Unit Interface>, "&
      " Resvd3 (0b11X) <Undefined behavior - Do Not Use> "&
                  " ), "&
   "SerDes_TX_Outputs ( "&       -- Output driver swing level
      "         off (0b00) <Powered down>,  "&
      "  Full_Swing (0b01) <100% Vdd Swing>, "&
      "   75%_Swing (0b10) <75% Vdd Swing>, "&
      " 52.7%_Swing (0b11) <52.7% Vdd Swing - Not valid for XAUI> "&
                   " ), "&
   "SerDesClockSettings ( "&     -- Only 2 valid settings
      "  125Mhz (0x07), "&
      "  100Mhz (0x15), "&
      " Invalid (Others) <Undefined behavior - Do Not Use> "&
                     " )" ;
```

In the first mnemonic group (named SerDes_Protocol), all of the eight possible values are enumerated explicitly or implicitly. The three bit fields named “Resvd1”, “Resvd2”, and “Resvd3” could have been omitted if desired as it is not required to enumerate all possible patterns, although, again, specific information tags were desired for these decodes. In the case of “Resvd3”, an X bit appears, but the two patterns implied, “110” and “111”, do not appear elsewhere in the list, satisfying the rules.

In the third group (SerDesClockSettings), the **“others”** keyword is used to assign a specific information tag to the 30 possible decodes (assuming it is assigned to a 5-bit register field) not already described explicitly. Note that the highest order 1 bit in the binary equivalent of the hexadecimal values specified will fit within a 5-bit field.

The mnemonic identifier “off” is used in both mnemonic groups SerDes_Protocol and SerDes_TX_Outputs, which have patterns of differing lengths. This is acceptable within a single BSDL or combination of BSDL and a User Package since within their mnemonic definitions, they are unique and of the same pattern lengths as their related patterns within that mnemonic definition.

Example 2

This example shows the same mnemonic attribute example provided in a user package (see B.10). The actual information content of the attribute is identical. Additional information (other than register mnemonics) may be provided in the package.

```
package MyCorp_SERDES_1_2_3 is
  use STD_1149_1_2013.all;
   -- { deferred constant } (see B.10) may be given here
end MyCorp_SERDES_1_2_3;
```

```
package body MyCorp_SERDES_1_2_3 is

   use STD_1149_1_2013.all;

   -- { extension declaration } (see B.8.24) may be given here
   attribute REGISTER_MNEMONICS of INIT_Example : entity is
      "SerDes_Protocol ( "&
         "   off  (0b000) <Powered down>, "&
         "   SATA (0b010) <Serial Advanced Technology Attachment>, "&
         "   SRIO (0b011) <Serial RapidIO>, "&
         "   XAUI (0b101) <10 Gbps Attachment Unit Interface>, "&
         " Resvd1 (0b100) <Reserved for Future Use>, "&
         " Resvd2 (0b11X) <Undefined behavior - Do Not Use> "&
         " ), "&
      "SerDes_TX_Outputs ( "&       -- Output driver swing level
         "         off (0b00) <Powered down>,  "&
         "  Full_Swing (0b01) <100% Vdd Swing>, "&
         "   75%_Swing (0b10) <75% Vdd Swing>, "&
         " 52.7%_Swing (0b11) <52.7% Vdd Swing - Not valid for XAUI> "&
         " ), "&
      "SerDesClockSettings ( "&     -- Only 2 valid settings
         "  125Mhz (0b00111), "&
         "  100Mhz (0b10101), "&
         " Invalid (Others) <Undefined behavior - Do Not Use> "&
         " )" ;
  -- { Register fields description } (see B.8.19) may be given here
  -- { Register assembly description } (see B.8.19) may be given here
  -- { <cell description constant> } (see B.10.1) may be given here

end MyCorp_SERDES_1_2_3 ;
```

Example 3

Some additional mnemonic descriptions, as they might be specified in a BSDL, which will be referenced in **REGISTER_FIELDS** and **REGISTER_ASSEMBLY** attribute examples:

```
   attribute REGISTER_MNEMONICS of INIT_Example : entity is

      "SerDesCXVddSelLevel ( "&
         " 1.8v (1) <1.8V power supply used>, "&
         " 1.5v (0) <1.5V power supply used>  "&
         " ), "&

      "Switch ( "&        -- Simple On/Off single bit field.
         " off (1) <Turn feature off ('0')>, "&
         "  on (0) <Turn feature on ('1')> "&
         " ), "&

      "PLLConfigValues ( "&
         " PLLsOff   (0b000) <All PLLs off>, "& -- Stop PLLs to save power
         " PLL1on    (0b001) <Only PLL1 on>, "&
         " PLL2on    (0b010) <Only PLL2 on>, "&
         " PLL12on   (0b011) <PLL1 & PLL2 on>, "&
         " PLL3on    (0b100) <Only PLL3 on>, "&
         " PLL13on   (0b101) <PLL1 & PLL3 on>, "&
```

```
         " PLL23on   (0b110) <PLL2 & PLL3 on>, "&
         " PLL123on  (0b111) <All PLLs on> "&
         " ), "&

-- IO voltage configuration. These input pins are read in the init_data
--   register because they must be set at power-up and checked before test.
      "IO_VSEL_Decodes  ( "&
         "B33_C33_L33  (0b00000) <BVdd=3.3V, CVdd=3.3V, LVdd=3.3V>, "&
         "B33_C33_L25  (0b00001) <BVdd=3.3V, CVdd=3.3V, LVdd=2.5V>, "&
         "B33_C33_L18  (0b00010) <BVdd=3.3V, CVdd=3.3V, LVdd=1.8V>, "&
         "B33_C25_L33  (0b00011) <BVdd=3.3V, CVdd=2.5V, LVdd=3.3V>, "&
         "B33_C25_L25  (0b00100) <BVdd=3.3V, CVdd=2.5V, LVdd=2.5V>, "&
         "B33_C25_L18  (0b00101) <BVdd=3.3V, CVdd=2.5V, LVdd=1.8V>, "&
         "B33_C18_L33  (0b00110) <BVdd=3.3V, CVdd=1.8V, LVdd=3.3V>, "&
         "B33_C18_L25  (0b00111) <BVdd=3.3V, CVdd=1.8V, LVdd=2.5V>, "&
         "B33_C18_L18  (0b01000) <BVdd=3.3V, CVdd=1.8V, LVdd=1.8V>, "&
         "B25_C33_L33  (0b01001) <BVdd=2.5V, CVdd=3.3V, LVdd=3.3V>, "&
         "B25_C33_L25  (0b01010) <BVdd=2.5V, CVdd=3.3V, LVdd=2.5V>, "&
         "B25_C33_L18  (0b01011) <BVdd=2.5V, CVdd=3.3V, LVdd=1.8V>, "&
         "B25_C25_L33  (0b01100) <BVdd=2.5V, CVdd=2.5V, LVdd=3.3V>, "&
         "B25_C25_L25  (0b01101) <BVdd=2.5V, CVdd=2.5V, LVdd=2.5V>, "&
         "B25_C25_L18  (0b01110) <BVdd=2.5V, CVdd=2.5V, LVdd=1.8V>, "&
         "B25_C18_L33  (0b01111) <BVdd=2.5V, CVdd=1.8V, LVdd=3.3V>, "&
         "B25_C18_L25  (0b10000) <BVdd=2.5V, CVdd=1.8V, LVdd=2.5V>, "&
         "B25_C18_L18  (0b10001) <BVdd=2.5V, CVdd=1.8V, LVdd=1.8V>, "&
         "B18_C33_L33  (0b10010) <BVdd=1.8V, CVdd=3.3V, LVdd=3.3V>, "&
         "B18_C33_L25  (0b10011) <BVdd=1.8V, CVdd=3.3V, LVdd=2.5V>, "&
         "B18_C33_L18  (0b10100) <BVdd=1.8V, CVdd=3.3V, LVdd=1.8V>, "&
         "B18_C25_L33  (0b10101) <BVdd=1.8V, CVdd=2.5V, LVdd=3.3V>, "&
         "B18_C25_L25  (0b10110) <BVdd=1.8V, CVdd=2.5V, LVdd=2.5V>, "&
         "B18_C25_L18  (0b10111) <BVdd=1.8V, CVdd=2.5V, LVdd=1.8V>, "&
         "B18_C18_L33  (0b11000) <BVdd=1.8V, CVdd=1.8V, LVdd=3.3V>, "&
         "B18_C18_L25  (0b11001) <BVdd=1.8V, CVdd=1.8V, LVdd=2.5V>, "&
         "B18_C18_L18  (0b11010) <BVdd=1.8V, CVdd=1.8V, LVdd=1.8V>, "&
         "Reserved     (others) <Reserved -- Do Not Use!> "&
         " ), "

-- Mnemonics for INIT_STATUS
   "InitCompletionValue  (Completed (0b00), "&
                          "Running (0b10), "&
                          "Stopped (0b01), "&
                        "NotStarted (0b11)), "&

   "ErrorCode (NoError (0b00), "&
               "  Err1 (0b01), "&
               "  Err2 (0b10), "&
               "  Err3 (0b11))";
```

B.8.19 Register fields description

The register description attributes define the possible hierarchical construction of a TDR, identifying fields within a TDR and such characteristics as the type of TDR cell used in each field, how the fields are reset and to what value, and what values should be written to or expected to be read from the fields. These fields may then be explicitly addressed by the Procedural Description Language (PDL) defined in Annex C.

These definitions may be included in the BSDL for a component, or in a BSDL user package body possibly provided by an IP supplier. The intent is to allow the IP or component designer to document the important characteristics of standard and public TDRs, or TDR segments, so that software can more easily be written to interact with the fields in the TDRs, and the results of operations involving the fields in the TDRs can be more easily predicted.

The register description attributes are optional. Including them in a component BSDL provides documentation of previously unknown structural details of the TDRs, making it practical for component and IP designers to support built-in test logic functions in board and system test. Among the standard TDRs, this capability allows support for programming the init_data register, which may vary from use to use in the board test environment, defining the types of resets controlled by the reset_select register, and allowing reasonably automated control of power or other domains and related excludable or selectable segments.

Like the register mnemonics attribute, the register description attributes may be defined in the BSDL or in a user package body. One exception is that the boundary-scan register can only be defined in the BSDL because the port names are required for the definition of the cells in the boundary segments or register. See B.8.14. Note that multiple occurrences of the register description attributes are allowed, which supports multiple sources of user package files with no requirement to merge them.

The **REGISTER_FIELDS** attribute defines and names fields within a register or register segment. The total length of the register or register segment is stated explicitly. By selecting specific bits of the register or register segment, the **REGISTER_FIELDS** attribute allows the definition of fields in a much larger register or register segment, and the definition of fields where the bits of the field are not contiguous in the register or register segment.

B.8.19.1 Specifications

Syntax

<register fields description> ::= **attribute REGISTER_FIELDS of** <target> **is**
<register fields string> <semicolon>

<register fields string>::= <quote> <register field list> { <comma> <register field list> } <quote>
<register field list>::= <reg or seg name> <left bracket> <reg or seg length> <right bracket>
<left paren> <register fields> <right paren>
<reg or seg name>::= <TDR> | <segment name>
<TDR>::= **BOUNDARY** | **BYPASS** | **DEVICE_ID** | **TMP_STATUS** |
ECID | **INIT_DATA** | **INIT_STATUS** | **RESET_SELECT** | <design specific TDR name>
<segment name>::= <VHDL identifier>
<design specific TDR name>::= <VHDL identifier>
<reg or seg length>::= <integer>
<register fields>::= <left paren> <register field element> <right paren>
{ <comma> <left paren> <register field element> <right paren> }
<register field element>::= <register field> | <prefix statement>
<register field>::= <extended field name> <field length> **is** <bit list and options>
<extended field name> ::= <prefix string> <field name>
<prefix string>::= { <prefix identifier> <period> }
<field name>::= <VHDL identifier>
<field length>::= <left bracket> <integer> <right bracket>
<bit list and options>::= <bit list> { <field options> }
<field options> ::= <type assignment> | <value assignment> | <reset assignment>
<bit list>::= <left paren> [<bit field> { <comma> <bit field> }] <right paren>
<bit field>::= <range> | <integer>
<prefix statement>::= **PREFIX** <integer> <prefix name>
<prefix name> ::= <prefix identifier> | <minus sign>

Rules

a) When the <target> of a <register fields statement> is an <entity target>, and when the <reg or seg name> also appears as a <register> element in a <register association> in the <register access description> or is otherwise defined by this standard, the <reg or seg length> shall:
   1) Match the specified length of a register defined in this standard (e.g., length 32 for DEVICE_ID).
   2) Match the length of the register as specified in the <register access description>.
   3) Define the length of the register when the length in the <register access description> is deferred (*).

b) When the <target> of a <register fields statement> is a <package target> or when the <reg or seg name> does not appear as a <register> element in a <register association> in the <register access description> or is not otherwise defined by this standard, the length of <reg or seg name> shall be the <reg or seg length>.

c) The bits within a <register field list> shall be numbered from the value of <reg or seg length> minus one down to zero, regardless of any ordering within the <bit list>, and with the bit numbered zero closest to TDO.

d) If one or more of the <prefix statement> is included in a <register field list>, then the <integer> value of each such statement shall be interpreted as the hierarchical level in ascending <integer> value order from left to right, and shall have effect until another <prefix statement> with the same or lower <integer> value is encountered or until the end of the <register field list>.

e) If one or more of the <prefix statement> is included in a <register field list>, then, at any time an <extended field name> is specified, one effective <prefix statement> shall have an <integer> value of 0, and all integer values from 0 to the highest value currently active shall have been specified by a <prefix statement>.

f) If a <prefix statement> is included in a <register field list> with a <prefix name> of <minus sign>, then the hierarchical level, as specified by the <integer> value, and all higher numbered hierarchical levels, shall be unspecified.

g) If one or more <prefix statement> is included in a <register field list>, then the <prefix string> of each such effective statement shall be prepended in ascending order of the <integer> value, with a period as separator, to the prefix string and field name in all subsequent <register field> statements to form the extended field name.

h) An <extended field name> shall be unique within a given <BSDL description> or <user package>.

   NOTE 1—Since the Standard Package is used in every BSDL and BSDL Package, names in that package are “global” and are reserved.

i) An <extended field name> shall be composed of zero or more <prefix identifier> fields in parent-to-child hierarchical order, followed by a <field name>, all separated by periods.

j) The <field length> shall be equal to the total number of bits in the <bit list> for that field.

k) If the <field length> is zero, then the <bit list> shall be an empty list “()”.

l) All bit numbers listed in the <bit list>, either as an <integer> or within a <range>, shall be unique within the <bit list> and numerically less than the associated <reg or seg length>.

   NOTE 2—Any bit of a test data register may be referenced in more than one <register field> but not more than once within a <register field>.

B.8.19.2 Description

The **REGISTER_FIELDS** attribute associates names with subsets of bits of a register named in the **REGISTER_ACCESS** attribute, or defined in this standard, or with subsets of bits of a newly named register segment when the name is not listed in the **REGISTER_ACCESS** attribute nor defined in this standard. Note that a segment may be excludable per 9.4 or not excludable.

The subset of bits for a field may be a single bit up to the entire register or register segment. The bits of a field may be listed in contiguous order or randomly ordered. All the bits of the register or register segment may be included in

the set of fields, or only a sparse subset of a longer register. Register or register segment bits may appear in more than one field. The register or register segment is of length L, and the range of the register or register segment bits for assigning to fields is assumed to be (L-1 DOWNTO 0), where bit 0 is closest to TDO in the scan chain.

In a similar way, the new named field is of length L, and when referring to bits of the field, they are assumed to have a range of (L-1 DOWNTO 0), and to be associated one-to-one, in order left to right, to the list of bits of the register or register segment in the field definition. The field may then be referred to by name, and any value assigned to the bits of the field will be mapped to the bits of the register or register segment.

For example, a simple register or register segment with all bits assigned to fields would look like the following. The numbers after the "IS" are the bit numbers from the register or register segment range.

NOTE—For clarity, no assignments are shown in these small examples. See B.8.20 for a description of and examples with assignments.

```
attribute REGISTER_FIELDS of MEMB_example : package is
"MBist [6] ( "&
  "(Algorithm  [3] IS  (5 DOWNTO 3 )), "&
  "(Command    [1] IS  (2)          ), "&
  "(Status     [2] IS  (1 DOWNTO 0) )  "&
  "      )";
```

There is significant and deliberate redundancy in the specification. The specified length of the register or segment name (MBist) and the range of (5 DOWNTO 0) is used to check the bit assignments of each field and to place each bit of each field within the register or register segment. In this case, a 6-bit segment was divided into three fields, all contiguous, and all bits of the segment were assigned to a field.

The bits of the register or register segment need not all be used, nor must all bits assigned to a field be contiguous, as shown in the following.

```
attribute REGISTER_FIELDS of init_example : entity is
 "init_seg [56] ( "&
    -- First 36 bits are not defined.
    "(Observe_IO_VSEL [5] IS (19 downto 15)), "&
    "(PLLPower [3] IS (5,3,1)), "&
    -- Rest of IP configuration register
    "(IPConfig [12] IS (14 downto 6,4,2,0)) )";
```

Bits of the register or register segment may also be assigned to more than one field. For example, the previous example may be altered as shown in the following. Bits 5, 3, and 1 are assigned to two fields. This might be done to allow setting the entire field to a default value and then overriding that default value just for the bits of the PLLPower field.

```
attribute REGISTER_FIELDS of init_example : entity is
 "init_seg [56] ( "&
    -- First 36 bits are not used.
    "(Observe_IO_VSEL [5] IS (19 downto 15)), "&
    "(PLLPower [3] IS (5,3,1)), "&
    -- All of IP configuration register
    "(IPConfig [15] IS (14 downto 0)) )";
```

Normally, BSDL only deals with the scan chain in terms of a chain name and the bit positions within the chain. There is another form of hierarchy that can be important to the designer or test engineer when performing debug or diagnosis: the logical hierarchy of the register field bits. The chain may cross many logical hierarchy boundaries in the logical netlist without that fact being recorded. However, for understanding of the effect of a value placed in or captured by a specific field, the engineer may want to know the logical hierarchy.

One simple way to put the logical hierarchy into the register field definitions is to simply include the entire logical hierarchy in the field name. There are two problems with this, as follows: First, a field name is defined as a <VHDL identifier>, which does not allow any of the characters normally used to delimit the hierarchy; and second, such a name is often such a long string as to be unwieldy. The remedy for both of these problems is the **PREFIX** keyword and values in the **REGISTER_FIELDS** attribute.

The **PREFIX** keyword takes two parameters: a number, which is the level within the logical hierarchy, and a logical hierarchy name for that level, which is normally the logical instance name of that level from the design source. The fully qualified logical hierarchy for a register field is each of the prefix values, in numerical order, concatenated with a period between them, and finally a period and the field name. This logical hierarchy is just an extension of the field name and is completely independent of the register assembly hierarchy.

```
attribute REGISTER_FIELDS of My_Chip : Entity is
  "Internal_Chain_33 [511] ( "&
    "(Prefix 0 TOP), "&
    "(Prefix 1 n1), "&
    "(Prefix 2 IO_Lane1), "&
    "(Prefix 3 RX_cntl), "&
    "(Prefix 4 deskew), "&
    "(Sequencer.state_machine_reg_Q [6] IS (510,509,508,501,502,505)), "&
    "(Command_reg_Q [8] IS (504,498,500,503,499,495,507,506)), "&
    "(Status_reg_Q [5] IS (493,492,497,494,496)), "&
    ...
    "      )";
```

In this example, note that when any level PREFIX is changed, the values of the lower numbered (higher in the hierarchy, containing) levels are retained and higher numbered (lower in the hierarchy, contained) levels are deleted. The order in which the levels are specified is therefore important. The extended field names are unique, even though the field name itself may be the same. The full extended field name for the three fields defined above would be:

```
TOP.n1.IO_Lane1.RX_cntl.deskew.Sequencer.state_machine_reg_Q
TOP.n1.IO_Lane1.RX_cntl.deskew.Command_reg_Q
TOP.n1.IO_Lane1.RX_cntl.deskew.Status_reg_Q
```

B.8.19.3 Examples

Referring to the register mnemonics that were defined in B.8.18.3, the following extensive example of init_data and init_status registers are defined flat (without hierarchy) using the **REGISTER_FIELDS** attribute.

NOTE—This extended example shows assignments for completeness; see B.8.20 for specifications and a description of the assignments.

```
attribute REGISTER_FIELDS of INIT_Example : entity is

-- Register Fields for INIT_DATA register
--   Bits (144 downto 140) observe the IO_VSEL pins, and are read-only bits.
--   All other bits are write-only.

-- The value captured from the IO_VSEL pins must be verified to be set
--   correctly prior to test with the Observe_IO_VSEL bits when the
--   init_data register is scanned.

  "init_data [180] ( "&
     -- Unused bits do not need to be referenced.
     -- Observed in init_data because must be set at power-up.
     "(Observe_IO_VSEL [5] IS (144 downto 140) "&
```

```
         "CAPTURES (IO_VSEL_Decodes (*)) NOPO), "&

      -- Enable PLLs
      "(PLLPower [3] IS (130,128,126) "&
         "CHRESET "&
         "RESETVAL(PLLConfigValues (PLL123on)) "&
         "DEFAULT (PLLConfigValues (PLL123on)) "&
         "SAFE    (PLLConfigValues (PLLsOff))  " &
         "NoPI), "&

      -- IP configuration register, see chip release documentation.
      -- No mnemonic provided, ******
      "(IPConfig [12] IS (139 downto 131,127,129,125) "&
         "CHReset "&
         "ResetVal(0xd29)  "&
         " Default(0xd29)  "&
         "    Safe(0xd29)), "&

      -- Transceiver Channels (18 in all)
      "(SerDesChannel_00 [3]   IS (124 downto 122) "&
         "DEFAULT (SerDes_Protocol(*)) nopi), "&
      "(SerDesChannelTX_00 [2] IS (121 downto 120) "&
         "DEFAULT (SerDes_TX_Outputs(*)) nopi), "&

      "(SerDesChannel_01 [3]   IS (119 downto 117) "&
         "DEFAULT (SerDes_Protocol(*)) nopi), "&
      "(SerDesChannelTX_01 [2] IS (116 downto 115) "&
         "DEFAULT (SerDes_TX_Outputs(*)) nopi), "&

-------------------------------------------------------------
-- Three additional channels removed for ease of reading.
-------------------------------------------------------------

      "(SerDesChannel_05 [3]   IS (99 downto 97) "&
         "DEFAULT (SerDes_Protocol(*)) nopi), "&
      "(SerDesChannelTX_05 [2] IS (96 downto 95) "&
         "DEFAULT (SerDes_TX_Outputs(*)) nopi), "&

      "(SerDesClkChannel1 [5] IS (94 downto 90) "&
         "DEFAULT (SerDesClockSettings (125Mhz)) nopi), "&

      "(SerDesClkChannel2 [5] IS (89 downto 85) "&
         "DEFAULT (SerDesClockSettings (125Mhz)) nopi), "&

      "(SerDesChannel_06 [3]   IS ( 85 downto  87) "&
         "DEFAULT (SerDes_Protocol(*)) nopi), "&
      "(SerDesChannelTX_06 [2] IS ( 86 downto  85) "&
         "DEFAULT (SerDes_TX_Outputs(*)) nopi), "&

-------------------------------------------------------------
-- Nine additional channels removed for ease of reading.
-------------------------------------------------------------

      "(SerDesChannel_16 [3]   IS ( 34 downto  32) "&
         "DEFAULT (SerDes_Protocol(*)) nopi), "&
      "(SerDesChannelTX_16 [2] IS ( 31 downto  30) "&
         "DEFAULT (SerDes_TX_Outputs(*)) nopi), "&
```

```
      "(SerDesChannel_17 [3]   IS ( 29 downto  27) "&
         "DEFAULT (SerDes_Protocol(*)) nopi), "&
      "(SerDesChannelTX_17 [2] IS ( 26 downto  25) "&
         "DEFAULT (SerDes_TX_Outputs(*)) nopi), "&

      "(SerDesClkChannel3 [5] IS (24 downto 20) "&
         "DEFAULT (SerDesClockSettings (125Mhz)) nopi), "&

      -- Power level supplied to SerDes internal gates;
      --   No default value or ports specified
      "(SerDesCXVddSel [1] IS (19) "&
         "DEFAULT (SerDesCXVddSelLevel (*)) nopi), "&

      -- Power up SerDes Test Receivers to enable correct SAMPLE operation
      "(SerDesSamplePowerUp [1] IS (18) "&
         "DEFAULT (Switch (off)) nopi), "&

      -- 2^^10 possible decodes, which apply to 1->150 pins.
      --   See Reference Manual, Clause 17.8.
      "(DDRTermSel [10] IS (17 downto 8) nopi), "&

      -- Reserved Field, which must be set to "00000000"
      "(ReservedField [8] IS (7 downto 0) "&
         "DEFAULT (0) " &
         "   SAFE (0) nopi) " &
               "), " &

-- Register Fields for INIT_STATUS register -
--   These bits are read-only per the standard
   "init_status [4] ( "&
      "(INITCompletionStatus [2] IS (0 to 1) nopo), "&
      "(INITErrorStatus [2]      IS (2 to 3) nopo) )" ;
```

A longer version of the PREFIX example is next. First is a simplified listing of an internal scan chain from an ATPG tool, with the number being the position relative to scan-out:

```
...
510  TOP.n1.IO_Lane1.RX_cntl.deskew.Sequencer.state_machine_reg_Q(5)
509  TOP.n1.IO_Lane1.RX_cntl.deskew.Sequencer.state_machine_reg_Q(4)
508  TOP.n1.IO_Lane1.RX_cntl.deskew.Sequencer.state_machine_reg_Q(3)
507  TOP.n1.IO_Lane1.RX_cntl.deskew.Command_reg_Q(1)
506  TOP.n1.IO_Lane1.RX_cntl.deskew.Command_reg_Q(0)
505  TOP.n1.IO_Lane1.RX_cntl.deskew.Sequencer.state_machine_reg_Q(0)
504  TOP.n1.IO_Lane1.RX_cntl.deskew.Command_reg_Q(7)
503  TOP.n1.IO_Lane1.RX_cntl.deskew.Command_reg_Q(4)
502  TOP.n1.IO_Lane1.RX_cntl.deskew.Sequencer.state_machine_reg_Q(1)
501  TOP.n1.IO_Lane1.RX_cntl.deskew.Sequencer.state_machine_reg_Q(2)
500  TOP.n1.IO_Lane1.RX_cntl.deskew.Command_reg_Q(5)
499  TOP.n1.IO_Lane1.RX_cntl.deskew.Command_reg_Q(3)
498  TOP.n1.IO_Lane1.RX_cntl.deskew.Command_reg_Q(6)
497  TOP.n1.IO_Lane1.RX_cntl.deskew.Status_reg_Q(2)
496  TOP.n1.IO_Lane1.RX_cntl.deskew.Status_reg_Q(0)
495  TOP.n1.IO_Lane1.RX_cntl.deskew.Command_reg_Q(2)
494  TOP.n1.IO_Lane1.RX_cntl.deskew.Status_reg_Q(1)
493  TOP.n1.IO_Lane1.RX_cntl.deskew.Status_reg_Q(4)
492  TOP.n1.IO_Lane1.RX_cntl.deskew.Status_reg_Q(3)
```

```
...
383  TOP.n1.IO_Lane2.RX_cntl.deskew.Sequencer.state_machine_reg_Q(5)
382  TOP.n1.IO_Lane2.RX_cntl.deskew.Sequencer.state_machine_reg_Q(4)
381  TOP.n1.IO_Lane2.RX_cntl.deskew.Sequencer.state_machine_reg_Q(3)
380  TOP.n1.IO_Lane2.RX_cntl.deskew.Command_reg_Q(1)
379  TOP.n1.IO_Lane2.RX_cntl.deskew.Command_reg_Q(0)
378  TOP.n1.IO_Lane2.RX_cntl.deskew.Sequencer.state_machine_reg_Q(0)
377  TOP.n1.IO_Lane2.RX_cntl.deskew.Command_reg_Q(7)
376  TOP.n1.IO_Lane2.RX_cntl.deskew.Command_reg_Q(4)
375  TOP.n1.IO_Lane2.RX_cntl.deskew.Sequencer.state_machine_reg_Q(1)
374  TOP.n1.IO_Lane2.RX_cntl.deskew.Sequencer.state_machine_reg_Q(2)
373  TOP.n1.IO_Lane2.RX_cntl.deskew.Command_reg_Q(5)
372  TOP.n1.IO_Lane2.RX_cntl.deskew.Command_reg_Q(3)
371  TOP.n1.IO_Lane2.RX_cntl.deskew.Command_reg_Q(6)
370  TOP.n1.IO_Lane2.RX_cntl.deskew.Status_reg_Q(2)
369  TOP.n1.IO_Lane2.RX_cntl.deskew.Status_reg_Q(0)
368  TOP.n1.IO_Lane2.RX_cntl.deskew.Command_reg_Q(2)
367  TOP.n1.IO_Lane2.RX_cntl.deskew.Status_reg_Q(1)
366  TOP.n1.IO_Lane2.RX_cntl.deskew.Status_reg_Q(4)
365  TOP.n1.IO_Lane2.RX_cntl.deskew.Status_reg_Q(3)
...
```

Next is the **REGISTER_FIELDS** attribute as a tool might have reconstructed the fields from the above listing. The extended field names are unique, even though the other level(s) may be the same due to multiple copies of part of the hierarchy within a single register.

```
attribute REGISTER_FIELDS of My_Chip : Entity is
  "Internal_Chain_33 [1511] ( "&
    "(Prefix 0 TOP), "&
    "(Prefix 1 n1), "&
    ...
    "(Prefix 2 IO_Lane1), "&
    "(Prefix 3 RX_cntl), "&
    "(Prefix 4 deskew), "&
    "(Sequencer.state_machine_reg_Q [6] IS (510,509,508,501,502,505)), "&
    "(Command_reg_Q [8] IS (504,498,500,503,499,495,507,506)), "&
    "(Status_reg_Q [5] IS (493,492,497,494,496))  "&
    "(Prefix 2 IO_Lane2), "& -- Prefix 3 & 4 are deleted
    "(Prefix 3 RX_cntl), "&  --  and have to be re-defined
    "(Prefix 4 deskew), "&   --  even if they don't change

    "(Sequencer.state_machine_reg_Q [6] IS (383,382,381,374,375,378)), "&
    "(Command_reg_Q [8] IS (377,371,373,376,372,368,380,379)), "&
    "(Status_reg_Q [5] IS (366,365,370,367,369)), "&
    ...
    "      )";
```

B.8.20 Register field assignment description

For each field, the type of TDR cells used may be described by type assignment keywords. For each field, values that the register will assume under specific conditions may be assigned using either binary patterns or mnemonics. These assignments can be used in the **REGISTER_FIELDS** attribute or the **REGISTER_ASSEMBLY** attribute, or both, as defined in B.8.19 and B.8.21.

B.8.20.1 Specifications

Syntax

<value assignment>::= <value keyword> <left paren> <assignment> <right paren>
<value keyword>::= **CAPTURES** | **DEFAULT** | **SAFE** | **RESETVAL** | <user extension>
<user extension>::= **USER** <colon> <user keyword>
<user keyword>::= <VHDL identifier>
<assignment>::= <assignment value> | <asterisk> | <minus sign>
<assignment value>::= <binary pattern> | <hex pattern> | <decimal pattern> | <mnemonic association>
<mnemonic association>::= [**PACKAGE** <package hierarchy> <colon>] <mnemonic group name>
<left paren> <mnemonic default> <right paren>
<mnemonic default>::= <mnemonic identifier> | <asterisk> | <minus sign>

<type assignment>::= **NOPI** | **NOPO** | **NOUPD** | **MON** | **PULSE0** | **PULSE1** | **DELAYPO** |
NORETAIN | **SHARED** | <user extension>

<reset assignment>::= **PORRESET** | **TRSTRESET** | **TAPRESET** | **CHRESET** |
DOMPOR | **HIERRESET** | <local reset assignment>
<local reset assignment>::= <reset type> <left paren> <reset ident> <right paren>
<reset type::= **RESETOUT** | **RESETIN**
<reset ident>::= <VHDL identifier>

<domain assignment>::= <association type> <left paren> <association name> <right paren>
<association type>::= **DOMAIN** | **DOMAIN_EXTERNAL** | **SEGMENT**
<association name>::= <VHDL identifier>

Rules

a) A <value assignment> defined both in a <register assembly statement> and in the <register fields statement> of a hierarchical register description shall not change the value assigned in the **REGISTER_FIELDS** attribute for either of the value keywords **CAPTURES** or **RESETVAL** except when:

1) The value in the <mnemonic default> assignment was deferred (*) or don't care (-).
2) The bit to be changed to 0 or 1 was defined in the <register field statement> as X.

NOTE 1—**DEFAULT** and **SAFE** values can be modified as desired as they do not represent the structural details of the register.

b) If a <value keyword> of **RESETVAL** is provided for a field, a <reset assignment> shall be provided for the field within the hierarchical register description.

c) Multiples of <value assignment>, if specified within a <bit list and options>, <instance and options>, or <field and options>, shall contain no more than one of each <value keyword> option.

d) The most significant 1 bit in the binary equivalent value of <assignment value> shall be located within the specified length of the associated register field.

NOTE 2—This allows a HEX value to be assigned to a register field with a length that is not an exact multiple of 4, as long as the excess most significant bits are 0. This can also be thought of as the equivalent decimal value being less than 2**^{length}.

e) If the binary equivalent value of <assignment value> has fewer bits than the specified length of the associated register field, the bits of the <value assignment> shall be right justified (closest to TDO) in the field and the remaining high-order bits set to 0 for the **DEFAULT** and **SAFE** keywords and to X for the **CAPTURES** or **RESETVAL** keywords.

f) The following combinations of <value keywords> and <type assignments> shall not be used:
   1) CAPTURES with NOPI.
   2) DEFAULT or SAFE with NOPO.

g) Multiples of <type assignment>, when specified within a <bit list and options>, <instance and options>, or <field and options>, shall contain no more than one of each <type assignment> option and the following combinations shall not be used:
   1) MON, PULSE1, PULSE0, or DELAYPO with either NOUPD or NOPO.
   2) PULSE1 with PULSE0.

h) No more than one <reset assignment> option of **PORRESET, TRSTRESET**, **TAPRESET**, **CHRESET,** or **HIERRESET** shall be specified within a <bit list and options>, <instance and options>, or <field and options>.

   NOTE 3—**DOMPOR**, **RESETOUT**, and **RESETIN** can be be used in combination with any one of the other <reset assignment> options.

i) For excludable segments controlled by an on-chip domain controller, at least one **DOMCTRL** field per domain and at least one **SEGSEL** field per excludable segment:
   1) Shall be instantiated in an <instance and value> of a <register assembly list>.
   2) Shall all have a <domain assignment> of **DOMAIN**.
   3) Shall all have the same <association name>.

j) For excludable segments controlled by an off-chip domain controller, at least one **SEGSEL** field per excludable segment shall:
   1) Be instantiated in an <instance and value> of a **REGISTER_ASSEMBLY** attribute.
   2) Have a <domain assignment> of **DOMAIN_EXTERNAL**.
   3) Have a **REGISTER_ASSOCIATION** attribute provided with a <port list> associating this field with the <port ID> of one or more controlling ports.

k) All of the set of **SEGSEL**, **SEGSTART**, and **SEGMUX** field instances controlling a single excludable segment shall have a <domain assignment> of **SEGMENT** and have the same <association name>.

l) The <reset assignment> keyword of **DOMPOR** shall only be used with a field contained in an excludable segment; and furthermore, the **SEGSEL** controlling the containing excludable segment shall have a <domain assignment> of **DOMAIN** or **DOMAIN_EXTERNAL**.

m) The <reset assignment> keyword of **HIERRESET** shall only be used with a field contained in an excludable segment.

n) The <reset assignment> keyword of **RESETOUT** shall only be used with a single bit field, which:
   1) Shall have one and only one of the other <reset assignment> keywords.
   2) Shall have a **RESETVAL** <values assignment> keyword with an <assignment value> of 1.
   3) Shall have a <type assignment> keyword of either **PULSE1** or **PULSE0** (see Figure 9-11).
   4) Shall have a <reset ident> that is unique among all uses of **RESETOUT**.

o) The <reset assignment> keyword of **RESETIN** shall have a <reset ident> that matches the <reset ident> of a **RESETOUT** <reset assignment> keyword on another cell.

p) For a single bit register field with both **RESETOUT** and **RESETIN** keywords, the reset output associated with the <reset ident> of the **RESETOUT** keyward shall not, directly or indirectly, cause the activation of the reset input assosiated with the **RESETIN** keyword.

   NOTE 4—This means that if one local reset resets another, they must be in a strict hierarchical tree without any loops.

q) When the **DOMAIN_EXTERNAL** <domain assignment> is used with a **SEGSEL** field instance in a <register assembly list>, no **DOMCTRL** field instance shall have the same <association name>.

r) Each instance of the **DOMCTRL** and **SEGSEL** fields in a <register assembly list> shall have a <reset assignment>, and shall not have a <type assignment>, <value assignment>, or <field selection assignment>.

s) No instance of the **SEGSTART** or **SEGMUX** fields in a <register assembly list> shall have a <type assignment>, <value assignment>, or <reset assignment>.

t) Only instances of the **SEGSEL**, **SEGSTART**, and **DOMCTRL** fields in a <register assembly list> shall have a <domain assignment>.

NOTE 5—**SEGSEL**, **SEGSTART**, **SEGMUX**, and **DOMCTRL** fields are defined in the Standard BSDL Package Body; see B.9.

u) The <mnemonic group name> used in a <value assignment> shall be defined in a <register mnemonics description> statement contained in the current BSDL or package, or in a user-supplied package referenced by a Use statement in the current file.

v) Within a <value assignment>, any <value keyword> of **DEFAULT**, **SAFE**, or **RESETVAL** using a <mnemonic group name> shall use the same <mnemonic group name>.

NOTE 5—A <mnemonic group name> used in a <value assignment> establishes a link between the register field and the <mnemonic group name>. The <mnemonic group name> used with **DEFAULT**, **SAFE**, or **RESETVAL** provides mnemonic values for writing to the field while the <mnemonic group name> used with **CAPTURES** provides mnemonic values for reading from the field. The same <mnemonic group name> or different ones may be used for reading and writing.

w) The <mnemonic identifier> used in a <value assignment> shall be defined, and a member of the mnemonics group named in the assignment shall be associated with a <binary pattern>, <hex pattern>, or <decimal pattern>.

NOTE 6—This means that a <mnemonic identifier> associated with the **others** keyword cannot be used in a <value assignment>.

x) An asterisk (*) used as the <mnemonic default> for the <mnemonic group name> in a <value assignment> shall associate the <mnemonic group name> with the field and indicate that the value is required, but deferred.

y) A minus-sign (-) used as the <mnemonic default> for the <mnemonic group name> in a <value assignment> shall associate the <mnemonic group name> with the field without assigning a value.

Recommendations

z) The **SAFE** value assignment should be used for default values when the component is in test mode with the system logic held in a safe state.

aa) The **DEFAULT** value assignment should be used for default values when the component is in mission mode.

bb) Any field with a <reset assignment> keyword of **RESETOUT** should also have a <reset assignment> of one of **CHRESET**, **DOMPOR**, **PORRESET**, or **RESETIN**.

NOTE 7—Use of these <reset assignment> keywords ensures that the fields are both reset at power-up and that resets during test operation are independent of the TAP controller *Test-Logic-Reset* state.

cc) Any field with a <reset assignment> keyword of **RESETIN** should have its reset implemented with synchronous reset techniques rather than with asynchronous reset techniques.

NOTE 8—This is to meet common IC Design For Test guidelines that asynchronous resets not be generated solely from internal logic.

B.8.20.2 Description

Values to be written or read may be assigned to a named field using PDL (see Annex C) or other language capable of reading and using BSDL and performing scans of TDRs. Fixed or default values also may be assigned with assignment keywords in each field definition.

When mnemonics are used as the values to be written or read, an association is defined between the TDR field and a specific mnemonic group. The <mnemonic group name>, if any, used with **DEFAULT**, **SAFE**, or **RESETVAL** keywords (only one mnemonic group can be used with these three keywords for a TDR field) is interpreted as providing mnemonic values for writing to the field; the <mnemonic group name>, if any, used with **CAPTURES** is interpreted as providing mnemonic values to be captured and read from the TDR field.

Different keywords describe the structure of the cells of the TDR field (which help in understanding and debugging interactions) and specify values to be shifted into or out of the register under specific circumstances. For clarity, the structural keywords (type and domain) are discussed first, followed by the value keywords.

The boundary-scan register is structurally defined with an ordered list of required values in the **BOUNDARY_REGISTER** or **BOUNDARY_SEGMENT** attributes (see B.8.14). Here, TDRs are specified with optional keywords or keyword-value pairs. Structural definition of the register fields in the BSDL or Packages is not required, but the information may simplify use and debug of the TDR, and simplify test programming and application.

The default TDR structure, if none of the type assignment keywords are used, is a *Capture-Shift-Update* cell (see Figure 9-6) without the optional reset to the update stage. The keywords **NOPI**, **NOPO**, and **NOUPD** restrict the capabilities of the scan-capture-update cells defined in 9.2, as shown in Figure 9-7 through Figure 9-9. The **PULSE0**, **PULSE1**, and **MON** keywords add the capabilities illustrated in Figure 9-10 and Figure 9-11. The **DELAYPO** keyword adds the capability for a delayed output, normally to avoid a race condition, as shown in Figure 9-19.

Whether or not the cell can be asynchronously reset, and the type of reset signal connected, is defined by the mutually exclusive keywords **PORRESET**, **TRSTRESET**, **TAPRESET**, and **CHRESET**. There is also the **HIERRESET** keyword for excludable fields, which are reset by the segment-switching cell, and **DOMPOR** keyword for excludable fields in a power domain that are reset when the domain is powered up. Whether the reset applies to the shift stage or the update stage is determined by the **NOUPD** keyword, not allowing a description of a TDR cell that resets both stages.

Additionally, it is possible to generate local reset signals from a single bit register field. This local reset field is required to have a reset specified for it, and when the local reset field is reset, the output from the field is activated to reset the cells that it controls. This means that a global reset, such as **TRSTRESET**, will cause all fields with any reset specified to be reset simultaneously; yet by scanning a bit into this field, only the locally controlled cells will be reset. The single bit local reset field would have the **RESETOUT** keyword and a reset identifier. Every field reset by a local reset would have the **RESETIN** keyword with the reset identifier that identifies the field controlling its reset. It is possible for a local reset field to have both the **RESETIN** and **RESETOUT** keywords, with different reset identifiers, so that hierarchical local resets are allowed. This distribution must be a strict tree with no loops.

Finally, the **SHARED** type keyword indicates that the TDR cell is shared with mission logic and should never be scanned when in mission mode. This is a structural characteristic of the register field, but primarily it affects test programs.

The **RESETVAL** keyword indicates both the fixed value of the field after the reset, and that a reset signal is required (as specified by the **PORRESET**, **TRSTRESET**, **TAPRESET**, **CHRESET**, **HIERRESET**, **DOMPOR**, and **RESETIN** keywords). The **CAPTURES** keyword indicates that the field captures one or more fixed bits. The domain keywords are used only with the predefined DomCtrl and SegSel cells and are used to define the fixed relationship among the excludable segments, the domain-control cell, and the domain controller.

The TDR cell structural <type assignment> keyword definitions are:

NOPI (No Primary Input) The TDR contents do not change during the *Capture-DR* TAP controller state (the cell does not capture). See Figure 9-7 and Figure 9-9 for examples of a TDR cell that would be described by this keyword.

NOPO (No Primary Output) The TDR contents do not affect any test or functional logic. See Figure 9-8 and Figure 9-9, which would be described by this keyword if the optional PO was not implemented.

NOUPD (No Update Stage) The parallel outputs, if any, of the register change during the *Shift-DR*, not the *Update-DR*, TAP controller state. If the field also has a reset assignment, then the reset applies to the shift stage. **NOPO** implies **NOUPD**. See Figure 9-8 and Figure 9-9.

MON (Self-Monitoring) The shift-capture stage captures the value in the Update stage, allowing verification of the value currently being driven on PO. **MON** implies **NOPI**. **MON** is incompatible with **NOUPD** and **NOPO**. See Figure 9-10 and Figure 9-11.

PULSE1 When a 1 is shifted into the cell, the output will go high (1) for a single TCK cycle after the *Update-DR* TAP controller state and then return to low (0). **PULSE1** is incompatible with **NOUPD** and **NOPO**. See Figure 9-11.

PULSE0 When a 1 is shifted into the cell, the output will go low (0) for a single TCK cycle after the *Update-DR* TAP controller state and then return to high (1). **PULSE0** is incompatible with **NOUPD** and **NOPO**. See Figure 9-11.

DELAYPO Any change in state of the output will occur up to two TCK cycles after the update stage changes state. This is required for all of the fields that control excludable or selectable segments, to avoid race conditions on update actions, but it may be used to document any register field that includes such a delay.

NORETAIN The register field cells may not hold their value when excluded or not selected, either as a selectable register segment or simply not selected for scan by the active instruction.

SHARED The register field cells are shared with mission mode logic and if scanned during mission mode could interfere with mission mode operation. Also, the data scanned into such register fields cannot be assumed to be held during mission mode.

The structural <reset assignment> keyword definitions are:

PORRESET (Power-on or Power-up Reset) The register contents and parallel outputs change in response to the on-chip POR signal, also used to reset the TAP controller (see 6.1.3).

TRSTRESET The register contents and parallel outputs change in response to the TRST* TAP port, also used to reset the TAP controller (see 6.1.3).

TAPRESET The register contents and parallel outputs change when the TAP controller enters the *Test-Logic-Reset* state, regardless of the state of the TMP controller, if it is provided. Either an on-chip POR or TRST* TAP port assertion will also force this reset.

CHRESET (Clamp-hold Reset) The register contents and parallel outputs change when the TAP controller enters the *Test-Logic-Reset* state and the Persistence controller is in the *Persistence-Off* state. Either an on-chip POR or a TRST* TAP port assertion will also force this reset. Note that the use of this keyword is an error if the optional TMP controller is not provided.

DOMPOR (Domain Power-on or Power-up Reset) The parallel output of a **SEGSEL**, contained in an excludable segment that may be powered-down, will be reset in response to a power-up reset signal local for the domain being powered-up. Note that this only occurs at power-up for the domain. (See Figure 9-16.)

HIERRESET (Hierarchical Reset) The parallel output of a **SEGSEL**, contained in an excludable segment, is reset when the containing excludable segment is excluded. (See Figure 9-17.)

RESETIN (Local Reset) A register field that is reset by a local reset control cell having the same reset identifier as this keyword.

RESETOUT (Local Reset Control) The output of the single bit field (which must be of type **PULSE1** or **PULSE0**, have another of the <reset assignment> keywords specified in the hierarchical structure, and have a **RESETVAL** of 1) controls the reset of other register fields paired by having the same reset identifier as this keyword.

The structural <domain assignment> keyword definitions are:

DOMAIN The associated fields control or are controlled by the named domain, which has an on-chip controller.

DOMAIN_EXTERNAL The associated fields are controlled by the named domain, which does not have an on-chip controller.

SEGMENT Associates by the specified name the **SEGSEL** with a **SEGSTART** when the **SEGSEL** is not in the same TDR with the excluded segment.

The following is an example **REGISTER_FIELDS** attribute specifying different fundamental cell types for different fields, all without the optional reset.

```
attribute REGISTER_FIELDS of Reg_Types : package is
  "Regs [6] ( "&
    "(ReadWrite [1] IS (5)), "&           -- Figure 9-6
    "(WriteOnly [1] IS (4) NoPI), "&      -- Figure 9-7
    "(ReadOnly  [1] IS (3) NoPO), "&      -- Figure 9-8
    "(ShiftOnly [1] IS (2) NoPI NoPO), "&  -- Figure 9-9
    "(SelfMon   [1] IS (1) MON), "&       -- Figure 9-10
    "(Pulse1Mon [1] IS (0) PULSE1 MON) "& -- Figure 9-11
    "      )";
```

If none of the **NOPI**, **NOPO**, **NOUPD**, **MON**, **PULSE1**, or **PULSE0** keywords are specified, then the register is assumed to be a full capture-shift-update register without the optional reset of the update stage. Combinations of type keywords can define specific, predictable behavior that would otherwise not be able to be assumed. A register with both the **TAPRESET** and **NOPI** assignment keywords specified, for instance, would have an expected value supplied by the **RESETVAL** keyword that would be scanned out after the reset. On TDRs that have not been scanned, tools can predict what bits are set or cleared prior to scanning the TDR based on the reset assignments, and possibly the state of the Test Persistence controller. This knowledge can be used to minimize the number of scans used to initialize registers where some fields are already initialized.

The behavior defined by the **MON** keyword allows any register that does not capture other specific data to return its current state. This improves the testability and the ability to verify the behavior of the test data register, and is preferred over a simple **NOPI**, in most cases.

The two pulse keywords provide an ability to gate the transfer of data from the test data register to another register (possibly in the system logic) or otherwise activate a process that requires a strobe after the update stages of the test

data register are stable. For instance, a test data register may have address and data fields with update stages, and a single bit "write-enable" cell with the pulse behavior. The update stages prevent the rippling address and data from propagating, and the pulse on the write-enable will notify the target logic that the rest of the data are available and stable. With a normal test data register cell, it would require up to three scans to replicate this behavior: Scan the data with write-enable off, scan again with write-enable on, and again with write-enable off.

The **DOMAIN** and **DOMAIN_EXTERNAL** keywords define domain control for excludable segments of TDRs. Multiple segments within a single TDR, or segments in multiple TDRs, could all be part of a single domain on the component. The value assigned with the **DOMAIN** or **DOMAIN_EXTERNAL** keyword identifies the domain that each domain-control cell and each segment-select cell is associated with. All domain-control cells with the same domain name would typically be logically ORed to provide the override to the on-chip domain controller, and the response of the domain controller would be connected to all segment-select cells with that same domain name, as shown in Figure B-12 and the following example.

There are three additional value keywords used just for assigning values to be shifted into or out of the register: **DEFAULT**, **SAFE**, and **CAPTURES**. One additional value keyword is used to assign the value forced into the field by a reset: **RESETVAL**.

Not all of the keywords are compatible. Specifically, **CAPTURES** is incompatible with **NOPI**, and **DEFAULT** or **SAFE** are incompatible with **NOPO**. Conversely, if a **RESETVAL** value keyword is used, then one of the reset type keywords (**PORRESET**, **TRSTRESET**, **TAPRESET**, **CHRESET, HIERRESET**, **DOMPOR**, or **RESETIN**) is required. **DOMPOR** is allowed in combination with one of the other reset type keywords.

In the following example, a reset is added to the PLLPower and IPConfig fields and a deferred capture value to the Observe_IO_VSEL field from an earlier example. This reset will occur (and the registers set to the specified values) at POR, if a TRST* TAP port is active, or if the TAP controller is in the *Test-Logic-Reset* state and the Persistence controller is in the *Persistence-Off* state.

```
attribute REGISTER_FIELDS of init_example : entity is
 "init_seg [56] ( "&
    -- First 36 bits are not used.
    "(Observe_IO_VSEL [5] IS (19 downto 15) "&
    --Pins that must be set at power-up on the board.
       "Captures(IO_VSEL_decodes(*)) NoPO), "& -- See Mnemonic Example 3
    "(PLLPower [3] IS (5,3,1) CHReset ResetVal(0b111)), "&
    -- Rest of IP configuration register
    "(IPConfig [12] IS (14 downto 6,4,2,0) CHReset ResetVal(0xd29)) "&
                ")";
```

When specified in a **REGISTER_FIELDS** attribute, the structural type keywords (**NOPI**, **NOPO**, **NOUPD**, **MON**, **PULSE1**, **PULSE0**, **SHARED**), like the length of a TDR segment in a field definition, cannot change at a higher level instantiation in a **REGISTER_ASSEMBLY** attribute in a way that changes the description of the cell design. These keywords are not allowed for a segment in a **REGISTER_ASSEMBLY** attribute, and semantic checks will check for consistency of keyword use.

The values (and mnemonic group name) for **CAPTURES**, **DEFAULT**, **SAFE**, and **RESETVAL** are separate. Each type of value to be assigned requires a value that can be provided by one of the following (any of the keywords could be used in the examples):

— A specified value.
  Example: **CAPTURES** (<mnemonic group name> (<mnemonic identifier>)) or **CAPTURES** (0x23)
— An indication that a value is required, but deferred (not yet specified).
  Example: **CAPTURES**(<mnemonic group name> (*)) or **CAPTURES** (*)

— An indication that no value is required (not specified and don't care).
Example: **CAPTURES** (<mnemonic group name> (-)) or **CAPTURES** (-)

Use of the <mnemonic group name>, even if no <mnemonic identifier> is specified, associates that mnemonic group with that type of value for the field. The use of the asterisk character for a value requires that a real value be supplied prior to scanning the register. The use of the minus-sign character is essentially a "don't care" but is useful for assigning the mnemonic group to the register.

Any specific value for a **CAPTURES** keyword is to be interpreted as a fixed capture value, to be expected *every* time the field is scanned out. The **CAPTURES** and **NOPI** keywords are incompatible.

Any specific value for a **DEFAULT** keyword is to be interpreted as a default value for system or test circuit operation, which will be written to the field until replaced by a new value. **DEFAULT** and **NOPO** are incompatible.

Any specific value for a **SAFE** keyword is to be interpreted as a default value for when the system logic is to be held in an inactive state. Thus, a register for controlling a PLL might have a **DEFAULT** value that puts the PLL into proper functional operation, and the **SAFE** value might turn the PLL off and put it in bypass mode. If a register field is reserved and not supposed to be used, then it is good practice to specify a **SAFE** value for the field. **SAFE** and **NOPO** are incompatible.

When a value is specified as an asterisk (*), then the value is required but deferred. That is, it must be supplied at some point, and it is not a don't care, but the value cannot be determined yet. For instance, a value may be required to initialize a component, but the correct value depends on usage in the system and there is no safe default. The value may be supplied later in a **REGISTER_ASSEMBLY** attribute referencing the register segment containing the field, but if not, it is expected to be supplied in the PDL (or other language) scanning a value into this register at the start of test.

When a value is specified as a minus sign (-), then it is a "don't-care" and equivalent to a default pattern of all X. This will normally be used when a mnemonic group name is being associated with a field and there is no preferred or required value.

If none of these value keywords are provided, the default is no mnemonic association and a "don't care" value.

The value assignments **DEFAULT**, **SAFE**, and with certain restrictions, **CAPTURES** may be used with a hierarchical instance of a register segment to add or change assignments made where the register segment was defined. For instance, they could be specified, or in this case overridden, as follows.

```
attribute REGISTER_ASSEMBLY of INIT_Example : entity IS
   "init_data (" &
      "(init_tail IS init_seg), "&
      "(array SerDesChannel(0 to 2) IS Channel), "&
      "(SerDesClk_0 IS ChClock "&
         "DEFAULT (SerDesClockSettings (100Mhz)) "&
         "SAFE    (SerDesClockSettings (100MHz)) ))";
```

As mentioned, keywords that define structures that are assigned to a field must be provided where the field is originally defined and cannot be changed at hierarchical levels. The keywords **NOPI**, **NOPO**, **NOUPD**, and the **RESETVAL** keywords may not be used where a previously defined field is instantiated hierarchically. The reset keywords **PORRESET**, **TRSTRESET**, **TAPRESET**, **CHRESET**, and **RESETIN** may be used at any level of the hierarchy and one will be required at some level when the **RESETVAL** keyword has been used. An IP provider may provide a reset value for his TDR, for example, but might not know the type of reset to which it will be connected, so the **RESETVAL** must be defined at the field level, but the type of reset may be determined in the instantiation of the field.

Figure B-10 illustrates a simple register structure with four fields and nested local resets. Only specific bits of fields are reset, so additional bits of those fields are assigned a **RESETVAL** of X, implying that those bits are not reset. The **REGISTER_FIELDS** attribute for this example follows the figure. The choice of **CHRESET*** in this example is arbitrary, as it could have been any of the other reset keywords. When **CHRESET*** is activated, field A will be set to 1, which will be driven out as a 0 due to that cell having a **PULSE0** type keyword (see Figure 9-11 for an example cell implementation). Field A then causes field B and the first bit of field D to be set and cleared, respectively. Field B is also a local reset control, and it will set the first bit of field C. Both fields A and B will clear on the first falling edge of TCK after **CHRESET*** deactivates. Scanning a 1 into either field A or B will cause a reset pulse on the corresponding output(s).

Whether a bit is set or cleared, or even reset at all, is determined by the **RESETVAL** value for the field. While the symbology of Figure B-10 suggests asynchronous set and reset, it is recommended that synchronous techniques be used for fields B, C, and D to minimize chip test generation problems in the component.

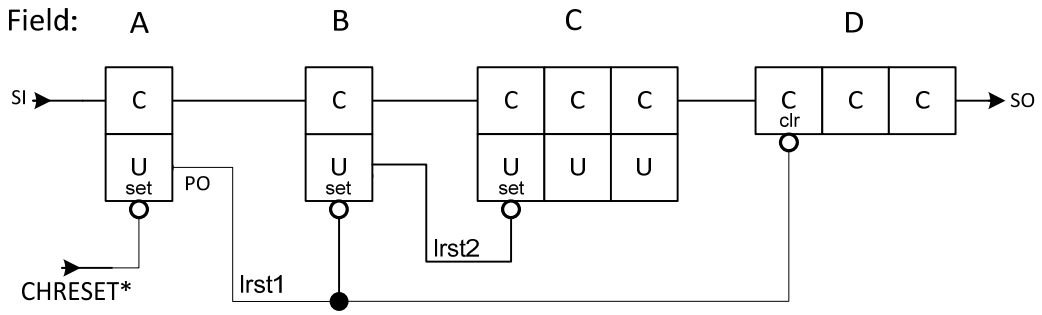


Figure B-10—Simple local reset structure

```
attribute REGISTER_FIELDS of MyReg : package is
  "Regs [8] ( "&
    "(A[1] IS (7) RESETOUT(lrst1) RESETVAL(0b1) CHRESET PULSE0 ) "&
    "(B[1] IS (6) RESETIN(lrst1) "&
                  "RESETOUT(lrst2) RESETVAL(0b1) PULSE0 ) "&
    "(C[3] IS (5 DOWNTO 3) RESETIN(lrst2) RESETVAL(0b1xx) ) "&
    "(D[3] IS (2 DOWNTO 0) RESETIN(lrst1) NOUPD RESETVAL(0b0xx) ) "&
    " )";
```

For value assignments that imply hard-wired values, namely, the capture or reset values, each level of the hierarchy must not change bits already specified as 0 or 1 in either a pattern or a mnemonic assignment. However, any bit originally defined as an X, or where the entire value was originally defined as “don’t-care” (-) or deferred (*), may be defined where the field is instantiated in a parent.

If a pattern or mnemonic value is specified in any **RESETVAL** or **CAPTURES** assignment when the field is defined, any bits set to 1 or 0 and any mnemonic association may not be changed at hierarchical levels above where they were originally defined. The value may be changed if it was deferred (*) or a don’t-care (-), or in any bit position originally specified as an X. In other words, in a multilevel hierarchy, each level must honor the bits already defined as 0 or 1 at a lower level for the **CAPTURES** and **RESETVAL** keywords. If the register field is designed such that a value is required, but the value may be defined by the user when the field is instantiated (through such means as input pins on a IP block are to be tied to create a capture or reset value), then the **REGISTER_FIELDS** assignment must be coded with the “deferred” token (*) in place of a specific value. This leaves the value open for the moment but allows it be filled in later by the same keyword with a specific value in a hierarchical instantiation.

DEFAULT and **SAFE** values may always be changed in hierarchical instantiations.

The following example expands on the previous example by adding **DEFAULT** and **SAFE** values to the PLLPower and IPConfig fields. For the PLLPower, the **SAFE** value powers the PLLs down. For the IPConfig field, all

assignments are to the same value to help ensure that the test software will not accidentally change it. In addition, while the IPConfig field would normally be write-only, it is used to capture an arbitrary and fixed value (hex "ABC") that allows the test software to verify that it is communicating with the correct register.

```
attribute REGISTER_FIELDS of init_example : entity is
 "init_seg [56] ( "&
    -- First 36 bits are not used.
    "(Observe_IO_VSEL [5] IS (19 downto 15) "&
       "Captures(*) NoPO), "&
    "(PLLPower [3] IS (5,3,1) "&
       "CHReset ResetVal(0b111) "&
       "         Default(0b111) "&
       "            Safe(0b000)), "&
    -- Rest of IP configuration register
    "(IPConfig [12] IS (14 downto 6,4,2,0) "
       "Captures(0xABC) "&
       "CHReset ResetVal(0xD29)  "&
       "         Default(0xD29)  "&
       "            Safe(0xD29)) "&
              ")";
```

The <field selection assignment> will be discussed and illustrated in the register assembly description.

B.8.21 Register assembly description

The **REGISTER_ASSEMBLY** attribute defines a register or register segment by concatenating register segments and fields in the order listed. The length of the assembled register or register segment is the sum of the lengths of the register segments included in the list. The register segments in the list may be defined in a **REGISTER_FIELDS** attribute or in a **REGISTER_ASSEMBLY** attribute, thereby providing a hierarchical register description. The top level of any hierarchy is a register named in the **REGISTER_ACCESS** attribute or defined in this standard.

Four special register fields are defined in the Standard BSDL Package Body **STD_1149_1_2013** in support of excludable segments, and these are intended for use in **REGISTER_ASSEMBLY** statements. The first two are the segment-select and segment-start cells named SegSel and SegStart, the third is the switching circuit named SegMux, and the fourth cell, called DomCtrl, is the domain-control cell, all as defined in 9.4.1.

Selectable segments are delineated, and the selection mechanism is defined, with keywords.

B.8.21.1 Specifications

Syntax

<register assembly description> ::= **attribute REGISTER_ASSEMBLY of** <target> **is**
    <register assembly string> <semicolon>

<register assembly string>::= <quote> <register assembly list>
    { <comma> <register assembly list> } <quote>
<register assembly list>::= <reg or seg name> <left paren> <register assembly elements> <right paren>
<register assembly elements>::= <left paren> <register element> <right paren>
    { <comma> <left paren> <register element> <right paren> }
<register element> ::= <instance and options> | <field and options> | <instance reference> |
    <selected segment element> | <boundary instance> | <using statement>

<instance and options> ::= <instance definition> { <field assignments> }
<instance definition> ::= <instance ident> **is** [**PACKAGE** <package hierarchy> <colon>]
    <reg or seg name>

<instance ident> ::= <segment ident> | <array ident>
<segment ident> ::= <VHDL identifier>
<array ident>::= **ARRAY** <array segment ident> <left paren> <range> <right paren>
<array segment ident>::= <VHDL identifier>

<field assignments> ::= <field value assignment> | <field reset assignment> |
<field domain assignment> | <field selection assignment>
<field value assignment> ::= [<field ident> <colon>] <value assignment>
<field reset assignment> ::= [<field ident> <colon>] <reset assignment>
<field domain assignment> ::= [<field ident> <colon>] <domain assignment>
<field ident> ::= { <instance name> <period> } <field name>
<instance name> ::= <segment ident> | <array instances>
<array instances> ::= <array segment ident> <bit list>

<field and options>::= <field name> <field length> { <field options> }

<instance reference> ::= <segment ident> | <array instance>
<array instance> ::= <array segment ident> <left paren> <index> <right paren>
<index> ::= <integer>

<selected segment element> ::= **SELECTMUX**
<left paren> <selectable segment> <right paren>
{ <comma> <left paren> <selectable segment> <right paren> }
<field selection assignment>
<selectable segment> ::= <instance and options> | <instance reference>

<field selection assignment> ::= <selection field> <selection values>
[<broadcast field> <broadcast values>]
<selection field> ::= **SELECTFIELD** <left paren> <field reference> <right paren>
<field reference> ::= { <instance reference> <period> } <field name>
<selection values> ::= **SELECTVALUES** <left paren> <segment selection>
{ <segment selection> } <right paren>
<segment selection> ::= <left paren> <instance reference> <colon> <field value>
{ <comma> <field value> } <right paren>
<field value> ::= <mnemonic identifier> | <binary pattern> | <hex pattern> | <decimal pattern>
<broadcast field> ::= **BROADCASTFIELD** <left paren> <field reference> <right paren>
<broadcast values> ::= **BROADCASTVALUES** <left paren> <broadcast selection>
{ <broadcast selection> } <right paren>
<broadcast selection> ::= <left paren> <instance reference> { <comma> <instance reference> }
<colon> <field value> { <comma> <field value> } <right paren>

<boundary instance> ::= <segment ident> **is**

[**PACKAGE** <package hierarchy> <colon>] <boundary segment name>

<using statement>::= **USING** <package prefix>
<package prefix> ::= <package hierarchy> | <minus sign>
<package hierarchy>::= <user package name> { <period> <user package name> }

Rules

a) Within a <register assembly list>, <register assembly elements> shall be ordered with the first listed segment closest to TDI, and the last closest to TDO.

b) When the <target> of a <register assembly statement> is an <entity target>, and when the <reg or seg name> also appears as a <register> element in a <register association> in the <register access description>,

or the <reg or seg name> is otherwise defined by this standard, the sum of the lengths of the nonexcludable segments listed in the <register assembly> shall either:

1) Match the length of the register when the length is otherwise defined and fixed in this standard.
2) For the **BOUNDARY** register, match the <register length> specified in the <boundary length stmt> or the <reset length> specified in the <assembled boundary length stmt>, whichever occurs.
3) Match the length of the register as specified in the <register access description>.
4) Define the length of the register when the length in the <register access description> is deferred (*).
5) Define the length of a register, which is defined in this standard and the length is not defined.

c) When the <target> of a <register assembly statement> is a <package target> or when the <reg or seg name> does not appear as a <register> element in a <register association> in the <register access description> or is not otherwise defined by this standard, the length of <reg or seg name> shall be the sum of the lengths of the nonexcludable segments listed in the <register assembly>.

d) For a <field and options> segment, the <field name> shall be unique within a given <register assembly list>.

e) All <register assembly elements> between a **SEGSTART** element and a **SEGMUX** element with the same **SEGMENT** <association name>, or if there is no **SEGSTART** element with the same **SEGMENT** <association name>, then between a **SEGSEL** element and a **SEGMUX** element with the same <association name>, shall be excludable as a unit.

NOTE 1—The **SEGSEL**, **SEGSTART**, and **SEGMUX**, all with the same **SEGMENT** <association name>, are not part of the excludable segment. These named fields are defined in the Standard BSDL Package Body (see B.9).

f) All <segment ident> and <array segment ident> elements that appear in a <register assembly list> (including a <selected segment element> list) shall:

1) Be unique within the BSDL or Package.
2) Be a name for a contiguous subset of bits as defined within the current <reg or seg name>.

g) When a <reg or seg name> is used in an <instance definition>, it shall be defined in a <register field list> or a <register assembly list> statement contained in the current BSDL or Package, or in a Package referenced by the <package hierarchy> specified in the most recent USING statement and as part of the <instance definition>.

h) Where multiple <array ident> statements refer to the same <VHDL identifier>, the associated <range> specifications shall not have any duplicate indices and there shall be no missing indices within the total range specified.

i) A <field ident> shall be composed of zero or more <instance name> fields in parent-to-child hierarchical order, followed by an <extended field name>, all separated by periods.

j) Each <instance name> of a <field ident> shall be an <instance ident> within the <register assembly list> with the <reg or seg name> given in the <instance and value> at each hierarchical level.

k) The <extended field name> of a <field ident> shall be an <extended field name> within a <register field list> or a <field name> within a <register assembly list>, in either case with the <reg or seg name> given in the <instance and value> of the last <instance ident>.

l) When an <instance reference> is used in a <register assembly list>, it shall not appear in the entire hierarchy of the TDR being defined in a way that could cause multiple copies of the referenced instance to be scanned simultaneously; that is, the referenced instance shall appear at most one time in any valid configuration of excludable or selectable segments for a single TDR.

NOTE 2—A single segment cannot be serially connected in more than one location within a TDR when the TDR is scanned. In the case of selectable segments, only one segment at a time is selected for connection between TDI and TDO, so a single segment can appear in multiple selectable segments of a single selectable segment structure. A segment may also appear in more than one TDR. Instance names must be unique only within a BSDL or BSDL Package, but the <instance definition> referred to by an <instance reference> must be in the same file, so there is no conflict if the <instance reference> matches an <instance definition> in a different file, even if they end up in the same TDR hierarchy.

m) The <selected segment element> shall only be included in the <register assembly list> of a design-specific register.

n) Every instance named in a <selectable segment> of a <selected segment element> shall be selectable by one or more value in the associated <selection field> and <broadcast field>, if any, and when selected shall establish a valid scan path through the <selected segment element>.

NOTE 3—For a <selectable segment> that is an array, all elements of the array are governed by the above rule.

o) A <field reference> shall be composed of zero or more <instance reference> fields in parent-to-child hierarchical order, followed by an <extended field name>, all separated by periods, and shall resolve to a register field with a length greater than zero and defined in a **REGISTER_FIELDS** or **REGISTER_ASSEMBLY** attribute.

p) The definition of the <field reference> of both a <selection field> and a <broadcast field>, if any, shall include a <reset assignment> and a <value assignment> of **RESETVAL** with a value or values that establish a valid default scan path through the <selected segment element>.

q) The definition of the <field reference> of both a <selection field> and a <broadcast field>, if any, shall include a <type assignment> of **DELAYPO** if both the <field reference> and a segment selected by the <field reference> are scanned simultaneously.

NOTE 4—For example, the IEEE 1500 wrapper instruction register (WIR) is not scanned simultaneously with any of the controlled wrapper data registers (WDRs), so this rule does not apply to the WIR. It may apply to the SelectWIR field, which selects between the WIR and the WDR.

r) The <mnemonic identifier> specified in a <selection values> list shall be a <mnemonic identifier> in the <mnemonic group> associated with the <field reference> in the <selection field> by the **RESETVAL** <value assignment>, and the <mnemonic identifier> specified in a <broadcast values> list shall be a <mnemonic identifier> in the <mnemonic group> associated with the <field reference> in the <broadcast field> by the **RESETVAL** <value assignment>.

s) The selection of any decode of the <selection field> or <broadcast field> not listed as a <field value> in the respective values list shall be undefined.

NOTE 5—This could be the selection of an undocumented (private) register segment, or just an unused code resulting in an open scan chain. There is no default selection for unused codes.

t) All <register assembly elements> of a <register assembly list> with a <reg or seg name> of **BOUNDARY** shall be a <boundary instance> or an instance of the **DOMCTRL**, **SEGSEL**, **SEGSTART**, or **SEGMUX** register fields defined in the Standard BSDL Package Body (see B.9).

u) All excludable segments in a <register assembly list> with a <reg or seg name> of **BOUNDARY** shall have at least one each of any associated **DOMCTRL** and **SEGSEL** fields in the same <register assembly list>.

NOTE 6—This requires that any excludable segments in the boundary-scan register can be controlled by cells in the boundary-scan register. There may be duplicate **DOMCTRL** and **SEGSEL** fields in the initialization data register.

v) All instances of the **DOMCTRL** and **SEGSEL** fields in a <register assembly list> with a <reg or seg name> of **BOUNDARY** shall only control excludable segments in the same <register assembly list>.

w) All excludable segments in the optional initialization data register shall have at least one each of any associated **DOMCTRL** and **SEGSEL** fields in the initialization data register.

x) All excludable segments in a public TDR (standard or design specific) shall have the associated **DOMCTRL** and **SEGSEL** fields in the same TDR or another public TDR other than the boundary-scan register.

y) A <package hierarchy> shall be composed of one or more <user package name> fields in parent-to-child hierarchical order, separated by periods.

z) A <package hierarchy> shall be used to identify a specific <mnemonic group name>, <reg or seg name>, or **<**boundary segment name**>** when the name was defined within the lowest level package of the <package hierarchy> and unless the name is known to be unique within the current BSDL or Package and all lower level Packages.

aa) When a <package hierarchy> is defined by a <using statement>, it shall be prepended to, when present, or used in lieu of, when not present, the <package hierarchy> in the <mnemonic group name>, <reg or seg name>, or <boundary segment name>, of all subsequent <instance and value> or <boundary instance> statements within the <register assembly list>, until replaced with a new <using statement>.

bb) When a <using statement> has a value of <minus sign> instead of a <package hierarchy>, then any current <package hierarchy> previously defined by a <using statement> shall be removed.

cc) When the value of <conformance identification> is **STD_1149_1_2001**, **STD_1149_1_1993**, or **STD_1149_1_1990**, the <instance definition> shall not be of a **DOMCTRL**, **SEGSEL**, **SEGSTART**, **SEGMUX** field.

Permissions

dd) Multiple <array ident> statements that refer to the same <VHDL identifier> may appear in any order in the <register assembly list>.

ee) When controlling excludable segments in the boundary-scan register or initialization data register, the **DOMCTRL** and **SEGSEL** fields may be duplicated in other public test data registers.

NOTE 7—The parallel outputs of all **DOMCTRL** or all **SEGSEL** cells controlling a single excludable segment would effectively be ORed together so that any one of them had the same effect as the others.

Recommendations

ff) When the <broadcast field> is provided, it should be the same <field reference> as the <selection field> in the same <selected segment element>.

gg) When the <broadcast field> is provided and is not the same <field reference> as the <selection field> in the same <selected segment element>, the **REGISTER_CONSTRAINTS** attribute should be used to document combinations of <field values> for the two registers that would result in a broken scan chain or other undesired behaviors.

B.8.21.2 Description

Like the **REGISTER_FIELDS** attribute, the **REGISTER_ASSEMBLY** attribute can be used to define one or more full registers named in the **REGISTER_ACCESS** attribute or defined in this standard, or of a register segment to be used in another **REGISTER_ASSEMBLY** attribute. The **REGISTER_ASSEMBLY** attribute lists instances of (i.e., "instantiates") register segments defined elsewhere in the BSDL or in a BSDL Package Body, which is the target of a **USE** statement in the BSDL. The length of the register or register segment is the sum of the lengths of the segments listed in the <register assembly list>, including the length of any currently selected selectable segment, and any included excludable segments. The documented minimum or default length is the sum of lengths of the nonexcludable segments (including the always nonexcludable **DOMCTRL** and **SEGSEL** fields) plus the length of any selectable segments selected by the reset value of the selection field. Each element in the <register assembly list> has to have an explicitly or implicitly defined length, so the length of the newly defined register or register segment may be calculated for any configuration of fixed, excludable, and selectable segments.

Unlike the **REGISTER_FIELDS** attribute, there is no redundant information to provide checking. All bits of the register or register segment must be defined. The various fields in the register or register segment are accessed by segment and field name rather than by counting bits within the register, as shown by the following simple example.

NOTE—For clarity, no field assignments are shown in these small examples. See B.8.20 for a description of and examples with field assignments.

```
attribute REGISTER_ASSEMBLY of INIT_Example : entity IS
   "init_data (" &
     -- TDI
     "(init_tail IS init_seg), "&
     "(SerDesChannel_00 IS Channel), "&
     "(SerDesChannel_01 IS Channel), "&
     "(SerDesChannel_02 IS Channel), "&
     "(SerDesClk_0 IS ChClock) )";
```

init_seg is shown in the discussion of the **REGISTER_FIELDS** attribute, and Channel and ChClock are defined in a later extended example (see B.8.21.3), but that is not critical to understanding **REGISTER_ASSEMBLY**. Note that no lengths are shown. The length of every segment is defined and known, so the length of the register or register segment defined with **REGISTER_ASSEMBLY** can be calculated from this definition.

Note in the preceding example that there are three instances of the "Channel" register segment, and they are all contiguous. There is an easier way to show this, as shown in the following.

```
attribute REGISTER_ASSEMBLY of INIT_Example : entity IS
   "init_data (" &
      "(init_tail IS init_seg), "&
      "(array SerDesChannel(0 to 2) IS Channel), "&
      "(SerDesClk_0 IS ChClock) )";
```

Arrays of segments can also be used to easily specify a range of contiguous bits that are not described. Defining a single bit register segment as unused, an array of them may be used to define an unused portion of a register in a **REGISTER_ASSEMBLY** attribute as shown at the end of the following.

```
attribute REGISTER_FIELDS of INIT_Example : entity is
   "reserved [1] ((do_not_use [1] IS (0)))";
attribute REGISTER_ASSEMBLY of INIT_Example : entity IS
   "init_data (" &
      "(init_tail IS init_seg), "&
      "(array SerDesChannel(0 to 2) IS Channel), "&
      "(SerDesClk_0 IS ChClock), "&
      "(array unused (7 DOWNTO 0) IS reserved) )";
```

Alternatively, the unused bits may be defined with an in-line field within the **REGISTER_ASSEMBLY** attribute as shown at the end of the following.

```
attribute REGISTER_ASSEMBLY of INIT_Example : entity IS
   "init_data (" &
      "(init_tail IS init_seg), "&
      "(array SerDesChannel(0 to 2) IS Channel), "&
      "(SerDesClk_0 IS ChClock), "&
      "(unused [8]) )";
```

This standard allows a register segment to be used in more than one TDR, and in more than one of a set of selectable segments. For example, in Figure 9-2, two segments are used to construct three TDRs. To support this use of a segment in more than one TDR or selectable segment, the name of an instance, already instantiated in a different TDR or selectable segment, may be referenced and used as an element in a register assembly. Note that the switching logic required to make this work in hardware is not specified; this simply describes the fact that this segment is used in two different TDRs, and that the segments are exactly the same set of register cells.

The three TDRs illustrated in Figure 9-2 could be coded as follows, with each box in the figure being multi-bit register fields, and where i1 and i2 are the segment instances that are used in two TDRs each:

```
attribute REGISTER_FIELDS of MyChip : entity IS
  "Front_Seg [9] ( "&
    "(a [3] IS (8 DOWNTO 6)), "&
    "(b [2] IS (5 DOWNTO 4)), "&
    "(c [4] IS (3 DOWNTO 0)) ), "&
  "Back_Seg  [9] ( "&
    "(x [2] IS (8 DOWNTO 7)), "&
    "(y [5] IS (6 DOWNTO 2)), "&
    "(z [2] IS (1 DOWNTO 0)) )";

attribute REGISTER_ASSEMBLY of MyChip : entity IS
  "FRONT_TDR ( "&
    "(i1 IS Front_Seg) ), "&  -- Create an instance of a segment
  "BACK_TDR ( "&
    "(i2  IS Back_Seg) ), "&  -- Create an instance of a segment
  "WHOLE_TDR ( "&
    "(i1), "&  -- Reference the existing segment instead of a new segment.
    "(i2) )" ; -- Reference the existing segment instead of a new segment.
```

REGISTER_FIELDS and **REGISTER_ASSEMBLY** attributes may be coded in the BSDL or in a user-supplied package body (see B.10). The string value of the attribute containing the field definitions is the same in all cases.

Prior to the existence of the **REGISTER_FIELDS** and **REGISTER_ASSEMBLY** attributes, all names in the BSDL and any standard or user packages referenced by "USE ..." statements in the BSDL were required to be unique. Now, user packages may come from diverse sources with no way of coordinating name assignments, and further, user packages may contain "USE ..." statements referencing other user packages. Clearly, it is impractical to impose a rule that all register field, register segment, and register instance definitions have unique names throughout the structure.

Some of the requirements have not changed: any names defined in the standard packages (package names starting with **STD_1149_**) must still be unique throughout the data structure.

Within BSDL and user packages, the rule is that the names be unique within the BSDL or within the package body. To refer to a name in another user package, the name is prefaced with the <package hierarchy>. The <package hierarchy> is a top-to-bottom hierarchy of package names, starting with the package used in the current BSDL or user package, concatenated by the period (.) character.

There are two ways to specify the hierarchy:

The first way is to simply specify the <package hierarchy> in the instance definition in the **REGISTER_ASSEMBLY** attribute with the optional "**PACKAGE** <package hierarchy> <colon>" value ahead of the register segment name.

The second way is to use the "**USING** <package hierarchy>" keyword and value within the **REGISTER_ASSEMBLY** attribute. All instances of register segment names, boundary segment names, and mnemonic group names following that keyword are treated exactly as if the "<package hierarchy> <colon>" value were used ahead of the name. A new **USING** assertion will change the <package hierarchy> value or remove it if "**USING -**" is specified. If a <package hierarchy> is supplied both with a **USING** assertion and prepended to the name, then the **USING** assertion <package hierarchy> will be prepended to the <package hierarchy> with the name. In the following example, the **USING** assertion affects the register segments Channel and Clock, and the mnemonic group names Protocol, and TX_Swing:

```
attribute REGISTER_ASSEMBLY of INIT_EXAMPLE : entity is
   "init_data ( "&
      "(USING MyCorp_SERDES_1_2_3), " &
      "(Array SerDesChannel(0 TO 5) IS Channel " &
```

```
      "SAFE(SerDes_Protocol(off)) " &
      "SAFE(SerDes_TX_Outputs(75%_Swing))), " &
   "(Array SerDesClk(1 TO 2) IS ChClock), " &
   "(USING -), " &
   ...
         "), " &
```

The **REGISTER_CONSTRAINTS** (see B.8.22) attribute and PDL (see Annex C) will use the instance names defined in the **REGISTER_ASSEMBLY** attribute for each instantiation of a register segment.

Excludable register segments and domain control

Four special register fields, named **DOMCTRL, SEGSEL, SEGSTART**, and **SEGMUX**, are defined in the Standard BSDL Package Body (see B.9) to support the use of excludable segments of registers (see 9.4). For public registers, only these fields may be used to delineate excludable segments, and these fields have been defined to meet all of the rules in 9.4. Within a register assembly, the **SEGSEL** or **SEGSTART** would be placed immediately before (closer to the start of the list and to TDI) the segments to be excluded, and the **SEGMUX** would be placed immediately after. Whatever segments are listed between those two, and not including the two, comprise the excludable segment. The default length of a register defined to include one or more excludable segments is the length with all excludable segments in their default excluded state. That is, they do not count toward the default length. Since the length of every named segment is known, calculation of the actual length of the register under any configuration of excludable segments, included or not, is straightforward.

When an excludable segment may exist in a domain where scanning may not be possible (such as a power domain that can be powered down), and control is provided on the component to make that domain scannable (a power controller that can provide power to the domain), then a **DOMCTRL** field may be provided in a register assembly. The domain-control cell in the **DOMCTRL** field is assumed to override the control of the domain in a way that will make the domain scannable, possibly after some delay. Note that there is no intention here of defining a general domain control structure controlled through the TAP, but just to give test logic override control of domains already existing in the mission logic so that the test logic within those domains may be controlled and exercised through the TAP.

A domain name is established by one or more **DOMCTRL** fields with a **DOMAIN** keyword followed by the domain name enclosed in parenthesis. The domain name is referenced by one or more **SEGSEL** fields with a **DOMAIN** keyword followed by the same name enclosed in parenthesis. This domain name is simply to identify all associated **DOMCTRL** and **SEGSEL** fields.

When the **DOMCTRL** cell precedes the **SEGSEL** cell, all in a single register segment, and both include the same domain name, the association is obvious. If the **DOMCTRL** cell is located in another register, then the software must find the register that includes the **SEGSEL** cells with the referenced domain name.

When the excludable segment is not immediately preceded by the **SEGSEL** field, including when the two are in different registers, then the zero length **SEGSTART** field is used to mark the beginning of the excludable segment. In order to associate the **SEGSEL SEGSTART** and **SEGMUX** fields controlling a single excludable segment with each other, the **SEGMENT** keyword with a segment identifier in parenthesis is required on all three types of fields.

See Figure B-12 and associated BSDL statements for an extensive example of these domain and segment control structures. Here is a simple example of the use of these fields to make the SerDes domain in the previous example excludable:

```
attribute REGISTER_ASSEMBLY of INIT_Example : entity IS
   "init_data (" &
      "(init_tail IS init_seg), "&
      "(SerDesPowerUp IS DomCtrl Domain(serdesPwr) TRSTReset), "&
      "(SerDesBegin IS SegSel Domain(serdesPwr) Segment(S1) TRSTReset), "&
        "(array SerDesChannel(0 to 2) IS Channel), "&
```

```
    "(SerDesClk_0 IS ChClock), "&
  "(SerDesEnd IS SegMux Segment(S1)), "&
  "(unused [8] NoPI NoPO) )";
```

The array of channels and the channel clock are all part of the excludable segment, and their lengths will not count toward the default length of "init_data" register defined here.

When assembling the boundary-scan register from segments, only segments defined by the **BOUNDARY_SEGMENT** attribute plus the four segment control fields defined in the Standard BSDL Package Body (DomCtrl, SegSel, SegStart, and SegMux) may be used. For example, assume that five boundary segments are defined: north, south, west, east1, and east2 (see Example 2 of B.8.14.2), and that the east2 segment may be powered down; then the definition of the boundary-scan register would be as follows.

```
attribute REGISTER_ASSEMBLY of BOUNDARY_Example : entity IS
   "boundary ( "&
      "(north       IS north), "&
      "(east1       IS east1), "&
      "(PowerUp     IS DomCtrl Domain(pwr3) TAPReset), "&
      "(East2_start IS SegSel  Domain(pwr3) Segment(E2) TAPReset), "&
        "(east2       IS east2), "&
      "(East2_End   IS SegMux Segment(E2)), "&
      "(south       IS south), "&
      "(west        IS west) )" ;
```

The maximum length of the boundary-scan register is the sum of all segment lengths, including the single bit DomCtrl and SegSel field segments. The default length is the sum of the lengths of all segments except the "east2" segment.

The value of the Domain assignment (`pwr3`) is an arbitrary name for the domain being controlled, and in return controlling the value captured by the SegSel cell with the matching name assignment. There may be multiple DomCtrl cells and multiple SegSel cells having the same Domain name assignment. For controlling segments in the boundary-scan register, the DomCtrl and SegSel cells could have been located anywhere in the boundary-scan register, and could be duplicated in the init_data register. The DomCtrl and SegSel cells could be in any public register for segments in TDRs other than the boundary-scan register.

Selectable register segments

Within a **REGISTER_ASSEMBLY** attribute, the segments listed, even those that are excludable, are connected in series in a single scan chain. Several keywords and their values are defined to support parallel scan structures (such as IEEE Std 1500) where multiple segments are in parallel and one is selected at any time to connect between the scan-in and the scan-out. These selectable register segment structures are treated as a single segment within the **REGISTER ASSEMBLY** attribute, and they are connected in series with any other segments defined in the attribute.

The selectable register segment is started with the **SELECTMUX** keyword, which marks the position in the scan chain where the scan-in data fans out to each of the selectable segments. This is followed by a list of parallel segments, all of which get the same scan-in data, which are followed in turn by the **SELECTFIELD** and **SELECTVALUES** keywords, and optionally the **BROADCASTFIELD** and **BROADCASTVALUES** keywords. These identify the encoded selection field, define the decodes of that field, and mark the location of the selection circuit that selects which segment is connected in series with the rest of the segments in the **REGISTER_ASSEMBLY** attribute (the end of the selectable register segment).

The keywords used to define a selectable register segment are:

SELECTMUX Specifies the start of a list of segments, each of which is individually selectable for scanning as opposed to all being in series. Effectively, these segments are in parallel (sharing the same scan-in signal) with a circuit such as a multiplexer to select which is connected to TDO.

SELECTFIELD Specifies the <reg or seg name> or <extended field name> that controls the selection of one of a set of selectable segments. **SELECTFIELD** is always used with **SELECTVALUES** and is used only with a **SELECTMUX** keyword in a <selected segment element>. If **BROADCASTFIELD** is also used in the same <selected segment element>, then **SELECTFIELD** only selects which segment is connected to the scan out of the element.

SELECTVALUES Associates a register segment <instance reference> with a comma separated list of <mnemonic identifier>, BINARY_PATTERN, DECIMAL_PATTERN, or HEX_PATTERN values. The values specified represent the values the **SELECTFIELD** register must contain to select the specified register segment.

BROADCASTFIELD Specifies the <reg or seg name> or <extended field name> that controls the selection of one of a set of selectable segments for scanning. **BROADCASTFIELD** is always used with **BROADCASTVALUES** and is used only with a **SELECTMUX** keyword in a <selected segment element>.

BROADCASTVALUES Associates a comma separated list of <mnemonic identifier>, BINARY_PATTERN, DECIMAL_PATTERN, or HEX_PATTERN values with a comma separated list of <register segment> instances, which receive scan-in data simultaneously in parallel. That is, these values enable the Shift_<TDR>, Capture_<TDR>, and Update_<TDR> signals of the recommended TDR interface in Table 9-1, or equivalent.

The circuit shown in Figure B-11 shows some of both the flexibility and the limitations of the selectable register segment.

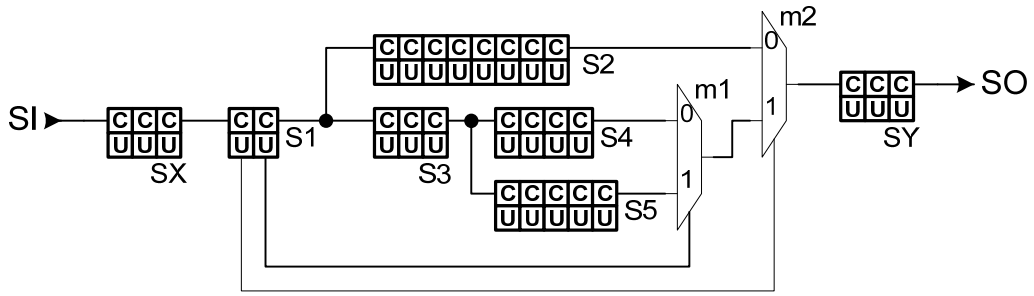


Figure B-11—Simple selectable register segment structure

First, note that there could be any number of fixed, excludable, or selectable segments in the position of segments SX and SY. These just represent the rest of the full **REGISTER_ASSEMBLY**. There are three unique scan paths through the selectable segment structure: S2, S3/S4, and S3/S5. Segment S1 is the selection field that selects which unique path will be taken, and it could be anywhere outside the selectable register segment structure and in this or another public TDR. The operation of this structure is fairly obvious. It could be documented in a couple ways. Clearly, the paths S3/S4 and S3/S5 involve multiple segments, but the selectable segment structure only allows single segments for each parallel path, so those paths need to be somehow described so they can be instantiated as single segments. First, S3, S4, and S5 could be described with a register assembly as a nested selectable register segment structure for S4 and S5. Alternatively, S3/S4 and S3/S5 could be described as separate register assemblies using an <instance reference> to include S3 in both segments.

The nested selectable segment register description follows:

```
attribute REGISTER_ASSEMBLY of Select_Example : entity IS
   "Seg_X2Y ("&
      "(SX [3]), "&
         "(S1 [2]), "&
      "(SELECTMUX "&
         "(S2 [8]), "&
         "(S345 IS Seg_345), "&
         "SELECTFIELD(S1) SELECTVALUES( (S2 : 0b0X) (S345 : 0b1X) ) "&
      "), "&
      "(SY [3]) "&
   "), "&
   "Seg_345 ("&
      "(S3 [3]), "&
      "(SELECTMUX "&
         "(S4 [4]), "&
         "(S5 [5]), "&
         "SELECTFIELD(S1) SELECTVALUES( (S4 : 0bX0) (S5 : 0bX1) ) "&
      ") "&
   ")";
```

The un-nested selectable register segment description (using an instance reference) follows:

```
attribute REGISTER_ASSEMBLY of Select_Example : entity IS
   "Seg_X2Y ("&
      "(SX [3]), "&
      "(SELECTMUX "&
         "(S2 [8]), "&
         "(S34 IS Seg_34), "&
         "(S35 IS Seg_35) "&
         "SELECTFIELD(S1) "&
         "SELECTVALUES( (S2 : 0b0X) (S34 : 0b10) (S35 : 0b11) ) "&
      "), "&
      "(SY [3]) "&
   "), "&
   "Seg_34 ( (S3 [3]), (S4 [4]) ), "&  -- S3 instantiated
   "Seg_35 ( (S3),     (S5 [5]) )";    -- S3 referenced
```

A well-documented use of this type of selectable segments is the wrapper serial port (WSP) of IEEE Std 1500. While the WSP is designed to be able to be controlled by an IEEE 1149.1 TAP, it was not possible prior to the 2013 version of this standard to document the register structure; it had to be a private register. Figure B-13 is a typical example of a WSP.

B.8.21.3 Examples

This is the same extensive example shown for the **REGISTER_FIELDS** attribute, but it is rewritten to show how a hierarchical **REGISTER_ASSEMBLY** attribute can make the register description more compact.

NOTE 1—This extended example shows assignments for completeness; see B.8.20 for specifications and a description of the assignments.

Initialization REGISTER_ASSEMBLY example

```
attribute REGISTER_FIELDS of INIT_Example : entity is
   -- IP configuration register, see chip release documentation.
```

```
   --   Done in a register field because of non-contiguous bits;
   --   The PLL enable bits are used and the rest not used in JTAG test.
   "configuration [15] ( " &
      "(IP_reg [12] IS (14 DOWNTO 6, 4, 2, 0) " &
         "DEFAULT (0x0DB) NoPI), " &  -- Required value for 1149 test.
      "(PLL_Enable [3] IS (5,3,1) SAFE (PLLConfigValues(PLLsoff)) NoPI) " &
                     " ), "&
   "Channel [5] ( "&
     "(Protocol [3] IS (2,0,1) DEFAULT (SerDes_Protocol(*))), "&
     "(TX_Swing [2] IS (3,4)   DEFAULT (SerDes_TX_Outputs(*))) "&
              " ), "&
   "ChClock [5] ( "&
      "(Setting [5] IS (4 downto 0) DEFAULT SerDesClockSettings (100Mhz)) "&
              " )" ;

attribute REGISTER_ASSEMBLY of INIT_EXAMPLE : entity is
   -- Register Assembly of INIT_DATA register
   "init_data ( "&
   -- TDI
      -- First 36 bits are unused.
      "(reserved1[36]), " &
      -- Observed in init_data because must be set at power-up.
      "(VSEL_bits [5] Captures(IO_VSEL_Decodes(*)) NoPO), " &
      "(IP_Config IS configuration), " &
      "(Array SerDesChannel(0 TO 5) IS Channel " &
         -- SAFE values determined by IC designer, not specified for IP.
         "SAFE( SerDes_Protocol(off)) " &
         "SAFE(  Serdes_TX_Outputs(75%_Swing))), " &
      -- Defaults are provided in the IP Package for the SerDes Clocks
      "(Array SerDesClk(1 TO 2) IS ChClock), " &
      "(Array SerDesChannel(6 TO 17) IS Channel " &
         -- SAFE values determined by IC designer, not specified for IP.
         "SAFE(  SerDes_Protocol(off)) " &
         "SAFE(  Serdes_TX_Outputs (75%_Swing))), " &
      "(Array SerDesClk(3 TO 3) IS ChClock), " &
      -- Register bits that are IP inputs, not part of IP itself.
      --   Power level supplied to SerDes internal gates;
      "(SerDesCXVddSel [1] DEFAULT (SerDesCXVddSelLevel(*)) NoPI), "&
      --   Powerup SerDes Test Receivers during SAMPLE operation
      "(SerDesSamplePowerUp [1] DEFAULT (SerDesSampleOvrd (off)) NoPI), "&
      -- 2^^10 possible decodes. See Reference Manual.
      "(DDRTermSel [10] SAFE(0) NoPI), "&
      -- Reserved Field
      "(reserved2[8]), " &
   -- TDO
            "), " &

   -- Register Assembly for INIT_STATUS register - read-only per the standard
   "init_status ( "&
      "(INITErrorStatus [2] " &
         "CAPTURES (ErrorCode(NoError)) NoPO), " &
      "(INITCompletionStatus [2] " &
         "CAPTURES (InitCompletionValue(Completed)) NoPO) " &
               ") ";
```

Boundary-scan example

Continuing the segmented boundary-scan register example from Example 2 of B.8.14.2, the segments defined there could be assembled, with the segment exclusion cells on the one segment, as follows.

```
attribute REGISTER_ASSEMBLY of Chip_2013 : entity is
   -- Register Assembly of Boundary-scan register
   "boundary ( "&
   -- TDI, starting in NE corner and going clockwise
      "(North_side IS north),  "&
      "(NEast_side IS east1),  "&
      "(PowerUp    IS DomCtrl Domain(pwr3) TAPReset), "&
      "(SEast_incl IS SegSel DOMAIN(pwr3) Segment(SE2) CHReset), "&
         "(SEast_side IS east2),  "&
      "(SEast_mux  IS SegMux Segment(SE2)), "&
      "(South_side IS south),  "&
      "(West_side  IS west) ) ";
```

Power-domain control example

The rules in 9.4 define the rules for creating excludable segments, but not the rules for how to document the domain control structure. To illustrate the documentation of excludable segments and domain control, Figure B-12 first shows the structure and the scan cells involved. For simplicity and clarity, power-domain level shifters, isolation cells, or other elements required for multiple power domains are not shown, nor any detail in the mission mode controls to the domain controller, including any possible input to select IEEE 1149.1 test mode.

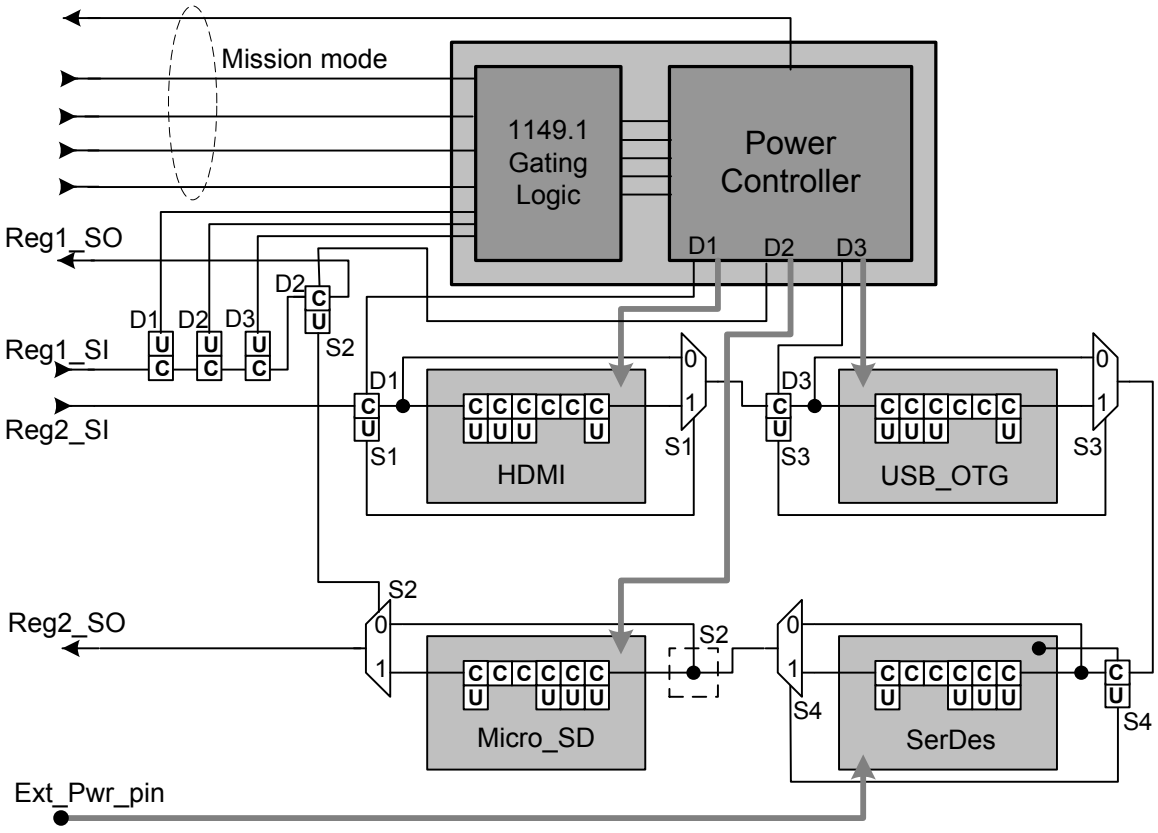


Figure B-12—Illustrative component power control structure

In Figure B-12, there are four power domains, designated D1 through D4, that can be turned off. Three domains are controlled by an on-chip power domain controller (D1, D2, and D3 provide power to HDMI, Micro_SD, and USB_OTG, respectively). Finally, one (D4 with Ext_Pwr_pin provides power to SerDes) is controlled from off the component through a dedicated power pin. There are also four excludable segments, designated S1 through S4. In the BSDL, the domain and segment designations are used to identify the associations among the various DomCtrl, SegSel, SegStart, and SegMux cells.

All of the domain-control cells (DomCtrl, for power domains D1 through D3) and one of the segment-select cells (SegSel, for power domain D2 and segment S2) are in a TDR named Reg1, and the other three segment-select cells (SegSel, for power domains D1, D3, and D4) are in a TDR named Reg2 along with the four excludable segments. At the SI input to the Micro_SD segment, the dashed square around the signal branch represents the zero-length segment-start field (SegStart), which contains no logic. The segment-select cell for that domain (D2) also has a segment name S2 to associate it with the segment-start field.

All domain-control and segment-select cells in Figure B-12 are labeled with the domain and segment names that they are associated with. The scan-in and scan-out are not labeled TDI and TDO since these may just be segments in a larger test data register. In the following example BSDL attributes, the same names and labels are used.

The segment-select cells each capture a signal indicating that the segment is ready to scan. For three domains (D1, D2, and D3), the signal comes from the power controller. The fourth domain (D4) is powered from off the component, and its associated segment-select cell will capture whether the power is on or not. There is no domain-control cell for this domain.

Placement of the DomCtrl and SegSel cells reflects the designer's choice. A designer could choose to place all domain-control and segment-select cells in the same TDR as the excludable segments, or in a completely different "configuration" TDR, or any combination in between. The only restrictions are that there must always be at least one scan cell in a TDR, even if all the segments are excluded, and that all the excludable segments, domain-control, and segment-select cells must be in public test data registers.

```
attribute REGISTER_ASSEMBLY of PwrDomStruc : entity IS
   "Reg1 ( "&
      "(hdmi_pwr     IS DomCtrl Domain(D1) CHReset), "&
      "(micro_sd_pwr IS DomCtrl Domain(D2) CHReset), "&
      "(usbotg_pwr   IS DomCtrl Domain(D3) CHReset), "&
      "(micro_sd_sel IS SegSel  Domain(D2) Segment(S2) CHReset)), "&
   "Reg2 ( "&
      "(hdmi_sel     IS SegSel  Domain(D1) Segment(S1) CHReset), "&
      "(hdmi         IS hdmi_seg), "&
      "(hdmi_mux     IS SegMux Segment(S1)), "&
      "(usb_otg_sel  IS SegSel  Domain(D3) Segment(S3) CHReset), "&
      "(usb_otg      IS usb_otg_seg), "&
      "(usb_otg_mux  IS SegMux Segment(S3)), "&
      "(SerDes_sel   IS SegSel  Domain_External(D4) Segment(S4) CHReset), "&
      "(SerDes       IS SerDes_seg), "&
      "(SerDes_mux   IS SegMux Segment(S4)), "&
      "(micro_sd_start IS SegStart Segment(S2)), "&
      "(micro_sd     IS microsd_seg), "&
      "(micro_sd_mux IS SegMux) Segment(S2) )";
Attribute Register_Association of PwrDomStruc : entity is
   "SerDes_sel : port(Ext_Pwr_pin) ";
```

NOTE 2—The definitiion of the segments named hdmi, usb_otg, SerDes, and micro_sd are not shown.

IEEE 1500 WSP Examples

A well-documented use of selectable segments is the WSP of IEEE Std 1500. While the WSP is designed to be able to be controlled by an IEEE 1149.1 TAP, it was not possible prior to the 2013 version of this standard to document the register structure; it had to be a private register. Figure B-13 is a typical example of a simple WSP.

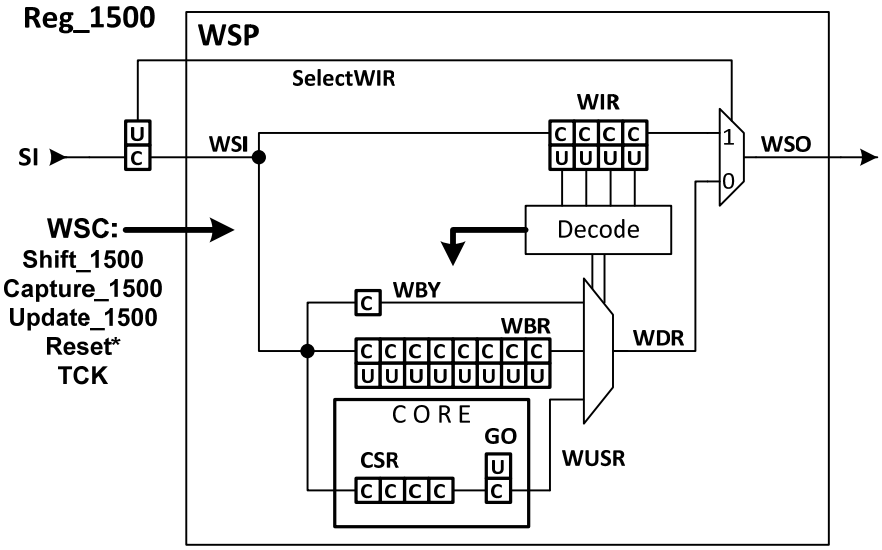


Figure B-13—Simple wrapper serial port

This design contains a single register cell (which is not part of the IEEE 1500-compliant WSP as shown by the box) to generate the required SelectWIR signal of the wrapper serial controls (WSCs), and a WSP, as defined in IEEE Std 1500. The WSP contains two selectable segment structures, the inner that selects between the available wrapper data register (WDR) and the outer that selects between the WDR and the wrapper instruction register (WIR). In this example, there are three WDRs: the required wrapper bypass (WBY) and wrapper boundary (WBR registers, and one design-specific wrapper user (Wusr) register in the core.

The wrapper scan in (WSI) signal fans out to all registers, and multiplexing circuits are used to select one register for connection to the wrapper scan out (WSO). The WSP inside the box is what is defined in IEEE Std 1500. Note that gating logic to control scan and update operations in the scan segments are not shown for simplicity.

This structure could be documented in a BSDL user package as follows.

```
-- Supplied by MyCorp for REG_1500 version 1.0

package REG_1500 is
   use STD_1149_1_2013.all;
end REG_1500;

package body REG_1500 is

   use STD_1149_1_2013.all;

Attribute REGISTER_MNEMONICS of REG_1500  : package is
  "WIR_decode  ( "&
    "WS_BYPASS  (0B0000) <Wrapper Bypass Instruction>, "&
    "WS_EXTEST  (0B0001) <Wrapper Serial External Boundary Instruction>, "&
    "WS_INTEST  (0B0010) <Wrapper Serial Internal Boundary Instruction>, "&
```

```
    "WS_BIST    (0B0100) <BIST Instruction>, "&
    "WP_ALL     (0B1xxx) <Wrapper Parallel instructions> "&
    " )" &                      -- end of WIR_decode
  " )";                         -- end of REGISTER_MNEMONICS

Attribute REGISTER_ASSEMBLY of REG_1500  : package IS
  "REG_1500 ( " &  -- The Select WIR bit and the Wrapper Serial Port
    -- Reset to WBY
    "(SELWIR [1] DelayPO ResetVal(0b0) TAPReset ),  "&
    "(WSP IS WSP_MUX) "&
  " ), "&                       -- end of REG_1500
  "WSP_MUX ( "&   -- The outer selectable segments: WIR and WDR
    "(SelectMUX "&
      -- Reset to WBY
      "(WIR IS WIR_Seg), "&
      "(WDR IS WDR_MUX) "&
      "SelectField (SELWIR) "&
      "SelectValues ((WIR : 0b1) (WDR : 0b0)) "&
    " ) "&                      -- end of SELECTMUX
  " ), "&                       -- end of WSP_MUX
  "WIR_Seg (  (WIR_field [4] "&
              "ResetVal(WIR_decode(WS_BYPASS)) TAPReset )    ), "&
  "WDR_MUX ( "&  -- The inner selectable segments: WBY, WBR, and Wusr
    "(SelectMUX "&
      "(WBY  IS Reg_WBY CAPTURES(0) ), "&
      "(WBR  IS Reg_WBR), "&
      "(WUSR IS Reg_WUSER) "&
      "SelectField (WIR) "&
      "SelectValues ("&
        "(WBY  : WS_BYPASS, WP_ALL) "&
        "(WBR  : WS_EXTEST, WS_INTEST) "&
        "(WUSR : WS_BIST) "&
        " ) "&                  -- end of SelectValues
    " ) "&                      -- end of SelectMUX
  " ), "&                       -- end of WDR_MUX
  "REG_WBY   ( (WBY[1] NOPO)), " &
  "REG_WBR   ( (WBR[8] )), " &
  "REG_WUSER ( (  CSR[4] NOUPD )," &
  "            (  GO [1] ResetVal(0b0) TapReset )   )" ;

end REG_1500;
```

When there are multiple identical cores, each with an identical wrapper, it is simple to connect them in series. When only one needs to be addressed, the others can select their bypass register to shorten the chain (that is their reset state). The following attribute statement from the component BSDL shows how to instantiate three WSP in series:

```
...
Attribute REGISTER_ASSEMBLY of MyChip_1500 : entity IS
  "Reg_1500_Series ( "&
    -- Other possible segments
    "(ARRAY WSP(1 TO 3) IS REG_1500) "&  -- See previous example
--   (ARRAY WSP(1 TO 3) IS Package REG_1500 : REG_1500) -- if name conflicts
    -- Other possible segments
  ")";
...
```

When there are multiple identical cores, each with an identical wrapper, it can be advantageous to scan the wrappers and run the tests in parallel. This, of course, requires that the wrapper or core be designed to hold the test results

until they can be captured and scanned out, one wrapper at a time. Figure B-14 shows three wrappers (each an instance of the Reg_1500 defined above) in parallel. All are scanned simultaneously whenever one of them is selected for scan-out (gating logic on the WSC not shown), and each one can be selected for scan out individually. In addition, all three can be bypassed.

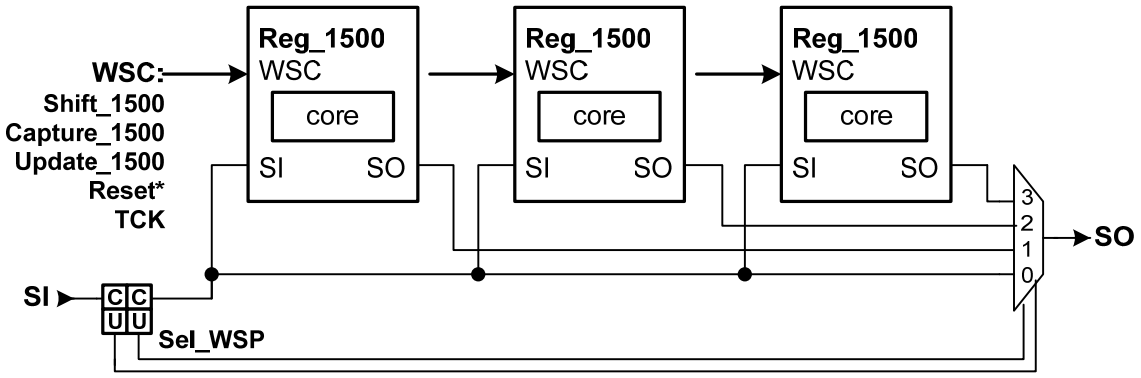


Figure B-14—Three wrappers in parallel

This structure could be documented in the component BSDL as follows.

```
...
Attribute REGISTER_MNEMONICS of MyChip_1500 : entity is
      "WSP  ( "&
         "   None   (0B00) <Bypass all WSPs>, "&
         "   WSP1   (0B01) <WSP1>, "&
         "   WSP2   (0B10) <WSP2>, "&
         "   WSP3   (0B11) <WSP3> "&
         "  )";
Attribute REGISTER_ASSEMBLY of MyChip_1500 : entity IS
  "WIRE ( (WIRE[0]) ), "&
  "Reg_1500_Parallel ( "&
    -- Other possible segments
    "(Sel_WSP[2] DelayPO ResetVal(WSP(None)) TAPReset ), "&
    "(SELECTMUX "&
      "(WIRE1 is WIRE), "&
      "(ARRAY WSP(1 TO 3) IS REG_1500) "&
      "SELECTFIELD (Sel_WSP) "&   -- 4:1 selection
      "SELECTVALUES ( "&
        "(WIRE1:None) (WSP(1):WSP1) (WSP(2):WSP2) (WSP(3):WSP3) ) "&
      "BROADCASTFIELD (Sel_WSP) "&
      -- Always scan data into all three WSP when any are selected.
      "BROADCASTVALUES ((WSP(1),WSP(2),WSP(3) : WSP1,WSP2,WSP3)) "&
    ")"&
    -- Other possible segments
  ")";
...
```

Note that the two selection fields (SelectField and BroadcastField) use the same physical field, which is the recommended practice. That field then selects both which WSP is connected to scan out and which of the WSPs will scan data. In this case, all WSPs scan at all times, which assumes that WSP registers are never required to hold their data when the others are being scanned. Results would have to be held in core registers for capture, if necessary.

B.8.22 Register constraint description

The optional **REGISTER_CONSTRAINTS** attribute is used to augment test, debug, and diagnosis by documenting structural constraints on values to be written to a TDR. Constraints define expressions that are evaluated as applicable prior to performing a scan of a TDR. If any of the applicable constraint check conditions evaluate to "TRUE," an undesired condition exists and the register scan should not be performed. A severity level and an information tag are provided to aid the test engineer in understanding the violation.

The expressions are based on SystemVerilog expressions as documented in 11.2 of IEEE Std 1800™-2012.[8] These expressions are also similar to expressions in Tcl, C, C++, and other languages. A subset of the Verilog operators is used, and for that subset, they follow the definitions and precedence of the Verilog expression operators. The evaluation of the expression follows the rules for such expressions with the exception that the "X" state of a value is treated as "don't-care," permitting a match to any value, instead of "unknown." As this type of expression is well defined, only the basics and the exceptions are defined in detail here.

B.8.22.1 Specifications

Syntax

<register constraints description> ::= **attribute REGISTER_CONSTRAINTS of** <target> **is**
    <constraints string> <semicolon>

<constraints string>::= <quote> <constraints list> { <comma> <constraints list> } <quote>
<constraints list>::= <constraint domain> <left paren> <constraint checks> <right paren>
<constraint domain> ::= <reg or seg name> | **entity** | **package**
<constraint checks>::= <left paren> <check expression> <right paren>
    <constraint severity> <information tag>
    { <comma> <left paren> <check expression> <right paren>
    <constraint severity> <information tag> }
<constraint severity> ::= **error** | **warning** | **info**

<check expression> ::= <short expression> | <binary expr>
<short expression> ::= <nested expr> | <unary expr> | <field reference> | <oper val>
<nested expr> ::= <left paren> <check expression> <right paren>
<unary expr> ::= <logical inv expr> | <bit-wise inv expr> | <one hot expr>
<logical inv expr> ::= <logical inversion> <short expression>
<bit-wise inv expr> ::= <bit-wise inversion> <short expression>
<one hot expr> ::= <one hot> <nested expression>
<binary expr> ::= <short expression> <binary operator> <check expression>
<binary operator> ::= <exponentiation> | <multiplication> | <division> | <remainder> |
    <addition> | <subtraction> | <right shift> | <left shift> | <less than> | <greater than> |
    <less than or equal> | <greater than or equal> | <equals> | <not equals> |
    <bit-wise and> | <bit-wise xor> | <bit-wise or> | <logical and> | <logical or>

<oper val> ::= <mnemonic pattern> | <binary pattern> | <hex pattern> | <decimal pattern>
<mnemonic pattern> ::= [[**PACKAGE** <package hierarchy> <colon>] <mnemonic group name>]
    <left brace> <mnemonic identifier> <right brace>

[8] IEEE publications are available from The Institute of Electrical and Electronics Engineers (http://standards.ieee.org/).

Rules

a) Any <constraint domain> of <TDR> shall be a previously defined register name as defined in this standard or in the **REGISTER_ACCESS** attribute.

b) A <constraint domain> of either **entity** or **package**, if used, shall match the object type of <target>.

c) The <constraint severity> of **error** shall indicate a condition requiring that the values not be scanned into the register.

NOTE 1—Response to <constraint severity> of **warning** or **info** is unspecified, and they may be user defined.

d) The <field reference> shall resolve to a previously defined register or register field as defined in this standard, or in the **REGISTER_ACCESS**, **REGISTER_FIELDS,** or **REGISTER_ASSEMBLY** attributes, and the value for evaluation shall be the data to be written to the register or register field.

e) The <mnemonic pattern> shall resolve to a mnemonic value previously defined in a **REGISTER_MNEMONICS** attribute.

f) A <check expression> shall contain at least one <field reference>.

g) A <field reference> shall be composed of zero or more <instance reference> fields in parent-to-child hierarchical order, each <instance reference> followed by a period, and followed by an <extended field name>.

h) The <field length> of an <field reference> shall not be zero.

i) All operands (<field value> and <field reference>) shall be converted into their binary equivalent value before evaluation by a bit-wise operator, into an arithmetic value before evaluation by an arithmetic operator, and into a logical value before evaluation by a logical operator.

NOTE 2—The rules and operands defined here do not match other languages using these operands exactly; any user intending to evaluate these constraints in existing evaluation software will need to process the operands to make them compatible with that software.

j) All operands shall have the logical value of "FALSE" if they have the numeric or binary equivalent value of zero, and the logical value of "TRUE" for any other value, including bits with a value of X.

k) The operator tokens used in the syntax and listed in the "Operator Token" column of Table B-5 shall be represented in BSDL by the character or characters shown in the "Operator" column of Table B-5.

l) Unless noted otherwise in Table B-5, all operators take two operands and use "infix" notation; that is, the operator binds both the operand or <nested expression> to its left and the operand or <nested expression> to its right.

m) A <check expression> enclosed within parenthesis shall be evaluated prior to evaluation of the remaining expression outside the parenthesis.

n) Evaluation order (precedence) shall be by row in Table B-5, where the operators of a given row shall be evaluated before the operators of any later row.

o) Evaluation order (precedence) for multiple operators from a single row of Table B-5 shall be from left to right in the expression.

p) **one_hot** is an added custom operator and shall return a logical TRUE if the operand has a binary equivalent value with one and only one 1 bit.

q) It shall be an error if any operand of arithmetic operators (as listed in Table B-5) contain a bit with value X.

r) <bit-wise and>, <bit-wise or>, <bit-wise xor>, and <bit-wise inversion> operators all take binary equivalent operands and produce a binary equivalent value by operating on 0, 1, and X values on a bit-position by bit-position basis using the Verilog conventions for such operators.

s) <equals> and <not_equal> operators take binary equivalent operands and produce a logical (TRUE, FALSE) value by operating on 0, 1, and X values on a bit-position by bit-position basis; and further, contrary to the Verilog conventions for such operators, they treat X as a "don't-care" matching 0, 1, or X.

t) Arithmetic operators (as listed in Table B-5) take numeric operands and produce either a numeric value or, where noted in the table, a logical value.

u) Logical operators (as listed in Table B-5) all take logical operands and produce a logical value.

Table B-5—Constraint expression operators

Precedence	Operator token	Operator	Operator type and (Comments)
1	<logical inversion> <bit-wise inversion> <one hot>	**!** ~ **one_hot**	Logical operator Bit-wise operator Custom operator (result is logical) (Operator binds to the token to its right.)
2	<exponentiation>	**	Arithmetic operator
3	<multiplication> <division> <remainder>	* / **%**	Arithmetic operators
4	<addition> <subtraction>	+ –	Arithmetic operators
5	<left shift> <right shift>	<< >>	Arithmetic operators (Right-hand operand is the number of shifts.)
6	<less than> <greater than> <less than or equal> <greater than or equal>	< > <= >=	Arithmetic operators (Result is logical.)
7	<equals> <not equal>	== **!=**	Bit-wise operators (Result is logical.)
8	<bit-wise and>	**&**	Bit-wise operator
9	<bit-wise xor>	^	Bit-wise operator
10	<bit-wise or>	|	Bit-wise operator
11	<logical and>	**&&**	Logical operator
12	<logical or>	||	Logical operator

B.8.22.2 Description

The optional **REGISTER_CONSTRAINTS** attribute allows checking for conditions needed to satisfy design constraints. The purpose is to assist in test development and debug by validating that register values are valid and meet the structural design conditions prior to being scanned to the register. The constraint evaluates to a logical value of TRUE or FALSE and, if TRUE, can be flagged as an error, a warning, or information, at the discretion of the designer. In addition, an information tag can be supplied to briefly describe the nature of the problem. A constraint flagged as an error is intended to prevent the register scan by stopping the test.

The typical type of constraint expected is a simple comparison of a field to some fixed value, possibly logically combined with a simple comparison of another field, usually in the same TDR, to some fixed value. However, since physical constraints on a register field or some combination of register fields could be just about anything, a fully capable expression evaluation capability is provided. The operators and the evaluation rules provided are similar to commonly available languages in software development, hardware development, and script development. In particular, they are part of Tcl, the language base for PDL (see Annex C). If constraints are defined for a component, then they need to be evaluated prior to scanning data into the component.

The constraints represent a statement of some structural characteristic of the design. As shown in the examples, it could be two power domains that cannot be powered up at the same time, or a power sequencing constraint between domains. It could also be programmable I/O where not all combinations of parameters are valid, even if the hardware would accept the invalid settings. It could be as simple as checking a test cycle count to a BIST engine where the register field will hold a larger value than the BIST engine can handle, or comparing upper and lower limit fields to verify that the upper is greater than the lower.

B.8.22.3 Examples

In this example, the IC contains two power domains: A and B. The IC design structure is such that power domain A cannot be turned on at the same time as power domain B. If the attempt is made, the hardware will ignore one of the requests, but which one may not be deterministic. Assuming PDA and PDB are DomCtrl cells in the init_data register, and both use a mnemonic group that includes a name "Override" for the value that would turn on the power domains, then we can express the requirement that both power domains not be turned on at the same time as follows:

```
attribute REGISTER_CONSTRAINTS of init_example : entity is
  "init_data (" &
     "( (PDA=={Override})&&( PDB == { Override } ) ) "&
        " ERROR <Domain A & B cannot both be ON at the same time> "&
   ")";
```

This next example again shows the flexible SerDes I/O channel, which can be programmed to support multiple protocols and multiple output voltage swings. However, some voltage swings are incompatible with certain protocols. This assumes the hardware will actually do the invalid combination, but the communication link will probably not work, even at low speed. See B.8.18.3 for the mnemonic definitions and B.8.21.3 for the field definitions.

```
attribute REGISTER_CONSTRAINTS of XYZ_SERDES : package is
   "init_data (" &
     " ((TX_Swing == {200mv}) && (Protocol == {SRIO}) ) "&
        " WARNING <A differential Swing of 200mv is not valid with SRIO. "&
        "The driver will do it, but communication may not work.> "&
   ")";
```

B.8.23 Register and power port association attributes

These optional attributes are used to augment test, debug, and diagnosis by providing information that may point to causes of incorrect behavior.

The **REGISTER_ASSOCIATION** attribute associates BSDL ports with register fields to indicate either that the content in the register fields may modify the behavior of the associated ports or that the associated ports may modify the values captured by the bits of the field.

The **POWER_PORT_ASSOCIATION** attribute associates BSDL ports with specific power ports (usually voltage reference ports) to indicate that the power port may modify the behavior of the associated ports.

B.8.23.1 Specifications

Syntax

<register association description> ::= **attribute REGISTER_ASSOCIATION**
of <target> **is** <register association string> <semicolon>

<register association string>::= <quote> <register association list>
{ <comma> <register association list> } <quote>
<register association list>::= <reg field or instance> <colon> <association list> { <association list> }
<reg field or instance> ::= <field or instance name> [<left paren> <index> <right paren>]
<field or instance name> ::= <extended field name> | <segment ident> | <array segment ident> | <TDR>
<association list> ::= <port list> | <info list> | <clock list> | <user list> | <unit>
<port list> ::= **port** <port association list>
<port association list> ::= <left paren> <port ID> { <comma> <port ID> } <right paren>
<info list> ::= **info** <left paren> <information tag> { <comma> <information tag> } <right paren>
<clock list> ::= **sysclock** <left paren> <port ID> { <comma> <port ID> } <right paren>

<user list> ::= **user** <user list name> <left paren> <single or multi list> <right paren>
<user list name> ::= <VHDL identifier>
<single or multi list> ::= <single word user list> | <multi-word user list>
<single word user list> ::= <VHDL identifier> { <comma> <VHDL identifier> }
<multi-word user list> ::= <information tag> { <comma> <information tag> }
<unit> ::= **unit** <left paren> <unit name> <unit definition> <right paren>
<unit definition> ::= <left brace> <unit value> [<unit scale>] [<unit link>] <right brace>
<unit name> ::= <VHDL identifier>
<unit value> ::= <hex pattern>
<unit scale> ::= <real>
<unit link> ::= <information tag>

<power port association description> ::= **attribute POWER_PORT_ASSOCIATION**
of <entity target> **is** <power port association string> <semicolon>

<power port association string>::= <quote> <power port association list>
{ <comma> <power port association list> } <quote>
<power port association list>::= <power port id> <colon> <port association list>
<power port id> ::= <port ID>

Rules

a) <field or instance name> shall be a previously defined register or register field as defined in this standard, in a **REGISTER_ACCESS**, **REGISTER_FIELDS** or **REGISTER_ASSEMBLY** attribute, or shall be an instance or array name previously defined in a **REGISTER_ASSEMBLY** attribute.

b) A <field or instance name> shall appear no more than once in a <register association list>.

c) Any <association list> shall contain, within parenthesis, either a single entry, which is associated with all bits of the <reg field or instance>, or a comma separated list of the same length as the <reg field or instance>, which is associated with the individual bits of the <reg field or instance> in order from closest to TDI to closest to TDO.

d) In any <clock list>, each <port ID> shall be a <port ID> identified as a system clock in a **SYSCLOCK_REQUIREMENTS** attribute.

e) Each <user list name> shall be unique within a <register association list>.

f) In any <unit>, the <unit value> shall be a <hex pattern> of 22 hex characters as defined in in 4.11 of IEEE Std 1451.0-2007.

NOTE—See Table B-6 for a partial definition. The table is for reference only.

g) In any <unit>, the optional <unit link> shall be a valid file (file://...) or Internet (http://...) URL locating a transducer electronic data sheet (TED) as defined in IEEE Std 1451.

h) <power port id> shall be a previously defined <port ID> in the <logical port description> with the <pin type> of **POWER_POS**, **POWER_NEG**, **POWER_0**, or **VREF_IN**.

i) The value of the <index> associated with a <field name> shall be less than the <field length> of the register field.

j) An <index> shall not be associated with a <segment ident>.

k) The value of the <index> associated with an <array segment ident> shall be a valid index into the <range> of the associated <array ident> in a **REGISTER_ASSEMBLY**.

l) An <index> shall be associated with a <power port id> only if the <port ID> was defined as type bit_vector, and it shall be a valid index into the <range> of the <port ID>.

m) No <port ID> value shall occur more than once in a given <port association list>.

n) Each <portID> in a <port association list> shall be a previously defined <portID> in the <logical port description>.

o) Each indexed <portID> in a <port association list> shall be a previous defined <portID> in the <logical port description> that also includes a <bit vector spec>, and the index shall lie within the <range> of the <bit vector spec>.

B.8.23.2 Description

The optional **REGISTER_ASSOCIATION** attribute allows annotating register bits or register fields with a specific information. The purpose is to assist in test, diagnostics, and debug.

The **port** list indicates which ports are affected by the register bits, or which ports affect the value captured by register bits. For example, the init_data register may have multiple fields that set impedances, pull-up or pull-down behavior, voltage swings, or any number of other analog characteristics, each for some set of ports. This attribute allows that relationship to be made explicit, and to identify exactly which bits of the init_data register (or other TDR) affect which ports.

The **sysclock** list is similar to the **port** list, except instead of specifying that the register bits capture these ports, it specifies that the register bits are clocked by this particular system clock port. The **SYSCLOCK_REQUIREMENTS** attribute identifies system clocks and specifies their usable frequency range and which instructions may require them. The **sysclock** list identifies which specific fields of a TDR are actually clocked by a particular system clock. This additional level of detail can be used to optimize testing.

The **REGISTER_ASSOCIATION** attribute is only used in the BSDL for an IC when the **port** or **sysclock** list is used since the port names to be associated with fields are not known in a package. Otherwise, multiple occurrences of <register association description> and <power port association description> are allowed.

The **info** list associates text strings with register fields or bits in a manner similar to the text strings associated with mnemonic values in the **REGISTER_MNEMONIC** attribute. For instance, this might be used to identify the use of the register in special test modes (memory BIST) or mission mode.

The **user** list allows the component supplier to provide one or more named lists of either single identifiers or text strings with register fields or bits. The name of the list would normally be the type of data being specified. This could allow design-specific or tool-specific information to be provided in an extensible format.

The **unit** specification allows the association of a unit description to a field to assist in interpreting the value contained in the field, especially when that field captures the output of some transducer. IEEE Std 1451.0 provides a standard way of defining units, and 4.11 of IEEE Std 1451.0-2007 is the source for this definition. The <unit name> is arbitrary; the <unit value> is a 22 character hex pattern comprising 11 eight-bit (2 hex character) fields described in Table B-6; the <unit scale> is a real number for scaling the digital integer value of the field; and the <unit link> is a file or Internet URL to a TED, which provides additional information, such as a nonlinear equations, and so on.

Table B-6—Unit value definitions

Field	Unit Description
1	Physical Unit Interpretation – Default to “00” (See 4.11 of IEEE Std 1451.0-2007)
2	(2 * <exponent of radians>) + 128
3	(2 * <exponent of steradians>) + 128
4	(2 * <exponent of meters>) + 128
5	(2 * <exponent of kilograms>) + 128
6	(2 * <exponent of seconds>) + 128
7	(2 * <exponent of amperes>) + 128
8	(2 * <exponent of kelvins>) + 128
9	(2 * <exponent of moles>) + 128
10	(2 * <exponent of candelas>) + 128
11	Default to “00”

Reference voltages are frequently key to the operation of IC I/O. **POWER_PORT_ASSOCIATION** documents in a machine-readable format the relationship of the reference voltages and the I/O, which depend on the input voltage. Even with initialization, failures of the I/O can occur simply because a **POWER_POS** or a **VREF_IN** voltage is not present or not turned on at the board level. For example, programmable devices may control voltage conversion circuits, and these programmable devices may not be programmed at the time test is performed. It is difficult for test engineers to determine the relationship of power and voltage references to particular I/O pins. **POWER_PORT_ASSOCIATION** resolves this issue and provides significant information to help in test, diagnosis, and debug.

B.8.23.3 Examples

The following associations might pertain to the fields defined in the Examples in B.8.19.3:

```
Attribute REGISTER_ASSOCIATION of init_example : entity is
   "VSEL_bits (4) : port (PwrUp_IO_VSEL(4)), "&
   "VSEL_bits (3) : port (PwrUp_IO_VSEL(3)), "&
   "VSEL_bits (2) : port (PwrUp_IO_VSEL(2)), "&
   "VSEL_bits (1) : port (PwrUp_IO_VSEL(1)), "&
   "VSEL_bits (0) : port (PwrUp_IO_VSEL(0)), "&
   "SerDesChannel(0) : port (IO_TXP(0),IO_TXN(0),IO_RXP(0),IO_RXN(0)), "&
   "SerDesChannel(1) : port (IO_TXP(1),IO_TXN(1),IO_RXP(1),IO_RXN(1)), "&
      . . .
   "SerDesChannel(16): port (IO_TXP(16),IO_TXN(16),IO_RXP(16),IO_RXN(16)), "&
   "SerDesChannel(17): port (IO_TXP(17),IO_TXN(17),IO_RXP(17),IO_RXN(17))";
```

For the VSEL_bits field (of the init_data register), the VSEL_bits field captures the values on five ports that control the voltage used to power various interfaces on the component, and it must be set properly at power-up prior to execution of *EXTEST*. In this case, there is a simple one-to-one relationship between the bits of the field and the ports. Note that this is specified bit-by-bit because the individual associations must be called out. Bus notation, that is, specifying a 5-bit register associated with a five bit bit_vector port, is not supported for fields.

The SerDesChannel is an array of register segments with each segment having two fields, both of which affect a specific set of four ports. Note that using an array range is not supported for arrays.

First, two memory BIST controller registers are individually associated with the clock that is needed to run both.

```
Attribute REGISTER_ASSOCIATION of MyChip : entity is
   "mem1.MBIST_CTRL : sysclock (F125MHz_in), "&
   "mem2.MBIST_CTRL : sysclock (F125MHz_in) ";
```

Next, the type of memory controller is associated with the memory controller register to assist in debug. Note that this assignment is made in a package body.

```
Attribute REGISTER_ASSOCIATION of ACME_MEMORIES_MBIST : package is
   " MBIST_CTRL : info (<ACME Memories BIST controller model 2Mx128>) ";
```

In some cases, there may be specific types of information associated with register bits. Here, in a stacked die situation, the TSV micro-bumps are listed for a six-bit register of control bits.

```
Attribute REGISTER_ASSOCIATION of MyChip : entity is
   "EXTERNAL_CTRL : user TSV_bump (F3,F6,F9,C4,C7,C10) ";
```

Where register fields are controlling or observing analog characteristics, for instance, the units can be specified. Here two eight-bit fields are specified, one for frequency and another for delay:

```
Attribute REGISTER_ASSOCIATION of mychip : entity is
```

```
      -- KHz is 1000 divided by seconds (field 6)
   "frequency : unit (KHz {0x00808080807E8080808000 1.0E+3} ), "&
      -- ms is .001 times seconds (field 6)
   "phase     : unit (ms  {0x00808080808280808080000 1.0E-3} )";
```

Finally, the following example shows three voltage reference ports associated with the ports they affect.

```
attribute POWER_PORT_ASSOCIATION of mydev : entity is
"DDR_REF1   :    ( DDR_DATA(7), "&
"                  DDR_DATA(6), "&
"                  DDR_DATA(5), "&
"                  DDR_DATA(4), "&
"                  DDR_DATA(3), "&
"                  DDR_DATA(2), "&
"                  DDR_DATA(1), "&
"                  DDR_DATA(0) ), "&
"IO_REF1     :   (  SERDES(0), SERDES(1) ), "&
"IO_REF2     :   (  SERDES(2), SERDES(3)  )";
```

B.8.24 User extensions to BSDL

Optional BSDL extensions provide a way to expand BSDL for proprietary needs without losing compatibility with the general definition of BSDL. The Standard BSDL Package **STD_1149_1_2013** defines a VHDL subtype **BSDL_EXTENSION** (as originally defined in Standard VHDL Package **STD_1149_1_1994**). It allows the user to define foreign attributes as being "BSDL extensions." These generally may be ignored by a BSDL parser. BSDL extensions appear in an entity description as the last portion before the (optional) **DESIGN_WARNING** (see B.8.25). In this manner, they may reference any data items defined previously.

B.8.24.1 Specifications

Syntax

<BSDL extensions> ::= <extension declaration> | <extension definition>
<extension declaration> ::= **attribute** <extension name> <colon> **BSDL_EXTENSION** <semicolon>
<extension definition> ::= **attribute** <extension name> **of** <target>
    is <extension parameter string> <semicolon>
<extension name> ::= <entity defined name> | <BSDL package defined name>
<entity defined name> ::= <VHDL identifier>
<BSDL package defined name> ::= <VHDL identifier>
<extension parameter string> ::= <string>

Rules

a) An <extension definition> shall appear after its corresponding <extension declaration>.

b) Any <VHDL identifier> appearing as a value of the <extension name> element in an <extension definition> shall appear also as the value of the <extension name> element of an <extension declaration> that occurs earlier in the BSDL description or in a BSDL package used in the BSDL description, and when the <extension declaration> occurs in a BSDL package, the <extension definition> shall appear in the body of the same BSDL package or in the BSDL description that uses that package.

c) Each <extension name> value in an <extension declaration> shall be unique even if the <extension name> values are defined in separate places, i.e., in separate user-supplied BSDL packages or one in the BSDL entity and one in a user-supplied BSDL package.

Permissions

d) An <extension name> element may appear in an <extension declaration> without appearing in any <extension definition> within a given BSDL description.

B.8.24.2 Description

The <extension declaration> may appear in the BSDL description itself or in a user-supplied BSDL package. The <extension definition> may appear in the BSDL description, regardless of where the declaration is, or in the body of the same user package as the declaration. If several BSDL extensions exist in the BSDL description, they may be intermixed in any manner as long as the declaration of an attribute precedes the definition of that attribute. The ability to define BSDL extensions in user-supplied BSDL packages allows for global definition of extensions.

When inventing names for <extension name> elements, take care to assure uniqueness of the names with respect to names created in other organizations that are also inventing extensions by choosing uncommon names not likely to be thought of by others. A company name could be appended to the name to maximize uniqueness.

B.8.24.3 Examples

Example 1

```
Package Global_extension is -- An example BSDL extension package
                                -- Does not define boundary cells,
                                --   just extensions
use STD_1149_1_2013.all;

-- Deferred constant declarations go here, if any (see B.10)

attribute First_extension  : BSDL_EXTENSION; -- Declare BSDL
attribute Second_extension : BSDL_EXTENSION; --  extensions here
attribute Third_extension  : BSDL_EXTENSION;

end Global_extension;

package body Global_extension is

-- Deferred constant definitions go here, if any (see B.10)

end Global_extension;
```

In the above example, a user-supplied BSDL package containing a BSDL extension is given; this package will be referenced by the entity of the next example.

Example 2

```
entity example is
generic (PHYSICAL_PIN_MAP : string := "DW_PACKAGE");

port (CLK:in bit; Q:out bit_vector(1 to 8);
D:in bit_vector(1 to 8);
GND: power_0 bit; VCC: power_pos bit;
OC_NEG:in bit; TDO:out bit; TMS, TDI, TCK:in bit);

use STD_1149_1_2013.all;
use Global_extension.all; -- Get declarations of
                          --   global extensions
```

. . . BSDL lines not relevant to this discussion are not being shown here . . .

```
-- Local declarations

attribute Local_extension1: BSDL_extension; --Declare local BSDL
attribute Local_extension2: BSDL_extension; --  extensions here

-- Now, define some proprietary extensions that were declared
--   in package - Global_Extension

attribute First_extension of example : entity is   -- Define attr.
" String of data " &                -- (global extension)
" in proprietary form. ";
attribute Second_extension of example: entity is
" More data, etc. ";

-- Local definition

attribute Local_extension1 of example : entity is  -- Define attr.
" Finally defined ";                  -- (local extension)

-- Optional design warning still located here --

end example;
```

In the above example, an entity is shown that uses global extensions as well as local extensions defined in the entity. Note that not all declared extensions are defined (e.g., `Third_extension`).

B.8.25 Design warning

A component designer may know of situations in which the system usage of a component can be subverted via the boundary-scan feature and cause circuit problems. As a simple example, a component may have dynamic system logic that requires clocking to maintain its state. Thus, clocking must be maintained when bringing the component out of system mode and into test mode for *INTEST*. The **DESIGN_WARNING** attribute is assigned a string message to alert future consumers of the potential for problems.

B.8.25.1 Specifications

Syntax

<design warning> ::= **attribute DESIGN_WARNING of** <target> **is** <string> <semicolon>

B.8.25.2 Description

The <design warning> may appear in the BSDL description itself or in a user-supplied BSDL package. This warning is for application-specific display purposes only. It is a textual message of arbitrary length with no specified syntax and is not intended for software analysis. No semantic checks are necessary.

B.8.25.3 Examples

```
attribute DESIGN_WARNING of My_IC:entity is
      "Dynamic device, " &
      "maintain clocking for INTEST.";
```

B.9 Standard BSDL Package STD_1149_1_2013

The following is the complete content of the Standard BSDL Package and Package Body: **STD_1149_1_2013**. This information defines the basis of BSDL and typically would be write-protected by a system administrator. An explanation of the cell definitions (e.g., **BC_1**, **BC_2**, etc.) in the package body is given in B.10,. BSDL descriptions that use <standard BSDL package identifier> **STD_1149_1_2013** must be processed using this Standard BSDL Package.

Note that what were formerly known, up to the 2001 version of this standard, as the Standard VHDL Package and the Standard VHDL Package Body are currently known as the Standard BSDL Package and Standard BSDL Package Body, respectively. This change was made in recognition of the fact that, as of the 2013 version of this standard, the syntax and semantics of BSDL are no longer a subset and standard practice of VHDL. However, the Standard BSDL Package remains pure VHDL, and its interpretation is defined in IEEE Std 1076.

```
--  STD_1149_1_2013    BSDL Package and Package Body
--
--  source       :  IEEE Std 1149.1-2013, B.9
--
-- NOTE-Where figures from the standard are cited,
--   the suffix 'c' denotes a control cell, and 'd' denotes a data cell.
--

package STD_1149_1_2013 is

-- Give component conformance declaration.

attribute COMPONENT_CONFORMANCE : string;

-- Give pin mapping declarations

attribute PIN_MAP : string;
subtype PIN_MAP_STRING is string;

-- Give TAP control declarations

type CLOCK_LEVEL is (LOW, BOTH);
type CLOCK_INFO  is record
  FREQ : real;
  LEVEL: CLOCK_LEVEL;
end record;

attribute  TAP_SCAN_IN   : boolean;
attribute  TAP_SCAN_OUT  : boolean;
attribute  TAP_SCAN_CLOCK: CLOCK_INFO;
attribute  TAP_SCAN_MODE : boolean;
attribute  TAP_SCAN_RESET: boolean;

-- Give instruction register declarations

attribute  INSTRUCTION_LENGTH : integer;
attribute  INSTRUCTION_OPCODE : string;
attribute  INSTRUCTION_CAPTURE : string;
attribute  INSTRUCTION_PRIVATE : string;

-- Give ID and USER code declarations
```

```
type ID_BITS is ('0', '1', 'x', 'X');
type ID_STRING is array (31 downto 0) of ID_BITS;

attribute IDCODE_REGISTER  :  ID_STRING;
attribute USERCODE_REGISTER:  ID_STRING;

-- Give register declarations

attribute REGISTER_ACCESS : string;
attribute REGISTER_MNEMONICS : string;
attribute REGISTER_FIELDS : string;
attribute REGISTER_ASSEMBLY : string;
attribute REGISTER_CONSTRAINTS : string;
attribute POWER_PORT_ASSOCIATION : string;
attribute REGISTER_ASSOCIATION : string;

-- Give boundary cell declarations

type BSCAN_INST is (EXTEST, SAMPLE, INTEST);
type CELL_TYPE is (INPUT, INTERNAL, CLOCK, OBSERVE_ONLY,
  CONTROL, CONTROLR, OUTPUT2, OUTPUT3, BIDIR_IN, BIDIR_OUT);
type CAP_DATA is (PI, PO, UPD, CAP, X, ZERO, ONE);
type CELL_DATA is record
  CT : CELL_TYPE;
  I  : BSCAN_INST;
  CD : CAP_DATA;
end record;
type CELL_INFO is array (positive range <>) of CELL_DATA;

-- Boundary cell deferred constants (see package body)

constant BC_0  : CELL_INFO;
constant BC_1  : CELL_INFO;
constant BC_2  : CELL_INFO;
constant BC_3  : CELL_INFO;
constant BC_4  : CELL_INFO;
constant BC_5  : CELL_INFO;
constant BC_7  : CELL_INFO;
constant BC_8  : CELL_INFO;
constant BC_9  : CELL_INFO;
constant BC_10  : CELL_INFO;

-- Boundary-scan register declarations

attribute BOUNDARY_LENGTH : integer;
attribute BOUNDARY_REGISTER : string;
attribute ASSEMBLED_BOUNDARY_LENGTH : array (0 to 1) of integer;
attribute BOUNDARY_SEGMENT : string;

-- Miscellaneous

attribute PORT_GROUPING : string;
attribute RUNBIST_EXECUTION : string;
attribute INTEST_EXECUTION : string;
attribute SYSCLOCK_REQUIREMENTS : string;
subtype BSDL_EXTENSION is string;
attribute COMPLIANCE_PATTERNS : string;
```

```
attribute DESIGN_WARNING : string;

end STD_1149_1_2013;  -- End of 1149.1-2013 Package


package body STD_1149_1_2013 is   -- Standard boundary cells

-- Generic cell capturing minimum allowed data

constant BC_0 : CELL_INFO :=
 ((INPUT,   EXTEST,  PI),      (OUTPUT2,  EXTEST,  X),
  (INPUT,   SAMPLE,  PI),      (OUTPUT2,  SAMPLE,  PI),
  (INPUT,   INTEST,  X),       (OUTPUT2,  INTEST,  PI),
  (OUTPUT3, EXTEST,  X),       (INTERNAL, EXTEST,  X),
  (OUTPUT3, SAMPLE,  PI),      (INTERNAL, SAMPLE,  X),
  (OUTPUT3, INTEST,  PI),      (INTERNAL, INTEST,  X),
  (CONTROL, EXTEST,  X),       (CONTROLR, EXTEST,  X),
  (CONTROL, SAMPLE,  PI),      (CONTROLR, SAMPLE,  PI),
  (CONTROL, INTEST,  PI),      (CONTROLR, INTEST,  PI),
  (BIDIR_IN,EXTEST,  PI),      (BIDIR_OUT, EXTEST, X ),
  (BIDIR_IN,SAMPLE,  PI),      (BIDIR_OUT, SAMPLE, PI),
  (BIDIR_IN,INTEST,  X ),      (BIDIR_OUT, INTEST, PI),
  (OBSERVE_ONLY, SAMPLE, PI),  (OBSERVE_ONLY, EXTEST, PI) );

-- Description for Figure 11-19, Figure 11-31, Figure 11-35c, Figure 11-35d,
Figure 11-37c, Figure 11-47d

constant BC_1 : CELL_INFO :=
 ((INPUT,   EXTEST,  PI),  (OUTPUT2,  EXTEST,  PI),
  (INPUT,   SAMPLE,  PI),  (OUTPUT2,  SAMPLE,  PI),
  (INPUT,   INTEST,  PI),  (OUTPUT2,  INTEST,  PI),
  (OUTPUT3, EXTEST,  PI),  (INTERNAL, EXTEST,  PI),
  (OUTPUT3, SAMPLE,  PI),  (INTERNAL, SAMPLE,  PI),
  (OUTPUT3, INTEST,  PI),  (INTERNAL, INTEST,  PI),
  (CONTROL, EXTEST,  PI),  (CONTROLR, EXTEST,  PI),
  (CONTROL, SAMPLE,  PI),  (CONTROLR, SAMPLE,  PI),
  (CONTROL, INTEST,  PI),  (CONTROLR, INTEST,  PI) );

-- Description for Figure 11-15, Figure 11-32, Figure 11-36c, Figure 11-36d,
Figure 11-38c,
--   Figure 11-39c, Figure 11-40(output) and Figure 11-42c

constant BC_2 : CELL_INFO :=
 ((INPUT,   EXTEST,  PI),  (OUTPUT2, EXTEST,   UPD),
  (INPUT,   SAMPLE,  PI),  (OUTPUT2, SAMPLE,   PI),
  (INPUT,   INTEST,  UPD),  -- Intest on output2 not supported
  (OUTPUT3, EXTEST,  UPD), (INTERNAL, EXTEST,  PI),
  (OUTPUT3, SAMPLE,  PI),  (INTERNAL, SAMPLE,  PI),
  (OUTPUT3, INTEST,  PI),  (INTERNAL, INTEST,  UPD),
  (CONTROL, EXTEST,  UPD), (CONTROLR, EXTEST, UPD),
  (CONTROL, SAMPLE,  PI),  (CONTROLR, SAMPLE,  PI),
  (CONTROL, INTEST,  PI),  (CONTROLR, INTEST,  PI) );

-- Description for Figure 11-16

constant BC_3 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),    (INTERNAL, EXTEST,  PI),
```

```
  (INPUT, SAMPLE,  PI),    (INTERNAL, SAMPLE,  PI),
  (INPUT, INTEST,  PI),    (INTERNAL, INTEST,  PI) );

-- Description for Figure 11-17, Figure 11-18, Figure 11-40(input)

constant BC_4 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),        -- Intest on input not supported
  (INPUT, SAMPLE,  PI),
  (OBSERVE_ONLY, EXTEST, PI),
  (OBSERVE_ONLY, SAMPLE, PI),  -- Intest on observe_only not supported
  (CLOCK, EXTEST,  PI),   (INTERNAL, EXTEST,  PI),
  (CLOCK, SAMPLE,  PI),   (INTERNAL, SAMPLE,  PI),
  (CLOCK, INTEST,  PI),   (INTERNAL, INTEST,  PI) );

-- Description for Figure 11-47c, a combined input/control

constant BC_5 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),   (CONTROL, EXTEST,  PI),
  (INPUT, SAMPLE,  PI),   (CONTROL, SAMPLE,  PI),
  (INPUT, INTEST,  UPD),  (CONTROL, INTEST,  UPD) );

-- Description for Figure 11-39d, a reversible cell
-- !! Not recommended; replaced by BC_7 below !!


-- Description for Figure 11-38d, self-monitor reversible
-- !! Recommended over cell BC_6 !!

constant BC_7 : CELL_INFO :=
 ((BIDIR_IN, EXTEST,  PI),  (BIDIR_OUT, EXTEST,  PO),
  (BIDIR_IN, SAMPLE,  PI),  (BIDIR_OUT, SAMPLE,  PI),
  (BIDIR_IN, INTEST,  UPD), (BIDIR_OUT, INTEST,  PI) );

-- Description for Figure 11-41, Figure 11-42d

constant BC_8 : CELL_INFO :=
 -- Intest on bidir not supported
 ((BIDIR_IN, EXTEST,  PI),  (BIDIR_OUT, EXTEST,  PO),
  (BIDIR_IN, SAMPLE,  PI),  (BIDIR_OUT, SAMPLE,  PO) );

-- Description for Figure 11-33

constant BC_9 : CELL_INFO :=
 -- Self-monitoring output that supports Intest
 ((OUTPUT2, EXTEST,  PO),  (OUTPUT3, EXTEST,  PO),
  (OUTPUT2, SAMPLE,  PI),  (OUTPUT3, SAMPLE,  PI),
  (OUTPUT2, INTEST,  PI),  (OUTPUT3, INTEST,  PI) );

-- Description for Figure 11-34

constant BC_10 : CELL_INFO :=
 -- Self-monitoring output that does not support Intest
 ((OUTPUT2, EXTEST,  PO),  (OUTPUT3, EXTEST,  PO),
  (OUTPUT2, SAMPLE,  PO),  (OUTPUT3, SAMPLE,  PO) );

-- Register segment field definitions for excludable segments and
--    (power) domain control.
```

```
attribute REGISTER_MNEMONICS of  STD_1149_1_2013 : package is
   "STD_MUX    (Include (1) < chain segment is included >, " &
   "            Exclude (0) < chain segment not included >), " &
   "STD_POWER  (On      (1) < Domain is functionally on >, " &
   "            Off     (0) < Domain is functionally off >), " &
   "STD_DOMSET (Override (1) < Force domain ON >, " &
   "            Normal   (0) < Domain in normal mode >) ";

attribute REGISTER_FIELDS of  STD_1149_1_2013 : package is
   "DOMCTRL[1] ( "&
      "(DOMCTRL[1] IS (0)  MON " &
      "                    DEFAULT(STD_DOMSET(Normal)) " &
      "                    RESETVAL(STD_DOMSET(Normal)) ) "&
      -- A reset type must be specified where this is instantiated
    "),"  &
   "SEGSEL[1] ( "&
      "(SEGSEL[1] IS (0)  DELAYPO " &
      "                    DEFAULT(STD_MUX(Exclude)) "&
      "                    RESETVAL(STD_MUX(Exclude)) "&
      -- A reset type must be specified where this is instantiated
      "                    CAPTURES(STD_POWER(-)) ) "&
    "),"  &
   "SEGMUX[0]   (( SEGMUX   [0] IS ()) ),"  &
   "SEGSTART[0] (( SEGSTART [0] IS ()) )";

end STD_1149_1_2013;  -- End of IEEE STD 1149.1-2013 Package Body
```

B.10 User-supplied BSDL packages

A user-supplied package and package body are used to express the behavior of user-designed boundary-scan register cells, mnemonics, and other test data register description attributes. It has a BSDL package section and a BSDL package body section.

The user-supplied BSDL package section is abbreviated compared to the Standard BSDL Package since the definitions of BSDL are supplied in the Standard BSDL Package specified by the <standard use statement>. The names of the cells that are defined in the BSDL package body must be given (they are called deferred constants).

A user-supplied BSDL package body is used for defining user-supplied boundary cells in the cell description constants, and BSDL extensions, register mnemonics, and other register description attributes may be supplied. A cell description constant is a specific VHDL constant record made up of a variable number of data triples containing VHDL enumerated types, and it specifies what the cell captures for each instruction.

NOTE—When writing a user-supplied BSDL package, it is possible to create an error if a later (e.g., 2001) construct such as a **BSDL_EXTENSION** is referenced that is not defined in an earlier-defined (e.g., 1990) Standard VHDL Package specified in the <standard use statement>. As with the BSDL itself, the standard use statement should reflect the compliance level of the BSDL user package and package body.

B.10.1 Specifications

Syntax

<user package> ::= <user package stmt> <user package body>

<user package stmt>::= **package** <user package name> **is** (see B.8.5.1)
<standard use statement> (see B.8.4)

{ <deferred constant> }
{ <extension declaration> } (see B.8.24)
end <user package name> <semicolon>

<deferred constant>::= **constant** <cell name> <colon> **CELL_INFO** <semicolon>

<user package body>::= **package body** <user package name> **is-**
<standard use statement> (see B.8.4)
{ <use statement> } (see B.8.5)
{ <cell description constant> }
{ <register mnemonics description> } (see B.8.18)
{ <register fields description> } (see B.8.19)
{ <register assembly description> } (see B.8.21)
{ <register constraints description> } (see B.8.22)
{ <register association description> } (see B.8.23)
{ <extension definition> } (see B.8.24)
[<design warning>] (see B.8.25)
end <user package name> <semicolon>

<cell description constant> ::= **constant** <cell name> <colon> **CELL_INFO**
<colon-equal> <left-paren> <capture descriptor list> <right-paren> <semicolon>
<capture descriptor list>::= <capture descriptor> { <comma> <capture descriptor> }
<capture descriptor>::= <left-paren> <cell context> <comma> <capture instruction> <comma>
<data source> <right-paren>
<cell context>::= **INPUT** | **OUTPUT2** | **OUTPUT3** | **INTERNAL** | **CONTROL** |
CONTROLR | **CLOCK** | **BIDIR_IN** | **BIDIR_OUT** | **OBSERVE_ONLY**
<capture instruction>::= **EXTEST** | **SAMPLE** | **INTEST**
<data source>::= **PI** | **PO** | **CAP** | **UPD** | **ZERO** | **ONE** | **X**

NOTE—Although the standard-defined instruction *PRELOAD* does operate the boundary-scan register, it is not listed as a <capture instruction> element since all cells capture unspecified data (**X**) when *PRELOAD* is in effect (see 8.7).

Rules

a) The <user package name> value shall be unique within a BSDL, or within a <user package>.
b) All <cell name> values shall be unique.
c) The <user package name> value in the <user package> shall match the <user package name> value in the <user package body>, and both shall match the <user package name> in the <use statement>. (See B.8.5.1.)
d) The <cell context> values shall be as given in Table B-7:

Table B-7—Cell context element values and meanings

Cell context value	Meaning
INPUT	Control-and-observe (supports *INTEST* instruction) or observe-only cell monitoring a system input
CLOCK	An Observe cell for clock pins (supports *INTEST* instruction)
OUTPUT2	Two-state output cell
OUTPUT3	Three-state output cell
CONTROL	Output enable or direction control cell
CONTROLR	**CONTROL** with preset/clear at *Test-Logic-Reset*
BIDIR_IN	Single-cell bidirectional pin acting as input
BIDIR_OUT	Single-cell bidirectional pin acting as output
INTERNAL	Internal cell, not associated with a system pin
OBSERVE_ONLY	Redundant observe-only cell without control, associated with a system pin or system logic input or output

e) The <data source> values shall be as given in Table B-8.

Table B-8—Data source element values and meanings

Data source value	Meaning
PI	Parallel input
PO	Parallel output (the output pad if a driver is present)
CAP	Capture flip-flop data
UPD	Update flip-flop (or latch) data
ZERO	Constant 0
ONE	Constant 1
X	Unknown data

NOTE 1—Figure B-15 gives a general model of the data source possibilities.

NOTE 2—In the following tables, an "L" indicates compliance. An "M" indicates compliance in the case of merged cells only. An "A" indicates compliance when the cell is an additional cell not mandated by this standard. A <capture descriptor> is a (<cell context> <capture instruction> <data source>) element. An example of a noncompliant <capture descriptor> is **(INPUT, EXTEST, UPD)**.

f) For a <capture descriptor> element with a <cell context> element equal to **INPUT**, **CLOCK**, or **BIDIR_IN**, compliant <data source> values for given <capture instruction> elements shall be as given in Table B-9.

Table B-9—Compliant capture sources for <cell context> of INPUT, CLOCK, and BIDIR_IN

<capture instruction>	PI	PO	UPD	CAP	X	ZERO	ONE
EXTEST	L	—	—	—	—	—	—
SAMPLE	L	—	—	—	—	—	—
INTEST	L	L	L	L	L	L	L

g) For a <capture descriptor> element with a <cell context> element equal to **OUTPUT2**, **OUTPUT3**, or **BIDIR_OUT**, compliant <data source> values for given <capture instruction> elements shall be as given in Table B-10.

Table B-10—Compliant capture sources for <cell context> of OUTPUT2, OUTPUT3, and BIDIR_OUT

<capture instruction>	PI	PO	UPD	CAP	X	ZERO	ONE
EXTEST	L	L	L	L	L	L	L
SAMPLE	L	L[a]	—	—	—	—	—
INTEST	L	—	—	—	—	—	—

[a] This combination is compliant with 2001 and later versions of BSDL. Beginning with the 2001 version, this standard now allows an output of the system logic to be sampled at the data output (pin) of the associated output buffer as well as at the data input of the associated output buffer [see rule a) and rule h) of 11.6.1].

h) For a <capture descriptor> element with a <cell context> element equal to **CONTROL** or **CONTROLR**, compliant <data source> values for given <capture instruction> elements shall be as given in Table B-11.

Table B-11—Compliant capture sources for <cell context> of CONTROL and CONTROLR

<capture instruction>	PI	PO	UPD	CAP	X	ZERO	ONE
EXTEST	L	L	L	L	L	L	L
SAMPLE	L	—	—	—	—	—	—
INTEST	L	—	M	—	—	—	—

i) For a <capture descriptor> element with a <cell context> element equal to **INTERNAL**, compliant <data source> values for given <capture instruction> elements shall be as given in Table B-12.

Table B-12—Compliant capture sources for <cell context> of INTERNAL

<capture instruction>	PI	PO	UPD	CAP	X	ZERO	ONE
EXTEST	L	L	L	L	L	L	L
SAMPLE	L	L	L	L	L	L	L
INTEST	L	L	L	L	L	L	L

NOTE 3—For a <cell context> of **INTERNAL**, a <capture descriptor> value of **PI** is essentially identical to **X** since internal cells do not capture anything other than constant 0s (**ZERO**), 1s (**ONE**), or the values previously shifted in **CAP**, **UPD**, or **PO**.

j) For a <capture descriptor> element with a <cell context> element equal to **OBSERVE_ONLY**, compliant <data source> values for given <capture instruction> elements shall be as given in Table B-13.

Table B-13—Compliant capture sources for <cell context> of OBSERVE_ONLY

<capture instruction>	PI	PO	UPD	CAP	X	ZERO	ONE
EXTEST	L	—	—	—	—	—	—
SAMPLE[a]	L	—	—	—	L --	L	L
INTEST	A	—	—	—	—	—	—

[a]Beginning with IEEE Std 1149.1-2013, this standard now allows redundant **OBSERVE_ONLY cells** to capture from PI, X, Zero, or One in *SAMPLE*. An example would be **OBSERVE_ONLY** cells on the pins of a differential receiver. *SAMPLE* is still required on the **INPUT** boundary-scan cell on the single-ended side of the differential receiver; however, to ease implementation of pin level **OBSERVE_ONLY** cells on the differential pins and the impact they may have on functional performance, only *EXTEST* needs to be supported.

k) No combination of a <cell context> value and a <capture instruction> value shall appear more than once in a single <capture descriptor list>.

l) The <cell name> value of a <cell description constant> in a <user package body> shall match the <cell name> value of a <deferred constant> in the <user package>, where the <user package body> and <user package> specify the same <user package name>.

NOTE 4—In the 1990 version of BSDL, the *RUNBIST* instruction was included as one of the <capture instruction> elements, but it does not appear in any version of this standard after that. This reflects the fact that *RUNBIST* may or may not reference the boundary-scan register and that the **RUNBIST_EXECUTION** attribute has been added to describe *RUNBIST* capture behavior.

m) All deferred constants included in the <user package stmt> shall be defined in the <user package body>.

n) The <standard use statement> in the <user package stmt> shall match the <standard use statement> in the <user package body>.

o) Only one package section and one package body section shall be included in a single BSDL package file.

p) Identifiers declared in design-specific BSDL packages shall not be defined more than once in a single BSDL package.

Recommendations

q) In order to minimize naming conflicts between multiple user-supplied packages, the <user package name> should contain a string that is unique to the supplier of the package.

NOTE 5—This could be a corporate name (or abbreviation), the unique name of a corporate product line, or other such unique identifier.

B.10.2 Description

A user-supplied package may be used to document certain information that may apply to many components, and a single package can then be used with multiple BSDL files. This package may be provided by the component designer or an IP provider. Most of the attribute statements allowed are also allowed in BSDL, and they are defined in those clauses. Unique to the package are the "CELL_INFO" constants, which define the behavior of user-supplied boundary-scan register cells.

Figure B-15 shows a generalized model of a boundary cell with all possible capture values listed in Table B-8 through Table B-13 that may be specified for each cell and instruction. As defined before, **PI** is the parallel input of the cell, **PO** the parallel output, and SI and SO are the serial input and output of the boundary-scan cell. The names inside the large multiplexer selecting the capture value correspond to the source names in the tables.

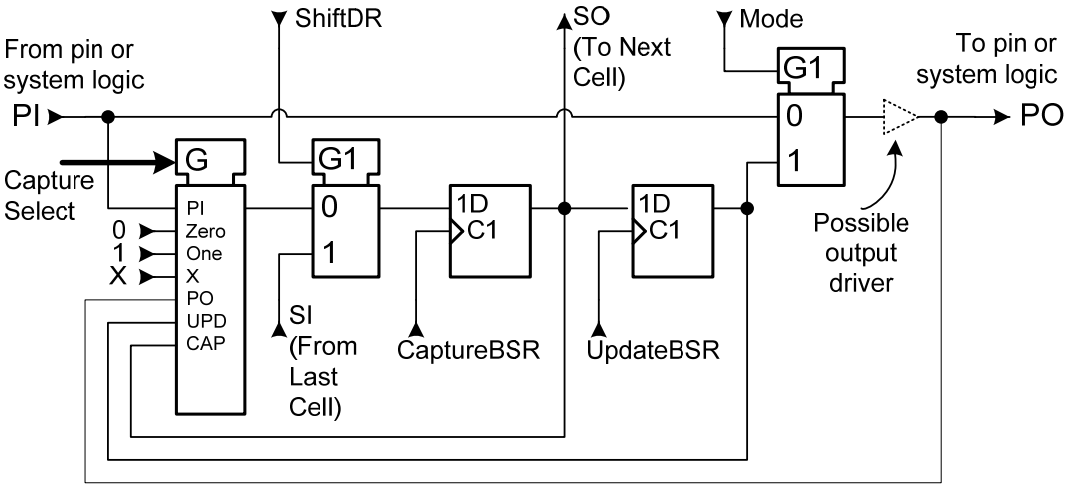


Figure B-15—Boundary-scan register cell showing possible capture sources

It is important to know the context of a cell to know how to interpret the data source. For example, the cell design in this standard at Figure 11-19 (called **BC_1**) can be used as an input cell (**INPUT**), an output cell for a two-state pin (**OUTPUT2**), a three-state pin (**OUTPUT3**), an internal cell (**INTERNAL**), or a control cell (**CONTROL**). This context determines how software interprets **PI** and **PO**. If the cell is used as an input (or is a bidirectional cell acting as an input), **PI** is interpreted as a system pin whose data are being captured. If the cell is used as an output (or a bidirectional cell acting as an output), **PI** is interpreted as the output from the system logic; during *EXTEST*, the cell would capture **X** unless a full simulation of the system logic were used to predict the system logic output. If the cell is used as an output, **PO** is the system pin; during *EXTEST*, the cell would capture board levels outside the component. When the cell is used as an input, **PO** will capture **X**.

With the exception of <cell context> values of **BIDIR_IN** and **BIDIR_OUT**, all the <cell context> values in Table B-7 map onto the like-named <function> values in Table B-4 and supporting text (see B.8.14.3.3). The <cell context> values of **BIDIR_IN** and **BIDIR_OUT** both map onto the <function> value **BIDIR**. The behavior of a **BIDIR** cell is dependent on whether it is currently set to be driving data out (**BIDIR_OUT**) or receiving data in (**BIDIR_IN**), as determined by the data value contained in the controlling cell identified by the <ccell> value.

B.10.3 Examples

Example 1

```
(INPUT, EXTEST, PI)
```

This can be read as "for this cell used as an **INPUT** cell, while *EXTEST* is in effect, the capture flip-flop loads the Parallel Input (**PI**) data during *Capture-DR* controller state."

Example 2

```
(BIDIR_IN, INTEST, UPD)
```

This can be read as "for this cell used as a bidirectional cell acting as an input (**BIDIR_IN**), while *INTEST* is in effect, the capture flip-flop loads the value of the Update flip-flop (or latch) data (**UPD**) during *Capture-DR* controller state."

Example 3

```
(OUTPUT2, SAMPLE, PI)
```

This can be read as "for this cell used as a (two-state) output cell (**OUTPUT2**), while *SAMPLE* is in effect, the capture flip-flop loads the Parallel Input (**PI**) data during *Capture-DR* controller state."

Example 4

```
(OUTPUT3, EXTEST, ZERO)
```

This can be read as "for this cell used as a (three-state) output cell (**OUTPUT3**), while *EXTEST* is in effect, the capture flip-flop loads a logic 0 (**ZERO**) during *Capture-DR* controller state."

User-supplied package for boundary-register cells

The following is an example of a user-supplied BSDL package that describes two new cells. These cells are able to capture constants (0 and 1) during certain situations. For example, used as outputs during *EXTEST*, they capture constant data rather than the system logic values (usually interpreted as "X"). By using these cells in the output cell positions of a boundary-scan register, it is possible to implement an informal ID code. They will capture a pattern of 1 and 0 bits.

```
-- User-defined package describing two new cells

package USER_PACKAGE is

use STD_1149_1_2013.all;      -- Get definition of "Cell_Info"

-- Boundary Cell deferred constants (defined in package body)

constant USER_0  : CELL_INFO;
constant USER_1  : CELL_INFO;

end USER_PACKAGE;                   -- End of user package


package body USER_PACKAGE is        -- User boundary cells

use STD_1149_1_2013.all;

constant USER_0 : CELL_INFO :=
```

```
((OUTPUT2, EXTEST,  ZERO),
 (OUTPUT2, SAMPLE,  PI),
 (OUTPUT3, EXTEST,  ZERO), (INTERNAL, EXTEST,  ZERO),
 (OUTPUT3, SAMPLE,  PI),   (INTERNAL, SAMPLE,  PI),
 (OUTPUT3, INTEST,  PI),   (INTERNAL, INTEST,  PI),
 (CONTROL, EXTEST,  ZERO), (CONTROLR, EXTEST,  ZERO),
 (CONTROL, SAMPLE,  PI),   (CONTROLR, SAMPLE,  PI),
 (CONTROL, INTEST,  PI),   (CONTROLR, INTEST,  PI) );

constant USER_1 : CELL_INFO :=
((OUTPUT2, EXTEST,  ONE),
 (OUTPUT2, SAMPLE,  PI),
 (OUTPUT3, EXTEST,  ONE),  (INTERNAL, EXTEST,  ONE),
 (OUTPUT3, SAMPLE,  PI),   (INTERNAL, SAMPLE,  PI),
 (OUTPUT3, INTEST,  PI),   (INTERNAL, INTEST,  PI),
 (CONTROL, EXTEST,  ONE),  (CONTROLR, EXTEST,  ONE),
 (CONTROL, SAMPLE,  PI),   (CONTROLR, SAMPLE,  PI),
 (CONTROL, INTEST,  PI),   (CONTROLR, INTEST,  PI) );

end USER_PACKAGE;                    -- End of user package body
```

User-supplied package body for internal registers

Register mnemonics, register fields, and port associations may be used to describe internal registers of reusable architectural blocks. The architectural blocks may be a design feature consistent across a family of devices or the blocks may be intellectual property designs provided by an IP provider such as SERDES I/O, SERDES built-in-self-test, or memory built-in self-repair.

The BSDL descriptions for these blocks can be provided in a user package body where they can be recorded once and referenced by a “use” statement in the BSDL of devices that incorporate these architectural blocks. An extended example is shown in D.1.1.

```
use SERDES_Defs.all;  -- Package Contains SERDES family register descriptions
```

B.11 BSDL example applications

B.11.1 Typical application of BSDL

The following example is a complete example BSDL for a simple octal tri-state buffer component, namely, the Texas Instruments SN74BCT8374 Octal D Flip-Flop, using BSDL version **STD_1149_1_2013**.

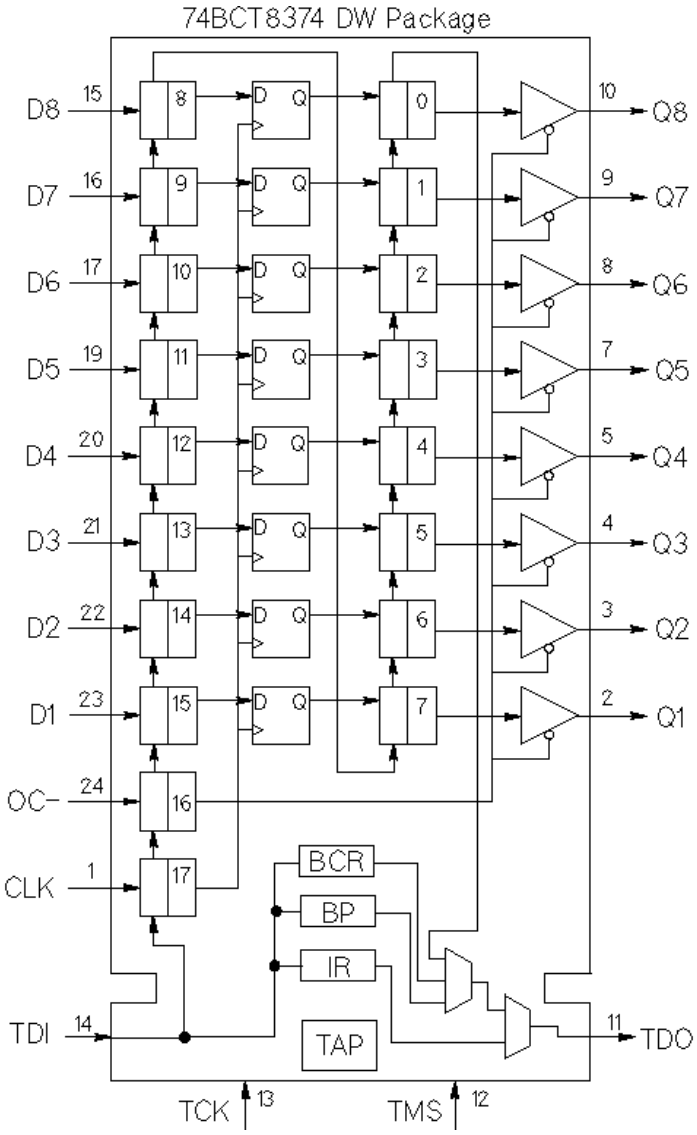


Figure B-16—Texas Instruments SN74BCT8374

```
entity ttl74bct8374 is
  generic (PHYSICAL_PIN_MAP : string := "DW");

  port (CLK:in bit; Q:out bit_vector(1 to 8); D:in bit_vector(1 to 8);
      GND: power_0 bit; VCC: power_pos bit;
      OC_NEG:in bit; TDO:out bit; TMS, TDI, TCK:in bit);

--Get IEEE Std 1149.1-2013 attributes and definitions
  use STD_1149_1_2013.all;

  attribute COMPONENT_CONFORMANCE of ttl74bct8374 : entity is
      "STD_1149_1_2013";

  attribute PIN_MAP of ttl74bct8374 : entity is PHYSICAL_PIN_MAP;
```

```
  constant DW:   PIN_MAP_STRING:="CLK:1, Q:(2,3,4,5,7,8,9,10), " &
         "D:(23,22,21,20,19,17,16,15)," &
         "GND:6, VCC:18, OC_NEG:24, TDO:11, TMS:12, TCK:13, TDI:14";

  constant FK:   PIN_MAP_STRING:="CLK:9, Q:(10,11,12,13,16,17,18,19)," &
         "D:(6,5,4,3,2,27,26,25)," &
         "GND:14, VCC:28, OC_NEG:7, TDO:20, TMS:21, TCK:23, TDI:24";

    attribute TAP_SCAN_IN    of TDI : signal is true;
    attribute TAP_SCAN_MODE  of TMS : signal is true;
    attribute TAP_SCAN_OUT   of TDO : signal is true;
    attribute TAP_SCAN_CLOCK of TCK : signal is (20.0e6, BOTH);

    attribute INSTRUCTION_LENGTH of ttl74bct8374 : entity is 8;

    attribute INSTRUCTION_OPCODE of ttl74bct8374 : entity is
       "BYPASS (11111111, 10001000, 00000101, 10000100, 00000001, 00000000),"
&
       "EXTEST (10000000)," &
       "SAMPLE (00000010, 10000010)," &
       "PRELOAD(00000010, 10000010),"&
       "INTEST (00000011, 10000011)," &
       "HIGHZ  (00000110, 10000110)," &
       "CLAMP  (00000111, 10000111)," &
       "RUNT   (00001001, 10001001)," &     -- Boundary run test
       "READBN (00001010, 10001010)," &     -- Boundary read normal
       "READBT (00001011, 10001011)," &     -- Boundary read test
       "CELLTST(00001100, 10001100)," &     -- Boundary self-test normal
       "TOPHIP (00001101, 10001101)," &     -- Boundary toggle-out test
       "SCANCN (00001110, 10001110)," &     -- BCR scan normal
       "SCANCT (00001111, 10001111)";       -- BCR scan test

    attribute INSTRUCTION_CAPTURE of ttl74bct8374 : entity is "10000001";

    attribute REGISTER_ACCESS of ttl74bct8374 : entity is
       "BOUNDARY (READBN, READBT, CELLTST)," &
       "BYPASS (TOPHIP, RUNT)," &
       "BCR[2] (SCANCN, SCANCT)";   -- 2-bit boundary control register

    attribute BOUNDARY_LENGTH of ttl74bct8374 : entity is 18;

    attribute BOUNDARY_REGISTER of ttl74bct8374 : entity is
      -- num cell  port      function   safe [input/ccell disval rslt]
        "17 (BC_1, CLK,      input,        X, OPEN1), " &
      -- Merged input/control on CELL 16
        "16 (BC_1, OC_NEG,   input,        X, OPEN1), " &
        "16 (BC_1, *,        control,      1), " &
        "15 (BC_1, D(1),     input,        X, OPEN1), " &
        "14 (BC_1, D(2),     input,        X, OPEN1), " &
        "13 (BC_1, D(3),     input,        X, OPEN1), " &
        "12 (BC_1, D(4),     input,        X, OPEN1), " &
        "11 (BC_1, D(5),     input,        X, OPEN1), " &
        "10 (BC_1, D(6),     input,        X, OPEN1), " &
        " 9 (BC_1, D(7),     input,        X, OPEN1), " &
        " 8 (BC_1, D(8),     input,        X, OPEN1), " &
      -- when cell 16 = 1, the Q outputs are at Hi-Z
```

```
        " 7 (BC_1, Q(1),     output3,      X,          16,   1,    Z)," &
        " 6 (BC_1, Q(2),     output3,      X,          16,   1,    Z)," &
        " 5 (BC_1, Q(3),     output3,      X,          16,   1,    Z)," &
        " 4 (BC_1, Q(4),     output3,      X,          16,   1,    Z)," &
        " 3 (BC_1, Q(5),     output3,      X,          16,   1,    Z)," &
        " 2 (BC_1, Q(6),     output3,      X,          16,   1,    Z)," &
        " 1 (BC_1, Q(7),     output3,      X,          16,   1,    Z)," &
        " 0 (BC_1, Q(8),     output3,      X,          16,   1,    Z)";

end ttl74bct8374;
```

B.11.2 Boundary-scan register description

The following examples illustrate a number of "special case" boundary-scan register structures.

B.11.2.1 Multiple cells per pin

Component pins can be serviced by more than one cell. Each cell can perform a different function. Note that this function is with respect to the boundary-scan register cell, not the component pin. For example, on a bidirectional pin (see Figure 11-37), one cell can serve as an input receiver while the other serves as an output driver. Additional **OBSERVE_ONLY** cells may be connected to any I/O pin.

The component shown in Figure B-17 will be used to illustrate a boundary-scan register with several **OBSERVE_ONLY** cells.

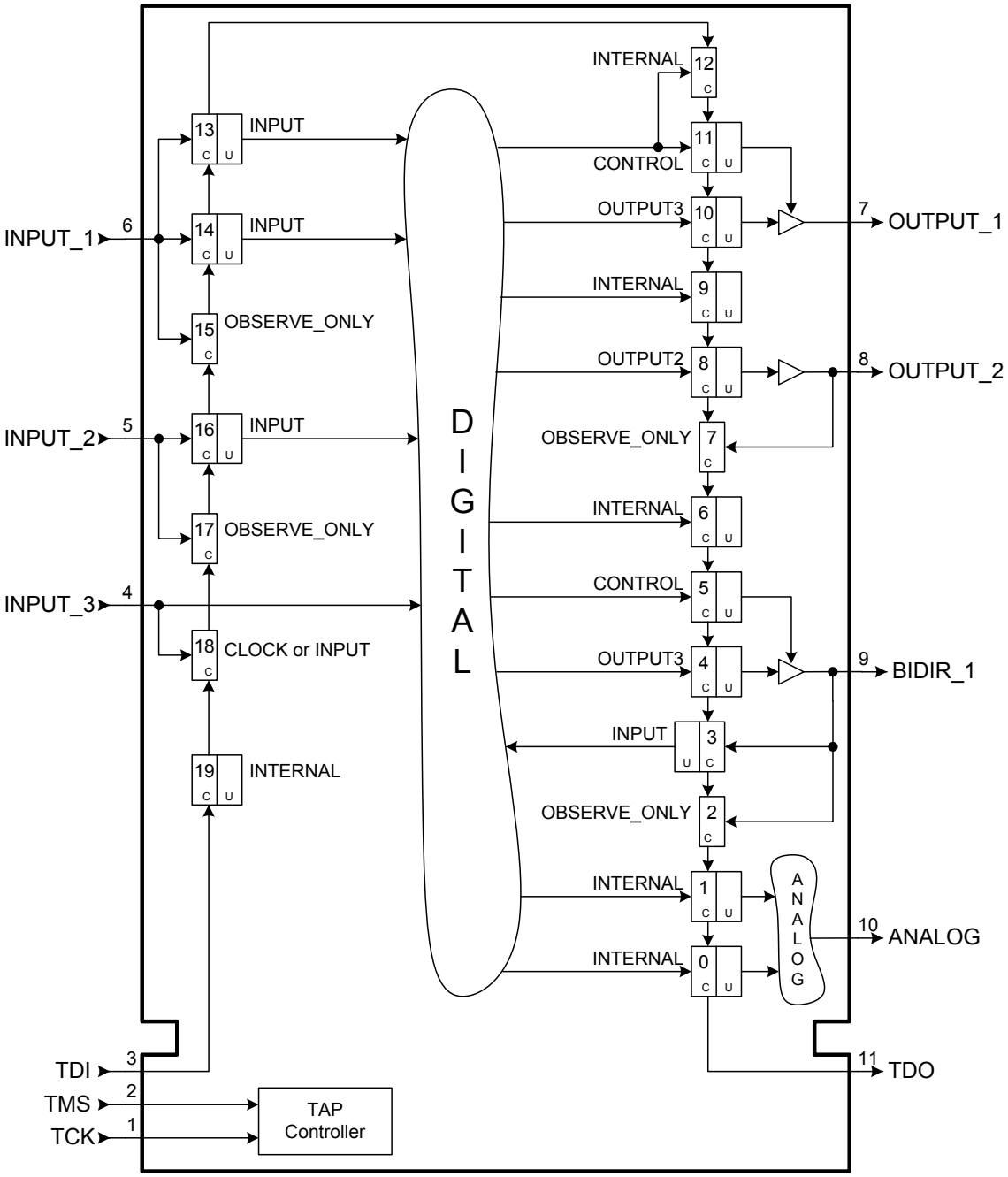


Figure B-17—Component that illustrates several OBSERVE_ONLY and INTERNAL cells

Cell 2 is an additional **OBSERVE_ONLY** cell associated with bidirectional pin 9 of the component.

Cell 7 is an additional **OBSERVE_ONLY** cell associated with output pin 8 of the component. Cells 7, 8, and 9 may have been a programmable two-cell bidirectional implementation that has been reprogrammed to a two-state output.

Cell 15 is an additional **OBSERVE_ONLY** cell associated with input pin 6 of the component. Cells 13 and 14 are also associated with pin 6, but they are described as **INPUT** cells and are connected to the system logic.

Cell 17 is an additional **OBSERVE_ONLY** cell associated with input pin 5 of the component. Cell 16 is the normal **INPUT** cell for pin 5.

If the component in Figure B-17 supports *INTEST*, and the signal `INPUT_3` is a repeating clock type, cell 18 must be of type **CLOCK**. If the component does not support *INTEST*, cell 18 could either be an **INPUT** cell or a **CLOCK** cell depending on the nature of the signal.

NOTE—The distinction here is that **INPUT** cells are required on inputs to the on-chip digital logic. **OBSERVE_ONLY** cells are used as optional and redundant cells where their omission does not affect the compliance of this standard. In the 2001 version of this standard, this distinction was accidently merged. After the 2001 version, **INPUT** cells and **OBSERVE_ONLY** cells are maintained as separate, **INPUT** cells are required on all system logic inputs, and **OBSERVE_ONLY** cells are allowed on pins where their omission does not affect the compliance of this standard.

B.11.2.2 Internal boundary register cells

Internal boundary register cells can be used to capture "constants" or system-logic-dependent values (0s and 1s) within a design. One proposed use of this possibility is capturing of a hard-coded value (perhaps in the first few bits of the boundary-scan register) as an informal identification code. Another application is to monitor power connections to verify whether they are receiving the correct input supply, and to capture a data bit based on the measured results. If the power connections are good, the data loaded will be 1, for example, while a faulty power input would cause a 0 to be captured. Internal cells, with either control-and-observe capability or observe-only capability, may sit at the boundary between digital and analog sections of the core circuitry. Finally, there may be "extra," unused cells in a programmable component (see Clause 11).

Note that this standard does not allow system logic to surround internal cells, as shown in Figure 11-43.

The component shown in Figure B-17 will be used to illustrate a boundary-scan register with several **INTERNAL** cells.

Cells 0 and 1 are **INTERNAL** cells between the digital system logic and the analog system functions. Note that cells 0 and 1 are not associated with pin 10.

Cells 6, 9, and 12 are cells that are observing signals from the system logic, are not associated with system pins, and are described as **INTERNAL** cells.

Cell 19 is an extra cell in the boundary-scan register. It is not observing the system logic or a system pin and is described as an **INTERNAL** cell.

The definition of the boundary-scan register for Figure B-17 is as follows:

```
attribute BOUNDARY_LENGTH of Figure_B11: entity is 20;
attribute BOUNDARY_REGISTER of Figure_B11: entity is
--
--num cell  port      function   safe [input/ccell disval rslt]
--
" 0  (BC_1, *,        internal,     0),                          "&
" 1  (BC_1, *,        internal,     1),                          "&
" 2  (BC_4, BIDIR_1,  observe_only, X),                          "&
" 3  (BC_1, BIDIR_1,  input,        X, OPEN1),                   "&
" 4  (BC_1, BIDIR_1,  output3,      0,  5,           0,    Z),   "&
" 5  (BC_1, *,        control,      0),                          "&
" 6  (BC_1, *,        internal,     X),                          "&
" 7  (BC_4, OUTPUT_2, observe_only, X),                          "&
" 8  (BC_1, OUTPUT_2, output2,      1),                          "&
" 9  (BC_1, *,        internal,     X),                          "&
"10  (BC_1, OUTPUT_1, output3,      0,  11,          0,    Z),   "&
```

```
"11  (BC_1, *,         control,      0),                          "&
"12  (BC_4, *,         internal,     X),                          "&
"13  (BC_1, INPUT_1,   input,        X, OPEN0),                   "&
"14  (BC_1, INPUT_1,   input,        X, OPEN0),                   "&
"15  (BC_4, INPUT_1,   observe_only, X),                          "&
"16  (BC_1, INPUT_2,   input,        X, OPEN1),                   "&
"17  (BC_4, INPUT_2,   observe_only, X),                          "&
"18  (BC_4, INPUT_3,   input,        X, OPENX),                   "&
"19  (BC_1, *,         internal,     X)                           ";
```

B.11.2.3 Merged cells

The component shown in Figure B-18 will be used to illustrate a boundary-scan register description in which special cases are handled. These special cases arise because this standard allows boundary-scan register cells to be merged when the system logic between them is null (see, for example, Figure 11-44 and Figure 11-45). Cells may be merged if the "logic" between them is simply a data path, such as a wire or buffer. When the merging is done, the resulting cell must obey a combination of the rules of the merged cells.

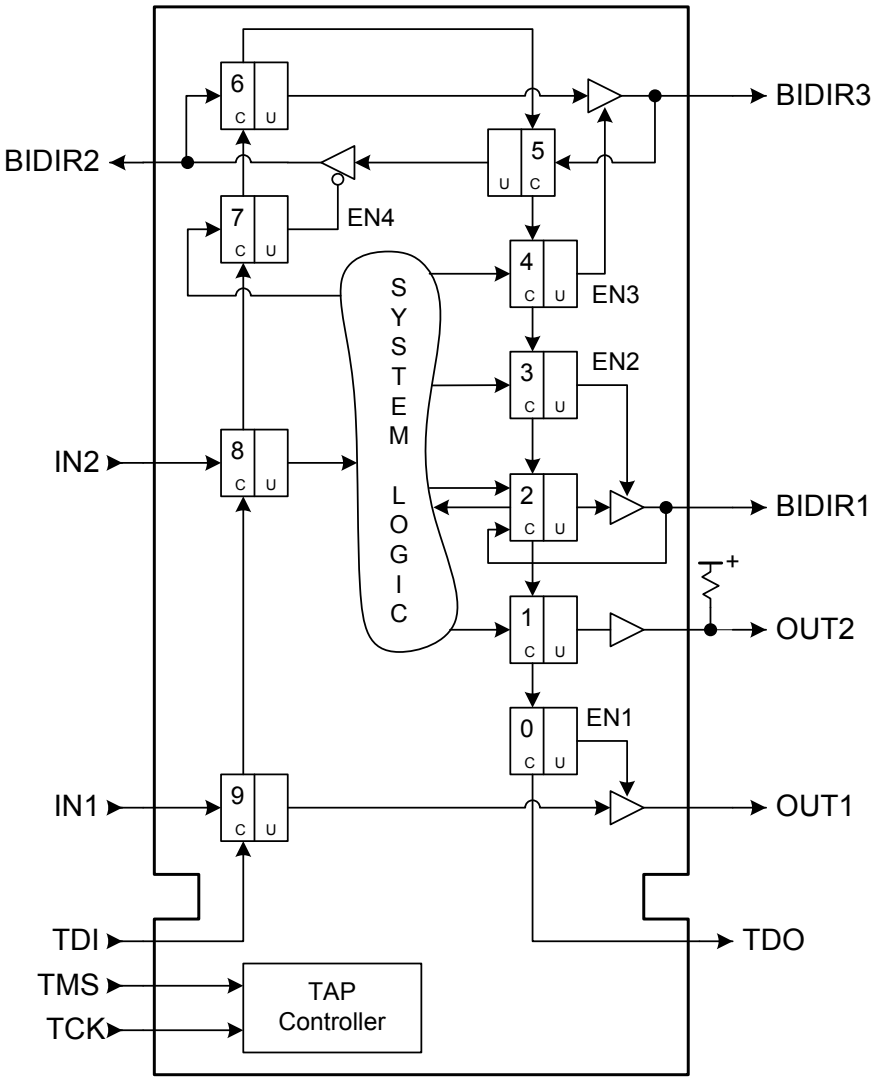


Figure B-18—Component that illustrates several merged cells

The definition of the boundary-scan register for Figure B-18 is as follows:

```
attribute BOUNDARY_LENGTH of Figure_B12: entity is 10;
attribute BOUNDARY_REGISTER of Figure_B12: entity is
--
--num cell port      function   safe[input/ccell disval rslt]
--
" 0  (BC_1, *,        control,    0),                          "&
" 1  (BC_1, OUT2,     output2,    1,  1,           1,   Weak1),"&
" 2  (BC_7, BIDIR1,   bidir,      X,  3,            0,    Z),"&
" 3  (BC_2, *,        control,    0),                          "&
" 4  (BC_1, *,        control,    0),                          "&
" 5  (BC_1, BIDIR3,   input,      X, OPEN1),                   "&
" 5  (BC_1, BIDIR2,   output3,    X,  7,            1,    Z),"&
" 6  (BC_1, BIDIR2,   input,      X, OPEN1),                   "&
" 6  (BC_1, BIDIR3,   output3,    X,  4,            0,    Z),"&
" 7  (BC_1, *,        control,    1),                          "&
" 8  (BC_1, IN2,      input,      X, OPEN0),                   "&
" 9  (BC_1, IN1,      input,      X, OPEN0),                   "&
" 9  (BC_1, OUT1,     output3,    X,  0,            0,    Z) ";
```

Cells 0, 4, and 7 are control cells located between the system logic and the enable for signals `OUT1`, `BIDIR2`, and `BIDIR3`. Notice that values are assigned to the "safe" subfields for these cells to cause the associated drivers to disable.

Cell 3 is the control for the reversible cell (see Figure 11-38c) used on the bidirectional signal `BIDIR1`. Notice its "safe" subfield is given the value that causes `BIDIR1` to be an input.

Cell 1 is a two-state output data cell. Note that it has the three extra fields, indicating that it controls its own open-collector, asymmetrical driver. By placing a 1 in cell 1, the driver at `OUT2` can be set to the inactive drive state, in which it will output the "**WEAK1**" state.

Cell 2 is the reversible cell of Figure 11-38d. This cell serves either as an input (if the control cell has turned off the output driver, meaning cell 3 produces a 0) or as the data source for the driver (if the output is enabled).

Note that the structures for `BIDIR2` and `BIDIR3` (see Figure 11-37) would allow observation of the driver, thus, allowing a simple consistency check.

Cell 5 (and similarly cells 6 and 9) has merged behavior—it serves as the input receiver for `BIDIR3` and as the data source for `BIDIR2`. As a result, the cell has two lines of description in the boundary-scan register description. The first gives its behavior as an input cell while the second describes its characteristics as an output cell. Note that cell **BC_1** used in this capacity must support both **INPUT** and **OUTPUT3** functions. This is reflected in the definition of **BC_1** where both functions exist for all instructions.

The example illustrated by Figure B-18 is extreme and includes several unusual cases. Most component implementations will be quite simple and routine, as illustrated by the example component description in B.11.1.

B.12 1990 version of BSDL

The 1990 version of BSDL is a de facto industry standard. The information presented in this clause is provided as a courtesy to tool implementers who wish to support BSDL descriptions written before the 2001 version was defined. This form of BSDL is a subset of the 2001 version except where noted below. New BSDL descriptions should not use the 1990 form.

In the 1990 version of BSDL, the following syntactic elements are not supported:

<component conformance statement>
<grouped port identification>
<compliance-enable description>
<runbist description>
<intest description>

<BSDL extensions>

In the 1990 version, **KEEPER** and **PRELOAD** were not recognized as reserved words. Also, in the 1990 version, cell types **BC_0**, **BC_7**, **BC_8**, **BC_9**, and **BC_10** need to be specified in a user-supplied BSDL package.

B.12.1 1990 Standard VHDL Package STD_1149_1_1990

The following is the complete content of Standard VHDL Package **STD_1149_1_1990**. Note that both the VHDL package and the package body are shown. This information defines the basis of BSDL and typically would be write-protected by a system administrator. An explanation of the cell definitions (e.g., **BC_1**, **BC_2**, etc.) in the package body is given in B.10.1. BSDL files that use <standard VHDL package identifier> **STD_1149_1_1990** are processed using this Standard VHDL Package.

NOTE 1—The figure references are to the 1990 edition of IEEE Std 1149.1.

NOTE 2—Where figures are cited, the suffix "c" is used to denote a control cell. The suffix "d" denotes a data cell.

```
package STD_1149_1_1990 is

-- Give pin mapping declarations

attribute PIN_MAP : string;
subtype PIN_MAP_STRING is string;

-- Give TAP control declarations

type CLOCK_LEVEL is (LOW, BOTH);
type CLOCK_INFO  is record
  FREQ : real;
  LEVEL: CLOCK_LEVEL;
end record;

attribute  TAP_SCAN_IN   : boolean;
attribute  TAP_SCAN_OUT  : boolean;
attribute  TAP_SCAN_CLOCK: CLOCK_INFO;
attribute  TAP_SCAN_MODE : boolean;
attribute  TAP_SCAN_RESET: boolean;

-- Give instruction register declarations

attribute  INSTRUCTION_LENGTH : integer;
attribute  INSTRUCTION_OPCODE : string;
attribute  INSTRUCTION_CAPTURE : string;
attribute  INSTRUCTION_DISABLE : string;
attribute  INSTRUCTION_GUARD : string;
attribute  INSTRUCTION_PRIVATE : string;
attribute  INSTRUCTION_USAGE : string;
attribute  INSTRUCTION_SEQUENCE : string;

-- Give ID and USER code declarations

type ID_BITS is ('0', '1', 'x', 'X');
type ID_STRING is array (31 downto 0) of ID_BITS;
```

```
attribute IDCODE_REGISTER  :  ID_STRING;
attribute USERCODE_REGISTER:  ID_STRING;

-- Give register declarations

attribute REGISTER_ACCESS : string;

-- Give boundary cell declarations

type BSCAN_INST is (EXTEST, SAMPLE, INTEST, RUNBIST);
type CELL_TYPE is (INPUT, INTERNAL, CLOCK,
 CONTROL, CONTROLR, OUTPUT2,
 OUTPUT3, BIDIR_IN, BIDIR_OUT);
type CAP_DATA is (PI, PO, UPD, CAP, X, ZERO, ONE);
type CELL_DATA is record
  CT : CELL_TYPE;
  I  : BSCAN_INST;
  CD : CAP_DATA;
end record;
type CELL_INFO is array (positive range <>) of CELL_DATA;

-- Boundary cell deferred constants (see package body)

constant BC_1  : CELL_INFO;
constant BC_2  : CELL_INFO;
constant BC_3  : CELL_INFO;
constant BC_4  : CELL_INFO;
constant BC_5  : CELL_INFO;
constant BC_6  : CELL_INFO;

-- Boundary-scan register declarations

attribute BOUNDARY_CELLS : string;
attribute BOUNDARY_LENGTH : integer;
attribute BOUNDARY_REGISTER : string;

-- Miscellaneous

attribute DESIGN_WARNING : string;

end STD_1149_1_1990;  -- End of IEEE Std 1149.1-1990 Package


package body STD_1149_1_1990 is   -- Standard boundary cells

-- Description for f10-12, f10-16, f10-18c, f10-18d, f10-21c

constant BC_1 : CELL_INFO :=
 ((INPUT,   EXTEST,  PI),  (OUTPUT2,  EXTEST,  PI),
  (INPUT,   SAMPLE,  PI),  (OUTPUT2,  SAMPLE,  PI),
  (INPUT,   INTEST,  PI),  (OUTPUT2,  INTEST,  PI),
  (INPUT,   RUNBIST, PI),  (OUTPUT2,  RUNBIST, PI),
  (OUTPUT3, EXTEST,  PI),  (INTERNAL, EXTEST,  PI),
  (OUTPUT3, SAMPLE,  PI),  (INTERNAL, SAMPLE,  PI),
  (OUTPUT3, INTEST,  PI),  (INTERNAL, INTEST,  PI),
  (OUTPUT3, RUNBIST, PI),  (INTERNAL, RUNBIST, PI),
  (CONTROL, EXTEST,  PI),  (CONTROLR, EXTEST,  PI),
```

```
  (CONTROL, SAMPLE,  PI),  (CONTROLR, SAMPLE,  PI),
  (CONTROL, INTEST,  PI),  (CONTROLR, INTEST,  PI),
  (CONTROL, RUNBIST, PI),  (CONTROLR, RUNBIST, PI) );

-- Description for f10-8, f10-17, f10-19c, f10-19d, f10-22c

constant BC_2 : CELL_INFO :=
 ((INPUT,   EXTEST,  PI),  (OUTPUT2, EXTEST,   UPD),
  (INPUT,   SAMPLE,  PI),  (OUTPUT2, SAMPLE,   PI),
  (INPUT,   INTEST,  UPD),  -- Intest on output2 not supported
  (INPUT,   RUNBIST, UPD), (OUTPUT2, RUNBIST, UPD),
  (OUTPUT3, EXTEST,  UPD), (INTERNAL, EXTEST,  PI),
  (OUTPUT3, SAMPLE,  PI),  (INTERNAL, SAMPLE,  PI),
  (OUTPUT3, INTEST,  PI),  (INTERNAL, INTEST,  UPD),
  (OUTPUT3, RUNBIST, PI),  (INTERNAL, RUNBIST, UPD),
  (CONTROL, EXTEST,  UPD), (CONTROLR, EXTEST, UPD),
  (CONTROL, SAMPLE,  PI),  (CONTROLR, SAMPLE,  PI),
  (CONTROL, INTEST,  PI),  (CONTROLR, INTEST,  PI),
  (CONTROL, RUNBIST, PI),  (CONTROLR, RUNBIST, PI) );

-- Description for f10-9

constant BC_3 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),    (INTERNAL, EXTEST,  PI),
  (INPUT, SAMPLE,  PI),    (INTERNAL, SAMPLE,  PI),
  (INPUT, INTEST,  PI),    (INTERNAL, INTEST,  PI),
  (INPUT, RUNBIST, PI),    (INTERNAL, RUNBIST, PI) );

-- Description for f10-10, f10-11

constant BC_4 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),  -- Intest on input not supported
  (INPUT, SAMPLE,  PI),  -- Runbist on input not supported
  (CLOCK, EXTEST,  PI),   (INTERNAL, EXTEST,  PI),
  (CLOCK, SAMPLE,  PI),   (INTERNAL, SAMPLE,  PI),
  (CLOCK, INTEST,  PI),   (INTERNAL, INTEST,  PI),
  (CLOCK, RUNBIST, PI),   (INTERNAL, RUNBIST, PI) );

-- Description for f10-20c, a combined input/control

constant BC_5 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),   (CONTROL, EXTEST,  PI),
  (INPUT, SAMPLE,  PI),   (CONTROL, SAMPLE,  PI),
  (INPUT, INTEST,  UPD),  (CONTROL, INTEST,  UPD),
  (INPUT, RUNBIST, PI),   (CONTROL, RUNBIST, PI) );

-- Description for f10-22d, a reversible cell

constant BC_6 : CELL_INFO :=
 ((BIDIR_IN, EXTEST,  PI),  (BIDIR_OUT, EXTEST,  UPD),
  (BIDIR_IN, SAMPLE,  PI),  (BIDIR_OUT, SAMPLE,  PI),
  (BIDIR_IN, INTEST,  UPD), (BIDIR_OUT, INTEST,  PI),
  (BIDIR_IN, RUNBIST, UPD), (BIDIR_OUT, RUNBIST, PI) );

end STD_1149_1_1990;  -- End of 1990 Package Body
```

B.12.2 Typical application of BSDL, 1990 version

The following example is for the Texas Instruments SN74BCT8374 Octal D Flip-Flop using version STD_1149_1_1990.

```
entity ttl74bct8374 is
  generic (PHYSICAL_PIN_MAP : string := "DW");

  port (CLK:in bit; Q:out bit_vector(1 to 8); D:in bit_vector(1 to 8);
    GND, VCC:linkage bit;

OC_NEG:in bit; TDO:out bit; TMS, TDI, TCK:in bit);

 use STD_1149_1_1990.all;--Get IEEE STD 1149.1-1990 attributes and
definitions

  attribute PIN_MAP of ttl74bct8374 : entity is PHYSICAL_PIN_MAP;

  constant DW:   PIN_MAP_STRING:="CLK:1, Q:(2,3,4,5,7,8,9,10), " &
         "D:(23,22,21,20,19,17,16,15)," &
         "GND:6, VCC:18, OC_NEG:24, TDO:11, TMS:12, TCK:13, TDI:14";

  constant FK:   PIN_MAP_STRING:="CLK:9, Q:(10,11,12,13,16,17,18,19)," &
         "D:(6,5,4,3,2,27,26,25)," &
         "GND:14, VCC:28, OC_NEG:7, TDO:20, TMS:21, TCK:23, TDI:24";

    attribute TAP_SCAN_IN    of TDI : signal is true;
    attribute TAP_SCAN_MODE  of TMS : signal is true;
    attribute TAP_SCAN_OUT   of TDO : signal is true;
    attribute TAP_SCAN_CLOCK of TCK : signal is (20.0e6, BOTH);

    attribute INSTRUCTION_LENGTH of ttl74bct8374 : entity is 8;

    attribute INSTRUCTION_OPCODE of ttl74bct8374 : entity is
       "BYPASS (11111111, 10001000, 00000101, 10000100, 00000001)," &
       "EXTEST (00000000, 10000000)," &
       "SAMPLE (00000010, 10000010)," &
       "INTEST (00000011, 10000011)," &
       "TRIBYP (00000110, 10000110)," &     -- Boundary Hi-Z
       "SETBYP (00000111, 10000111)," &     -- Boundary 1/0
       "RUNT   (00001001, 10001001)," &     -- Boundary run test
       "READBN (00001010, 10001010)," &     -- Boundary read normal
       "READBT (00001011, 10001011)," &     -- Boundary read test
       "CELLTST(00001100, 10001100)," &     -- Boundary self-test normal
       "TOPHIP (00001101, 10001101)," &     -- Boundary toggle-out test
       "SCANCN (00001110, 10001110)," &     -- BCR Scan normal
       "SCANCT (00001111, 10001111)";       -- BCR Scan test

    attribute INSTRUCTION_CAPTURE of ttl74bct8374 : entity is "10000001";
    attribute INSTRUCTION_DISABLE of ttl74bct8374 : entity is "TRIBYP";
-- Now obsolete, see note below.
    attribute INSTRUCTION_GUARD   of ttl74bct8374 : entity is "SETBYP";
-- Now obsolete, see note below.
--NOTE—The INSTRUCTION_SEQUENCE and INSTRUCTION_USAGE attributes would appear
--here, but they are now obsolete.

    attribute REGISTER_ACCESS of ttl74bct8374 : entity is
```

```
        "BOUNDARY (READBN, READBT, CELLTST)," &
        "BYPASS (TOPHIP, SETBYP, RUNT, TRIBYP)," &
        "BCR[2] (SCANCN, SCANCT)";   -- 2-bit Boundary Control Register

    attribute BOUNDARY_CELLS of ttl74bct8374 : entity is "BC_1";
-- Now obsolete, see note below.
    attribute BOUNDARY_LENGTH of ttl74bct8374 : entity is 18;

--NOTE-The "attribute INSTRUCTION_DISABLE of ttl74bct8374:entity is
"TRIBYP";"
--statement is the same as the HIGHZ instruction defined in this standard and
--the "attribute INSTRUCTION_GUARD of ttl74bct8374:entity is "SETBYP";"
--statement is the same as CLAMP instruction. The "attribute BOUNDARY_CELLS
of
--ttl74bct8374:entity is "BC_1";" statement has been removed from later
--versions of BSDL, since the cells being used can be identified while
--processing the BOUNDARY_REGISTER attribute.

    attribute BOUNDARY_REGISTER of ttl74bct8374 : entity is
      -- num cell  port      function   safe [ccell disval rslt]
        "17 (BC_1, CLK,      input,        X),  " &
        "16 (BC_1, OC_NEG,   input,        X), " &    -- Merged input/control
        "16 (BC_1, *,        control,      1), " &    -- Merged input/control
        "15 (BC_1, D(1),     input,        X),  " &
        "14 (BC_1, D(2),     input,        X),  " &
        "13 (BC_1, D(3),     input,        X),  " &
        "12 (BC_1, D(4),     input,        X),  " &
        "11 (BC_1, D(5),     input,        X),  " &
        "10 (BC_1, D(6),     input,        X),  " &
        " 9 (BC_1, D(7),     input,        X),  " &
        " 8 (BC_1, D(8),     input,        X),  " &
-- cell 16 @ 1 -> Hi-Z
        " 7 (BC_1, Q(1),     output3,      X,  16,   1,    Z)," &
        " 6 (BC_1, Q(2),     output3,      X,  16,   1,    Z)," &
        " 5 (BC_1, Q(3),     output3,      X,  16,   1,    Z)," &
        " 4 (BC_1, Q(4),     output3,      X,  16,   1,    Z)," &
        " 3 (BC_1, Q(5),     output3,      X,  16,   1,    Z)," &
        " 2 (BC_1, Q(6),     output3,      X,  16,   1,    Z)," &
        " 1 (BC_1, Q(7),     output3,      X,  16,   1,    Z)," &
        " 0 (BC_1, Q(8),     output3,      X,  16,   1,    Z)";

 end ttl74bct8374;
```

B.12.3 Obsolete syntax

Several attributes were enumerated in the 1990 version of BSDL that were all string-valued. They are obsolete in versions of BSDL after 1994. The **INSTRUCTION_GUARD** and **INSTRUCTION_DISABLE** attributes were made obsolete by the standardization of the *CLAMP* and *HIGHZ* instructions. The **BOUNDARY_CELLS** attribute was found to be unnecessary. With regard to the **INSTRUCTION_SEQUENCE** and **INSTRUCTION_USAGE** attributes, agreement was never reached on either their scope or the applications they were intended to handle. These two attributes have been dropped.

All these obsolete attributes had string values, and so a tool intended to ignore statements defining the attributes can be designed to ignore string-valued attributes with the obsolete names. Their syntax is provided in Syntax to assist users in the development of tools that can read both the version of BSDL documented in this annex and the earlier draft version of the language.

B.12.3.1 Syntax

attribute <obsoleted attribute name> **of**
<component name> <colon> **entity is** <obsoleted string> <semicolon>

<obsoleted attribute name>::= **INSTRUCTION_GUARD** | **INSTRUCTION_DISABLE** |
INSTRUCTION_SEQUENCE | **INSTRUCTION_USAGE** | **BOUNDARY_CELLS**
<obsoleted string>::= <string>

B.12.4 Miscellaneous points on 1990 version

In parsing a 1990 version of BSDL, a violation of the condition described in semantic check p) of B.8.14.1 should result in the issuance of a warning message by parsing software. This acknowledges that the 1990 edition of this standard allowed a control cell to fan out to more than one driver, and each driver could be enabled with an independent choice of control value. Semantic check p) of B.8.14.1 reflects the strengthening of this standard to require specifically that all drivers controlled by a single control cell are disabled by the same value.

The VHDL package body shown in B.12.1 has a fourth value of <capture instruction>, with a value of **RUNBIST** shown in the <capture descriptor> elements (see B.10.1). This value has been removed in versions of BSDL after 1994, since these versions now provides for RUNBIST description (see B.8.15).

In the 1990 version of BSDL, the device ID register was named **IDCODE** in the **REGISTER_ACCESS** attribute. In this annex, this name has been changed to **DEVICE_ID** to match the definition used elsewhere in this standard.

In the 1990 version of BSDL, the following values of given BSDL syntactical elements did not exist:

— <cell context> value **OBSERVE_ONLY**
— <disable result> value **KEEPER**
— <function> value **OBSERVE_ONLY**
— <instruction name> value **PRELOAD**

B.13 1994 version of BSDL

The 1994 version of BSDL was defined by IEEE Std 1149.1b-1994. The information presented in this clause is provided to help tool implementers support BSDL descriptions written before the 2001 version was defined. This form of BSDL is a subset of the later versions except where noted below. New BSDL descriptions should not use the 1994 form.

In the 1994 version, **KEEPER** and **PRELOAD** were not recognized as reserved words. Also, in the 1994 version, cell types **BC_8**, **BC_9**, and **BC_10** need to be specified in a user-supplied BSDL package.

B.13.1 Standard VHDL Package STD_1149_1_1994

The following is the complete content of Standard VHDL Package **STD_1149_1_1994**. Note that both the VHDL package and the package body are shown. This information defines the basis of BSDL and typically would be write-protected by a system administrator. An explanation of the cell definitions (e.g., **BC_1**, **BC_2**, etc.) in the package body is given in B.10.1. BSDL descriptions that use <standard VHDL package identifier> **STD_1149_1_1994** are processed using this Standard VHDL Package.

NOTE—The figure references are to the 1993 edition of IEEE Std 1149.1.

```
package STD_1149_1_1994 is

-- Give component conformance declaration
```

```
attribute COMPONENT_CONFORMANCE : string;

-- Give pin mapping declarations

attribute PIN_MAP : string;
subtype PIN_MAP_STRING is string;

-- Give TAP control declarations

type CLOCK_LEVEL is (LOW, BOTH);
type CLOCK_INFO  is record
  FREQ : real;
  LEVEL: CLOCK_LEVEL;
end record;

attribute  TAP_SCAN_IN   : boolean;
attribute  TAP_SCAN_OUT  : boolean;
attribute  TAP_SCAN_CLOCK: CLOCK_INFO;
attribute  TAP_SCAN_MODE : boolean;
attribute  TAP_SCAN_RESET: boolean;

-- Give instruction register declarations

attribute  INSTRUCTION_LENGTH : integer;
attribute  INSTRUCTION_OPCODE : string;
attribute  INSTRUCTION_CAPTURE : string;
attribute  INSTRUCTION_PRIVATE : string;

-- Give ID and USER code declarations

type ID_BITS is ('0', '1', 'x', 'X');
type ID_STRING is array (31 downto 0) of ID_BITS;
attribute IDCODE_REGISTER  :  ID_STRING;
attribute USERCODE_REGISTER:  ID_STRING;

-- Give register declarations

attribute REGISTER_ACCESS : string;

-- Give boundary cell declarations

type BSCAN_INST is (EXTEST, SAMPLE, INTEST);
type CELL_TYPE is (INPUT, INTERNAL, CLOCK, OBSERVE_ONLY,
 CONTROL, CONTROLR, OUTPUT2,
 OUTPUT3, BIDIR_IN, BIDIR_OUT);
type CAP_DATA is (PI, PO, UPD, CAP, X, ZERO, ONE);
type CELL_DATA is record
  CT : CELL_TYPE;
  I  : BSCAN_INST;
  CD : CAP_DATA;
end record;
type CELL_INFO is array (positive range <>) of CELL_DATA;

-- Boundary cell deferred constants (see package body)

constant BC_0  : CELL_INFO;
```

```
constant BC_1  : CELL_INFO;
constant BC_2  : CELL_INFO;
constant BC_3  : CELL_INFO;
constant BC_4  : CELL_INFO;
constant BC_5  : CELL_INFO;
constant BC_6  : CELL_INFO;
constant BC_7  : CELL_INFO;

-- Boundary-scan register declarations

attribute BOUNDARY_LENGTH : integer;
attribute BOUNDARY_REGISTER : string;

-- Miscellaneous

attribute PORT_GROUPING : string;
attribute RUNBIST_EXECUTION : string;
attribute INTEST_EXECUTION : string;
subtype BSDL_EXTENSION is string;
attribute COMPLIANCE_PATTERNS : string;
attribute DESIGN_WARNING : string;

end STD_1149_1_1994;  -- End of 1149.1-1994 Package


package body STD_1149_1_1994 is   -- Standard boundary cells

-- Generic cell capturing minimum allowed data

constant BC_0 : CELL_INFO :=
 ((INPUT,   EXTEST,  PI),      (OUTPUT2,  EXTEST,  X),
  (INPUT,   SAMPLE,  PI),      (OUTPUT2,  SAMPLE,  PI),
  (INPUT,   INTEST,  X),       (OUTPUT2,  INTEST,  PI),
  (OUTPUT3, EXTEST,  X),       (INTERNAL, EXTEST,  X),
  (OUTPUT3, SAMPLE,  PI),      (INTERNAL, SAMPLE,  X),
  (OUTPUT3, INTEST,  PI),      (INTERNAL, INTEST,  X),
  (CONTROL, EXTEST,  X),       (CONTROLR, EXTEST,  X),
  (CONTROL, SAMPLE,  PI),      (CONTROLR, SAMPLE,  PI),
  (CONTROL, INTEST,  PI),      (CONTROLR, INTEST,  PI),
  (BIDIR_IN,EXTEST,  PI),      (BIDIR_OUT, EXTEST, X ),
  (BIDIR_IN,SAMPLE,  PI),      (BIDIR_OUT, SAMPLE, PI),
  (BIDIR_IN,INTEST,  X ),      (BIDIR_OUT, INTEST, PI),
  (OBSERVE_ONLY, SAMPLE, PI),  (OBSERVE_ONLY, EXTEST, PI) );

-- Description for f10-18, f10-29, f10-31c, f10-31d, f10-33c, f10-41d

constant BC_1 : CELL_INFO :=
 ((INPUT,   EXTEST,  PI),  (OUTPUT2,  EXTEST,  PI),
  (INPUT,   SAMPLE,  PI),  (OUTPUT2,  SAMPLE,  PI),
  (INPUT,   INTEST,  PI),  (OUTPUT2,  INTEST,  PI),
  (OUTPUT3, EXTEST,  PI),  (INTERNAL, EXTEST,  PI),
  (OUTPUT3, SAMPLE,  PI),  (INTERNAL, SAMPLE,  PI),
  (OUTPUT3, INTEST,  PI),  (INTERNAL, INTEST,  PI),
  (CONTROL, EXTEST,  PI),  (CONTROLR, EXTEST,  PI),
  (CONTROL, SAMPLE,  PI),  (CONTROLR, SAMPLE,  PI),
  (CONTROL, INTEST,  PI),  (CONTROLR, INTEST,  PI) );
```

```
-- Description for f10-14, f10-30, f10-32c, f10-32d, f10-35c

constant BC_2 : CELL_INFO :=
 ((INPUT,   EXTEST,  PI),  (OUTPUT2, EXTEST,   UPD),
  (INPUT,   SAMPLE,  PI),  (OUTPUT2, SAMPLE,   PI),
  (INPUT,   INTEST,  UPD),  -- Intest on output2 not supported
  (OUTPUT3, EXTEST,  UPD), (INTERNAL, EXTEST,  PI),
  (OUTPUT3, SAMPLE,  PI),  (INTERNAL, SAMPLE,  PI),
  (OUTPUT3, INTEST,  PI),  (INTERNAL, INTEST,  UPD),
  (CONTROL, EXTEST,  UPD), (CONTROLR, EXTEST, UPD),
  (CONTROL, SAMPLE,  PI),  (CONTROLR, SAMPLE,  PI),
  (CONTROL, INTEST,  PI),  (CONTROLR, INTEST,  PI) );

-- Description for f10-15

constant BC_3 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),    (INTERNAL, EXTEST,  PI),
  (INPUT, SAMPLE,  PI),    (INTERNAL, SAMPLE,  PI),
  (INPUT, INTEST,  PI),    (INTERNAL, INTEST,  PI) );

-- Description for f10-16, f10-17

constant BC_4 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),        -- Intest on input not supported
  (INPUT, SAMPLE,  PI),
  (OBSERVE_ONLY, EXTEST, PI),
  (OBSERVE_ONLY, SAMPLE, PI),  -- Intest on observe_only not supported
  (CLOCK, EXTEST,  PI),   (INTERNAL, EXTEST,  PI),
  (CLOCK, SAMPLE,  PI),   (INTERNAL, SAMPLE,  PI),
  (CLOCK, INTEST,  PI),   (INTERNAL, INTEST,  PI) );

-- Description for f10-41c, a combined input/control

constant BC_5 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),   (CONTROL, EXTEST,  PI),
  (INPUT, SAMPLE,  PI),   (CONTROL, SAMPLE,  PI),
  (INPUT, INTEST,  UPD),  (CONTROL, INTEST,  UPD) );

-- Description for f10-35d, a reversible cell
-- !! Not recommended; replaced by BC_7 below !!

constant BC_6 : CELL_INFO :=
 ((BIDIR_IN, EXTEST,  PI),  (BIDIR_OUT, EXTEST,  UPD),
  (BIDIR_IN, SAMPLE,  PI),  (BIDIR_OUT, SAMPLE,  PI),
  (BIDIR_IN, INTEST,  UPD), (BIDIR_OUT, INTEST,  PI) );

-- Description for f10-34d, self-monitor reversible
-- !! Recommended over cell BC_6 !!

constant BC_7 : CELL_INFO :=
 ((BIDIR_IN, EXTEST,  PI),  (BIDIR_OUT, EXTEST,  PO),
  (BIDIR_IN, SAMPLE,  PI),  (BIDIR_OUT, SAMPLE,  PI),
  (BIDIR_IN, INTEST,  UPD), (BIDIR_OUT, INTEST,  PI) )
```

B.14 2001 version of BSDL

B.14.1 Standard VHDL Package STD_1149_1_2001

```
package STD_1149_1_2001 is

-- Give component conformance declaration

attribute COMPONENT_CONFORMANCE : string;

-- Give pin mapping declarations

attribute PIN_MAP : string;
subtype PIN_MAP_STRING is string;

-- Give TAP control declarations

type CLOCK_LEVEL is (LOW, BOTH);
type CLOCK_INFO  is record
  FREQ : real;
  LEVEL: CLOCK_LEVEL;
end record;

attribute  TAP_SCAN_IN   : boolean;
attribute  TAP_SCAN_OUT  : boolean;
attribute  TAP_SCAN_CLOCK: CLOCK_INFO;
attribute  TAP_SCAN_MODE : boolean;
attribute  TAP_SCAN_RESET: boolean;

-- Give instruction register declarations

attribute  INSTRUCTION_LENGTH : integer;
attribute  INSTRUCTION_OPCODE : string;
attribute  INSTRUCTION_CAPTURE : string;
attribute  INSTRUCTION_PRIVATE : string;

-- Give ID and USER code declarations

type ID_BITS is ('0', '1', 'x', 'X');
type ID_STRING is array (31 downto 0) of ID_BITS;
attribute IDCODE_REGISTER  :  ID_STRING;
attribute USERCODE_REGISTER:  ID_STRING;

-- Give register declarations

attribute REGISTER_ACCESS : string;

-- Give boundary cell declarations

type BSCAN_INST is (EXTEST, SAMPLE, INTEST);
type CELL_TYPE is (INPUT, INTERNAL, CLOCK, OBSERVE_ONLY,
 CONTROL, CONTROLR, OUTPUT2,
 OUTPUT3, BIDIR_IN, BIDIR_OUT);
type CAP_DATA is (PI, PO, UPD, CAP, X, ZERO, ONE);
type CELL_DATA is record
  CT : CELL_TYPE;
```

```
  I  : BSCAN_INST;
  CD : CAP_DATA;
end record;
type CELL_INFO is array (positive range <>) of CELL_DATA;

-- Boundary cell deferred constants (see package body)

constant BC_0  : CELL_INFO;
constant BC_1  : CELL_INFO;
constant BC_2  : CELL_INFO;
constant BC_3  : CELL_INFO;
constant BC_4  : CELL_INFO;
constant BC_5  : CELL_INFO;
constant BC_6  : CELL_INFO;
constant BC_7  : CELL_INFO;
constant BC_8  : CELL_INFO;
constant BC_9  : CELL_INFO;
constant BC_10  : CELL_INFO;

-- Boundary-scan register declarations

attribute BOUNDARY_LENGTH : integer;
attribute BOUNDARY_REGISTER : string;

-- Miscellaneous

attribute PORT_GROUPING : string;
attribute RUNBIST_EXECUTION : string;
attribute INTEST_EXECUTION : string;
subtype BSDL_EXTENSION is string;
attribute COMPLIANCE_PATTERNS : string;
attribute DESIGN_WARNING : string;

end STD_1149_1_2001;  -- End of 1149.1-2001 Package


package body STD_1149_1_2001 is   -- Standard boundary cells

-- Generic cell capturing minimum allowed data

constant BC_0 : CELL_INFO :=
 ((INPUT,   EXTEST,  PI),      (OUTPUT2,  EXTEST,  X),
  (INPUT,   SAMPLE,  PI),      (OUTPUT2,  SAMPLE,  PI),
  (INPUT,   INTEST,  X),       (OUTPUT2,  INTEST,  PI),
  (OUTPUT3, EXTEST,  X),       (INTERNAL, EXTEST,  X),
  (OUTPUT3, SAMPLE,  PI),      (INTERNAL, SAMPLE,  X),
  (OUTPUT3, INTEST,  PI),      (INTERNAL, INTEST,  X),
  (CONTROL, EXTEST,  X),       (CONTROLR, EXTEST,  X),
  (CONTROL, SAMPLE,  PI),      (CONTROLR, SAMPLE,  PI),
  (CONTROL, INTEST,  PI),      (CONTROLR, INTEST,  PI),
  (BIDIR_IN,EXTEST,  PI),      (BIDIR_OUT, EXTEST, X ),
  (BIDIR_IN,SAMPLE,  PI),      (BIDIR_OUT, SAMPLE, PI),
  (BIDIR_IN,INTEST,  X ),      (BIDIR_OUT, INTEST, PI),
  (OBSERVE_ONLY, SAMPLE, PI),  (OBSERVE_ONLY, EXTEST, PI) );

-- Description for f11-18, f11-30, f11-34c, f11-34d, f11-36c, f11-46d
```

```
constant BC_1 : CELL_INFO :=
 ((INPUT,   EXTEST,  PI),  (OUTPUT2,  EXTEST,  PI),
  (INPUT,   SAMPLE,  PI),  (OUTPUT2,  SAMPLE,  PI),
  (INPUT,   INTEST,  PI),  (OUTPUT2,  INTEST,  PI),
  (OUTPUT3, EXTEST,  PI),  (INTERNAL, EXTEST,  PI),
  (OUTPUT3, SAMPLE,  PI),  (INTERNAL, SAMPLE,  PI),
  (OUTPUT3, INTEST,  PI),  (INTERNAL, INTEST,  PI),
  (CONTROL, EXTEST,  PI),  (CONTROLR, EXTEST,  PI),
  (CONTROL, SAMPLE,  PI),  (CONTROLR, SAMPLE,  PI),
  (CONTROL, INTEST,  PI),  (CONTROLR, INTEST,  PI) );

-- Description for f11-14, f11-31, f11-35c, f11-35d, f11-37c,
-- f11-38c, f11-39(output) and f11-41c

constant BC_2 : CELL_INFO :=
 ((INPUT,   EXTEST,  PI),  (OUTPUT2, EXTEST,   UPD),
  (INPUT,   SAMPLE,  PI),  (OUTPUT2, SAMPLE,   PI),
  (INPUT,   INTEST,  UPD),  -- Intest on output2 not supported
  (OUTPUT3, EXTEST,  UPD), (INTERNAL, EXTEST,  PI),
  (OUTPUT3, SAMPLE,  PI),  (INTERNAL, SAMPLE,  PI),
  (OUTPUT3, INTEST,  PI),  (INTERNAL, INTEST,  UPD),
  (CONTROL, EXTEST,  UPD), (CONTROLR, EXTEST, UPD),
  (CONTROL, SAMPLE,  PI),  (CONTROLR, SAMPLE,  PI),
  (CONTROL, INTEST,  PI),  (CONTROLR, INTEST,  PI) );

-- Description for f11-15

constant BC_3 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),    (INTERNAL, EXTEST,  PI),
  (INPUT, SAMPLE,  PI),    (INTERNAL, SAMPLE,  PI),
  (INPUT, INTEST,  PI),    (INTERNAL, INTEST,  PI) );

-- Description for f11-16, f11-17, f11-39(input)

constant BC_4 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),        -- Intest on input not supported
  (INPUT, SAMPLE,  PI),
  (OBSERVE_ONLY, EXTEST, PI),
  (OBSERVE_ONLY, SAMPLE, PI),  -- Intest on observe_only not supported
  (CLOCK, EXTEST,  PI),   (INTERNAL, EXTEST,  PI),
  (CLOCK, SAMPLE,  PI),   (INTERNAL, SAMPLE,  PI),
  (CLOCK, INTEST,  PI),   (INTERNAL, INTEST,  PI) );

-- Description for f11-46c, a combined input/control

constant BC_5 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),   (CONTROL, EXTEST,  PI),
  (INPUT, SAMPLE,  PI),   (CONTROL, SAMPLE,  PI),
  (INPUT, INTEST,  UPD),  (CONTROL, INTEST,  UPD) );

-- Description for f11-38d, a reversible cell
-- !! Not recommended; replaced by BC_7 below !!

constant BC_6 : CELL_INFO :=
 ((BIDIR_IN, EXTEST,  PI),  (BIDIR_OUT, EXTEST,  UPD),
  (BIDIR_IN, SAMPLE,  PI),  (BIDIR_OUT, SAMPLE,  PI),
  (BIDIR_IN, INTEST,  UPD), (BIDIR_OUT, INTEST,  PI) );
```

```
-- Description for f11-37d, self monitor reversible
-- !! Recommended over cell BC_6 !!

constant BC_7 : CELL_INFO :=
 ((BIDIR_IN, EXTEST,  PI),  (BIDIR_OUT, EXTEST,  PO),
  (BIDIR_IN, SAMPLE,  PI),  (BIDIR_OUT, SAMPLE,  PI),
  (BIDIR_IN, INTEST,  UPD), (BIDIR_OUT, INTEST,  PI) );

-- Description for f11-40, f11-41d

constant BC_8 : CELL_INFO :=
 -- Intest on bidir not supported
 ((BIDIR_IN, EXTEST,  PI),  (BIDIR_OUT, EXTEST,  PO),
  (BIDIR_IN, SAMPLE,  PI),  (BIDIR_OUT, SAMPLE,  PO) );

-- Description for f11-32

constant BC_9 : CELL_INFO :=
 -- Self-monitoring output that supports Intest
 ((OUTPUT2, EXTEST,  PO),  (OUTPUT3, EXTEST,  PO),
  (OUTPUT2, SAMPLE,  PI),  (OUTPUT3, SAMPLE,  PI),
  (OUTPUT2, INTEST,  PI),  (OUTPUT3, INTEST,  PI) );

-- Description for f11-33

constant BC_10 : CELL_INFO :=
 -- Self-monitoring output that does not support Intest
 ((OUTPUT2, EXTEST,  PO),  (OUTPUT3, EXTEST,  PO),
  (OUTPUT2, SAMPLE,  PO),  (OUTPUT3, SAMPLE,  PO) );

end STD_1149_1_2001;  -- End of IEEE STD 1149.1-2001 Package Body
```

Annex C

(normative)

Procedural Description Language (PDL)

C.1 General information

This clause is informative and intended to provide basic background.

The *INIT_SETUP* and *INIT_SETUP_CLAMP* instructions require a file type other than BSDL because the init_data register must be loaded with different values from use to use of a component. Rather than require just a data file, this standard uses a very simple procedural file called level-0 Procedural Description Language (PDL). Using a procedural file also allows the component provider to specify reset sequences for the system logic using the *IC_RESET* instruction and reset_select register, procedures for recovery of an ECID value, procedures for controlling power domains in test, and procedures for documenting the use of design-specific test data registers.

There is a common problem in the board and system test industry where most internal test features on the components are hidden behind private instructions, and are therefore not available to higher level packaging tests. This prevents reuse of these tests, and the reuse of such test features after the component leaves the integrated circuit (IC) foundry was not directly supported in the earlier IEEE 1149.1 environments. While this standard remains a component-level standard, one of the goals of the 2013 version of IEEE Std 1149.1 is to make it reasonable for intellectual property (IP) and integrated circuit (IC) providers to document some of these internal tests so they may be used at higher levels of packaging.

To accomplish this goal, the IP or IC provider can use the new BSDL register attributes (see B.8.19 through B.8.23) in either a BSDL for an IC or a BSDL Package for an IP, to describe the TDR controlling the test. Optionally, mnemonics for the various test controls and options (see B.8.18) can be defined to help ensure correct data are written to and read from the register or register fields. Then the test sequence to run the test can be provided in a PDL. PDL supports writing and reading registers or register fields without having to know the details of instruction opcodes or the exact structure and length of the test data register of which the register field is a part.

PDL is defined in this standard to allow IC and IP providers to document vendor-independent test data register access procedures that are callable from other test languages. It is not intended to be a full board or IC test language.

C.1.1 Purpose

The purpose of PDL defined in this standard is to load and unload registers and register fields in a manner that is independent of package level. It is a completely TDR-centric language. When using the version of PDL defined here, PDL is dependent on the component BSDL (including used Packages) for the names of the registers or register fields, as well as other information such as instruction opcodes required to access those registers. PDL written for a test feature in reusable IP, or in a component, may be automatically reused at higher levels of packaging without modification.

PDL is not meant to be a general-purpose IEEE 1149.1 board test language. It is a language for documenting the operation of TDRs in a standard way, with an eye on more uniformly defined DFT, which is done through writes and reads of TDRs designed and described in a standard way.

The register descriptions in BSDL and Packages, in combination with PDL procedures, provides a mechanism for capturing and documenting both the structure of internal test features and the procedures to use them, in a reusable and verifiable form.

C.1.2 Dependence on Tool Command Language (Tcl)

PDL follows the conventions of Tcl, and PDL commands are intended to be extensions of Tcl. Level-0 PDL does not allow embedded Tcl commands, while Level-1 does. In both cases, Tcl conventions for procedure syntax, name substitution, and decimal and real numbers are used, for example.

C.1.3 Dependence on Boundary Scan Description Language (BSDL)

PDL references registers, register fields, mnemonics, and instructions defined in BSDL and BSDL Packages. For these tokens, they will follow the definitions in Annex B.

C.2 PDL concepts and use model

This clause is informative in nature to provide a context for the formal definition of commands in the next clause. It is intended to illustrate PDL use and is not intended to dictate or restrict PDL use.

C.2.1 Use model introduction

Figure C-1 shows an example board. Despite the fact that this is a component standard, the example starts with a board, both because of the need to illustrate the use of PDL in support of multiple uses of the same component requiring different initialization and because some of the reuse concepts of PDL are easiest to understand in a board context.

The example board uses four IC: U1 through U4. U3 and U4 are two uses, or instances, of the same component type (CHIP_C). Both U1 and U2 (uses, or instances, of CHIP_A and CHIP_B, respectively) use the same memory with BIST (MEMB) from an IP supplier. U3 and U4 (CHIP_C) have a SerDes macro (SERDES) from another IP provider that supports multiple protocols and frequencies. There are three levels of object hierarchy: board, component (or IC), and IP.

The SERDES and MEMB IP have BSDL Packages defining the test data register segments in the IP, and reusable PDL documenting use of their test features (memory BIST and SerDes BIST), which can be used at the IC and board levels. The PDL for the SERDES also supports initialization with an init_data register segment. This IP Package and PDL are reusable and not specific to any single component. For instance, both CHIP_A (U1) and CHIP_B (U2) contain the internal IP MEMB and use the same Package and PDL for that IP.

The three component types (CHIP_A, CHIP_B, and CHIP_C) on the board each have BSDL, which “use” the appropriate IP Packages, and fully define both the test data registers and the instructions that select them. Components that support the initialization instructions and the init_data register may provide template PDL procedures for modification by the test engineer for each use of those components, or the board test engineer may write initialization PDL for each use manually or with the help of a tool that can read and display the options coded in the BSDL and BSDL Packages. For the sake of illustration, this example will show board-level procedures in PDL, although they could be written in whatever language is appropriate for the test environment. In addition to the procedures shown, component-level PDL may contain procedures for initialization, functional reset (using the reset_select register), ECID recovery, as well as design-specific reusable component test features.

At board test, initialization of U1, U3, and U4 is needed to program the I/O on the components, and perhaps other characteristics as well. Initialization is unique for each of these instances, and may be performed in a single board procedure or in separate PDL procedures for each instance that is called from the board-level procedure. U2 is a simple component that does not require initialization, and does not support the *IC_RESET* instruction or other component-level tests that are meant to be reused in board test. The memory BIST in U2 is supported by the Package and PDL from the IP supplier, but there are no additional IC level PDL files.

Since the PDL test procedures are defined in terms of the named registers and not bit positions within the overall scan chain, they become reusable throughout the hierarchy.

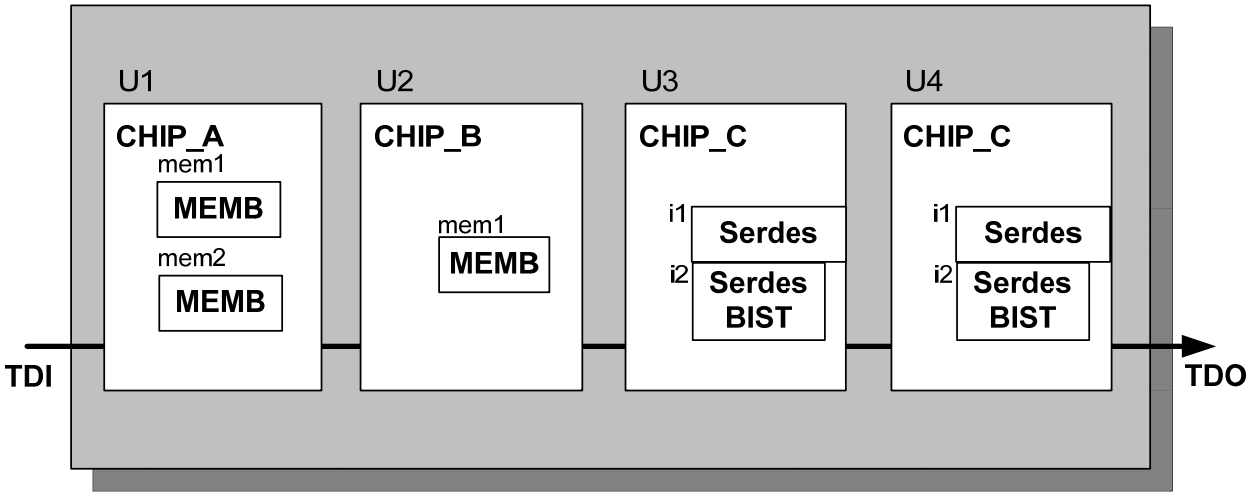


Figure C-1—PDL example board

Figure C-2 shows some simple segments of PDL procedures for the U3 and U4 uses shown in Figure C-1. Note that the SerDes IP on CHIP_C is programmable, and on this board the two uses of CHIP_C must be initialized for two different communication protocols: U3 for XAUI and U4 for SRIO. There needs to be separate "init_setup" procedures for U3 and U4 that are created by the test engineering team for the board. In addition, there is an "init_setup" procedure for CHIP_C that takes care of preparing the component for test (shutting down PLLs, etc.). Each of these three "init_setup" procedures is associated with the specific object (type or instance) and can be called separately.

Figure C-2 also shows how that PDL is then called from a board-level PDL procedure, and further how the board-level PDL could be optimized to load both instances with a single scan. Detailed examples illustrating the pertinent elements of all these procedures are provided in C.5. Note in the board-level PDL, the iCall commands. Use of the -direct parameter allows calling a procedure associated with this one instance. Without that parameter, a procedure associated with the object, of which this is an instance, will be called. This is how different initialization values may be supplied to multiple instances of one object (CHIP_C instances U3 or U4, in this case). See C.3.8.1 for more details.

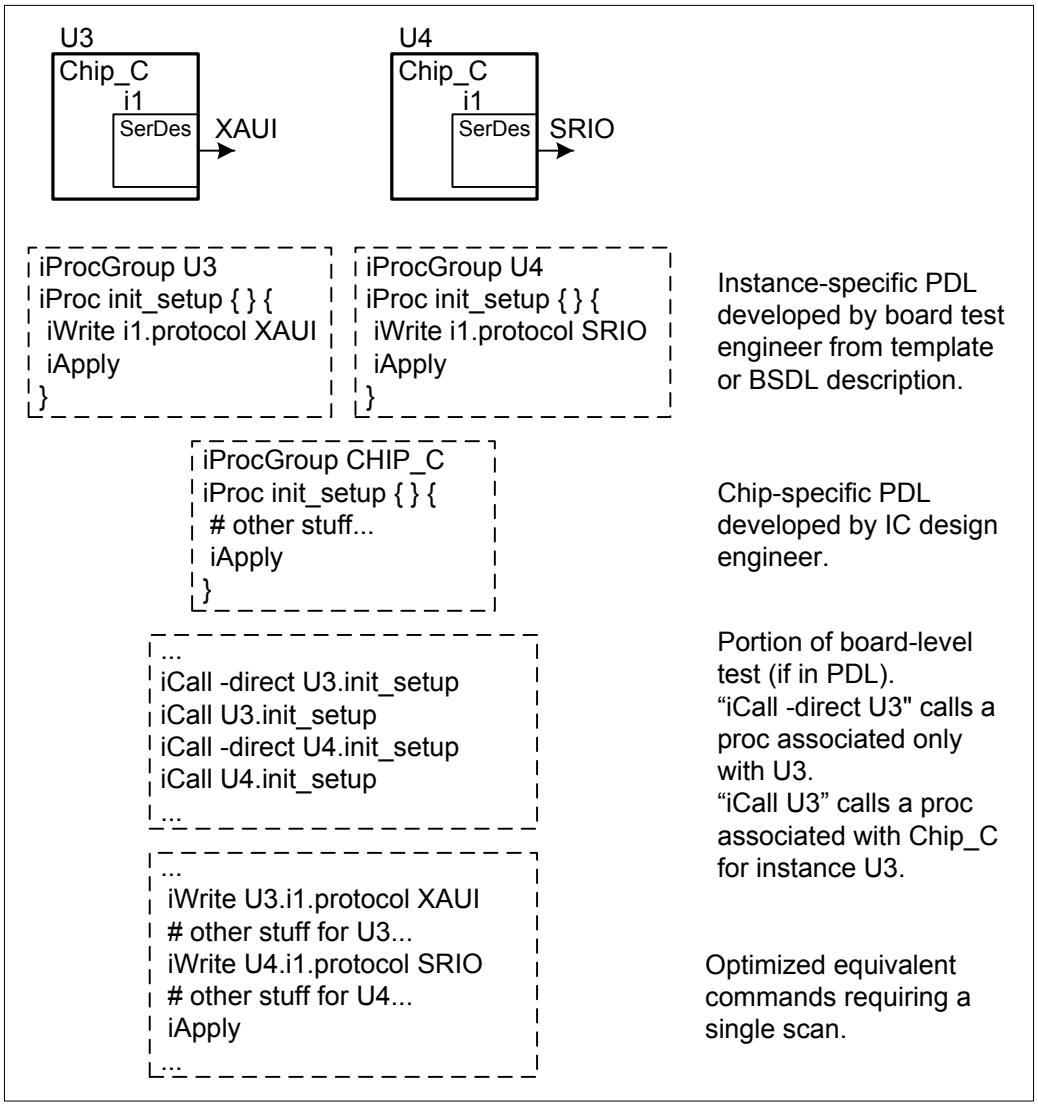


Figure C-2—PDL example detail

C.2.2 PDL levels

There are two defined levels of PDL: Level-0 and Level-1. Level-0 PDL is designed to provide an advantage when used for production test procedures. The Level-0 PDL operations are designed to run faster, and the commands translate more easily to tester hardware capabilities than Level-1 PDL. Diagnostics, reading registers, and conditional operations, however, are often better suited for Level-1 PDL. Both follow the conventions of the Tool Command Language (Tcl), with multiple restrictions in Level-0 and with fully intermixed Tcl in Level-1.

C.2.2.1 Level-0 PDL

Level-0 PDL is intended to support “load-and-go” automatic test equipment. It consists of procedures that operate on test data registers and contains commands for writing to and comparing expected values from registers, but it does not return the data captured in the registers. It provides very limited flow control. PDL commands cause transactions with the test data registers and communicate the details of how to apply useful sequences of operations. A linear listing of the PDL commands applied to a test data register over time will thus provide a complete transcript of that test data register activity.

C.2.2.2 Level-1 PDL

Level-1 PDL is intended to support diagnostic, debug, and test procedures where interactive operation is needed. Therefore, Level-1 PDL includes all Level-0 PDL commands plus additional commands suitable for returning values read from registers into variables, plus the full capabilities of the Tcl for manipulating such information and making sophisticated decisions. Level-1 PDL defines all PDL commands as extensions to Tcl, enabling flow control, variables, and data structures as defined in Tcl.

C.2.3 PDL procedures

To create a callable test procedure in PDL, the PDL commands used to interact with a register are wrapped in a simple procedural syntax (similar to the format used by Tcl):

```
iProc <proc_name> <proc_options> { <arguments> } { <PDL commands> }
```

At its simplest, the command **iProc** is followed by a procedure name, which only needs to be unique for the object it is associated with. Next come procedure options, if any, which describe the procedure, followed by a mandatory pair of curly braces, which may optionally contain a space separated list of arguments to pass values into the procedure. Finally, another mandatory pair of curly braces contains the body of the procedure, which consists of PDL commands. Commands are separated by a semicolon or an end-of-line, whichever occurs first. If included, any options document some of the intent of this procedure. If included, the optional arguments are used to substitute data (either names or values) into the body of the procedure.

For the purposes of organization, compactness, and clarity, PDL procedures may be organized hierarchically; i.e., a PDL procedure may be called by another PDL procedure using the **iCall** command. Since there are no return values or variable manipulation in Level-0 PDL, recursion is not possible. In the following discussion, position within the PDL instance hierarchy is defined as being rooted at the *top* and grows in a *downward* direction. Therefore, the phrase *A lower than B* implies B contains A. Similarly, the phrase *A higher than B* implies that A contains B and that A is closer to the root.

Each PDL procedure, when it is defined, is associated with the specific object (a BSDL entity or package name, or a specific instance of an object), which is specified by the preceding **iProcGroup** command, and are associated with a specific instance of that object when called by the **iCall** command.

Several predefined procedure names exist that have special meaning and purpose. To minimize confusion with similar instruction and register names, these procedure names will always be shown quoted in the text. The procedures named “init_setup” and “init_run” are used to describe the design-specific I/O interconnection test initialization procedures via the *INIT_SETUP, INIT_SETUP_CLAMP,* and *INIT_RUN* instructions. The identification of these procedures is important as these procedures will be automatically used to configure the I/O prior to going into *EXTEST*.

The procedure named “ecid” describes how to access an on-chip electronic identification value.

The predefined procedure named “main” can be used at any level of the hierarchy to indicate a procedure that the IP or IC provider recommends for design-specific testing. Thus, a board or system test tool simply needs to determine the objects present in the board or system from the BSDL and the used BSDL Packages, read the PDL files associated with the objects, and then execute the **iProc** “main” associated with each object to automatically get a basic level of test for each object. The procedure “main” would normally be called after the completion of init_setup, init_run, and *EXTEST*, as appropriate.

In IC testing, IP component PDL could also be used to test the IP components in the IC.

```
iProc  init_setup  { }  {
      <commands>
}
```

```
iProc  init_run  { }  {
      <commands>
}

iProc  ecid  { }  {
      <commands>
}

iProc  main  { }  {
      <commands>
}
```

C.2.4 Read and write with capture-shift-update sequence

In an IEEE 1149.1 compliant component, every scan of a TDR is a simultaneous read-and-write operation. The *Capture-DR* operation loads the register with the ***read*** data, the *Shift-DR* operation makes the read data observable while shifting in ***write*** data, and the *Update-DR* operation completes the write to any update latches in the register. In PDL, the **iRead** (or **iScan**) command, if needed, specifies the data expected to be captured and shifted out so it can be compared to the actual read data from the scan. The **iWrite** (or **iScan**) command, if needed, specifies new data to be shifted into the register. The shift-out of the captured data and shift-in of the new data are overlapped simultaneously during the *Shift-DR* state of the TAP controller state machine.

Multiple **iRead** and **iWrite** (or **iScan**) commands accumulate write and expect data for a single TDR, but those data are not used until an **iApply** command actually performs a scan. For the register that has had one or more **iRead** or **iWrite** (or **iScan**) commands specified since the last **iApply**, the **iApply** command will first load a new instruction if the current instruction does not select the specified register and will take action to include any excluded register segments that are referenced by the **iRead**, **iWrite**, or **iScan** commands, and then shift the data register. The instruction required for each TDR, the position of the named register or register field within the TDR, and any excludable segments are all defined in the BSDL for the component.

C.2.5 Register state definition

The current content of the IR and various TDRs within the unit under test are not directly visible from outside the component. Since PDL does not require that every bit of a TDR be specified prior to every register scan, and even encourages setting values only for select fields within the TDR, PDL assumes that the current state, both write data and expect data, and for Level-1 PDL, captured data and bit-by-bit fail data, of all scannable registers in the component is maintained internally. **iRead**, **iWrite**, and **iScan** commands then modify this state and the **iApply** command applies this modified state to the physical registers.

The accumulation and maintenance of write and expect data could be implemented in many ways, but for illustration, Figure C-3 assumes it is a static record structure called a scan frame. Within the scan frame, Level-0 PDL maintains two values for each IR and publically accessible TDR: a write (stimulus) value to be written to TDI when the next scan of that register takes place; and an expected (read) value to be compared to the value read from TDO during the next scan. Each of these values is the same width as the register itself.

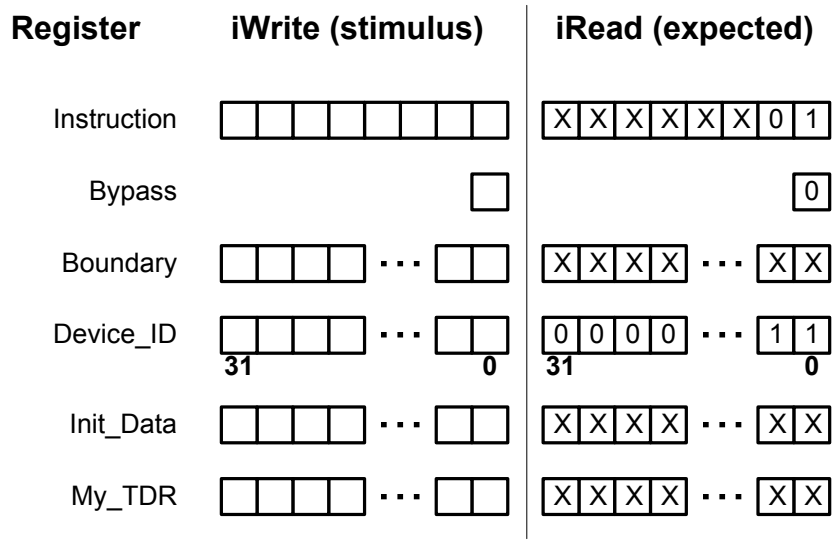


Figure C-3—PDL scan frame

Prior to execution, the write and expect data will be initialized as required by this standard and as specified in the BSDL attributes. PDL uses the **iWrite** (or **iScan**) command to modify values in the write data and the **iRead** (or **iScan**) command to modify values in the expect data. The modification continues for as many **iRead**, **iWrite**, or **iScan** commands as are encountered prior to an **iApply** command, and all **iRead**, **iWrite**, and **iScan** commands must reference fields of the same TDR. When multiple **iRead**, **iWrite**, or **iScan** commands modify the same bits, the last value specified is used. This convention can be used to set all the bits in a register to a single value (e.g., all 0s) and then to set an individual bit to the opposite value (e.g., 1s):

```
iWrite my_reg  0            ; # Pad with zeros for full register
iWrite my_reg(234) 0b1      ; # Set bit (234) to '1'
iApply
```

NOTE—Registers may be modified as above within a given PDL procedure. If procedures are executed in parallel, such access to the same field of a common register from two different PDL procedures is not allowed.

When an **iApply** command is executed, PDL will move the data for the entire TDR to the TDI to TDO scan path, including scanning the instruction register, when necessary, moving the TAP through the appropriate states, etc. In addition, PDL maintains two Boolean error flags called the accumulating and the local fail flags that report errors either for the entire test or since the last **iApply** command, respectively. This is all transparent to the PDL procedure itself.

In optional Level-1 PDL, the **iApply** command will also record the captured data and bit-by-bit fail data so that any of these data (write, expect, capture, and fail) may be accessed with the Level-1 PDL **iGet** command and used in Tcl code.

Figure C-4 shows the **iApply** command data flow from the PDL maintained data to the unit under test (UUT). The UUT could be a single component or a chain of components as shown in the use model, Figure C-1. While this figure illustrates the concept as if it were hardware, all of this could be done by software as well. The **iRead**, **iWrite**, and **iScan** commands modify the default data in the database for the various fields, which is then assembled by the **iApply** command into the output, expect, and mask data for the TDR to be scanned. In Level-1 PDL, the **iApply** command also collects the capture data and the bit-by-bit fail data for the database. In either case, unmasked mis-compares are used to set one or both of the fail flags.

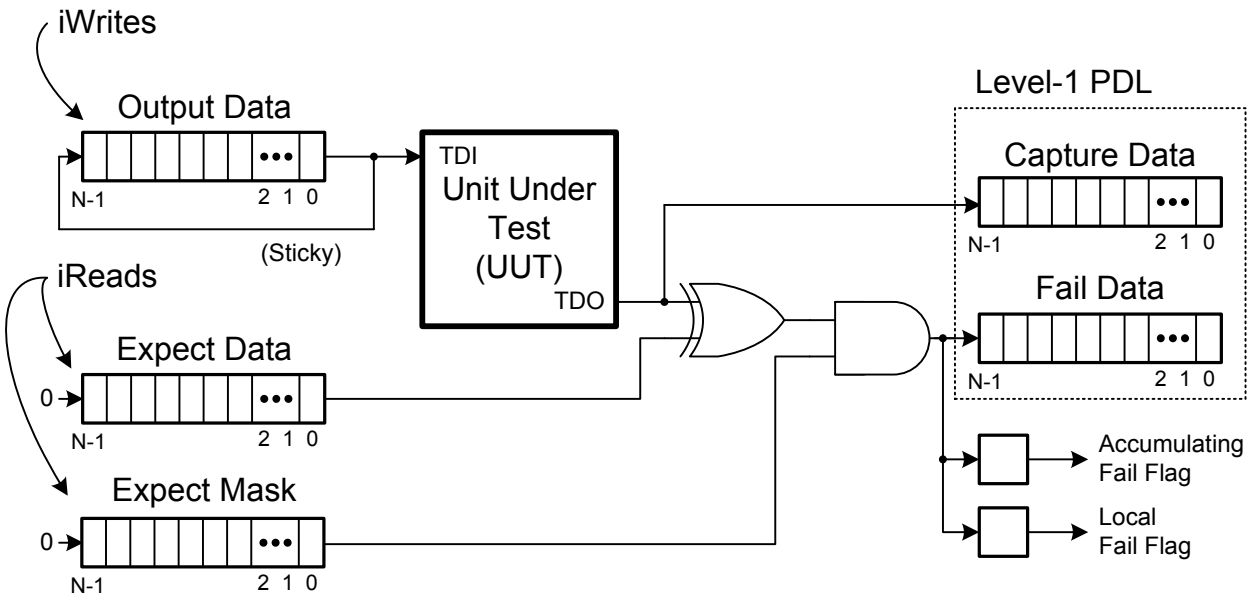


Figure C-4—Data flow during an iApply command

The first of the two "Fail Flag" registers in Figure C-4 accumulates fail information over multiple **iApply** commands, possibly for the entire test, while the second accumulates fail information only for a single **iApply** command. These two fail flags differ only in the source of their "Clear Fail" source.

After the completion of an **iApply**, the write data are retained unmodified, since the TDR may have to be written again later after only a part of the register has been modified by an **iWrite** or **iScan** command. However, since only the bits specified by an **iRead** or **iScan** command prior to the **iApply** (or the default fixed capture bits specified in this standard or with CAPTURES values in the register access or field definitions) are compared to the read scan frame data, and since the expect value is generally independent of history, the expect data are reset to that default value after completion of an **iApply** command.

Each register or register field that can be referenced in an **iRead** or **iWrite** command has a precise calculable location (position and length) within the scan chain of the unit under test. This information is derived from the component BSDL and, if needed, the board netlist. Each register or register field that can be referenced in an **iScan** command has a precise position within the scan chain, but only its initial length is known, and therefore, the current length must be specified each time the command is used.

Where excludable segments exist in a TDR, the location and state of those segments (included or excluded), and the location of the controlling **DOMCTRL** and **SEGSEL** fields, will need to be maintained as well. If the write or expect data for a currently excluded segment are changed, then the following **iApply** command will need to perform the scans necessary to include the segment before performing the data scan. In Level-1 PDL, captured and fail data will only exist for included and selected portions of the TDR, and excluded or unselected fields will have to be held to the last value captured, if any, or set to a state indicating that they have not been scanned and, therefore, no data are available. (An **iRead** command with the name of a field in the excluded or unselected segment, but no data value, can cause the segment to be included and then captured.)

C.2.6 Level-0 PDL commands

Table C-1 lists the Level-0 PDL commands defined in this standard. Detailed syntax and semantic checks for each command are defined in C.2.9.

Table C-1—PDL Level-0 commands

Command	Parameters	Purpose
Procedure definition		
iSource	<PDL file name>	Include a PDL file containing procedure definitions, which is read as if the contents are inline. All PDL procedures must be defined before they are called.
iPDLLevel	<level> **-version** <1149.1_version>	Identify PDL level and version for all of the following procedures in a file.
iProcGroup	<Entity, Package, or instance name>	Identify the object or instance with which the following PDL iProc procedures are associated.
iProc	<proc_name> <options> { <arguments> } { <commands> }	A PDL procedure.
Test setup		
iSetInstruction	<instruction>	Select instruction for a register accessed by multiple instructions.
iClock	<ClockName> **-period** <seconds> <**on**|**off**>	Define the system clock.
iClockOverride	<IntClockName> **-source** <ClockName> **-freqMultiplier** <real> <**on**|**off**>	Define on-chip clock multipliers.
iPrefix	<instance_or_null>	Specify (partial) hierarchy for registers.
Test execution		
iWrite	<register or instance> <value>	Queue data to be written.
iRead	<register or instance> [<value>]	Queue data to be compared with what is read.
iApply	[options] [<label>]	Execute queued operations.
iScan	<register or instance> <length> **-si** <si_data> **-so** <so_data>	Queue data for “black box” register or segment.
Flow control		
iCall	[**-direct**] [<instance>.]<proc_name> [<argument values>]	Transfer execution to a PDL procedure associated with the object associated with the named or current instance; or associated directly with the instance.
iRunLoop	<delayspec>	Issue a number of clocks or wait an absolute time.
------ iLoop		Mark the beginning of a conditional loop.
iUntil	**-match** | **-mismatch** [**-maxloop** <maxcnt> [<text]]	Loop until any iApply -nofail in the loop meets the specified condition or the maximum loop count is reached.
ifTrue		Execute commands based on last iApply –nofail.
ifFalse		Execute commands based on last iApply –nofail.
ifEnd		End of conditional execution.
Optimization		
iMerge	**-begin** | **-end**	Provides guidance on where tools can optimize multiple

Command	Parameters	Purpose
		PDL procedures.
iTake	<resource>	Tag an object to be 'in-use' to provide guidance where tools can optimize PDL.
iRelease	<resource>	Release an object from 'in-use' to provide guidance where tools can optimize PDL.
Miscellaneous		
} **iSetFail**	[**-quit**] [<text>]	Set the status to "FAIL"; stop the test if the -quit parameter is specified, and output the <text>.
iNote	**-comment** | **-status** <text>	Creates a tool identifiable comment intended to pass either detailed annotation information to the output vectors (**-comment**) or execution status information to the system (**-status**).
Low-level commands		
iTRST	**-on** | **-off**	Assert or de-assert the TAP TRST* pin, and remain in TLR state after TRST* is off. Only executed at the top level of the current unit under test.
dures that are run stand-alone. **iTMSreset**		Enter TLR state via TMS. Only executed at the top level of the current unit under test.
iTMSidle		Enter RTI state via TMS.
NOTE—This table is informative, and no attempt is made to define the syntax. Angle brackets (< >) are used to show elements that are design-specific, as opposed to language keywords. The vertical bar (|) is used to separate a list of options.		

The **iRead**, **iWrite**, and **iApply** commands do most of the work in creating scans of the UUT. The **iScan** command supports queuing data for registers that are not documented in detail in the BSDL. The rest of the commands are needed in order to write procedures supporting this core capability.

The procedure definition commands provide the definition and context necessary so that named procedures can be called, and their basic definition is understood. The **iProc** command, mentioned earlier, defines a procedure. **iPDLLevel** establishes the PDL level and the compliance of following procedures. **iProcGroup** associates procedures with specific objects or instances in the UUT so they can be identified when called by an **iCall** command. **iSource** allows multiple files to be pulled together to define all the procedures for a UUT.

The instruction to be used to access a register is determined from the instruction to register associations in the BSDL. If a register can be accessed by multiple instructions (such as the bypass or boundary-scan registers), then the instruction associated with the register is specified either by default or by a preceding **iSetInstruction** command.

Clocks can be identified and described using the **iClock** and **iClockOverride** commands.

The **iPrefix** command identifies an instance that subsequent commands will use to reference register fields.

Level-0 PDL supports a limited form of flow control. First, control can be passed from one procedure to another using the **iCall** command. Delays, to allow for clocked processes in the UUT, for instance, can be inserted with the **iRunLoop** command. When a register field is scanned with an **iApply** command, the captured value will be compared to a fixed value, and the system will record whether the comparison matched or not. Looping and if-then-else commands (**iLoop**, **iUntil**, **ifTrue**, **ifFalse**, and **ifEnd**) allow conditional execution of some set of commands

based on the result of that comparison. The **iSetFail** command allows the accumulating fail flag to be set to "FAIL" if the PDL software detects a failing condition.

Level-1 PDL supports the full flow control capabilities of Tcl.

PDL procedures may be executed in parallel when there are multiple objects in the scan chain, each supported with a PDL procedure. While the details of how the various commands in the multiple procedures are to be aligned is beyond the scope of this standard, the **iMerge**, **iTake**, and **iRelease** commands allow a PDL coder to provide information to assist the process.

The **iNote** command allows text to be inserted in any compiled test vectors to help guide the test engineers.

Supporting software must *not* implement entering the *Test-Logic-Reset* TAP controller state at any time except in response to the PDL **iTMSreset** or **iTRST** commands. This preserves data in the registers and helps prevent interference between PDL procedures, which may otherwise operate in parallel. Note that the use of these reset commands is heavily restricted as, in most test situations, any general reset will be performed as part of the preparation for test, and does not need to be repeated during the test.

For those rare cases when they are needed, some additional low-level commands are provided to alter the normal operation. The **-shiftPause** parameter on the **iApply** command can cause a register to be scanned multiple times without going through the *Update-DR* and *Capture-DR* states for each shift. This supports pausing the shift to permit *Read-Modify-Write* sequences, as well as some low-level tests of register integrity and connectivity. The **iTMSidle** command and the **-skipRTI** flag on the **iApply** command force or prevent going through the *Run-Test/Idle* TAP controller state when an instruction similar to *RUNBIST* is active.

C.2.7 Specification of names and values

There are several policies defining how register names and their associated values must be specified in PDL:

- Omission of the register's bit range is acceptable when the intent is to address the register's full width.
- Underspecification of a value (not enough bits) will result in the assignment of the specified bit values to the least significant bits of the associated register. Upper bits are padded to zero for an **iWrite**, and to "X" for an **iRead**.

```
# Reg16bit is currently defaulted to all '0'
iWrite Reg16bit 0x555     ; # 16 bit register is now set to 0x0555
```

- Overspecification of a value (too many bits) is allowed only if the overflow bits are all 0.

The commented examples below indicate the range of usages and the policies applied. All register fields are 10 bits wide and contained in a single TDR.

```
iProc main_pdl { } {
  iWrite reg_a 0 ;            # implicit reg width, decimal value
  iWrite reg_a(3) 1 ;         # overwrite of single bit of a multi-bit reg
  iApply ;                    # TDR 1

  iRead reg_p 0x000 ;         # implicit reg width; lead "0b00" truncated
  iRead reg_q 0 ;             # implicit reg width; decimal value
  iRead reg_r 2 ;             # implicit reg width; decimal value
  iRead reg_s 0xA ;           # implicit reg width; hex value (padded)
  iRead reg_x ;               # no value specified, expect default
  iRead reg_y 0x25A  ;        # implicit reg width, lead "0b00" truncated
  iApply ;                    # TDR 2
}
```

C.2.8 Retargeting

Every **iProc** command, at the time it is defined, is associated with a specific object (BSDL entity or package name) or instance. When that **iProc** command is called, it will always be called for a specific instance. That instance holds a unique position in the object hierarchy of the UUT.

The **iProc** command, when written, cannot know what the UUT instance hierarchy is, and in fact, an **iProc** command may need to be used for multiple instances in any one hierarchy, and for many different hierarchies. Therefore, at the time that the **iProc** command is called, PDL assumes the full hierarchical instance path of the current procedure (called the context instance path, or context path for short) is an implicit argument to the called procedure, and this hierarchical path is pre-pended to all register references (**iWrite**, **iRead**, etc.) and to all called procedure names (**iCall**) called within the procedure. This allows the **iProc** command to be written once for a reusable object, and to be automatically re-targeted to specific instances when used. This allows PDL to find the appropriate register and register field definitions.

iProcGroup is the command that associates the PDL procedures to a specific BSDL entity name or Package name (for reusable IP on a component), when they are defined. **iProcGroup** also may link the PDL procedures to a specific instance in the hierarchy when different instances require a different procedure, such as may happen for the init_data register.

For each component (IC or IP), there are one or more instances in the unit under test (UUT). Each register or field instance may be addressed by the hierarchically concatenated instance names from the UUT down to the containing instance, which is finally concatenated to the register or register field name from the BSDL or BSDL Package. This means that all register names are local to the specific object, and the specific hierarchy of instances must be used to identify the specific object containing the named register. Obviously, when addressing a register, the instruction register inherits the same hierarchy, but just down to the IC (TAP) level, as indicated by having a BSDL entity <component name> associated with the object.

Similarly, procedures called with the **iCall** command require the same hierarchically concatenated instance names down to the instance of the object associated with the PDL being called. This inheriting of the hierarchical instance names allows PDL written for a lower level object to be reused at higher levels.

The hierarchy of concatenated instances below the level of the object associated with the current procedure may be explicitly added to each register or procedure name, or it may be supplied for use with multiple registers by the **iPrefix** command, in addition to the inheritance of the context instance path above the current level as described above. Table C-2 shows a simple example of how the instance path is built up through a series of **iCall** and **iPrefix** commands. Note that at each level, the PDL only includes levels of hierarchy contained within the object associated with that procedure. Each level of hierarchy adds its own instance(s) to the full path. At the time of execution, the "flat" interpretation provides a full path from the UUT down to each called procedure or register referenced in the current procedure. Also note that the **iPrefix** command affects the path to the registers but not called procedures.

Table C-2—Handling PDL procedure hierarchy

	Hierarchical PDL procedures	"Flat" interpretation
1	`# Top PDL (Context Path is empty)`	`# Effective retargeting`
2	`#   U3 is instance of Chip_C`	
3	`  iCall U3.init_setup`	`  iCall U3.init_setup`
4		
5	`# IC PDL (Context Path is U3)`	
6	`#   i1 is instance of SERDES`	
7	`  iCall i1.init_setup`	`  iCall U3.i1.init_setup`
8		
9	`# IP PDL (Context Path is U3.i1)`	
10	`#   SERDES contains chan(0 TO    7)`	
11	`  iPrefix chan(0)`	
12	`  iCall powercheck`	`iCall  U3.i1.powercheck`
13	`  iWrite protocol XAUI`	`iWrite U3.i1.chan(0).protocol XAUI`
14	`  iWrite TX_swing full`	`iWrite U3.i1.chan(0).TX_swing full`

C.2.9 Simple PDL Example

Referring again to Figure C-1, an example U3.PDL file with two procedures is shown below. The two procedures are "init_setup," which is specific to instance U3 of IC Chip_C, and "init_run," which is specific to all instances of the IC Chip_C. A more extensive example for Figure C-1 is in C.5.

U3.PDL

```
iPDLLevel 0 -version STD_1149_1_2013  ; # level-0 PDL only
iProcGroup U3     ; # Associate the following procs with the chip instance

# this procedure becomes U3.init_setup internally to PDL
iProc  init_setup -export { }  {
  iPrefix  i1
  iWrite  Clock       125Mhz     ; # use of BSDL mnemonics
  iWrite  Voltage     0x40       ; # use of hex values
  iWrite  Protocol    XAUI       ; # use of BSDL mnemonics
  iApply
}

iProcGroup Chip_C ; # Associate the following procs with the chip type
# this procedure becomes Chip_C.init_run internally to PDL
iProc  init_run { }  {
  iRunLoop 10000  ;                # 10,000 TCK cycle delay
  iRead   i1.init_status(1)   Pass; # use of single register bit
  iApply
}

#end of file
```

A board-level PDL could then use the following to execute *INIT_SETUP* for U3.

```
iSource U3.PDL
iPDLLevel 0 -version STD_1149_1_2013  ; # level-0 PDL only
iProcGroup board
iProc top { } {
  iCall -direct U3.init_setup ; # Do not look up type from instance
```

```
  iCall U3.init_run           ; # Look up type (Chip_C) from instance (U3)
}
# end of file
```

The first line of the PDL indicates what level of PDL is present. It is then followed by commands that source other files with required procedures, then a command that specifies what object these procs belong to, and then a procedure that calls the procedures init_setup and init_run. Note that the call to U3.init_setup uses the -direct parameter to indicate that this procedure is instance specific, not object specific. No lookup of the instance to determine the object is performed for this call. “Init_setup” procedures are often instance specific, rather than applying to all instances of an object.

A PDL may also describe the operation of IP blocks. The following example illustrates the use of **iProcGroup** for object types rather than for instances. The first PDL in this example is MEMB.PDL and represents the procedures associated with the registers in the package MEMB. Figure C-1 shows IC U1 with two instances of MEMB: mem1 and mem2.

MEMB.PDL

```
iPDLLevel 0 -version STD_1149_1_2013  ; # level-0 PDL only
iProcGroup  MEMB   ; # Associate procs to object MEMB

iProc start_bist {bistreg bist_mode} {
  iWrite $bistreg 0b1           ; # $bistreg chooses register
  iWrite BIST_mode $bist_mode   ; # $bist_mode: argument substitution
  iApply
  iRunLoop 10000
}

iProc run_bist -export { } {
  # software converts instance path U1.mem1 to a type lookup
  iCall start_bist BIST_engage(42) 0b110 ; # U1.mem1 or U1.mem2
  iRead bist_sig  Pass
  iApply
}


# end of file
```

A PDL for devices of type CHIP_A can be described as follows.

Chip_A.PDL

```
iSource MEMB.PDL   ; # Bring in MEMB procedures so they can be called.
iPDLLevel 0 -version STD_1149_1_2013  ; # level-0 PDL only
iProcGroup CHIP_A  ; # type CHIP_A, which has a BSDL CHIP_A.BSDL

# iProc “main” is normally executed after interconnect test
iProc main { } {
# mem1 and mem2 are instances of memb.
#   Note that these could be run sequentially or in parallel.
iMerge -begin
  iCall mem1.run_bist; # Call MEMB.run_bist for instance mem1
  iCall mem2.run_bist; # Call MEMB.run_bist for instance mem2
iMerge -end
}
# end of file
```

A PDL for a board with U1 of type CHIP_A

```
iSource CHIP_A.pdl      ; #  find and define all CHIP_A procedures
iPDLLevel 0 -version STD_1149_1_2013  ; # level-0 PDL only
# U1 is an instance of CHIP_A
iCall U1.main ;# call CHIP_A.main for instance U1

# end of file
```

C.3 PDL Level 0 command reference

The Syntax, Rules, Recommendations, and Permissions in this xlause are normative, and they provide the formal definition of Level-0 PDL. Introductory text, Notes, and Examples are descriptive.

C.3.1 Understanding a PDL “string”

All of PDL is considered to be a string containing one or more commands. Semicolons and newlines are command separators. Each command is composed of a sequence of words, and the words are separated by one or more space or horizontal tabulation characters. Level-0 PDL follows a compatible subset of the conventions of Tcl as described here. Level-1 PDL incorporates all of Tcl, which goes beyond the following description.

A command is evaluated in two steps. First, the command (the collection of characters between command separators) is broken into words with whitespace as separators. The backslash-newline substituting is performed before the command is broken into words, and the other substitutions described below (backslash, command, and value) are performed after the command is broken into words. These substitutions are performed in the same way for all commands. Second, the first word of the command is used to locate a command procedure to carry out the command, and all of the remaining words of the command are passed to the command procedure. Each PDL command will interpret the words passed to it according to the syntax (BNF) and semantics (Rules, Recommendations, and Permissions) defined in this annex. These rules apply *after* all substitutions are performed.

The only backslash substitution supported in Level-0 PDL is the backslash-newline-whitespace character sequence. These characters are replaced by a single space character. In effect, two lines are merged into one. This backslash sequence is unique in that it is replaced before the command is actually broken into words. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it is not enclosed in braces or quotes.

Command substitution, that is, starting a word with a bracket ([ ), is not supported in Level-0 PDL.

Value substitution occurs if a word contains a dollar sign ($). The dollar sign and (in Level-0 PDL) the following PDL identifier are replaced with the value of the PDL identifier prior to use. The value is substituted verbatim, and no further substitution processing is performed. For example:

```
iProc myproc { reg value } {
  iWrite my$reg $value
}
-------------------
...
iCall myproc Reg5 0x55 ; # In a higher level proc, writes myreg5.
...
```

In Level-0 PDL, substitution variables can only be defined as an argument in a procedure definition, as shown in the above example. The values may only be defined either in the call of the procedure, as shown, or as default values in the procedure definition. Due to these restrictions, the $name(index) and ${non-PDL name} forms of value substitution are not supported in PDL0. (${PDL name} may be used.)

The above substitutions within a word do not affect the boundaries of words in a command. For example, during variable substitution, the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

When a word starts with a double-quote (") character, then the word will be terminated by the next double-quote character. Semicolons, close brackets, whitespace, or newline characters, within the word, are treated as ordinary characters and included in the word. Substitutions (variable, command, and backslash—limited as just described above in Level-0 PDL) are performed within the word as described below. The double-quotes are removed prior to passing the word to a command processor.

If the first character of a word is an open brace ({), then the word is terminated by the matching close brace (}). No substitutions are performed on the characters between the braces except for the backslash-newline substitutions described above, nor do semicolons, newlines, close brackets, whitespace, and so on receive any special interpretation. The word will consist of exactly the characters between the outer braces. The braces are not retained as part of the word.

There is one significant difference between the use of quotes and braces: If a substitution variable is included, then if it is in a word enclosed by quotes, the substitution takes place before the word is passed to the command function, but if it is in a word enclosed in curly braces, then the substitution does not take place. For example:

```
  iProc myproc { value1 } {
    iWrite rega "$value1"
    iWrite regb {$value1}
}
--------------------
...
iCall myproc 0x55 ; # In a higher level proc
...
```

The result is that rega will get the value 0x55, and regb will not as the string $value1 is not a compliant value for the iWrite command.

If a pound sign character (also called the hash character) (#) appears at a point where PDL is expecting the first character of the first word of a command, then the pound sign character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears in the position of the beginning of a command.

There is no separate PDL element called a "string," and every "word" may be enclosed in double quote marks ("") or curly braces ({ }) or not. The quotes or curly braces are not retained as part of the word. For example, the following two lines are equivalent:

```
iWrite  myreg 0x55
"iWrite" {myreg} "0x55"
```

C.3.2 BNF conventions

In this annex, the BNF syntax is used only to define the PDL commands and their arguments. The syntax of PDL commands is presented in this standard in the same modified Backus-Naur form (BNF) used for BSDL. The definition is as follows:

— Any item enclosed in chevrons (i.e., between the character < and the character >) is the name of a syntax item (a token) that will be defined in this annex or in Annex B. To assist in differentiating PDL tokens from BSDL tokens, all PDL tokens use underscores instead of spaces between words. Example: <VHDL identifier> versus <PDL_identifier>.

— Items enclosed between braces (i.e., between the character { and the character }) can either be omitted or included one or more times. Where the brace characters are required in the PDL syntax, they will be shown by the tokens <L_brace> and <R_brace>. In Table C-1, the tokens { and } are shown explicitly for compactness.

— Items enclosed between square brackets (i.e., between the character [and the character]) can be either omitted or included only one time.

— Parenthesis [i.e., the characters (and)] are not used in this BNF. Where the parenthesis characters are required in the PDL syntax, they will be shown by the tokens <L_paren> and <R_paren>.

— Text shown in **bold Helvetica** type shall be included exactly as it is presented in this annex except that command arguments preceded by a minus sign (-) are case-insensitive..

— Alternative syntaxes (choices) are separated by a vertical bar (|).

— The symbol ::= is read as "is defined as."

— Whitespace (spaces, tabulation, carriage returns, etc.) is used in these BNF descriptions to provide enhanced readability of the BNF and is not part of the syntax.

C.3.3 PDL lexical elements and common syntax

C.3.3.1 Lexical element specifications

The lexical elements form the atomic, axiomatic elements of the language.

General rules

a) PDL shall follow the conventions of Tcl except as noted in this standard.

NOTE 1—The "conventions of Tcl" specifically include the processing of the input string into commands and words, with substitutions, and the calling of the command so identified. See C.3.1 for an overview and multiple sources on the Internet for details.

b) The following tokens are used in the syntactical and semantic descriptions:
   1) <space> shall be a single blank space character.
   2) <white_space> shall be any combination of one or more space or horizontal tabulation characters.
   3) <newline> shall be any combination of nonprinting characters used by the current file system to move following text to the start of the next line when printed or displayed, or the end of the file (see <EOF>).
   4) <backslash> shall be the character \.
   5) <semicolon> shall be the character **;**.
   6) <hash_mark> shall be the character **#**.
   7) <period> shall be the character **.**.
   8) <comma> shall be the character **,**.
   9) <colon> shall be the character **:**.
   10) <plus_sign> shall be the character +.
   11) <minus_sign> shall be the character -.
   12) <L_brace> and <R_brace> shall be the characters { and }, respectively.

c) <comment> shall start with a <hash_mark> character ("#"), if and only if it appears at the point where PDL is expecting the first character of a command name (the first non-whitespace character on a line or after a semicolon), and ends with a <newline> character; and all characters between those two characters are ignored.

NOTE 2—If they follow a PDL statement on the same line, a semicolon is required as a separator. No comment can be inserted in the middle of a command. Comments may extend over multiple lines using the <backslash><newline> character pair or a new comment may be started on each line.

d) <EOF> shall represent the end of a file, which is treated as a <newline> and parsing stops.

e) Only PDL command names, <PDL_identifiers>, and <substitution variables> shall be case-sensitive.

f) The length of a line in PDL shall be unlimited.

g) <ct> shall indicate a command terminator consisting of a <newline> (without a preceding <backslash>), <semicolon>, or <EOF>.

h) If there are multiple commands on a single line, or a command followed by a comment, they shall be separated by <semicolon>.

i) A single x or X character in scan data shall define a don't-care bit in binary and four (4) bits of don't care in hexadecimal.

NOTE 3—Scan data with random don't-care bits will likely require a binary representation. A hex value retrieved by the **iGet** command and with both known and don't-care bits in a hex digit is an error and that error is represented as undefinable with the character U, which is not a compliant value in <hex_numX>. Decimal numbers cannot contain don't-care bits by definition.

j) Scan data shall be defined with the least significant bit physically closest to the output scan port, and scan data always shift from the most significant bit toward the least significant.

Numeric literal rules

k) <bin_num1> shall be a contiguous string of characters starting with 0b followed by one of the set [01] followed by zero or more of the characters in the set [01_<white_space><newline>], and if <white_space> or <newline> appears in the middle of a value, then the value shall be enclosed in double-quotes or curly braces.

l) <bin_numX> shall be a contiguous string of characters starting with 0b followed by one of the set [01xX] followed by zero or more of the characters in the set [01xX_<white_space><newline>], and if <white_space> or <newline> appears in the middle of a value, then the value shall be enclosed in double-quotes or curly braces.

m) <hex_num1> shall be a contiguous string of characters starting with 0x followed by one of the set [0-9a-fA-F] followed by zero or more of the characters in the set [0-9a-fA-F_<white_space><newline>], and if <white_space> or <newline> appears in the middle of a value, then the value shall be enclosed in double-quotes or curly braces.

n) <hex_numX> shall be a contiguous string of characters starting with 0x followed by one of the set [0-9a-fA-FxX] followed by zero or more of the characters in the set [0-9a-fA-FxX_<white_space><newline>], and if <white_space> or <newline> appears in the middle of a value, then the value shall be enclosed in double-quotes or curly braces.

NOTE 4—An **iWrite**, **iRead**, and **iScan** may use numbers in hexadecimal or binary with a clarity separator of one or more underscore characters or spaces. An example is 0b1000__1001. This and bit values of X are not Tcl number compatible but are allowed in application-specific extensions such as iWrite and iRead. An example using spaces might be "0b1000 1001" or {0b1000 1001}.

o) <dec_num1> shall be a contiguous string of characters in the set [0-9]; multiple character values shall not start with 0, and they shall have a numeric value in the range of $(2^{32}-1)$ down to zero.

NOTE 5—A multicharacter decimal number beginning with 0 is interpreted as an octal number in Tcl.

p) All numeric values (integer or real) shall be non-negative.

Identifier rules

q) <VHDL identifier> shall be as defined in rule a) of B.5.4.

r) <PDL_identifier> shall be a contiguous string of characters starting with any character in the set [a-zA-Z] followed by one or more of the characters in the set [a-zA-Z0-9_], and it is case sensitive.

NOTE 6—PDL identifiers allow multiple underscores in a row and at the end of the identifier; VHDL identifiers do not. PDL identifiers are case sensitive, and VHDL identifiers are not. Otherwise the definitions are the same.

s) <substitution variable> shall be the special character dollar-sign ($) pre-pended without whitespace to a <PDL identifier>, optionally enclosed in braces, and terminated by any character not allowed in a <PDL identifier>, including the optional closing brace.

NOTE 7—In Level-0 PDL, a <substitution variable> may only be defined as an argument to a procedure being defined in an **iProc** command. In Level-1 PDL, a <substitution variable> may also be defined using Tcl constructs. Substitution variables are case sensitive.

Text string rules

t) <text> shall be a single word composed of a contiguous set of one or more of the characters [0-9a-fA-F], plus the special characters tilde (**~**), tic (**`**), at-sign (**@**), dollar-sign ($), percent-sign (**%**), caret (**^**), ampersand (&), asterisk (*), underscore (_), minus-sign (-), plus-sign (**+**), equal-sign (**=**), and vertical-bar (|), colon (:), apostrophe ('), chevrons (< >), comma (,), period (.), exclamation-mark (!), question-mark (?), backslash (\), and slash (/), and if the text is enclosed in quotes or curly braces, then it may also include <white_space> and <newline> characters.

C.3.3.2 Substitutions

Level-0 PDL only supports a subset of the Tcl language substitution capability. Level-1 PDL supports the full Tcl capability. Specifically, Level-0 PDL does not support:

- Command substitution
- Backslash substitution, other than <backslash><newline>
- Variable substitution names that are not <PDL_identifiers>
- Variable substitution in the command name

Variable substitution is allowed in any word of a command other than the command itself. This includes both syntactic tokens and dash-parameter keywords. In many cases, it is possible that the result of a substitution could create words that violate the syntax or semantics for the command, and which must be caught by the command processor as it checks the incoming words. In many cases, such as a mandatory keyword, substitution may not be useful, but it is allowed unless the result is invalid for the command.

In Level-0 PDL, the <substitution_variable> name can only be defined as an argument to an **iProc** command, and the value can only be assigned either as a default value in the **iProc** command argument or as the argument of an **iCall** command. This means that a <substitution_variable> may only be used in commands within an **iProc** command. In Level-1 PDL, Tcl constructs can define <substitution_variable> names and values at any point, so there is no such restriction.

The definition of a Level-0 PDL <substitution_variable> allows the <PDL_identifier> to be enclosed in braces. This is needed only when the <substitution_variable> is embedded in a string that does not otherwise properly delimit the <substitution_variable>. Consider the variable “xyz” in the following example:

```
iProc myproc {xyz} {
  iNote -status "abc$xyz def\n" ; # Compliant, space delimiter
  terminates xyz
```

```
    iNote -status "abc$xyzdef\n" ; # Non-compliant, no delimiter
    terminating xyz
    iNote -status "abc${xyz}def\n" ; # Compliant, braces used as xyz
    delimiter
}
```

It is not possible to properly identify the <substitution_variable> in the second **iNote** command because the end of the <PDL_identifier> is not delimited by a character not allowed in a <PDL_identifier>. This can be resolved by inserting braces around the <PDL_identifier> as shown in the third line. The curly braces are removed when the substitution is performed.

Rules

a) Other than explicit restrictions for Level-0 PDL, any discrepancy between the rules of this substitutions clause and standard Tcl implementations shall be resolved by using the Tcl implementation.

b) The <backslash><newline>[<white_space>] character sequence within a command shall be replaced by a single <space> character prior to breaking the command into words, causing the next line of text to be concatenated to the current line.

NOTE 1—Tcl, and therefore PDL Level-1, also support using the backslash character (\) to define or substitute various text formatting characters, Unicode characters (single and double byte), and "escape" characters that would normally not be allowed within a word. PDL level-0 commands do not support such uses of the backslash, although the backslash character is allowed in a <text> word.

c) In Level-0 PDL, no word shall start with the "[" character; that is, command substitution shall not be supported.

d) A <substitution_variable>, including any braces around the <PDL_identifier>, shall be replaced by its value after the command has been broken into words, but prior to passing it as an argument to the command processor and subsequent checking of syntactical and semantic rules.

NOTE 2—If a <substitution_variable> has a value that embeds whitespace or newline characters, the result of the substitution is still considered a single word. For example, the value "able baker" is considered a single word when used in a PDL command, not two. In some cases, use of whitespace or newline characters in a substitution value will result in syntax or semantic errors for the PDL command.

e) In Level-0 PDL, a <substitution_variable> shall be allowed in any word of a command within an iProc except the command name itself.

f) $PDL_INSTANCE_PATH shall be a predefined <substitution_variable> with a value of the <instance_path> on the call to the current procedure.

g) $PDL_CONTEXT_PATH shall be a predefined <substitution variable> with a value of the full context path to the current procedure.

NOTE 3—One possible use of these predefined substitution variables is in an **iNote** command to display the current path or current instance.

h) If a <text> string is enclosed in double quotes (""), <backslash><newline>[<whitespace>] and variable substitution shall be performed within the string for Level-0 PDL, all Tcl defined substitutions shall be performed for Level-1 PDL, the double quotes shall be removed, and the resulting character string (including any whitespace or newline characters) shall be passed as a single word to the PDL command processor.

NOTE 4—" myparm " is NOT the same as "myparm" or myparm. Everything between the quotation marks is part of the "word" that is passed, including leading or trailing spaces. Similarly, "able baker" is a single word, not two words. The same applies for braces. This is in conformance with Tcl usage.

i) If a <text> string is enclosed in a pair of braces ({ }), <backslash><newline>[<whitespace>] substitution shall be performed, but other backslash, variable, and command substitutions shall *not* be performed within the string, the pair of braces shall be removed, and the resulting character string (including any whitespace or newline characters) shall be passed as a single word to the PDL command processor.

NOTE 5—Long binary and hex values that are broken across multiple lines can be passed into **iWrite**, **iRead**, and **iScan** commands if the entire value is enclosed in a single pair of either quote marks or curly braces. This is a Tcl construct where the space and newline separated values are seen as one value, and may be used with any binary, hex, or text value. Such enclosing braces or quotes are stripped before the value is passed to the command function. Note that this is very different from the BSDL approach of quoting partial strings on single lines and then concatenating them.

C.3.3.3 Common syntax

Syntax

<register_inst> ::= <TDR_spec> | <field_instance>
<TDR_spec> ::= <TDR> [<array_index_list>]
<field_instance> ::= [<instance_path> <period>] <full_field_name>
<instance_path> ::= { <instance_ident> <period> } <instance_ident>
<instance_ident> ::= <segment ident> | <array_spec>
<array_spec> ::= <array segment ident> <array_index_list>
<array_index_list> ::= <L_paren> <array_index> { <comma> <array_index> } <R_paren>
<array_index> ::= <dec_num1> | <dec_range>
<dec_range> ::= <dec_num1> <colon> <dec_num1>
<full_field_name> ::= <extended field name> [<array_index_list>]

<rvalue> ::= <dec_num1> **<**period**>** <dec_num1> [**e [** <sign>] <dec_num1>]
<sign> ::= <plus_sign> | <minus_sign>
<seconds> ::= <rvalue>

Rules

a) The <register_inst> shall contain no embedded whitespace or new line characters,
b) The <rvalue> shall contain no embedded whitespace or new line characters.
c) If <sign> is not specified, the exponent shall default to a positive value.
d) All bit numbers listed in the <array_index_list>, either as an <integer> or within a <range>, shall be unique and a value within the <range> of the associated <array ident> or within the range (<field length> – 1 downto 0) of the associated <TDR> or <extended field name>, all as defined in BSDL.

NOTE—While the individual tokens are linked, many of the tokens in the above syntax are defined in the BSDL **REGISTER_FIELDS** or **REGISTER_ASSEMBLY** attributes in B.8.19.1 and B.8.21.1. The instance hierarchy, in particular, is defined in **REGISTER_ASSEMBLY** attributes.

Description

A <register_inst> may be either a TDR as defined in the **REGISTER_ACCESS** attribute, or a register field instantiated in a **REGISTER_ASSEMBLY** attribute. In both cases, it is possible to address a subset of bits within the register using a combination of bit numbers and ranges.

When a TDR is defined in a **REGISTER_ASSEMBLY** attribute, the order of entries in the list defines an implied bit ordering starting with most significant bit (i.e., TDR length – 1) and ending with zero (0).

Therefore, for an **iRead** or **iWrite** command, bits within a TDR can be referenced in any of several ways: (1) by the TDR name, possibly with an index or index range; or (2) by an field instance hierarchy, possibly with an index or index range. The TDR or field name alone implies the entire TDR or field. For example:

```
attribute REGISTER_ASSEMBLY of example : entity is:
TDR_A ( " &
    " ( dataA[2] ), " &  -- same as TDR_A(4:3)

" ( dataB[3]), "&    -- same as TDR_A(2:0)
")" ;
```

This register is of length 2 + 3 = 5 bits; Therefore, the pairs of **iWrite** commands below access the same bit in the TDR:

```
iWrite TDR_A(3) 0b1
iWrite dataA(0) 0b1
iApply
iWrite TDR_A(1) 0b1
iWrite dataB(1) 0b1
iApply
```

C.3.3.4 PDL reserved words

PDL is a context-sensitive language and, as a result does not, in general, require any reserved words.

Rules

a) All identifiers that start with **STD_1149_** shall be reserved.

Recommendations

b) All PDL command names, in any text case, should not be used as identifiers in order to minimize confusion.

C.3.4 PDL File

A PDL file has a specific structure. Comments, as defined in the lexical elements, may appear anywhere in the file where a command may appear.

Syntax

<PDL_File> ::= { <iSource_cmd> } <iPDLLevel_cmd> <iProc_group>
{ [<iPDLLevel_cmd>] <iProc_group> } <EOF>

<iProc_group> ::= <iProc_group_0> | <iProc_group_1>
<iProc_group_1> ::= [<Tcl_cmds> <iProcGroup_cmd>] [<Tcl_cmds>] { <iProc_cmd> <Tcl_cmds> }
<iProc_group_0> ::= [<iProcGroup_cmd>] <iProc_cmd> { <iProc_cmd> }

Rules

a) The <PDL_File> shall be Level-0 PDL from the beginning through the first **iPDLLevel** command, which will set the PDL level for subsequent commands.
b) A <iProc_group_1> shall not appear following an **iPDLLevel** command setting the PDL level to 0.
c) The token <Tcl_cmds> is not defined in this standard, except that it shall represent one or more valid commands in the current Tcl environment for Level-1 PDL.

Permissions

d) In Level-1 PDL (an <iProc_group_1>), Tcl commands may appear as allowed by Tcl rules.

C.3.5 Procedure definition commands

These commands exist at the top level of a PDL file, and they establish the compliance and association of the procedures, as well as define the procedures themselves.

C.3.5.1 iSource command

The **iSource** command is used to import PDL files containing **iProc** procedures that may then be called from procedures in the current PDL file. All of the PDL procedures necessary for test need not be contained in one file, but **iProc** commands in separate PDL files are not visible until they have all been included by using an **iSource** command. It is the only way to include IC- or IP-level PDL files such that the procedures contained in them are made available to the calling PDL program.

NOTE 1—For Level-0 PDL procedures running in a Tcl environment, many of the substitution rules of C.3.3.2 are not detectable by the time that the words are passed to the PDL command processors. The **iSource** command, in addition to finding and opening the named file, can also scan the file contents for violations of those rules.

Syntax

<iSource_cmd> ::= **iSource** <text> <ct>

Rules

a) All **iSource** commands shall appear before any other command in each PDL file.
b) <text> shall be the valid file name of a PDL file in the current file system, and further shall be enclosed in quotes or curly braces if it includes any embedded whitespace or new line characters.
c) When the sourced PDL file contains **iSource** commands, no **iSource** command in the sourced PDL file or any PDL file sourced below it shall source the current PDL file.

   NOTE 2—The sourced files must form a tree, not a loop.

d) All procedures called from within an **iProc** command in a PDL file shall be previously defined either by an **iProc** command in the current PDL file or by an **iProc** command within a PDL file previously sourced by an **iSource** command in the PDL file.

Example

```
iSource XYZ_IO.PDL
iSource "my IO.PDL" ;# quotes or braces around file names with spaces in them
```

C.3.5.2 iPDLLevel command

The **iPDLLevel** command is the first command of each PDL file following any **iSource** commands. The **iPDLLevel** command indicates both the PDL level and the version of PDL that it is compliant with. This command is not allowed inside a PDL procedure (inside an **iProc** command) and constrains all procedure definitions in that file following the **iPDLLevel** command up until another **iPDLLevel** command or the end of the file. The command may appear more than once within the file to set the PDL level and version for following procedure definitions.

This standard only requires support for PDL that is Level-0 and conforms to the **STD_1149_1_2013** version. PDL can be viewed as just an extension of Tcl, and it is expected that other standards will define their own versions of PDL or extensions of **STD_1149_1_2013** PDL. In addition, it is possible that others will define and support

extensions. The **iPDLLevel** command provides the means to identify such extensions, and for those that do not support such extensions to skip them by ignoring all commands until the next **iPDLLevel** command or <EOF>, whichever comes first.

Syntax

<iPDLLevel_cmd> ::= **iPDLLevel** <level> **-version** <version_string> <ct>

<level> **::= 0** | **1**
<version_string> ::= **STD_1149_1_2013** | <PDL_identifier>

NOTE 1—As other standards and later versions of this standard support PDL, additional <version_string> identifiers will be added.

Rules

a) The **iPDLLevel** command shall appear as the first command after any optional **iSource** commands in each PDL file.
b) The **iPDLLevel** command shall not appear within an **iProc** command.
c) For any <level> and <version_string>, all following procedures (**iProc** commands) up to the next **iPDLLevel** command shall contain only PDL commands conforming to those two specifications.
d) Any tool claiming compliance with this standard shall accept PDL with a specified <level> of **0** and a <version_string> of **STD_1149_1_2013**.

NOTE 2—Support of PDL Level-1 or of versions other than **STD_1149_1_2013** are not required for a tool to claim compliance with this standard.

Permissions

e) The **iPDLLevel** command may appear as needed to indicate which procedures conform to a specific <level> and <version_string> pair.

Example

```
iPDLLevel 0 -version STD_1149_1_2013  ; # level-0 1149.1 PDL only

iPDLLevel 1 -version My_Std_1149_1 ;    # level-1, with user extensions
```

C.3.5.3 iProcGroup command

The **iProcGroup** command associates PDL procedures following the command to a specific object or instance. This relationship is established for all procedures until the next **iProcGroup** command is encountered. This command enables tools to manage libraries of PDL procedures by associating specific procedures with each object and, when necessary, instance in the system, where PDL procedures for different objects (components or IP) or instances may have duplicate procedure names. The association of a specific instance of an object to an object type is done separately based on board netlists, use of IP packages, register field definitions in BSDL, and so on. The **iProcGroup** command is required for all **iProc** commands except those at the top level.

The **iProcGroup** command is not required for the top-level PDL procedures, and if an **iProcGroup** command does not appear before an **iProc** command, then the procedure being defined is assumed to be associated with the current unit under test. (The context instance path of such a procedure is the null string.) Procedures defined without a previous **iProcGroup** command are not reusable at higher levels of hierarchy.

Syntax

<iProcGroup_cmd> ::= **iProcGroup** <object_or_instance> <ct>

<object_or_instance> ::= <component name> | <user package name> | <instance_path>

NOTE—<component_name> is defined in BSDL B.8.1.1. <user package name> is defined in BSDL B.10.1.

Rules

a) The **iProcGroup** command shall not appear within an **iProc** command.
b) The **iProcGroup** command shall associate all following procedure definitions (**iProc** commands), up to the next **iProcGroup** command, with the <object or instance>.
c) <object or instance> shall be the name of an object or instance in the currently defined instance hierarchy.
d) <component name> and <user package name> shall be as defined in B.8.1.1 and B.10.1, respectively.
e) An **iProcGroup** value established within a file shall be cleared at the end of the file.

Permissions

f) The **iProcGroup** command may appear as needed to change the object or instance affiliation of subsequent procedures.

Example

```
# MYIO.PDL file
iPDLLevel 0 -version STD_1149_1_2013
iProcGroup MYIO

# MYIO Procedures
iProc setup -export { swingval } {
  iWrite swing $swingval
}

iProc AC_Mode -export { onoff } {
  iWrite ACMODE $onoff
}

#end of file, iProcGroup cleared.

# main.pdl
...
iSource myio.pdl
iPDLLevel 0 -version STD_1149_1_2013
iProcGroup MYIC

iProc myproc { } {
  ...
}
```

C.3.5.4 iProc command

The **iProc** command defines a named, callable, PDL procedure with descriptive optional parameters and optional arguments that may be passed to the procedure. Following Tcl practice, the argument name is intended to be used as

a substitution variable within the procedure, and any default value for an argument cannot itself contain a substitution variable (there is no way to define it outside of the iProc command).

Several procedure names have special meaning: “init_setup” and “init_run” to be used for automated initialization of components. If the *INIT_STATUS* or *INIT_RUN* instructions are coded in the associated BSDL, then the test software will expect the corresponding procedure name in the associated PDL. Procedure “ecid” describes the writes and reads necessary to access the ECID code of an IC. The procedure name “main” may be used at any level, and includes all tests recommended for execution by the object designer. At the board level, this procedure (if present) will normally be called after interconnect test (*EXTEST*) is complete.

Syntax

<iProc_cmd> ::= **iProc** <proc_name> { <proc_option> } <L_brace> **[** <proc_arguments>] <R_brace>
<L_brace> <proc_command> { <proc_command>} <R_brace> <ct>

<proc_option> ::= **-export** | **-TMSreset** | **-TRSTreset** | **-mission** | **-noninvasive** |
-noninteractive | <information>
<information> ::= **-info** <text>
<proc_name> ::= <PDL_identifier> | **init_setup** | **init_run** | **ecid** | **main**
<proc_arguments> ::= { <no_default_argument> } { <default_argument>}
<no_default_argument> ::= <argument_name>
<default_argument> ::= <L_brace> <argument_name> <argument_default> <R_brace>
<argurment_name> ::= <PDL_identifier>
<argument_default> ::= <text>
<proc_command> ::= <L0_command> | <L1_command>
<L0_command> ::=
<iWrite_cmd> | <iRead_cmd> | <iScan_cmd> | <iApply_cmd> |
<iCall_cmd> | <iRunLoop_cmd> | <iLoop_cmd> | <iUntil_cmd> |
<ifTrue_cmd> | <ifFalse_cmd> | <ifEnd_cmd> |
<iPrefix_cmd> | <iSetInstruction_cmd> | <iClock_cmd> | <iClockOverride_cmd> |
<iMerge_cmd> | <iTake_cmd> | <iRelease_cmd> |
<iNote_cmd> | <iSetFail_cmd> |
<iTRST_cmd> | <iTMSreset_cmd> | <iTMSidle_cmd>
<L1_command> ::= <iGet_cmd> | <iGetStatus_cmd> | <Tcl_cmds>

Rules

a) If the <level> stated in the most recent **iPDLLevel** command is **0**, then neither an <L1 command> nor a Tcl command shall appear within the **iProc**.

b) The token <Tcl_cmds> is not defined in this standard, except that it shall represent any valid command in the current Tcl environment for Level-1 PDL.

c) The <proc_name> shall be unique among procedure names within an <iProc_group>.

d) If the *INIT_SETUP, INIT_SETUP_CLAMP,* and/or *INIT_RUN* instructions exist in the BSDL, then a set of PDL procedures shall be associated with the BSDL by an **iProcGroup** command and procedures with the names init_setup and/or init_run, respectively, shall be provided in that set.

e) If provided, at any object level, procedure names init_setup and/or init_run and all procedures invoked from them by **iCall** commands:

   1) Shall prepare the associated object or instance for interconnect test (e.g., *EXTEST*) initialization and perform the initialization, respectively.
   2) Shall be Level-0 PDL procedures.
   3) Shall not contain any reset commands.

f) If provided, at any object level, procedure name “main” shall execute a preferred set of tests as defined by the object provider.

NOTE 1—In a board test scenario, any "main" procedure would normally be executed after completion of interconnect test.

g) Any procedure with the name "ecid" shall be a Level-1 PDL procedure that accesses and returns the Electronic Chip Identification value.

NOTE 2—Processing the "ecid" procedure is not required for a tool to claim compliance with this standard.

h) The init_setup, init_run, and ecid procedures, if provided, shall not contain an **iTRST** or **iTMSReset** command at any level of the procedure hierarchy; that is, they shall not have the **-TRSTreset** or **-TMSreset** keywords.

i) The <proc options> are descriptive and shall be interpreted as follows:

1) **-export:** this procedure is to be visible to and able to be called directly by the PDL user.

NOTE 3—When this option is not specified, the **iProc** name is visible only within the <iProc_group> in which it appears and can be called only by an **iProc** within that <iProc_group>.

2) **-TMSreset:** required if this procedure or a procedure called from this procedure contains an **iTMSreset** command.

3) **-TRSTreset:** required if this procedure or a procedure called from this procedure contains an **iTRST** command.

4) **-mission**: this procedure is intended to be executed while the associated object is in mission mode, and may alter mission mode behavior.

5) **-noninvasive**: this procedure is intended to be executed in either test or mission modes and will not interfere with or change any mission mode behavior.

6) **-noninteractive**: this Level-1 PDL procedure only contains Level-0 PDL and Tcl commands that can be rolled out to Level 0 PDL, and is able to be treated as Level-0 PDL in a tool that also supports Level-1 PDL; in particular, it contains no commands that access the status or the captured data.

NOTE 4—A Level-1 PDL procedure with the **-noninteractive** keyword can be converted into a compliant Level-0 PDL.

7) **-info** <text>: the <text> contains additional description of this procedure, similar to the <text> with an **iNote** command.

j) The <proc options> **-mission**, and **-noninvasive** shall be mutually exclusive.

k) In Level-0 PDL, no iProc command shall have an <argument_name> of args.

NOTE 5—The name "args" has special meaning for a iProc and for a Tcl proc. It is processed differently than other user-defined argument names.

l) The end of an **iProc** shall clear any **iPrefix** value, and the context instance path present before the procedure was called shall be restored.

Predefined procedure names

There are four predefined procedure names: init_setup, init_run, ecid, and main.

The init_setup and init_run procedures are the PDL procedures used to define the execution of the *INIT_SETUP* or *INIT_SETUP_CLAMP,* and of the *INIT_RUN* instructions, respectively. While it is possible to use those instructions to perform different initializations for different tests, these procedure names should be reserved for performing any initialization needed for interconnect test. The init_setup procedure is used to write and read all needed fields defined in the init_data register, including any with deferred values in the BSDL. The init_run procedure is used to perform any sequential operations necessary to bring the component to the proper known and safe state required for testing to proceed. See 8.17 for general information on initialization instructions, 8.18 for information on the

INIT_SETUP or *INIT_SETUP_CLAMP* instruction, 8.19 for information on the *INIT_RUN* instruction, Clause 14 for the init_data register, and Clause 15 for the init_status register.

The init_setup and init_run procedures must be capable of running in any environment able to perform *EXTEST*, so they cannot be interactive procedures, which access data returned from the unit under test, either data scanned out or the accumulating status. They can contain Tcl structures that, when interpreted, generate multiple Level-0 PDL statements, referred to as unrolling the Level-1 PDL to Level-0 PDL; at which point, the procedure can be treated as a Level-0 PDL procedure. These procedures must not contain any reset commands. While the intent is that initialization be performed using just the init_data and init_status registers, there is no restriction on accessing other registers as needed.

The ecid procedure also must not contain any reset commands and must be coded in Level-1 PDL since it returns a value. It can be run at any time needed. The main procedure is completely unrestricted and would be a set of tests to be run in addition to and usually after the interconnect test.

Any number of **iSource** commands may appear before the first **iPDLLevel** command in the file.

C.3.6 Test setup commands

C.3.6.1 iPrefix command

The purpose of the **iPrefix** command is to support temporary instance hierarchy within the object associated with the current PDL procedure. The specified value provides an instance path name prefix for all subsequent register and register field names. If defined, *all* following register or register field names will be prefixed with the <partial_path> value plus a trailing period (.).

An **iPrefix** command with the minus sign as an argument clears the current prefix.

Since the language is procedural, this command must be entered prior to any command that needs the instance path prefix. Note that if only one or two names need the instance path prefix while many others do not, it may be simpler to omit the command and include the instance path on only those names that require it.

The **iPrefix** command within a procedure only references hierarchy below the object associated with the procedure. This is necessary for reuse of the PDL procedure in different environments. Hierarchy down to the object associated with the procedure, called the context instance path, is made available when the procedure is called and an **iPrefix** command inside the procedure simply adds additional level(s). An **iPrefix** value is not passed down to a called procedure, but it is retained for use after the called procedure returns. **iPrefix** does not affect the target of an **iCall**. At the end of the procedure, any **iPrefix** value established in the procedure is cleared.

If a <partial_path> value is specified by an **iPrefix** command, it will be appended to the context instance path, established when the procedure was called, and then that combination will be prepended to the following <register_inst> register or register field references and to <port ID> IC port name references.

Note that while the value of an **iPrefix** command has a specific structure, no checking is normally done on the value. It is just a string. Only when the full name is assembled and passed to a command such as **iWrite** or **iRead** is the fully qualified name checked against the relevant objects in the structural database built from the BSDL.

Syntax

<iPrefix_cmd> ::= **iPrefix** <instance_or_null> <ct>
<instance_or_null> ::= <partial_path> | <minus_sign>
<partial_path> ::= <instance_path> | <prefix_path> | <compound_path>
<compound_path> ::= <instance_path> <period> <prefix_path>
<prefix_path> ::= { <prefix identifier> <period> } <prefix identifier>

Rules

a) The <partial_path> value shall contain no whitespace or new line characters.
b) The <partial_path> value shall be prepended to all following <register_inst> or <port ID> occurrences within the current PDL, and it shall use a <period> as separator.
c) An <instance_or_null> value of <minus_sign> shall clear the current <partial_path>; that is, it shall set it to the null string.
d) The <partial_path> value shall be cleared at the end of the current procedure.

Examples

```
# U1 is an instance of object "Chip"
iCall U1.myproc myinst2 ; # Top level call to "Chip.myproc" PDL procedure
# iPrefix is now null
iCall U1.nextproc    ;
------------------------
#Chip PDL
iProcGroup Chip

iProc myproc { mypath } {
  iPrefix myinst1           ; # identifies an instance path below chip
  iRead bootreg 0x55        ; # now iRead U1.myinst1.bootreg
  iPrefix $mypath           ; # If $mypath is null, generates an error
  iRead bootreg 0x44        ; # now iRead U1.myinst2.bootreg
  iRead myreg 0b1           ; # now iRead U1.myinst2.myreg
  iPrefix -                 ; # removes the local instance path
  iRead myinst3.jregA 0b0   ; # now iRead U1.myinst3.jregA
  iPrefix myinst1           ; # any prefix is cleared at the end of the iProc
}

iProc nextproc { } {
  iNote -status "inside nextproc iPrefix is not defined\n"
}
```

C.3.6.2 iSetInstruction command

The purpose of the **iSetInstruction** command is to remove the ambiguity of which instruction to use when accessing a register or register field that can be selected for scan by multiple instructions. The obvious example is the boundary-scan register, which can be scanned by *PRELOAD, SAMPLE, EXTEST,* or *INTEST,* among the standard instructions. There are defaults for all registers that can be selected by multiple instructions, but changes will be needed at times.

The command takes a single value of the instruction name. This pairs the instruction to the register that the instruction selects for scan per the definitions in this standard or in the **REGISTER_ACCESS** BSDL attribute. This pairing is inherited by all child processes in the call hierarchy below the current procedure, and it remains in effect until explicitly changed by another **iSetInstruction** command.

The **iSetInstruction** command can only appear in a procedure associated with a BSDL object by an **iProcGroup** command since the instruction names are not known otherwise.

Syntax

<iSetInstruction_cmd> ::= **iSetInstruction** <instruction name> <ct>

Rules

a) The **iSetInstruction** command shall only be used at the TAP (chip) level, that is, in procedures associated with an **iProcGroup** command to a <component name> as defined in BSDL.

b) If an **iSetInstruction** command is not specified in the calling hierarchy at or above the current procedure, the default instruction for the boundary register shall be *PRELOAD* if it exists; otherwise it shall be *SAMPLE*, the default instruction for the bypass register shall be *BYPASS*, the default instruction for the device_id register shall be *IDCODE*, the default instruction for the init_data register shall be *INIT_SETUP*, and the default for a design-specific TDR shall be the first instruction named for the register in the **REGISTER_ACCESS** attribute.

c) The assignment of a specific instruction to access a TDR by an **iSetInstruction** command shall persist until another **iSetInstruction** command changes the setting or the end of the procedure containing the **iSetInstruction** command; and further, upon completion of a procedure containing an **iSetInstruction** command, the assignment that existed before that procedure was called shall be restored.

Examples

```
# Put the chip into test mode for init.
  iSetInstruction init_setup_clamp ;
# Prepare to release test-mode persistence.
  iSetInstruction clamp_release ;   # Selects bypass register.
```

C.3.6.3 iClock and iClockOverride commands

The purpose of the **iClock** command is to define a clock at any level of packaging. The purpose of the **iClockOverride** command is to define the relationship between two clocks. One or both of these commands must be included in the PDL hierarchy if a clock is specified in an **iRunLoop** command.

The **-period** parameter indicates a minimum period with units of seconds that, after inversion, also defines a maximum system clock frequency in Hz. Note that unless otherwise specified by the **-off** parameter, all system clocks are assumed to be free-running and therefore always available. When a clock is multiplied by an internal PLL or other frequency multiplier or divider prior to being used, the frequency relationship is defined by the **-freqMultiplier** parameter. Normally, the source clock would come from an IC port, and the result of the multiplication would be the clock used, for example, in an IP block.

These commands set the local fail flag to indicate whether the system has determined there is an issue with the named clock or not. PDL does not have a mechanism to escape from a clocked delay if the clock is not available, so when the system determines that the clock is not available, this command will set the local fail flag to "FAIL" and the PDL conditional statements can be used to bypass the part of the procedure requiring that clock, or use the **iSetFail** command to abort the procedure.

Syntax

<iClock_cmd> ::= **iClock** <clock_name> [**-period** <seconds>] [<clock_state>] <ct>

<iClockOverride_cmd> ::= **iClockOverride** <int_clock> **-source** <clock_name>
-freqMultiplier <rvalue> [<clock_state>] <ct>

<clock_name>::= <src_clock> | <int_clock>
<src_clock> ::= <port ID>
<int_clock> ::= <PDL_identifier>
<clock_state> ::= **-on** | **-off**

Rules

a) If the fully qualified name formed by concatenation of the context instance path passed to the current procedure, any current <partial path> established by an **iPrefix** command, and the <clock_name> value of this command is a clock name defined by the **SYSCLOCK_REQUIREMENTS** BSDL attribute, <clock_name> shall be a <src_clock>.

b) If <clock name> is a <src_clock>, and the system determines that the specified clock is not available, the local fail flag shall be set to "FAIL" by the **iClock** and **iClockOverride** commands; in all other cases, the local fail flag shall be set to "PASS" by these commands.

NOTE 1—This standard does not formally define how the system would determine that the specified clock signal was not available, but some ways would include that the clock pins were unconnected in a BSDL pin map or tied off in the board netlist, or that the test system could not provide the required clocks. Setting the local fail flag allows use of **ifTrue** and **ifFalse** commands to bypass code dependent on the unavailable clock signal.

c) If the fully qualified name formed by concatenation of the context instance path passed to the current procedure, any current <partial path> established by an **iPrefix** command, and the <clock_name> value of this command is not a name defined by the **SYSCLOCK_REQUIREMENTS** BSDL attribute, <clock_name> shall be an <int_clock>.

d) An <int_clock> shall be an arbitrary name for a clock that is only valid within the current procedure.

NOTE 2—The clock name may be passed down through lower levels of PDL as an argument on iCall commands. Where it is used this way, and does not match a port name defined by the **SYSCLOCK_REQUIREMENTS** BSDL attribute, it will be treated as a local name within each PDL procedure.

e) In any clock distribution structure, possibly across multiple procedures, built of one or more **iClock** and **iClockOverride** commands, at least one **iClock** command shall have the **-period** parameter, and instances of the **-period** parameter in such a structure shall be compatible with each other, and shall be compatible with the frequency range specified in the **SYSCLOCK_REQUIREMENTS** BSDL attribute, if appropriate.

f) The <clock_state> shall default to **-on,** and the <clock_state> set to **-off** shall document the desire that the clock be shut off at its source.

g) For the **iClockOverride** command, the period of the <int_clock> shall be equal to the period of the <clock_name> divided by the **freqMultiplier** value.

Examples

```
; #(500 MHz source clock)
iClock MySclk -period 2.0e-09 ; # minimum period: 2.0 ns
```

Below are PDL snippets for an IP (MEMB) and the component using it (CHIPA). The PDL for the IP is supplied by the IP vendor, and it includes an **iRunLoop** command for the memory BIST. Both the sys_clock (the component port) and the int_clock (used in the IP) are defined in the component level, in this case.

```
 # MEMB IP PDL from vendor AbCD
...
iProc membist {clk_name} {
  ...
  iApply

  iRunLoop 1000000 -sck $clk_name
  ...
}

-----------------------------------
```

```
# CHIPA PDL from vendor XYZinc
#   Uses two instances of MEMB: mem1 and mem2
#   Clock names have to be passed down as arguments because they are
#   not visible in lower level PDL otherwise.
...
iProc main { } {
  ...
  iClock F125MHz -period 8.0e-9 ; # F125MHz is a CLOCK port on CHIPA
  # mem2 is an instance of memb
  iCall mem2.membist F125MHz ; # call memb.membist for instance mem2
  ...
  iClockOverride memclk -source F125MHz -FreqMultiplier 2.0
  # mem1 is an instance of memb
  iCall mem1.membist memclk ; # call memb.membist for instance mem1
  ...
}
```

C.3.7 Test execution commands

C.3.7.1 iRead and iWrite commands

The purpose of the **iRead** command is to define data to be compared with the data shifted out of the named register instance during a subsequent **iApply** command. The purpose of the **iWrite** command is to define data to be shifted into the named register instance during a subsequent **iApply** command. These two commands manage the data to be shifted.

Multiple **iRead** and **iWrite** commands that access the same TDR can be entered prior to an **iApply** command. Each **iRead** and **iWrite** command modifies the current scan frame data. Therefore, multiple references to the same register can overwrite the values set by previous commands on a bit-by-bit basis. No data are shifted into or out of a register unless a subsequent **iApply** command is executed.

Syntax

<iRead_cmd> ::= **iRead** <register_inst> [<expect_value>] <ct>

<iWrite_cmd> ::= **iWrite** <register_inst> <write_value> <ct>

<expect_value> ::= <valueX>
<valueX> ::= <bin_numX> | <hex_numX> | <dec_num1> | <mnemonic identifier>
<write_value> ::= <value1> | **-reset** | **-default** | **-safe**
<value1> ::= <bin_num1> | <hex_num1> | <dec_num1> | <mnemonic identifier>

NOTE 1—-<mnemonic identifier> is defined in BSDL (see B.8.18).

Rules

a) The **iRead** command shall modify the expect data for the specified register or register field instance, and if a register field is modified by multiple **iRead** commands before an **iApply** command, the result shall be that of the last **iRead** command on a bit-by-bit basis.
b) The **iWrite** command shall modify the write data for the specified register or register field instance, and if an instance is modified by multiple **iWrite** commands before an **iApply** command, the result shall be that of the last **iWrite** command on a bit-by-bit basis.

c) The expect data shall be initialized on a bit-by-bit basis as follows prior to the first **iRead** command and after each **iApply** command, and for only the specified register field upon execution of an **iRead** command with no <valueX> specified:

1) To any capture value mandated by this standard.
2) Otherwise, to any "CAPTURES" value specified on a register access or register field definition in either the BSDL or the Package associated with the object (see <value assignment>, B.8.20.1).
3) Otherwise, to the "don't-care" value of X.

NOTE 2—Since **iApply** is only required to compare the **bits** specified by the cumulative **iRead** commands since the last **iApply**, including the implied **iRead** of all prespecified capture values, the expect data do not need to be preserved after an **iApply**. They may be set back to the initialized values.

d) Before the first **iApply** command for a specific TDR, any undefined write data shall be initialized on a bit-by-bit basis as follows:

1) For the boundary-scan register, to the "safe" value specified for each cell in the BOUNDARY_REGISTER attribute.
2) Otherwise, to the logic value of 0.

NOTE 3—Only expect data are allowed to contain X or x values. X or x would be an error in the write, capture, or fail data at the time of or after an **iApply** command.

e) The <register_inst> shall not resolve to a zero-length field.

f) If the **-reset** parameter is specified on an **iWrite** command instead of on a specific value, the write data for that <register_inst> shall be set to:

1) The <safe bit> element of each <cell entry> (see B.8.14.3) with a <function> of **ControlR**.
2) The **RESETVAL** value, if specified, for each field in the <register inst> other than a <cell entry> (see <value assignment> B.8.20.1).

g) If the **-default**, or **-safe** parameter is specified on an **iWrite** command instead of on a specific value, the write data for that instance shall be set to:

1) The <safe bit> element of each <cell entry>.
2) The **DEFAULT**, or **SAFE** value, respectively, and if specified, for each field in the <register inst> other than a <cell entry> (see <value assignment> B.8.20.1).

h) If the binary equivalent of the <write_value> contains an X in any bit position, then that bit position in the data queue shall be unchanged.

i) An **iRead** command followed by the **DEVICE_ID** register name without an expected value shall set the expected value to the first specified value of the attribute **IDCODE_REGISTER**.

j) The fully qualified name formed by concatenation of the context instance path passed to the current procedure, any current <partial_path> established by an **iPrefix** command, and the <register_inst> value of this command shall be the name of a register or register field defined in the current instance hierarchy.

k) Any underscore, <whitespace>, or <newline> characters embedded in a <value1>or <valueX> shall be stripped; that is, the underscore, <whitespace>, and <newline> characters do not count as a character in the numeric value.

l) The most significant 1 bit in the binary equivalent value of <value1> or <valueX> shall be located within the length specified for the <register_inst>.

NOTE 4—This allows a hexadecimal value to be assigned to a register field with a length that is not an exact multiple of 4, as long as the excess most significant bits are 0.

m) If the binary equivalent value of <value1> or <valueX> has fewer bits than the specified number of bits of the register, the bits shall be right justified (closest to TDO) in the register and the remaining high-order bits shall be set to 0 for an **iWrite** command and to X for an **iRead** command.

n) The <mnemonic identifier> value shall not evaluate to **others**.

Examples

```
iRead Bypass      ; # check for default bypass register of zero
iApply
iRead device_id   ; # check the first device_id in the BSDL
iApply
```

Examples with internal registers are shown below. All registers are defined as length 10.

```
iRead myReg1 0b0101011010     ; # set expect data
iApply                        ; # compare all 10 bits

iRead myReg1(7) 0b0           ; # set new expect data
iApply                        ; # compare myReg1 to xx0xxxxxx

iRead myreg3                  ; # expect default data
iApply                        ; # compare all 10 bits

iRead myreg1 0b010X0x01   ; # X-filled, compare to 0bXX010x0x01
iRead myinst.myreg2 0b11  ; # X-filled, compare to 0bXXXXXXXX11

# Next is OK because highest order binary '1' is in reg length
iRead myreg3 0x1xd        ; # Hex value, compare to 0b01xxxx1101
iApply

iRead myreg3              ; # default expect data
iApply

#  Next statement is an Error: highest order '1' bit outside reg length
iRead myreg3 0xfxd        ; # 0b1111xxxx1101 exceeds length of reg
iApply

iSetInstruction PRELOAD
iWrite boundary             ; # preload with "safe" values or '0'
iApply
```

The boundary register has multiple instructions that access the register. The **iSetInstruction** command instructs the tool to load the *PRELOAD* instruction into the .instruction register prior to setting the boundary register to its default "safe" value.

All registers in the following example are defined as length 10:

```
iWrite myreg1 0b1111111101    ; # value length equals register
iWrite myreg2 0b11            ; # same as 0b0000000011
iWrite inst1.myreg3 0xffd     ; # Invalid, hi-order bit outside register
```

C.3.7.2 iApply command

The purpose of the **iApply** command is to perform a scan of the TDR referenced by **iWrite**, **iRead**, or **iScan** commands since the last **iApply** command. All **iWrite**, **iRead**, and **iScan** commands between **iApply** commands must reference register fields that are part of the same TDR. To perform the scan of the referenced TDR, the **iApply** command may have to change the active instruction and perform additional scans of TDRs to include or select segments that contain referenced fields.

Note that it is perfectly acceptable to reference two selectable or excludable segments that are mutually exclusive. The **iApply** command then has to select and scan each segment, one at a time, and the order in which they will be

scanned is not defined. If there are dependencies (data written to one segment affect the captured data in another segment) between the selectable or excludable segments and other segments in the TDR, the results may differ depending on the exact order in which they were scanned. The PDL writer must exercise care in such situations to make sure that the code is completely deterministic.

When any TDR is scanned, initialized expect data, as modified by **iRead** or **iScan** commands for the referenced TDR, is compared with the returned value, and accumulating and local fail flags, as appropriate, are set to "FAIL" if there is a miscompare. Simultaneously, the accumulated write data are written to the TDR. Note that the write data or expect data for a referenced register need not have changed; just that an **iWrite**, **iRead**, or **iScan** command referenced that register. If no **iRead**, **iWrite**, or **iScan** commands were specified since the previous **iApply,** then the same TDR, with the current configuration of excludable and selectable segments, is scanned with the accumulated write data and with initialized and unmodified expect data bits for the current configuration.

If the TDR to be scanned is not selected for scan by the currently active instruction, then the **iApply** command performs an instruction register (IR) scan to make the TDR the selected TDR. If a field referenced in an **iRead**, **iWrite**, or **iScan** command is currently excluded or not selected, the **iApply** command generates the necessary scans (possibly including changing the active instruction) to include or select the needed segments. For every TDR scan, the **iApply** command also checks any register constraints (see B.8.22) that apply before scanning the register, and aborts the scan and the test if the constraint evaluates to true and has **error** severity. Finally, a data scan operation for the referenced register is performed.

The label associated with an **iApply** command is arbitrary and only for reference.

The iApply can terminate in any TAP controller state desired, although that would normally be either the *Run-Test/Idle* or *Pause-DR* states. The behavior of the **iApply** command can be altered in several ways:

- The **-nofail** parameter changes what expected data are used for comparisons and how the two fail flags are set.
- The **-skipRTI** parameter supports instructions that perform specific actions in the *Run-Test/Idle* TAP controller state by not entering that state while loading the instruction and test data registers.
- The **-shiftPause** parameter pauses at the end of the *Shift-DR* TAP controller state and causes the next **iApply** command to resume shifting without intervening update or capture operations.

The **-nofail** parameter alters the behavior of the **iApply** command by limiting the comparisons to only the expect data queued by any **iRead** or **iScan** commands since the previous **iApply** command, and limiting the response to any miscompares to setting the local fail flag. This is to allow this **iApply** command to set the condition for an **iUntil**, **ifTrue**, or **ifFalse** command to interrogate without triggering the normal response to miscompares.

The **iApply** command manages two fail flags in PDL. First, an accumulating fail flag indicating overall PASS/FAIL status, which is initialized to PASS prior to the start of PDL execution and set to FAIL by miscompares between the data being scanned out and the expect data. This flag is not affected by comparisons in an **iApply -nofail** command. Note that the **iSetFail** and **iUntil** commands may also set the accumulating fail flag to FAIL. Second, a local fail flag that is used for the **iUntil**, **ifTrue**, and **ifFalse** commands that is set to PASS prior to each **iApply** command and set to FAIL only if the expect data supplied by **iRead** or **iScan** commands since the previous **iApply** command miscompares.

Beyond setting the accumulating fail flag, the handling of any miscompares of **iRead** or **iScan** data to the data actually scanned out of the unit under test is not specified in this standard. That said, it should be noted that the procedure can be written using a conditional command to force termination of the test process if required. For instance, an **iRead** of the initialization data register may check that critical initialization input pins are set correctly, and testing might have to be terminated if that comparison fails.

The **-skipRTI** parameter forces the TAP controller to not go to or through the *Run-Test/Idle* state during the execution of this instance of the **iApply** command. This is primarily used to help avoid initiating a test that starts in the *Run-Test/Idle* TAP controller state of the currently active instruction. The TAP controller state machine may

then be directed to the *Run-Test/Idle* state with the **iTMSidle** command when required to start the test. Note that this parameter does not imply or assume any particular default path through the state machine, only that whatever the normal transition path is, it will be modified (if necessary) to skip the *Run-Test/Idle* state for all scans needed to complete the command. At the end of the **iApply -skipRTI** command, the *Pause-DR* TAP controller state is used to wait for the next command.

Every **iApply** command causes the TDR to be shifted by the total length of the TDR. The **-shiftPause** parameter on one or more **iApply** commands allows the TDR to be shifted by some multiple of the total length of the TDR, once per **iApply** command. On the first **iApply -shiftPause** command, the TDR performs a capture and shift, subsequent **iApply -shiftPause** commands only perform the shift, and the first **iApply** command without the **-shiftPause** parameter will perform a shift and update. In effect, the **-shiftPause** parameter concatenates the shifts of the **iApply** commands with the parameter to the shift of the following **iApply** command. This mode is common in many programmable components.

An example "pseudo-code" illustrating the flow of the **iApply** command is shown in Annex E.

Syntax

<iApply_cmd> ::= **iApply** { <iApply_parm> } [<label>] <ct>

<iApply_parm> ::= **-nofail** | **-skipRTI** | **-shiftPause**
<label> ::= <PDL_identifier>

Rules

a) If the register fields referenced in **iWrite**, **iRead**, and **iScan** commands since the last **iApply** command are in excludable segment(s) that are currently excluded, or selectable segment(s) that are not currently selected, the **iApply** command shall perform any additional data register scans needed to include or select the segment before performing the data register scan of the referenced fields; and furthermore, if there are multiple such register fields to be scanned that cannot be included or selected at the same time, then they will be included or selected in sequence and the order in which each will be included or selected and then scanned is undefined.

b) If the register field(s) to be scanned is in a TDR that is not selected for scan by the currently active instruction, before performing the data register scan, the **iApply** command shall perform an instruction register scan to make active an instruction that does select the required TDR, using the instruction defined in the **iSetInstruction** defaults or explicitly specified by an **iSetInstruction** command when more than one instruction could be used.

NOTE 1—This applies to data register scans needed to include or select register segments as well as the referenced register fields.

c) If there were no **iWrite**, **iRead**, and **iScan** commands since the last **iApply** command, then the TDR selected by the currently active instruction in its current configuration of excludable and selectable segments shall be scanned using the accumulated write data and the default expect data.

d) For any data register scan, the **iApply** command shall perform the following steps:
   1) Clear the local fail flag prior to performing the first scan of the referenced TDR.
   2) Shift the write data accumulated for the current configuration of the referenced TDR to the scan-in of the TDR.
   3) Compare the data from the scan-out of the referenced TDR with only the expect data accumulated for the current configuration of the TDR since the last **iApply**, setting the local fail flag appropriately.
   4) If the **-nofail** parameter is not specified, compare the scan-out of the referenced TDR with the default expect data for the current configuration of the TDR as modified by the expect data accumulated since the last **iApply**, setting the accumulating fail flag appropriately.

5) If the procedure is Level-1 PDL, capture and maintain the scan-out and bit-by-bit failure values of the current configuration of the referenced TDR, replacing any previously captured and fail values for the same register segments.

e) All **iWrite**, **iRead**, and **iScan** commands since the last **iApply** command, if any, shall reference register fields in only one TDR.

f) If the TDR to be scanned is within the domain of any register constraints (see B.8.22; including any constraints specified for the entity or package), the **iApply** command shall evaluate those constraints, and if any such constraint with a <constraint severity> of **error** evaluates to TRUE, the scan shall not be performed and the test shall be stopped.

NOTE 2—If Level-0 PDL is compiled, constraint checking only needs to be done at compile time. Constraints defined for a domain that does not include the TDR being scanned need not be evaluated. The accumulating and local fail flags are not affected by the result of constraint checking. The meanings and actions to be taken for <constraint severity> of **warning** or **info** are not defined by this standard.

g) The <iApply_parm> parameters shall appear either zero or one times each.

h) If the **-skipRTI** parameter is specified, the **iApply** command shall not pass through the *Run-Test/Idle* TAP controller state between any of the scans required to complete the command and shall end in the *Pause-DR* TAP controller state after the last data register scan.

i) After the execution of an **iApply** command, the accumulated write data defined by all preceding **iWrite** and **iScan** commands shall be retained with the exception of a register field defined as assignment type **PULSE0** or **PULSE1**; in which case, the field shall be reset to 0.

NOTE 3—After execution of an **iApply** command, rule c) of C.3.7.1 applies to expect data.

j) In a Level-1 PDL procedure, any time the **iApply** command is executed, the capture data shall be updated with data from TDO for the scanned TDR, and sufficient data shall be retained to provide a map of each register bit that miscompares.

k) When the **-shiftPause** parameter is specified, an **iApply** command shall, upon completion of shifting the TDR, pause the shift by traversing the *Shift-DR* > *Exit1-DR* > *Pause-DR* state sequence and end in the *Pause-DR* state.

l) When shift is currently paused due to a prior **iApply** command with the **-shiftPause** parameter, an **iApply** command shall resume shifting by traversing the *Pause-DR* > *Exit2-DR* > *Shift-DR* state sequence.

m) All **iRead**, **iWrite**, **iScan**, and **iSetInstruction** commands defining data for a sequence of **iApply** commands from the first **iApply** command with the **-shiftPause** parameter through the first **iApply** command without the **-shiftPause** parameter shall reference the same TDR using the same instruction.

n) The last **iApply** command of a procedure shall not have the **-shiftPause** parameter.

o) The effect of an **iApply** command shall be the same as if the commands following the **iApply** command are not executed until the **iApply** command has completed.

p) If a TDR can be selected for scanning by more than one instruction, the default instruction or the instruction specified for this register using the **iSetInstruction** command shall be used by the **iApply** command.

Recommendations

q) When excludable register segments are controlled from a different TDR than the one containing the excludable segment, they should be explicitly opened before being scanned by an **iApply** command.

Examples

```
# myreg [12] ... CAPTURES 0bxxxx_xxxx_xx01 <= Error bits.
iWrite myreg  0xf0d    ; # same as 0b111100001101
iApply                 ; # apply write data
```

```
# Rescan to ensure that error bits of default capture value have not changed.
#   Because field is not excludable/selectable, no iRead required.
iApply                ; # repeat last scan.

# write to specific device instances
iWrite U1.mybscell 0x1
iWrite U1.myinternalreg 0x1
iApply

iWrite myreg 0x1
iApply FirstReg ;   # tag this iApply with the label FirstReg

# Init_status register is two bits ("Pass" and "Done")
iRead init_status 0x11
iApply   ;
```

To verify a TDR (perhaps after inclusion of an excluded segment), the following Level-0 PDL could be used:

```
...
iWrite myTDR 0b10011 ;  # pattern is right justified, '0' filled
iApply -shiftPause ;    # No update
iRead  myTDR 0b10011
iWrite myTDR -safe ;    # Could write any desired data here.
iApply ;    # No capture, Error indicates the register is not correct.
...
```

C.3.7.3 iScan command

The purpose of the **iScan** command is to define write and expect data for a "private" or "black box" register or register segment in an IC or IP block. A "private" register could be defined in the BSDL **REGISTER_ACCESS** attribute, including an initial length, but without any additional structural information in a BSDL **REGISTER_FIELDS** or **REGISTER_ASSEMBLY** attribute. A "private" register segment is defined in a BSDL **REGISTER_FIELDS** or **REGISTER_ASSEMBLY** attribute as a single contiguous field without any detail. Note that the **iScan** command cannot be used with "private" instructions unless the registers are documented in the **REGISTER_ACCESS** attribute.

A test procedure using an **iScan** command must take full control of the "private" register or register segment, including any possible excludable segments. The **iScan** command supplies the length and both write and expect data for the "private" register segment, which will be applied by the next **iApply** command. As the length can change after each **iApply**, the length and write data may not be correct for use in a later **iApply** command. Whenever a "private" register or register segment is used that is simply undocumented, but does not change length or violate any of the rules of this standard, **iWrite** and **iRead** are preferable to **iScan**.

The **iScan** command can be intermixed with **iWrite** and **iRead** commands prior to an **iApply** command as long as all such commands reference fields in the same TDR. However, once the **iScan** command is used for a register or register segment, it must be used prior to every subsequent **iApply** command accessing that TDR until the end of the procedure or a reset is issued to restore the initial length. This is required since the write data and even the length of the "private" register or register segment are unknown from one **iApply** to the next. It is required that the final **iScan** of a procedure provide the correct current length and write data for the register or register segment so that other procedures may scan that TDR without having to use **iScan** commands.

A length value defines the length of both the input and output scan data. All four (4) values must be defined for every **iScan** command.

Syntax

<iScan_cmd> ::= **iScan** <register_inst> <length> **-si** <scan_data_in> **-so** <scan_data_out> <ct>

<length> ::= <dec_num1>
<scan_data_in> ::= <bin_num1> | <hex_num1>
<scan_data_out> ::= <bin_numX> | <hex_numX>

Rules

a) The most significant 1 bit in the binary equivalent value of <scan_data_in> and <scan_data_out> shall be located within the <length> value, and if the length of the binary equivalent value is less than the <length> value, PDL shall right-justify the value (closest to TDO) and pad the value with zeros for <scan_data_in> and with X for <scan_data_out>.

NOTE 1—This allows a HEX value to be assigned to a register field with a length that is not an exact multiple of 4, as long as the excess most significant bits are 0.

b) Any underscore, <whitespace>, or <newline> characters embedded in <scan_data_in> and <scan_data_out> shall be stripped; that is, the underscore, <whitespace>, and <newline> characters do not count as characters in the scan data.

c) The **iScan** command shall specify the length and modify both the expect data and the write data for the specified register or register field instance, and if a register field is modified by multiple **iScan**, **iRead**, and **iWrite** commands before an **iApply** command, the <length> specified by all such **iScan** commands shall be the same and the resulting write and expect data shall be that of the last command on a bit-by-bit basis.

d) The fully qualified name formed by concatenation of the context instance path passed to the current procedure, any current <prefix path> established by an **iPrefix** command, and the <register_inst> value of this command shall be the name of a register or register field defined in the current instance hierarchy.

e) Within an **iProc**, the final **iScan** command for a particular <register inst> shall provide the correct current length and write data for that <register inst> to allow subsequent scanning of the TDR.

NOTE 2—This allows for "private" registers or register fields with variable length or variable configurations. If the register or register field changes length or configuration, then a final **iScan** with the correct length value and write data must be performed so that the correct information is known to PDL. Within an **iProc**, prior to that final **iScan** command for a particular <register inst>, any **iApply** command scanning the TDR must have an **iScan** command first to set the length and write data if the length or configuration of the register or register segment does not match the previous **iScan** command.

Examples

```
# myreg1, 2, and 3 are fields in a single TDR.
iScan myreg1 8 -si 0b01010101 -so 0b11xx00xx
iScan myreg2 8 -si 0b11 -so 0b11    ; # scan-in zero-filled to 0b00000011
                                      # scan-out 'X'-filled to 0bxxxxxx11
iScan myreg3 6 -si 0xfd -so 0xxx    ; # Error: length mismatch
iApply
```

C.3.8 Flow-control commands

C.3.8.1 iCall command

The **iCall** command passes control to (calls) another PDL procedure, suspending the current procedure until the called procedure returns. The procedure name is fully qualified by the instance to which it applies. The instance is used to look up the associated object in order to identify the specific procedure to call, unless the **-direct** parameter

is specified; in which case, the procedure is assumed to be associated directly with the instance. Each **iCall** would concatenate the instance path of the target procedure to the context instance path of the calling procedure and pass that to the called procedure as the context instance path for the called procedure.

Syntax

<iCall_cmd> ::= **iCall** [**-direct**] <proc_string> { <argument_value> } <ct>

<proc_string> ::= [<instance_path> <period>] <proc_name>
<argument_value> ::= <text>

Rules

a) The <instance_path> shall contain only instance names of objects that are instantiated within the object associated with the calling **iProc**.
b) The <proc_name> shall be a PDL procedure associated with the object type or instance selected by the context instance path prepended to the <instance_path>.
c) The <proc_string> shall contain no embedded whitespace or new line characters.
d) The <instance_path> shall be appended to the context instance path of the current (calling) procedure to form the full context instance path for the called procedure.

NOTE—The context instance path passed into the top procedure is the null string.

Examples

	Portion of U1 PDL Code	MYIO.PDL
1	iSource MYIO.PDL ;# this first->	
2	iPDLLevel 0 -version \	iPDLLevel 0 -version \
3	STD_1149_1_2013	STD_1149_1_2013
4	iProcGroup U1	iProcGroup MYIO ; # MYIO Procedures
5		
6	iProc INIT_SETUP {	iProc setup { swingval } {
7	# configure IOs of type MYIO	iWrite swing $swingval
8	iCall IO(1).setup SATA	}
9	iCall IO(1).AC_mode OFF	
10	iCall IO(2).setup SRIO	iProc AC_Mode { onoff } {
11	iCall IO(2).AC_Mode OFF	iWrite ACMODE $onoff
12	iApply	}
13	}	# iProcGroup MYIO cleared upon
14	# iProcGroup cleared upon	# return from iSource
	# return from PDL	<EOF>
	<EOF>	

A board-level PDL might then include the following, in order to use the above:

```
iSource U1.PDL ; iSource U2.PDL ; # source files for each chip on board
  ...
  iCall -direct U1.INIT_SETUP ; # Call init_setup for instance U1
  iCall ...           ;# subsequent iCalls
  iCall U2.INIT_SETUP ; # Call init_setup for instance U2
  ...
-------------------------------------------

# A PDL for chip type ABCIC could be as follows:

iPDLLevel 0 -version STD_1149_1_2013  ; # level-0 PDL only
```

```
iProcGroup ABCIC

iProc init { IO Protocol } {
  iWrite $IO.Channel.Protocol $protocol
}

# IO1 through IO4 are instances a "SERDES" within the chip type ABCIC.
#   See Channel and REGISTER_MNEMONICS for definitions.
iProc init_setup -export { } {
  iCall init IO1 SATA
  iCall init IO2 SRIO
  iCall init IO3 SATA
  iWrite IO4.Channel.Protocol SATA  ; # Same as "iCall init IO4 SATA".
  iApply
}

# EOF

# A subsequent call for the init_setup procedure of U1 of type ABCIC is

  iCall U1.init_setup ; # Lookup instance U1, call ABCIC.init_setup for U1
```

C.3.8.2 iRunLoop command

The purpose of the **iRunLoop** command is to delay any additional register loads or unloads, either by waiting a minimum absolute amount of time or by generating a minimum number of clock cycles. A typical application for the **iRunLoop** command is to generate clocks for a component hardware state machine or built-in self-test (BIST). Of the instructions defined in this standard, PDL procedures in support of *RUNBIST, INIT_RUN, ECIDCODE*, and *INTEST* might require the **iRunLoop** command. If multiple **iRunLoop** commands are specified, additional clock cycles or wait time will be generated. The command simply documents the delay required, but it does not change anything in the unit under test.

If a cycle count is specified but the **-sck** parameter is not, it is assumed that the clock is the test clock (TCK). If a system clock is specified, the clock name must have been specified in a previous **iClock** or **iClockOverride** command.

Since the clock name supplied with the **-sck** parameter may be the input to an IP block, it may be the name for an internal clock net, which may be the result of internal clock multiplication documented with an **iClockOverride** command. See C.3.6.3 for a description of how to determine the appropriate clock count in this situation.

Absolute time can be specified in seconds as a real number, and it may be the preferred way of specifying a delay since the cycle time of TCK is often unknown and system clocks may be difficult to count, which could result in an estimated delay rather than in actual counting. The delay value defines the minimum time or number of cycles of the specified clock to be executed; greater delays or overcounting must be allowed.

When waiting for a fixed time, or counting system clock cycles, the recommendation to stop TCK to the test logic in order to reduce the noise in the system can be documented by using the **-tck_off** parameter. The allowed stop states are defined in the BSDL.

Syntax

<iRunLoop_cmd> ::= **iRunLoop** <delayspec> <ct>

<delayspec> ::= <time_spec> | <cycle_count>
<time_spec> ::= **-time** <seconds> [**-tck_off**]

<cycle_count> ::= <dec_num1> [**-sck** <clock> [**-tck_off**]]
<clock> ::= <src_clock> | <int_clock>

Rules

a) Either <time_spec> or <cycle count> shall be interpreted as a minimum value.
b) If a <cycle count> is specified, but the **-sck** parameter is not, the TCK test clock shall be counted.
c) If the **-time** parameter is specified, the time shall be expressed as a real number of seconds.
d) If the **-sck** parameter is specified, the <clock> shall be a <clock name> of a preceding **iClock** command or an <int_clock> of a preceding **iClockOverride** command in this procedure or a higher calling procedure.

NOTE—The **iClock** or **iClockOverride** command may be at a higher level of the PDL hierarchy; in which case, the <clock> name would be passed in as an argument on the **iProc** command.

e) The effect of the **iRunLoop** command shall be the same as if no further commands in the current procedure are executed until the **iRunLoop** command returns.

Recommendations

f) If the **-tck_off** parameter is specified, the TCK signal to the unit under test should be held at a static state defined in the BSDL **TAP_SCAN_CLOCK** attribute during the execution of the **iRunLoop** command.

Examples

```
iRunLoop 2000  ;        # generate 2,000 TCK clocks
iRunLoop -TIME 100.0e-6 ; # clock (with TCK) for 100 microseconds

-----------------------------------------------------------------
# MEMB IP PDL from vendor AbCD
...
iProc membist {clk_name } {
  ...
  iApply
  iRunLoop 1000000 -sck $clk_name ; # clk_name is defined in calling PDL
  ...
}
-----------------------------------
# CHIPA PDL from vendor XYZinc
#   Uses two instances of MEMB: mem1 and mem2
...
iProc main { } {
  ...
  iClock F125MHz -period 8.0e-9
  # mem2 is an instance of memb
  iCall mem2.membist F125MHz
  ...
  iClockOverride memclk -source F125MHz -FreqMultiplier 2.0
  # mem1 is an instance of memb
  iCall mem1.membist memclk
  ...
}
-----------------------------------------------------------------
```

C.3.8.3 iLoop and iUntil commands

The paired **iLoop** and **iUntil** commands allow the user to specify PDL statements to be repeated until there is either a match or mismatch of expected values. The PDL statements between the **iLoop** and **iUntil** commands are executed at least once. When the **iUntil** command is encountered, PDL either continues past the **iUntil** command or returns to the **iLoop** command based on the local fail flag and the **-match** or **-mismatch** parameter.

An **iApply** command normally sets the accumulating fail flag when expect data mismatch. To prevent that, the **-nofail** parameter is coded on the **iApply** command immediately preceding the **iUntil** command so that the result of the comparison is captured only in the local fail flag. The condition to be tested is specified within the loop by one or more **iRead** or **iScan** commands preceding the final **iApply** command within the loop. Even within the loop, **iRead** or **iScan** commands associated with earlier **iApply** commands do not affect the conditional test.

The **iUntil** may never reach a satisfactory condition to exit the loop. IP or IC vendors supplying PDL may not be able to properly predict the length of time a particular **iApply** is going to take; hence, the timeout values specified by the PDL writer may be too short to allow proper looping or too long and then impact test time. The optional **-maxloop** <maxcnt> limit may be supplied by the PDL writer to provide an escape from the loop.

iLoop and **iUntil** provide a looping construct that is solely reliant on expect data comparisons. “While” and “For” loops provided by Tcl in Level-1 PDL provide access to variables and mathematical expressions of those variables, which are not available in Level-0 PDL.

NOTE—This style of looping has similarities to “matchloops” on IC ATE, but in this context, they are different in that the matching or mismatching is done on an IEEE 1149.1 register value. The pitfalls of edge placement, skew, and timing common for ATE matchloops using IC I/O pins running at speed are not present.

Syntax

<iLoop_cmd> ::= **iLoop** <ct>
<iUntil_cmd> ::= **iUntil** <condition> [**-maxloop** <maxcnt> [<text>]] <ct>
<condition> ::= **-match** | **-mismatch**
<maxcnt> ::= <dec_num1>

Rules

a) The **iLoop** and **iUntil** commands shall always appear as a pair, and always in that order.
b) At least one **iRead** or **iScan** and one **iApply** command shall appear between the **iLoop** and **iUntil**, possibly within a procedure called inside the loop.
c) The following commands shall not appear between an **iLoop** and **iUntil** or within any procedure called between an **iLoop** and **iUntil**:
   1) The conditional commands **iLoop**, **iUntil**, **ifTrue, ifFalse**, and **ifEnd**; that is, conditional commands may not be nested.
   2) The low-level commands **iTMSreset**, **iTRST**, and **iTMSidle**.
d) The following commands shall not appear between an **iLoop** and **iUntil** but may appear within a procedure called between an **iLoop** and **iUntil**: **iSetInstruction**, **iClock**, and **iClockOverride**.
e) When the **iUntil -mismatch** command is executed, control flow shall return to the matching **iLoop** command only if all expect data specified by **iRead** or **iScan** commands before the last **iApply** within the loop did not miscompare with the data scanned out of the TDR (i.e. the local fail flag is PASS).
f) When the **iUntil -match** command is executed, control flow shall return to the matching **iLoop** command if any expect data specified by **iRead** or **iScan** commands before the last **iApply** within the loop did miscompare with the data scanned out of the TDR (i.e. the local fail flag is FAIL).
g) If the **-maxloop** <maxcnt> on an **iUntil** command is exceeded, the accumulating fail flag (retrieved by the **iGetStatus** command) shall be set to FAIL and the <text>, if provided, shall be reported in the diagnostic data for this test failure.

Example A

In this example, a vendor XYZ is supplying a routine to be called to configure the vendor's SERDES IP for *INIT_SETUP*.

```
iProc XYZ_SERDES { } {

iWrite Swing Full_swing
iWrite Protocol XAUI
iWrite WE 0
iWrite RD 1
iApply
iWrite WE 1
iApply
iRunLoop 100  ;# clock the statemachine

iLoop
  iWrite RD 0
  iApply
  iWrite RD 1   ;# Toggle RD on SERDES until RDY goes high
  iApply
  iRead  RDY 1
  iApply -nofail
iUntil -match -maxloop 5 "SERDES initialization timed out"

#; I/O are ready to use
}
```

Example B

```
iProc XYZ_EXIO { } {

# Loop until non-zero voltage appears

iLoop ;# repeat

  iWrite ADDR   VREFADDR
  iWrite WE  0
  iApply
  iWrite WE  1
  iApply

  iRead  VREF-VOLTAGE 0x00  ;#  Loop until VREF is ON
                            ;#  any non-zero value
  iApply -nofail
iUntil -mismatch -maxloop 10  "VREF-VOLTAGE never turned on"

}
```

C.3.8.4 ifTrue, ifFalse and ifEnd commands

ifTrue and **ifFalse** commands allow Level-0 PDL branching based on the local fail flag, which reflects the comparison of captured data with expected data of the last **iApply**. **ifTrue** and **ifFalse** may appear in any order to allow preferred positioning of PDL commands. The **ifEnd** command marks the end of the conditional commands.

ifTrue and **ifFalse** are designed for the IP or IC PDL writer to provide a basic ability to test for an expected value and perform an action based on that conditional result. It provides a basic flow control mechanism for Level-0 PDL that is dependent only on expect data comparisons. It does not need variables, and therefore, it is more contained than the Level-1 PDL or Tcl "if-else" commands.

An **iApply** command normally generates a failure when expect data mismatch. To prevent that, the **-nofail** parameter is coded on the **iApply** command immediately preceding the **ifTrue** or **ifFalse** commands so that the result of the comparison is captured only in the local fail flag without generating a failure. If the last **iApply -nofail** had no miscompares, then the commands between the **ifTrue** and either **ifFalse** or **ifEnd** are executed. Otherwise the commands between **ifFalse** and **ifTrue** or **ifEnd** are executed. The order of the **ifTrue** and **ifFalse** commands is immaterial, and either can be omitted or have an empty set of commands following it.

Syntax

<ifTrue_cmd> ::= **ifTrue** <ct>
<ifFalse_cmd> ::= **ifFalse** <ct>
<ifEnd_cmd> ::= **ifEnd** <ct>

Rules

a) No more than one **ifTrue** command and/or one **ifFalse** command shall be associated with one subsequent **ifEnd** command.
b) **ifTrue**, **ifFalse**, and **ifEnd** shall not be allowed after an **iMerge -begin** and before an **iMerge -end** command.
c) The **iLoop**, **iUntil**, **ifTrue**, **ifFalse**, and **ifEnd** commands shall not appear between any two of **ifTrue**, **ifFalse**, and **ifEnd**.
d) **iApply**, **iWrite**, **iRead**, and **iScan** shall be the only commands allowed between any two of **ifTrue**, **ifFalse**, and **ifEnd**.
e) For the **ifTrue** command, when the local fail flag is PASS, the commands between the **ifTrue** command and either the **ifFalse** or the **ifEnd** command, whichever occurs first, shall be executed.
f) For the **ifFalse** command, when the local fail flag is FAIL, the commands between the **ifFalse** and either the **ifTrue** or the **ifEnd** command, whichever occurs first, shall be executed

Example A

In this example, XYZ company produces a SERDES IO. SERDES IO is compatible with ABC standard 1.0 and 2.0. Two new ABC standards are in discussion. XYZ company likes to keep a small version number as part of the I/O such as to be able to recognize changes from one version of the IP to another.

```
-- Excerpts from BSDL Package Body
attribute REGISTER_MNEMONICS of SerdesH : package is
   "TERM    (Test       (0b110) < Test Mode Termination >, " &
   "          CML       (0b011) < CML Termination >, "&
   "          Dis       (0b001) < Termination current disabled >, "&
   "          rsrvd     (Others) <Reserved - Undefined behavior >), " &
   "ONOFF    (ON        (1) < enable >,  " &
   "          OFF       (0) < off >), " &
   "CMMV     (Norm_cm         (1) < Normal mission mode >,  " &
   "          Test_cm         (0) < Test mode CMMV set to 0V >), " &
   "SWING    (1200mV    (0b11) <Boundary Scan Output Swing, mVdfpp>, " &
   "          1000mV    (0b10), " &
   "          800mV     (0b01), " &
   "          700mV     (0b00))" ;&
```

```
attribute REGISTER_FIELDS of SerdesH : package is
 "init_data[11] ( "&
-- TDI
-- "*" = Value is required but deferred to BSDL level
 "(VERSION [4] IS  (10 DOWNTO 7 )   NOPO  ), "&
 "(UPD     [1] IS  (6)   DEFAULT(ONOFF(OFF) RESETVAL(ONOFF(OFF)) NOPI
PULSE0), "&
 "(TERM2   [3] IS  (3, 5, 4 )     DEFAULT(TERM(*)) NOPI  ), "&
 "(TERM1   [3] IS  (5 DOWNTO 3 )  DEFAULT(TERM(*)) NOPI ), "&
 "(CMMV    [1] IS  (2)            DEFAULT(CMMV(*)) NOPI ), "&
 "(SWING   [2] IS  (1 DOWNTO 0)   DEFAULT(SWING(*)) NOPI )  "&
 "      )";

#Start of PDL for Package.
iProc INIT_SETUP { } {
iRead VERSION 0b01
iApply -nofail

ifTrue      ;# The first version uses TERM1
   iWrite TERM1 Test
ifFalse     ;#  Version 2 the bits are swizzled
   iWrite TERM2 Test ; # set the bits differently on this rev
ifEnd

iWrite SWING 800mv
iWrite CMMV  Test_cm
iWrite UPD      ON
iApply
iWrite UPD     OFF      ;# prevents further updates
iApply
}
```

Example B

In this example, the IC vendor has two different settings based on the IC fuse values.

```
iProc initial { }  {
iRead fuse(0) 0b0

iApply -nofail

ifTrue      ;# Rev 0 has several features not enabled
   iCall init_setup1
ifFalse
   iCall init_setup2
ifEnd
}
```

C.3.9 Optimization commands

C.3.9.1 iMerge command

The **iMerge** command is used to identify which called procedures could be optimized to minimize the number of scans required. Typically, these would be procedures that are independent and operate on different fields of the same TDR; in which case. only a single scan would be required to satisfy **iApply** commands in multiple procedures. This

command is not intended to either require or restrict merging by PDL; it simply documents the PDL provider's belief that this block of procedure calls is a prime candidate for merging.

By default, all procedures are sequential. This standard does not define any merging algorithms, and merging is not required for compliance. It is also compliant to do merging outside the **iMerge** block. The primary requirement for merging is that the end results of merged scans be the same as the results without any merging.

Syntax

<iMerge_cmd> ::= **iMerge** <block_flag> <ct>
<block_flag> ::= <**-begin**> | <**-end**>

Rules

a) Both the **iMerge -begin** and **iMerge -end** commands shall appear if either does, and the **iMerge -begin** command shall appear before an **iMerge -end** command.
b) No other PDL command shall exist between an **iMerge -begin** and **iMerge -end** other than the **iCall**, **iNote**, **iTake**, or **iRelease** commands.
c) If procedures called between an **iMerge -begin** and **iMerge -end** are merged, the state of the write data and the UUT at the **iMerge -end** command shall be identical to the state that would exist if the merging had not occurred and the procedures were executed sequentially in the stated order.

Example

In this example, a single TDR contains defined fields for two cores with PLLs, and one I/O IP block with AC coupling, which can be turned off as shown in Figure C-5.

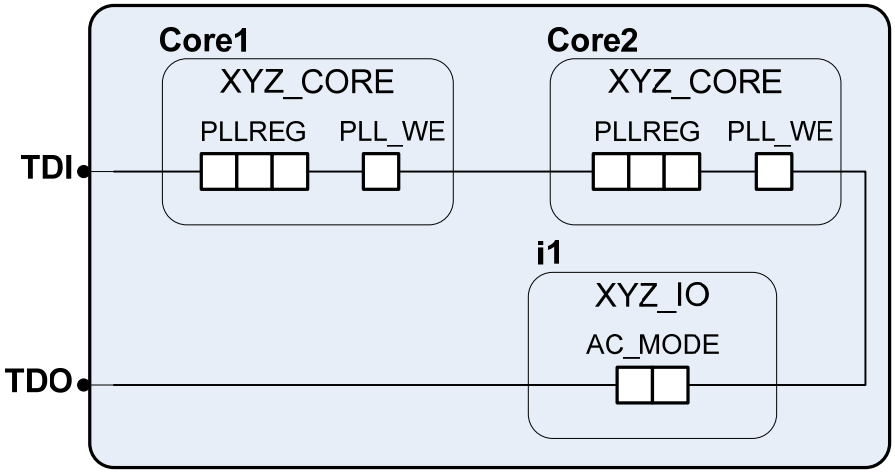


Figure C-5—iMerge example

First, here are the two files from the XYZ IP provider for setting up the PLL and IO objects. The PLL in the XYZ_CORE requires a rising edge on the WE bit to move the scanned data into the PLL control registers, meaning two scans of the PLL_WE register field are required.

```
iPDLLevel 0 -version STD_1149_1_2013  ; # level-0 PDL only
iProcGroup XYZ_CORE

iProc setpll { val } {
# PLL_WE is 1 bit and PLLREG is 1 bit
```

```
  iWrite PLL_WE  0
  iWrite PLLREG $val
  iApply
  iWrite PLL_WE  1
  iApply
}
#end of file


iPDLLevel 0 -version STD_1149_1_2013  ; # level-0 PDL only
iProcGroup XYZ_IO

iProc setACMode { val } {
  iWrite AC_MODE $val
  iApply
}
#end of file
```

Here are some of the top-level test commands calling the above procedures. If there was no optimization, then there would be five scans of the TDR (one for each **iApply** in each called procedure; the example ignores the possibility of needing to deal with IR scans or excluded segments in this example):

```
iSource XYZ_CORE.PDL
iSource XYZ_IO.PDL
...
iMerge -begin
# U1.Core1 is an instance of XYZ_CORE
  iCall U1.Core1.setpll OFF  ;# XYZ_CORE one
# U1.Core2 is an instance of XYZ_CORE
  iCall U1.Core2.setpll OFF  ;# XYZ_CORE two
# U1.i1 is an instance of XYZ_IO
  iCall U1.i1.setACMode OFF  ;# XYZ_IO one
iMerge -end
...
```

Given the procedure calling sequence in the top-level commands, there would be five scans of this TDR. While allowed, it is hardly optimal. Since the procedures are independent of each other, and operate on different instances of the register fields, ideally it would be beneficial to have these optimized for execution. During optimization, the **iMerge** command suggests that the PDL processor merge the three calls by aligning the scans of the TDR in each procedure, which results in two TDR scans as shown in the table.

	U1.Core1.PLLREG	U1.Core1.PLL_WE	U1.Core2.PLLREG	U1.Core1.PLL_WE	U1.i1.AC_MODE
Scan1	**OFF**	**0**	**OFF**	**0**	**OFF**
Scan2	**OFF**	**1**	**OFF**	**1**	**OFF**

C.3.9.2 iTake and iRelease commands

The **iTake** command specifies a resource, which is to be exclusively reserved for use by a single PDL routine. The **iRelease** command relinquishes control of a resource, which had been exclusively reserved by an **iTake** command.

The **iTake** and **iRelease** commands, like the **iMerge** command, simply provide guidance to PDL so that any merging can be optimized without resource conflicts. It would be an error if two procedures requiring the same

resource at the same time were scheduled to execute in parallel. Some resources, particularly a <port ID>, are only available to PDL procedures associated with an IC. If no merging is being performed, they are ignored.

Syntax

<iTake_cmd> ::= **iTake** <resource> <ct>
<iRelease_cmd> ::= **iRelease** <resource> <ct>
<resource> ::= <reg_resource> | <port_resource> | <name_resource>
<reg_resource> ::= **-register** <register_inst>
<port_resource> ::= **-port** <port ID>
<name_resource> ::= **-name** <PDL_identifier>

Rules

a) The **iRelease** command shall follow an **iTake** command for the same <resource>.
b) If the **iTake** command appears within a procedure, an **iRelease** command for the same <resource> shall appear before the end of the procedure.
c) Two procedures shall not be merged in a way that both take the same resource at the same time.
d) For a <port_resource>, the fully qualified name, formed by the concatenation of the context instance path passed to the current procedure, any current <partial path> established by an **iPrefix** command, and the <port ID> value of this command shall match a <port name> defined in the current instance hierarchy.
e) For a <reg_resource>, the fully qualified name, formed by the concatenation of the context instance path passed to the current procedure, any current <partial path> established by an **iPrefix** command, and the <resource> value of this command shall be the name of a register or register field defined in the current instance hierarchy.
f) For a <name_resource>, a given <PDL_identifier> shall be used to mean the same resource wherever used in the set of PDL procedures for a UUT.

NOTE—This <name_resource> is not defined in BSDL.

Example

```
...
iProcGroup PLL

iProc PLLCONFIGURE { val } {
  iWrite PLLEN $val
  iApply
}

...

iProc IOSetup { }  {

  iTake  PLLEN    ; # if this routine is merged, need PLLEN to be ON
                  ; #   until I/O setup can complete
  iWrite PLLEN ON ; # enable PLL such that I/O state machines are clocking
  iApply
  iWrite ACMode OFF
  iApply
  iLoop
    iRead  Rdy
    iApply -nofail
  iUntil -match
```

```
  iRelease PLLEN

  iWrite Swing 400mv ; # no state machine required, direct voltage setting
  iApply             ; # this iApply can be merged
}
#end of file

# top level PDL
...
iMerge -begin        ;#   merging is requested, but the iTake prevents it.
  # i1 is an instance of PLL
  iCall i1.IOSetup
  iCall i1.PLLCONFIGURE OFF
iMerge -end
...
```

Both of the procedures PLLCONFIGURE and IOSetup attempt to set the PLLEN field, and to conflicting values. The **iTake** command in IOSetup reserves the PLLEN field for its exclusive use, and the **iApply** command in PLLCONFIGURE cannot be merged until the final **iApply** in IOSetup, which then allows the fields Swing and PLLEN to be set at the same time.

C.3.10 Miscellaneous commands

C.3.10.1 iNote command

PDL provides two mechanisms for comments. Strings intended for documenting PDL source are delimited by the <hash_mark> character "#" and the <newline> character. **iNote**, with the **-comment** parameter, is a command for passing detailed comments to any output vector format to improve readability and debug of such vectors. For instance, if PDL is being converted into scan-load vectors for an IC tester, it may be possible to embed this comment information in the final vector set. **iNote** with the **-status** parameter is a command for passing status or progress comments to the system.

As always, substitution variables appropriate for the current PDL level can appear in the <text>.

Syntax

<iNote_cmd> ::= **iNote** <purpose> <text> <ct>
<purpose> ::= **-comment** | **-status**

Recommendations

a) The **iNote -comment** command should be used to provide detailed comments in any PDL that may be used to generate test vectors.
b) The **iNote -status** command should be used to denote important milestones in the test flow.

Example A

```
#this comment is not processed, the iNote commands below are.
iNote -comment "Re-setting WE from 1 to 0\n"
iNote -status "Completed writing initialization data to I/O\n"
iWrite WE 0
iApply
# end
```

Example B

```
iProc Write_WE { val } {

iNote -comment "Setting WE = $val\n"
iWrite WE $val
iApply

}
```

C.3.10.2 iSetFail command

Generally, expected results are compared with actual results and a fail flag is set when the two are different. Each time this happens, the tester can simply record enough information to allow later diagnostics or can stop the test. PDL maintains a copy of the accumulating fail flag, and the **iSetFail** command allows that copy of the accumulating fail flag to be set, the optional **-quit** parameter specifies whether the tester should continue or stop, and the optional <text> string can contain information supporting debug and diagnostics. The **iSetFail** command is intended to be used within conditional commands that test for conditions beyond the automatic comparison of captured with expected data that occurs during an **iApply**. The **-quit** parameter is recommended if the failing condition could damage a component.

Syntax

<iSetFail_cmd> ::= **iSetFail** [**-quit**] [<text>] <ct>

Rules

a) The **iSetFail** command shall set the accumulating fail flag of the test to FAIL.
b) If the **-quit** parameter is also provided, the test shall stop.

Recommendations

c) The <text>, if provided, should be treated as diagnostic information for the failure.
d) The **-quit** parameter should be used if the detected failure condition could damage any components in the unit under test.

Examples

```
...
iSetFail -quit {Chip temperature 3 is out of range, quitting.}
...

...
iSetFail "SRIO voltage on $instance incorrectly set to $swing."
#   Quotes permit $parm substitution
...
```

C.3.11 Low-level commands

PDL is, by design, a very TDR-centric language. Details such as which instruction needs to be loaded and TAP controller state sequences are deliberately hidden. There are, however, a few low-level commands provided for special circumstances. These include the reset commands (**iTRST** and **iTMSreset**) and a command to return from any other TAP controller state to the *Run-Test/Idle* (**iTMSidle**). The **-skipRTI** and **-shiftPause** parameters of the **iApply** command also support low-level functions.

These commands are intended to support specific capabilities of this standard not directly supported by the typical **iApply** type scan. They are not a general low-level test capability. For example, resetting the entire unit under test is typically handled by the test environment, and therefore, use of reset commands is generally discouraged within PDL procedures intended for reuse.

In general, none of these commands can be merged for test scan optimization, but they must be in procedures that are run stand-alone.

C.3.11.1 iTMSreset and iTRST commands

The purpose of the **iTMSreset** command is to move the TAP controller to the *Test-Logic-Reset* state by holding the TMS TAP port high for at least five consecutive rising edges of the TCK TAP port. It does not use the TRST* TAP port.

The purpose of the **iTRST** command is to move the TAP controller to the *Test-Logic-Reset* state by holding the TRST* TAP port low. It does not use the TCK and TMS TAP ports.

In either case, write data for all test data register fields with a specified reset type are set to the specified reset value as appropriate for the type of reset defined for the register field (see B.8.19). *Test-Logic-Reset* becomes the starting TAP controller state for the next command.

Note that these commands are intended to be used sparingly and typically in test procedures that exercise the test logic itself and verify that it is functional. In particular, such routines might test the response of the TAP and TMP controllers and the instruction and test data registers to the reset commands.

Normally, both the process of preparing the unit under test (UUT) for test, including any initializing resets, and the process of returning the UUT to nontest operation, including any needed resets, is performed outside of any specific test procedures such as init_setup. Use of either reset command within a procedure also effectively prevents that procedure from being executed in parallel with other procedures. They are, therefore, not allowed in any of the predefined procedures defined in this standard (see **iProc**, C.3.5.4).

Syntax

<iTMSreset_cmd> ::= **iTMSreset** <ct>

<iTRST_cmd> ::= **iTRST** <on_off> <ct>
<on_off> ::= **-on** | **-off**

Rules

a) A reset (moving the TAP controller state to *Test-Logic-Reset*) shall not be performed by any procedure except in response to an explicit **iTMSreset** or **iTRST** command.
b) The **iTRST -on** command shall cause the TRST* TAP port, if provided, to be driven to its active state (logic low).
c) The **iTRST -off** command shall cause the TRST* TAP port, if provided, to be driven to its inactive state (logic high).

NOTE 1—An **iRunLoop** command can be used between the **iTRST -on** and **iTRST -off** commands to provide a minimum of a few TCK clock cycles of active TRST*. The timing of the **iTRST -on** and **iTRST -off** commands, back to back, is undefined and could be very brief.

d) The **iTMSreset** command shall cause the TMS TAP port to be driven to a logic high for a minimum of five consecutive rising edges of the test clock TCK.
e) The **iTMSreset** or **iTRST** commands shall set the appropriate **RESETVAL** values into the write data as specified in the BSDL (see B.8.20) and shall set the currently active instruction as specified in the BSDL.

f) The **iTMSreset** or **iTRST** commands shall not appear in any of the predefined procedures specified in C.3.5.4.

g) The **iTMSreset** or **iTRST** commands shall not appear within a procedure containing the **iMerge** command or within any procedure called from a procedure within the **iMerge** command.

NOTE 2—The TRST*, TMS, and TCK TAP ports may be shared across multiple components in a unit under test, so the reset commands, even if written in the PDL for a single component, may affect multiple components and effectively are equivalent to an **iTake** command for all such components and the resources on those components.

h) All procedures containing the **iTMSreset** or **iTRST** commands shall be defined by an **iProc** command with the **-TMSreset** or **-TRSTreset** parameter, respectively.

NOTE 3—The use of these parameters identifies procedures that include a reset command, enabling tools to identify and limit these procedures to the special purposes for which they are intended.

Recommendations

i) When a register field is no longer needed by a PDL procedure, the register field should be set to an appropriate state using an **iWrite** command possibly with one of the parameters **-reset**, **-safe**, or **-default**.

Example

```
# turn power on
iTRST -On          ; # Assert TRST* low
iRunLoop 1000      ; # wait until power is stable
iTRST -Off         ; # de-assert TRST*

# Reset the test logic
iTMSreset      ; # move to the Test-Logic-Reset TAP controller state
```

C.3.11.2 iTMSidle command

The **iTMSidle** command simply moves the TAP controller state to the *Run-Test/Idle* TAP controller state. This may be used, for example, to start a test that requires the *Run-Test/Idle* TAP controller state (such as *RUNBIST*), or to leave the *Test-Logic-Reset* TAP controller state after one of the reset commands.

Syntax

<iTMSidle_cmd> ::= **iTMSidle** <ct>

Rules

a) If the current TAP controller state is not *Run-Test/Idle*, the **iTMSidle** command shall traverse any allowed sequence of TAP controller states to reach the *Run-Test/Idle* state.

Examples

```
# Reset the test logic
...
iTMSreset      ; # move to the IEEE 1149.1 state of Test-Logic-Reset
iRunLoop 30    ; # Stay in Test-Logic-Reset for thirty TCK cycles
iTMSidle       ; # move to the IEEE 1149.1 state of Run-Test/Idle
...
```

```
# Perform memory BIST in Run-Test/Idle TAP controller state
...
iWrite memBist ... ;     # Load bist instruction and load control register
iApply -skipRTI ;        # Do not go through RTI yet!
iTMSidle ;               # Start the test
...
```

C.4 PDL Level 1 command reference

The Syntax, Rules, Recommendations, and Permissions in this clause are normative and provide the formal definition of Level-1 PDL commands. Introductory text, Notes, and Examples are descriptive.

Level-0 PDL lacks the ability to return data from a register, to use variables, logical expressions, while/for loops, and other constructs that may be necessary to fully describe the operation of some on-chip IP blocks and their related diagnostics. This clause describes the additional commands that are part of Level-1 PDL in this standard. Level-1 PDL includes the commands of Level-0 PDL, Level-1 PDL, and the open-source Tool Command Language (Tcl).

Support of Level-1 PDL is not required for compliance with this standard. PDL intended to describe production test procedures may not be able to support the additional complexity of Tcl, where PDL intended to describe interactive debug procedures would profit from such support.

No single language is perfect for all situations, or preferred by all practitioners, but the choice of Tcl as the base for Level-1 PDL is not arbitrary. The Tcl language is already in extensive use in the engineering community, and Tcl is itself extensible; it is a publicly available language and interpreter. IEEE 1801 specifies new Tcl commands for describing the power intent of a design. Tcl procedures can be interfaced with other languages as needed. As a result, it is an acceptable language for reuse and interchange of procedure documentation.

At the same time, nothing in this standard or in the Tcl language itself would preclude calling PDL from other languages, or translating PDL into another language. As far as this standard is concerned, PDL is the language for documenting and exchanging IEEE 1149.1-compliant procedures in a standard format.

There are many sources of information on Tcl on the Internet as well as in textbooks and articles. There is no formal standard.[9]

There is a data retrieval command and a status retrieval command in the Level-1 PDL commands as shown in Table C-3.

[9] For a summary of current Tcl language syntax, see the following Internet locations: http://wiki.tcl.tk/10259 and http://wiki.tcl.tk/299. For more details on using the Tcl language, see the following Internet location: http://sourceforge.net/projects/tcl/.

Table C-3—PDL Level-1 commands

Command	Parameters	Purpose
iGet	<register> [**-si** | **-so** | **-expect** | **-fail**] [**-hex** | **-bin** | **-dec** | **-mnem**]	Return a Tcl string representing the value associated with a register in the specified format.
iGetStatus	[**-clear**]	Get the PASS/FAIL status since the last time it was cleared.

C.4.1 Level-1 PDL operation

Just like Level-0 PDL, Level-1 PDL maintains data for writing to the register or register field (set via an **iWrite** or **iScan** command) and data that are expected from the register (set via an **iRead**, **iScan**, or BSDL **CAPTURES** specification). In addition, Level-1 PDL maintians TDO data scanned out during the most recent scan (captured by an **iApply** command), and sufficient information to return the bit-by-bit failure information (also captured by an **iApply** command).

Other than this change in the behavior of the **iApply** command, all Level-0 PDL commands behave the same in Level-1 PDL as they do in Level-0 PDL.

C.4.2 iGet command

Comparing expected data to received data in Level-0 PDL has limitations. In many test scenarios, the expected data are not known or a single expected value is not possible. Level-1 PDL includes an **iGet** command, which acts like a function and returns the value of a register or register segment from the values currently maintained by Level-1 PDL procedures. Returning and manipulating values requires variables and a complete language, so this standard specifies Tcl to be used with the PDL commands.

There is a difference between Level-0 PDL and Level-1 PDL that is important to the operation of the **iGet** command. As noted, Level-0 PDL itself is restricted so that it may be compiled (or translated, the two terms are used synonymously here) into test programs suitable for manufacturing (production) test. These test programs usually run on automated test equipment (ATE), sometimes referred to as load-and-go or memory-behind-pins testers. While Level-0 PDL will maintain the write and expect data during compilation of a procedure, the potential output languages typically do not support maintaining such data during execution of the compiled test program. Thus, the **iGet** command cannot rely on being able to access that write or expect data maintained by a compiled Level-0 PDL procedure, even if called from a Level-1 PDL procedure. The **iGet** command is therefore restricted to returning values maintained by Level-1 PDL procedures, or Level-0 PDL procedures that are interpreted as if they were Level-1 PDL procedures. On occasion, in an interactive environment, this may require an additional scan of a TDR already scanned in a compiled Level-0 PDL procedure in order to retrieve data for the **iGet** command to access.

Level-0 PDL procedures can also be called from Level-1 PDL procedures in an interactive mode where they are interpreted as if they are Level-1 PDL procedures instead of using the precompiled version, and the **iGet** command could then access the data maintained by the combination of the Level-0 and Level-1 PDL procedures.

iGet returns a Tcl string representing the current value associated with a register and maintained by a Level-1 PDL procedure. By default, **iGet** returns the data most recently scanned out (the **-so**) of the given register in the hexadecimal (**-hex**) format.

Optionally the write (**-si**), expected (**-expect**), or miscompared bit-by-bit (**-fail**) values may be specified as arguments to return additional information about the register.

The empty string will be returned as an error indication when the mnemonic format is requested but PDL cannot match the register value to a mnemonic identifier, and when expected data containing don’t-care bits (X or x) is

requested in the decimal format. When the hexadecimal format is requested, the character U, which is not an allowed hexadecimal character, will be returned for any hexadecimal character that cannot be represented due to don't-care bits.

A radix may be specified using the keyword **-hex** for hexadecimal, **-bin** for binary, **-dec** for decimal, or **-mnem** for mnemonic. **iGet** is used in conjunction with **iApply**, **iWrite**, **iRead**, and/or **iScan** commands. The **iRead** command followed by an **iApply** command forces the target register to be scanned such that the contents of the register can be examined.

Syntax

<iGet_cmd> ::= **iGet** [<data_source>] [<format>] <register_inst> <ct>
<data_source> ::= **-si** | **-so** | **-expect** | **-fail**
<format> ::= **-hex** | **-bin** | **-dec** | **-mnem**

Rules

a) The **iGet** command shall return a string, for the specified <register inst>, of the appropriate current value (based on the <data_source> parameter) in the format specified (by the <format> parameter), and if either of these parameters is not specified, then the default for <data_source> shall be **-so** and the default for <format> shall be **-hex**.

b) The fully qualified name formed by concatenation of the context instance path passed to the current procedure, any current <partial path> established by an **iPrefix** command, and the <register_inst> value of this command shall be the name of a register or register field defined in the current instance hierarchy.

c) When the **-so** <data_source> parameter is specified or no <data_source> is specified, the **iGet** command shall return the value captured for the specified register during the most recent **iApply** in a Level-1 PDL procedure, or shall return the character X on a bit-by-bit basis if some or all bits of the specified register have not been captured by a previous **iApply** command in a Level-1 PDL procedure or a Level-0 PDL procedure treated as a Level-1 PDL procedure.

NOTE 1—X would not normally appear in captured data. There could be a situation where a segment is excluded, and therefore has never been scanned, while the rest of the register is scanned, and X is substituted for the excluded and never scanned bits.

d) When the **-si** <data_source> parameter is specified, the **iGet** command shall return the current write value of the specified register accumulated in a Level-1 PDL procedure or a Level-0 PDL procedure treated as a Level-1 PDL procedure.

e) When the **-expect** <data_source> parameter is specified, the **iGet** command shall return the current expect value of the specified register accumulated in a Level-1 PDL procedure or a Level-0 PDL procedure treated as a Level-1 PDL procedure.

NOTE 2—Initialization of register fields is defined in C.3.11.1 and in C.3.7.1. After an **iApply**, the expect data reverts to the initialized state as expected values set with **iRead** are cleared after an **iApply**.

f) When the **-fail** <data_source> parameter is specified, the **iGet** command shall return a value with a 0 for each bit of the specified register that matched or where the expected data were X, and return a 1 for each bit that that did not match, for the most recent execution of an **iApply** command in a Level-1 PDL procedure or a Level-0 PDL procedure treated as a Level-1 PDL procedure.

g) When the **-mnem** <format> parameter is specified, the **iGet** command shall return the <mnemonic identifier> with a value matching the current <data_source> value from the <mnemonic definition> associated with the <register_inst> of this command, with the match to be performed as follows:

1) Any X in the mnemonic value shall match any value in the corresponding position of the data value.
2) The highest order 1 bit in the mnemonic value shall fit within the length of the <register inst>.

3) If the mnemonic value length is less than the length of the <register inst>, then the mnemonic value will be right-justified and padded with 0 bits prior to the comparison.

h) When the **-mnem** <format> parameter is specified, a <register inst> that does not have a mnemonic definition for the given field of a register or with a data value that does not match a mnemonic shall return the single character string U.

i) When a **-bin** value is being returned, the first two characters of the string shall be 0b.

j) When a **-hex** value is being returned, the first two characters of the string shall be 0x.

k) When a **-hex** value is being returned, and the register or register field length is not an exact multiple of 4 bits, the extra high-order bits in the hex value shall be set to 0.

l) When a **-hex** value is being returned, and a hex digit contains a mixture of bits that are X and 0 or 1, then the hex digit shall be set to U (Undefinable, an error).

m) When a **-dec** <format> parameter is specified, and the register or register field value contains an X or x character, the value returned shall be the single character string U.

Example A

NOTE—In all of the examples, the **iGet** and **iGetStatus** commands are shown enclosed in square brackets. This is Tcl syntax that forces the evaluation of what is in the square brackets before the rest of the expression, essentially passing the result of the expression inside the square brackets to the rest of the expression.

```
iRead U1.device_id ; # Initialized to first value from BSDL
iApply
```

Assume the device_id register has the following values in the scan frame after the above **iApply**:

Register	-si (iWrite)	-expected (iRead)	-so
device_id	0x55555555	0xX4345601	0x14345601

```
set result [iGet U1.device_id] ; # Default is -so -hex
  puts "Device ID = $result"
```

The output would be Device ID = 0x14345601.

```
set result [iGet -si U1.device_id]
  puts "The value written to DEVICE_ID is $result"
```

The output would be “The value written to DEVICE_ID is 0x55555555.”

Example B

```
iRead U1.device_id
iApply

set result [iGet U1.device_id]

   if {$result == 0x14345601} {
      iCall U1.run_testv1           ;# run the procs that work with rev1
      }
   elseif {$result == 0x04345601} {
      iCall U1.run_testv0           ;# run the older tests which are limited
      }
   else {
```

```
      puts "Found new U1.device_id = $result. No tests were executed"
      }
```

Example C

```
# swing and protocol are register fields associated with mnemonic
#   descriptions having the values shown in the comments.
set val1 [iGet -si -mnem swing]     ; # 500mv, 1000mv 1500mv
set val2 [iGet -si -mnem protocol]  ; # SRIO, SATA

if { $val1 == "500mv" && $val2 == "SRIO" } {
      puts "The I/O cannot be set to 500mv in SRIO mode"
} ; # This test could be done before the iApply to check for correct values.
```

Example D

```
iProc ecid { } {

iWrite ECIDEN 1  ; # ECIDEN is a one-bit enable bit for reading the ECID
iLoop
  iRead  ECIDRDY 1 ; # Status bit
  iApply -nofail
iUntil -match

iRead ECID ; # Not needed if ECIDEN, ECIDRDY and ECID are in the same TDR.
iApply     ; #   The ECID value will be there when exiting the loop.
return [iGet ECID]
}
...
set result [iCall U1.ecid]
puts "ECID = $result"
```

The output would be ECID = 0x10014356, assuming that is the value for the ECID.

Example E

```
# routine to read the strapping input pins and set I/O voltage for
# related I/O to proper voltage for interconnect testing

iProcGroup XYZ_IO

iProc set_IO_voltage { } {
  iRead IO_VSEL_Pins ; # Five bit value tells us the strapping for I/O
  iApply             ; # No expected value, just get it.

  # If pins are not set correctly, smoke can result.
  set result [iGet IO_VSEL_Pins]
  if {$result == 0b100} {          ;# each bit sets a different voltage
    iWrite voltage 1.8V
    iApply
    }
  elseif {$result == 0b010} {
    iWrite voltage 2.5V
    iApply
```

```
    }
  elseif {$result == 0b001} {
    iWrite voltage 3.3V
    iApply
    }
  else {
    iSetFail -quit "Invalid value $result on IO_VSEL pins, terminate test."
}
#end of file
```

Assuming that the above procedure was exported, then the test engineer in his board-level procedure might include the equivalent of:

```
# board level call to set io1
#   U1.IO1 is an instance of XYZ_IO
iCall  U1.IO1.set_IO_voltage
```

Example F

This example shows how an IC or IP PDL provider might add error checking to the procedure. Note that this would be before the **iApply** command that actually scanned the values into the register.

```
...;  # Other iWrite commands for this register.

set val1 [iGet -si -mnem U1.DOMSELA]     ; # DOM A ON
set val2 [iGet -si -mnem U1.DOMSELB]     ; # DOM B ON
if { $val1 == "ON" && $val2 == "ON" } {
      puts "ERROR Domain A cannot be turned on when Domain B is on"
      iWrite U1.DOMSELA OFF
      # return   - turn one of them off and optional return
}

iApply
...
```

Example G

```
iProc ecid {eFuseLoadTime} {

#Pulse System reset
#  (reset_enable, reset_control are fields in the reset_select TDR)
iWrite reset_enable  0
iWrite reset_control 0
iApply
iRunLoop 10  ; # Wait for 10 TCK cycles
iWrite reset_control 1
iApply

# eFuse load starts on de-assertion of reset
iRunLoop -time $eFuseLoadTime
iRead ECID ;  # No expect data, just fetch the register value.
iApply

return [iGet ECID]
}
```

C.4.3 iGetStatus command

The **iGetStatus** command returns a four-character string of PASS or FAIL indicating the accumulated status of the PDL application at the time of the call. The **iGetStatus** command is used to get the current pass/fail status based on any miscompares that have been detected since the last time the status was set to PASS.

The accumulated status is set to PASS at the beginning of the top-level PDL program. It becomes FAIL when an **iApply** produces miscompares based on expected data whether set with **iRead**, with **iScan**, or by initialization rules. The accumulated status may also be set to FAIL by the **iSetFail** command. The **-clear** option of the **iGetStatus** command clears the accumulating fail flag back to PASS after the current value has been returned.

Syntax

<iGetStatus_cmd> ::= **iGetStatus** [**-clear**] <ct>

Rules

a) The **iGetStatus** command shall return the accumulating fail flag as a string value of PASS if there have been no miscompares in any **iApply** command since the status was last cleared, and shall return a string value of FAIL otherwise.
b) If the **-clear** parameter is specified, the accumulating fail flag shall be cleared after determining the value to return.

Example

Consider the example below. RegA and RegB are 3-bit fields in the same TDR and always capture 0b111.

```
iWrite RegA 0b000
iWrite RegB 0b111
iRead RegA 0b100
iRead RegB 0b010
iApply
set result [ iGetStatus ]
if { $result == "FAIL" } { puts "test failed" }
```

iGetStatus returns FAIL because there were miscompares.

C.5 Example BSDL and PDL for the use model

The following are stripped down examples of the different files that would be provided for the Use Model board in Figure C-1 reproduced in Figure C-6 for convenience, and discussed in C.2.1. These files come from multiple sources: the suppliers of the IP, the component suppliers, and the board test engineer.

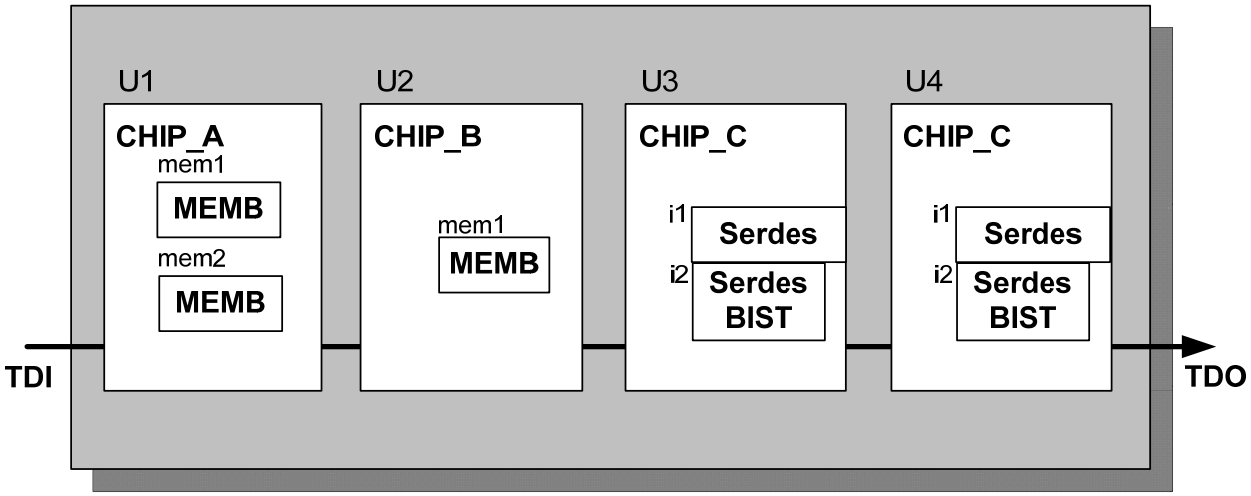


Figure C-6—Example circuit board

C.5.1 BSDL Packages for IP

These files permit the IP supplier to control the definition of the TDR or TDRs in the IP.

MEMB

```
package MEMB is
   use STD_1149_1_2013.all;
end MEMB;

package body MEMB is
   use STD_1149_1_2013.all;
   ...
attribute REGISTER_MNEMONICS of MEMB : entity is
   "Mode                  (chkbrd   (0B000) <Checkerboard>, "&
   "                       Walk1    (0B001) <Walk 1/0 >, "&
   "                       GalPat   (0B010) <GALPAT >, " &
   "                       GALROW   (0B011) <FC on Write recovery>, "&
   "                       GALCOL   (0B100) <FC on Write recovery>, "&
   "                       MATS+    (0B101) < March Algorithm >, "&
   "                       MOVI     (0B110) < Moving Invert >, "&
   "                       March_C- (0B111) <Unlinked CFins >), "&
   "Run                   (Start    (1),  " &
   "                       Stop     (0) ), " &
   "Result                (Pass     (0B11), " &
   "                       Fail     (0B01), " &
   "                       Not_Done (0BX0))";

   attribute REGISTER_FIELDS of MEMB : package is
      "MBist [6]( "&
         "( Algorithm  [3] IS  (5 DOWNTO 3 )  DEFAULT  (Mode  (Walk1)) ), "&
         "( Command    [1] IS  (2)            DEFAULT  (Run   (Stop )) ), "&
         "( Status     [2] IS  (1 DOWNTO 0)   CAPTURES (Result(Pass )) ), "&
         " )";
   ...
end MEMB;
```

SERDES

```
package SERDES is
   use STD_1149_1_2013.all;
end SERDES;

package body SERDES is
   use STD_1149_1_2013.all;
   ...
   attribute REGISTER_MNEMONICS of SERDES : package is
      "SerDes_Protocol  (off   (0b000) <I/Os powered down>, "&
         "               SATA  (0b010) <SATA>, "&
         "               SRIO  (0b011) <Serial RapidIO>, "&
         "               XAUI  (0b100) <XAUI>, "&
         "               Rsvd1 (0b101) <Undefined, do not use>, "&
         "               Rsvd2 (0b11X) <Undefined, do not use>),"&
      "SerDesClkSettings(F125Mhz (0b00111), "&
         "               F100Mhz (0b10101), "&
         "               Invalid (Others) <Do not use!>),"&
      "OnOff            (ON (1), OFF (0))";
   attribute REGISTER_FIELDS of SERDES : package is
      "serdes_init [8] ( "&
         "(Protocol [3] IS (2 DOWNTO 0) DEFAULT(SerDes_Protocol(off)) ), " &
         "(CHClock [5]  IS (7 DOWNTO 3) SAFE(SerDesClkSettings(F125MHz)))" &
      "serdes_bist [4] ( "&
         "(Local_Loopback [1] IS (3) DEFAULT(OnOff(ON))  ), " &
         "(BER_en         [1] IS (2) DEFAULT(OnOff(OFF)) ), " &
         "(GoDone         [1] IS (1) DEFAULT(OnOff(OFF)) ), " &
         "(Pass           [1] IS (0)) )";
   ...
end SERDES;
```

C.5.2 BSDL files for components

These BSDL files must be supplied by the component supplier.

Chip_A

```
entity Chip_A is
   ...
   port (
      ...
      sysclock_100MHz   : in bit;
      ...
      )
   Use std_1149_1_2013.all;
   Use memb.all;
   ...
   attribute INSTRUCTION_OPCODE of Chip_A : entity is
      "RAMBIST    (00000001)," &
      "INIT_SETUP (01000000)," &
      "INIT_RUN   (01000001)," &
      "IC_RESET   (00001111)," &
      ...
   attribute REGISTER_ACCESS of Chip_A : entity is
      "RAMBIST_CTL [12] (RAMBIST)" ;  -- Two MEMB in series
```

```
   ...
   attribute BOUNDARY_REGISTER of Chip_A : entity is
      ...
      " 193 (BC_4, sysclock_100MHz, clock, X, OPEN1)," &
      ...

   attribute SYSCLOCK_REQUIREMENTS of Chip_A is

   "(sysclock_100Mhz, 100e6, 100e6, RAMBIST)";


   -- Build RAMBIST_CTL from MEMB Package and register fields (mem1 & mem2).

   attribute REGISTER_MNEMONICS of Chip_A : entity is

   "RSTCTL ( assert (0), de-assert(1)  ) " ;


   attribute REGISTER_FIELDS of Chip_A : entity IS
   "RESET_SELECT[3] ( "&
     "(Reset_Hold     [1] IS  (0)  DEFAULT  (RSTCTL (de-assert))  ), "&
     "(Reset_Enable   [1] IS  (1)  DEFAULT  (RSTCTL (de-assert))  ), "&
     "(Reset_Control  [1] IS  (2)  DEFAULT  (RSTCTL (de-assert))  )  "&
     "      )";

   ...
   attribute REGISTER_ASSEMBLY of Chip_A : entity IS
     "RAMBIST_CTL ( " &
        "(Mem1 IS MBist ), " &
     "(Mem2 IS MBist ) " &
     " )";
   ...
   attribute Register_Association of Chip_A : entity is
     "Mem1 : sysclock (sysclock_100MHz), " &
     "Mem2 : sysclock (sysclock_100MHz) ";
   ...
end Chip_A ;
```

Chip_B

```
entity Chip_B is
   ...
   port (
      ...
      sclock      : in bit;
      ...
      );

Use std_1149_1_2013.all;
Use memb.all;
   ...
   attribute INSTRUCTION_OPCODE of Chip_B : entity is
      "MEMBIST  (001001)," &
   ...
   attribute REGISTER_ACCESS of Chip_B : entity is
      "MEMBIST_REG [6] (MEMBIST)" &
```

```
   ...
   attribute BOUNDARY_REGISTER of Chip_B : entity is
      ...
      "  73 (BC_4, sclock,    clock, X, OPEN1)," &
      ...
     ;

   attribute SYSCLOCK_REQUIREMENTS of Chip_B is

   "(sclock, 100e6, 120e6, MEMBIST)";


   -- Build MEMBIST_REG from MEMB Package and MBIST fields (mem1).
   ...
   Attribute REGISTER_ASSEMBLY of Chip_B : entity IS
     "MEMBIST_REG ( " &
     "(Mem1 IS MBist ) " &  -- register segment name from MEMB package file
     " )";
   ...
   Attribute Register_Association of Chip_B : entity is
     "Mem1 : sysclock (sclock ) ";
   ...
end Chip_B ;
```

Chip_C

```
entity Chip_C is
   ...
   port (
      ...
      sysclk                : in       bit;
      ...
      );
   Use std_1149_1_2013.all;
   Use serdes.all;

   ...
   attribute INSTRUCTION_OPCODE of Chip_C : entity is
      "SERDES_TEST (11110001)," &
      "INIT_SETUP  (01100000)," &
      "INIT_RUN    (01100001)," &
      ...
   attribute REGISTER_ACCESS of Chip_C : entity is
      "SERDES_CTL [4] (SERDES_TEST)" ;
   ...
   attribute BOUNDARY_REGISTER of Chip_C : entity is
      ...
      " 220 (BC_4, sysclk,    clock, X, OPEN1)," &
      ...
      ;
   ...
attribute SYSCLOCK_REQUIREMENTS of Chip_C is

 "(sysclk, 156.25e6, 156.25e6, SERDES_TEST, INIT_SETUP, INIT_RUN)";

   ...
```

```
-- Build SERDES_CTL from SERDESIO Package and serdes_bist fields.
-- Build init_data from SERDESIO Package and serdes_init field.
   ...
Attribute REGISTER_ASSEMBLY of Chip_C : entity IS
  "INIT_DATA ( " &
    "(i1 IS serdes_init ) " &  -- register segment from SERDES package file
    " )," &
  "SERDES_CTL ( " &
    "(i2 IS SERDES_BIST ) " &
    " )";
   ...
   Attribute Register_Association of Chip_C : entity is
   "i1 : sysclock (sysclk ), " &
   "i2 : sysclock (sysclk ) ";
   ...
end Chip_C ;
```

C.5.3 PDL files supplied by IP supplier

These files permit the IP supplier to control how test features in the IP get used at higher level packages.

MEMB

```
# MEMB.pdl
iPDLLevel 0 -version STD_1149_1_2013
iProcGroup MEMB
iProc memory_bist {alg clk} {

# "(Algorithm  [3] IS  (5 DOWNTO 3 )  DEFAULT  (Mode  (Walk1))  ), "&
# "(Command    [1] IS  (2)            DEFAULT  (Run   (Stop ))  ), "&
# "(Status     [2] IS  (1 DOWNTO 0)   CAPTURES (Result(Pass ))  )  "&

  iWrite Algorithm $alg
  iWrite Command    Start
  iApply
  iRunLoop 10000 -sck $clk
  iRead Status Pass
  iApply
}
```

SERDES

```
# SERDES.pdl
iPDLLevel 0 -version STD_1149_1_2013
iProcGroup SERDES
iProc serdes_bist { Local BER } {

  #      "(Local_Loopback [1] IS (3) DEFAULT(OnOff(ON))  ), " &
  #      "(BER_en         [1] IS (2) DEFAULT(OnOff(OFF)) ), " &
  #      "(GoDone         [1] IS (1) DEFAULT(OnOff(OFF)) ), " &
  #      "(Pass           [1] IS (0)) )";

  iWrite Local_Loopback $Local
  iWrite BER_en         $BER
  iWrite GoDone       ON
  iApply
```

```
  iRunLoop 100000 ;# use TCK cycles

  iRead GoDone   0  ;# 0 for DONE 1 for 'busy'
  iRead Pass     1  ;# 1 for pass 0 for fail
  iApply
}
```

C.5.4 PDL files supplied by component supplier

Chip_A

```
# Chip_A.pdl
iSource memb.pdl
iPDLLevel 0 -version STD_1149_1_2013
iProcGroup CHIP_A

# procedure causes IC level reset via the 1149.1 TAP
# This clears IC after on-chip tests are run
iProc fuctional_reset { } {

#   "(Reset_Hold     [1] IS  (0)  DEFAULT  (RSTCTL (de-assert))  ), "&
#   "(Reset_Enable   [1] IS  (1)  DEFAULT  (RSTCTL (de-assert))  ), "&
#   "(Reset_Control  [1] IS  (2)  DEFAULT  (RSTCTL (de-assert))  )  "&

  iWrite Reset_Enable  assert
  iWrite Reset_Control assert
  iApply

  iRunLoop 10

  iWrite Reset_Control de-assert
  iWrite Reset_Enable  de-assert
  iApply
}

# procedure to hold internal IC reset de-asserted during on-chip tests
# and block any external board level resets while tests are running
iProc protect_reset { } {

  iWrite Reset_Enable  assert
  iWrite Reset_Control de-assert
  iApply
}

iProc main { }  {
  iClock sysclock_100MHz -period 1.0e-8

# block board level resets to IC so tests will execute unimpeded
  iCall protect_reset

  iCall mem1.memory_bist MATS+ sysclock_100Mhz
  iCall mem2.memory_bist MATS+ sysclock_100Mhz

  iCall functional_reset
}
```

Chip_B

```
# Chip_B.pdl
iSource memb.pdl
iPDLLevel 0 -version STD_1149_1_2013
iProcGroup CHIP_B
iProc main { }  {
  iClock sclock -period 8e-9
  iCall mem1.memory_bist MATS+ sclock
}
```

Chip_C

```
# Chip_C.pdl
iSource serdes.pdl
iPDLLevel 0 -version STD_1149_1_2013
iProcGroup CHIP_C
iProc main { }  {
  iCall i2.serdes_bist ON ON
}
```

C.5.5 PDL files coded by test engineer

The test engineer must take “template” initialization PDL files, if provided by the component supplier, and modify it for the specific instances on the board. As with IP, if there are one or more design-specific TDRs and test features in the IC, the IC supplier may also provide PDL procedures for those.

The test engineer might also create a top-level PDL for the unit under test (UUT).

U1

```
# U1.PDL

iSource CHIP_A.pdl
iPDLLevel 0 -version STD_1149_1_2013

iProcGroup U1 ; # Applies to all iProc commands

iProc init_setup { } {
   ...
}

iProc init_run { } {
   ...
   iLoop
     iRead init_status 0b11
     iApply -nofail
   iUntil -match
}
iProc fuctional_reset { } {
   ...
}
```

U2

No additional PDL is needed.

U3

```
# U3.PDL
iSource Chip_C.pdl
iPDLLevel 0 -version STD_1149_1_2013
iProcGroup U3

iProc init_setup { } {
   iWrite U3.i1.PROTOCOL XAUI
   iWrite U3.i1.CHClock F100Mhz
   ...
}
iProc init_run { } {
   iClock Sysclock_100MHz -period 10.0e-9 ; # minimum period: 10.0 ns
   iRead init_status 0b00
   iApply

   iRunLoop 25000000 -sck Sysclock_100MHz ; # 25 million system clocks

   iRead init_status 0b11
   iApply
}
```

U4

```
# U4.PDL
iSource SERDES.pdl
iPDLLevel 0 -version STD_1149_1_2013

iProcGroup U4 ;     # This duplicates same lines in U3.PDL.

iProc init_setup { } {
   iWrite U4.i1.PROTOCOL SRIO
   iWrite U4.i1.CHClock F100Mhz
   ...
}

iProc init_run { } {
   iClock Sysclock_100MHz -period 10.0e-9 ; # minimum period: 10.0 ns
   iRead init_status 0b00
   iApply

   iRunLoop 25000000 -sck Sysclock_100MHz ; # 25 million system clocks

   iRead init_status 0b11
   iApply
}
```

UUT

```
# UUT.PDL   This file could be in some other test language.
iSource U1.pdl
iSource Chip_B.pdl      ;# needed for U2
iSource U3.pdl
iSource U4.pdl
iPDLLevel 0 -version STD_1149_1_2013
```

```
iProc power_up { } {
   iNote –status "Power-on Reset in progress.\n"
   iTRST -ON
   iRunLoop 1000
   iTRST -OFF
}

iProc init_setup { } {
   iNote -status "Initializing for test.\n"
   iCall -direct U1.init_setup
   iWrite U2.bypass 0b0
   iCall -direct U3.init_setup
   iCall -direct U4.init_setup
   iSetInstruction BYPASS
   iApply ;                # Explicit parallel optimization
}

iProc init_run { } {
   iWrite U2.bypass 0b0 ; set instruction register of U2
   iApply

# init_run routines may not be able to be optimized
   iCall -direct U1.init_run
   iCall -direct U3.init_run
   iCall -direct U4.init_run
}

# A board level main could be generated by a test generation tool relying on
# standardized iProc main to contain chip specific test procedures.
# All tests to be run after EXTEST.
iProc main { } {
   iNote –status "Memory test in progress.\n"
   iCall U1.main
   iCall  U2.main
   iNote –status "Serdes test in progress.\n"
   # U3.i1 and U4.i1 are instances of serdes
   iCall U3.main
   iCall U4.main
   iNote –status "Tests Complete, resetting for functional operation.\n"
   iCall U1.functional_reset
}
```

Annex D

(informative)

Integrated examples of BSDL and PDL

The following examples show both the BSDL and the PDL for a single example. The first covers initialization, the second for use of multiple IEEE 1500 wrapper serial ports.

D.1 Initialization example structure and procedures

D.1.1 Initialization example using register description attributes

An example application of the new register documentation attributes appears in several clauses in this annex. For clarity, this material is consolidated, organized, and expanded here to provide a complete example of how the various new attributes are intended to support test data register (TDR) documentation. To illustrate such use, the SerDes documentation is placed in a user package as if it is provided by an IP supplier.

The init_data and init_status registers are newly specified registers in this standard that created the need for detailed TDR documentation. The initialization data register may include various excludable segment controls that need to be set up as part of initialization, and controls needed to set the functional logic in a safe state. The specific values to be set in many of the fields of the init_data register are unknown at the time the BSDL is created, so the value to be written or read is deferred and must be selected by the test engineer based on the specifics of the component usage on the board design. See D.1.2 for the PDL procedures that would support initialization of this component.

Figure D-1 illustrates the hard IP that has been provided by the IP supplier in the example user package body. In addition to a single SERDES port and a single data PLL (data PLL not shown), a full 9-bit byte with parity and an embedded data PLL is also provided. The init_data fields are shown within both the SERDES and the PLL. The boundary cell in the SERDES is also shown, although the IP provider cannot describe the boundary segment because the I/O port names are not known. The IP supplier has manually documented the boundary register pair as **BC_8** and **BC_2** in comments. (The boundary-scan chain is not used in the PLL but is wired through to maintain ease of wiring.) The package documenting the supplied IP follows Figure D-1.

Note that the byte_init_data register segment defined in the package body has a bit length of $10 \times 5 = 50$ bits.

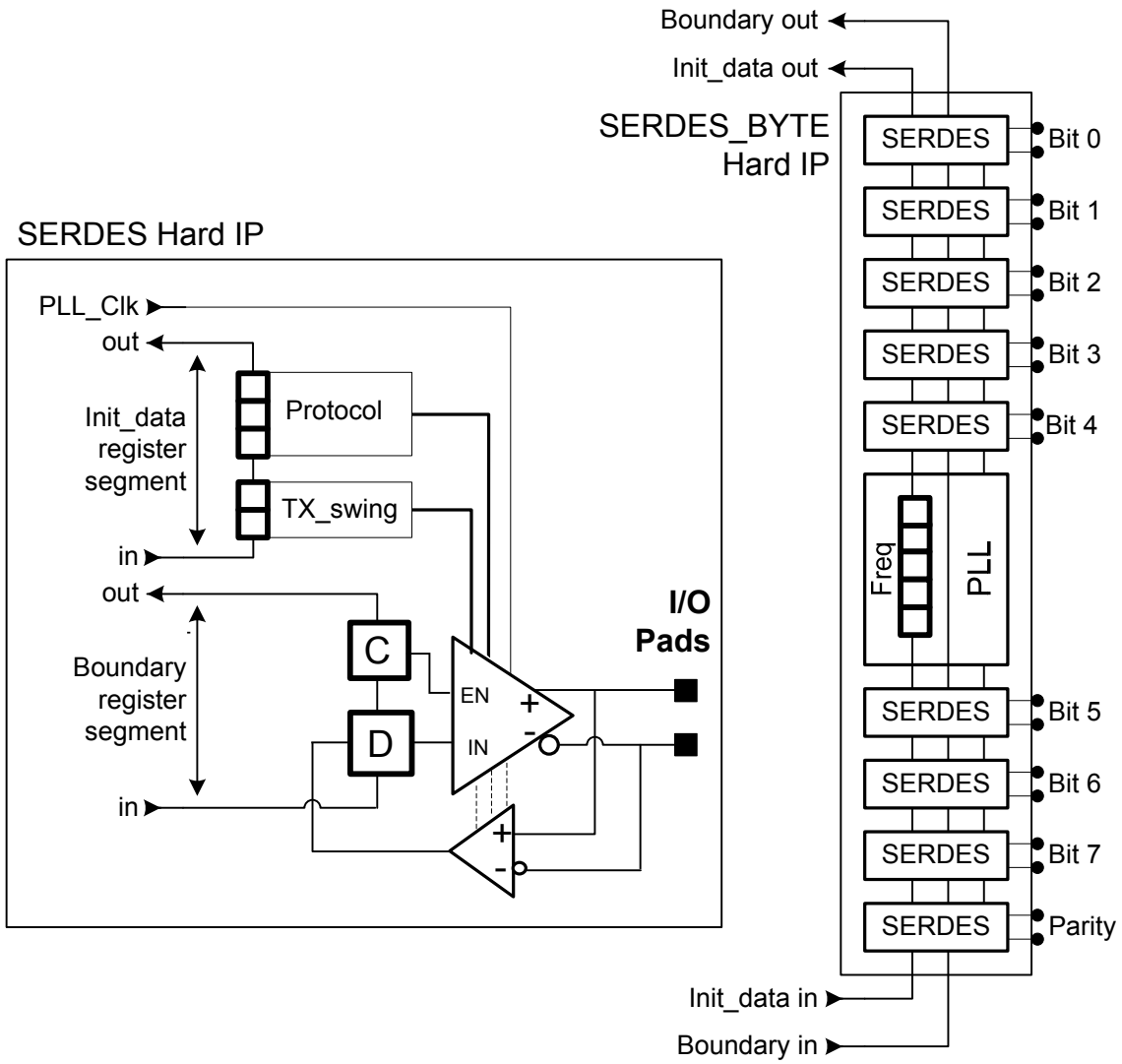


Figure D-1—Hard SerDes IP defined in a package

```
package MyCorp_SERDES_1_2_3 is
   use STD_1149_1_2013.all;
end MyCorp_SERDES_1_2_3;

package body MyCorp_SERDES_1_2_3 is
   use STD_1149_1_2013.all;

   attribute REGISTER_MNEMONICS of MyCorp_SERDES_1_2_3 : package is
      "SerDes_Protocol ( "&
         "   off  (0b000) <Powered down>, "&
         " Resvd0 (0b001) <Reserved for future use>, "&
         "   SATA (0b010) <Serial Advanced Technology Attachment>, "&
         "   SRIO (0b011) <Serial RapidIO>, "&
         "   XAUI (0b101) <10 Gbps Attachment Unit Interface>, "&
         " Resvd1 (0b100) <Reserved for future use>, "&
         " Resvd2 (0b11X) <Reserved for future use> "&
         " ), "&
```

```
      "SerDes_TX_Outputs ( "&       -- Output driver swing level
         "         off (0b00) <Powered down>,  "&
         "  Full_Swing (0b01) <100% Vdd Swing>, "&
         "   75%_Swing (0b10) <75% Vdd Swing>, "&
         " 52.7%_Swing (0b11) <52.7% Vdd Swing - Invalid for XAUI> "&
         " ), "&

      "SerDesClockSettings ( "&     -- Only 2 valid settings
         "  125Mhz (0b00111), "&
         "  100Mhz (0b10101), "&
         " Invalid (Others) <Undefined behavior - Do Not Use> "&
         " )" ;

   attribute REGISTER_FIELDS of MyCorp_SERDES_1_2_3 : package is

      "Channel [5] ( "&
        "(Protocol [3] IS (2,0,1) SAFE (SerDes_Protocol(*))), "&
        "(TX_Swing [2] IS (3,4)   SAFE (SerDes_TX_Outputs(*))) "&
                 " ), "&

      "ChClock [5] ( "&
         "(Freq [5] IS (4 downto 0) "&
            "DEFAULT (SerDesClockSettings (100Mhz))) "&
                 " )" ;

-----------------------------
-- NOTE: in each Channel, there are two boundary scan cells:
-- num cell  port    function safe [ccell disval rslt]
--TDO
--  n (BC_2,   *,    control,  0),
-- n+1(BC_8, port_id, bidir,   X,     n,    0,    Z),
--TDI
-----------------------------

   attribute REGISTER_ASSEMBLY of MyCorp_SERDES_1_2_3 : package is

      "byte_init_data (" &
         --TDI
         "(parity IS Channel), "&
         "(array byte(7 DOWNTO 5) IS Channel), "&
         "(byte_pll IS ChClock), "&
         "(array byte(4 DOWNTO 0) IS Channel) "&
         --TDO
                    " )" ;
-- The “byte_init_data” register segment is 50 bits long.
--  5 bits for the ChClock and 5 bits for each of 9 Channel instances.

end MyCorp_SERDES_1_2_3 ;

<EOF>

#######################################

-- Chip name is INIT_Example.
entity INIT_Example IS
   ...
```

```
PORT (
   ...
   DBus7 : inout     bit_vector (8 downto 0);
   DBus6 : inout     bit_vector (8 downto 0);
   DBus5 : inout     bit_vector (8 downto 0);
   DBus4 : inout     bit_vector (8 downto 0);
   DBus3 : inout     bit_vector (8 downto 0);
   DBus2 : inout     bit_vector (8 downto 0);
   DBus1 : inout     bit_vector (8 downto 0);
   DBus0 : inout     bit_vector (8 downto 0);
   ...
   VSEL_pin:      in    bit_vector(0 TO 4);
   ...
   Vdd_IO1: POWER_POS   bit;
...
USE STD_1149_1_2013.all;
USE Mycorp_SERDES_1_2_3.all;
...
attribute BOUNDARY_SEGMENT of INIT_Example : entity is
  "upper_four_byte_bus [72] ( "&
-- num  cell  port      function safe [input/ccell disval rslt]
  " 71 (BC_8, DBus7(8), bidir,    X,          70,    0,    Z), "&
  " 70 (BC_2,   *,      control,  0),                          "&
  " 69 (BC_8, DBus7(7), bidir,    X,          68,    0,    Z), "&
  " 68 (BC_2,   *,      control,  0),                          "&
  " 67 (BC_8, DBus7(6), bidir,    X,          66,    0,    Z), "&
  " 66 (BC_2,   *,      control,  0),                          "&
  " 65 (BC_8, DBus7(5), bidir,    X,          64,    0,    Z), "&
  " 64 (BC_2,   *,      control,  0),                          "&
  " 63 (BC_8, DBus7(4), bidir,    X,          62,    0,    Z), "&
  " 62 (BC_2,   *,      control,  0),                          "&
    ...
  "  9 (BC_8, DBus4(4), bidir,    X,           8,    0,    Z), "&
  "  8 (BC_2,   *,      control,  0),                          "&
  "  7 (BC_8, DBus4(3), bidir,    X,           6,    0,    Z), "&
  "  6 (BC_2,   *,      control,  0),                          "&
  "  5 (BC_8, DBus4(2), bidir,    X,           4,    0,    Z), "&
  "  4 (BC_2,   *,      control,  0),                          "&
  "  3 (BC_8, DBus4(1), bidir,    X,           2,    0,    Z), "&
  "  2 (BC_2,   *,      control,  0),                          "&
  "  1 (BC_8, DBus4(0), bidir,    X,           0,    0,    Z), "&
  "  0 (BC_2,   *,      control,  0)                           "&
  "   )";

attribute BOUNDARY_SEGMENT of INIT_Example : entity is
  "lower_four_byte_bus [72] ( "&
-- num  cell  port      function safe [input/ccell disval rslt]
  " 71 (BC_8, DBus3(8), bidir,    X,          70,    0,    Z), "&
  " 70 (BC_2,   *,      control,  0),                          "&
  " 69 (BC_8, DBus3(7), bidir,    X,          68,    0,    Z), "&
  " 68 (BC_2,   *,      control,  0),                          "&
  " 67 (BC_8, DBus3(6), bidir,    X,          66,    0,    Z), "&
  " 66 (BC_2,   *,      control,  0),                          "&
  " 65 (BC_8, DBus3(5), bidir,    X,          64,    0,    Z), "&
  " 64 (BC_2,   *,      control,  0),                          "&
  " 63 (BC_8, DBus3(4), bidir,    X,          62,    0,    Z), "&
  " 62 (BC_2,   *,      control,  0),                          "&
```

```
      ...
    "  9 (BC_8, DBus0(4), bidir,    X,           8,    0,    Z), "&
    "  8 (BC_2,   *,      control,  0),                          "&
    "  7 (BC_8, DBus0(3), bidir,    X,           6,    0,    Z), "&
    "  6 (BC_2,   *,      control,  0),                          "&
    "  5 (BC_8, DBus0(2), bidir,    X,           4,    0,    Z), "&
    "  4 (BC_2,   *,      control,  0),                          "&
    "  3 (BC_8, DBus0(1), bidir,    X,           2,    0,    Z), "&
    "  2 (BC_2,   *,      control,  0),                          "&
    "  1 (BC_8, DBus0(0), bidir,    X,           0,    0,    Z), "&
    "  0 (BC_2,   *,      control,  0)                           "&
    "   )";
...
attribute REGISTER_MNEMONICS of INIT_Example : entity is

   "SerDesCXVddSelLevel ( "&
      " 1.8v (1) <1.8V power supply used>, "&
      " 1.5v (0) <1.5V power supply used>  "&
      " ), "&

   "Switch ( "&        -- Simple On/Off single bit field.
      " off (1) <Turn feature off ('0')>, "&
      "  on (0) <Turn feature on ('1')> "&
      " ), "&

   "InitCompletionCode ( "& -- INIT completion codes.
      " NotStarted   (0b00) <Waiting for initialization.>, "&
      " ResetPLL     (0b10) <Resetting the system PLL.>, "&
      " ResetMemory  (0b01) <Resetting the memory.>, "&
      " Completed    (0b11) <Initialization complete.> "&
      " ), "&

   "InitErrorCode ( "& -- INIT error status codes.
      " NoError    (0b00) <No errors.>, "&
      " PLLfail    (0b10) <System PLL did not reset.>, "&
      " MemoryFail (0b01) <Memory did not reset.>, "&
      " AllFail    (0b11) <System PLL and Memory did not reset.> "&
      " ), "&

   "PLLConfigValues ( "&
      " PLLsOff   (0b000) <All PLLs off>, "& -- Stop All PLLs
      " PLL1on    (0b001) <Only PLL1 on>, "&
      " PLL2on    (0b010) <Only PLL2 on>, "&
      " PLL12on   (0b011) <PLL1 & PLL2 on>, "&
      " PLL3on    (0b100) <Only PLL3 on>, "&
      " PLL13on   (0b101) <PLL1 & PLL3 on>, "&
      " PLL23on   (0b110) <PLL2 & PLL3 on>, "&
      " PLL123on  (0b111) <All PLLs on> "&
      " ), "&

   "SerDesSampleOvrd (off (0), on (1)), "&

-- IO voltage configuration. These input pins are read in the
--   init_data register because they must be set at power-up
--   and checked before test.
    "IO_VSEL_Decodes  ( "&
      "B33_C33_L33  (0b00000) <BVdd=3.3V, CVdd=3.3V, LVdd=3.3V>, "&
```

```
         "B33_C33_L25  (0b00001) <BVdd=3.3V, CVdd=3.3V, LVdd=2.5V>, "&
         "B33_C33_L18  (0b00010) <BVdd=3.3V, CVdd=3.3V, LVdd=1.8V>, "&
         "B33_C25_L33  (0b00011) <BVdd=3.3V, CVdd=2.5V, LVdd=3.3V>, "&
         "B33_C25_L25  (0b00100) <BVdd=3.3V, CVdd=2.5V, LVdd=2.5V>, "&
         "B33_C25_L18  (0b00101) <BVdd=3.3V, CVdd=2.5V, LVdd=1.8V>, "&
         "B33_C18_L33  (0b00110) <BVdd=3.3V, CVdd=1.8V, LVdd=3.3V>, "&
         "B33_C18_L25  (0b00111) <BVdd=3.3V, CVdd=1.8V, LVdd=2.5V>, "&
         "B33_C18_L18  (0b01000) <BVdd=3.3V, CVdd=1.8V, LVdd=1.8V>, "&
         "B25_C33_L33  (0b01001) <BVdd=2.5V, CVdd=3.3V, LVdd=3.3V>, "&
         "B25_C33_L25  (0b01010) <BVdd=2.5V, CVdd=3.3V, LVdd=2.5V>, "&
         "B25_C33_L18  (0b01011) <BVdd=2.5V, CVdd=3.3V, LVdd=1.8V>, "&
         "B25_C25_L33  (0b01100) <BVdd=2.5V, CVdd=2.5V, LVdd=3.3V>, "&
         "B25_C25_L25  (0b01101) <BVdd=2.5V, CVdd=2.5V, LVdd=2.5V>, "&
         "B25_C25_L18  (0b01110) <BVdd=2.5V, CVdd=2.5V, LVdd=1.8V>, "&
         "B25_C18_L33  (0b01111) <BVdd=2.5V, CVdd=1.8V, LVdd=3.3V>, "&
         "B25_C18_L25  (0b10000) <BVdd=2.5V, CVdd=1.8V, LVdd=2.5V>, "&
         "B25_C18_L18  (0b10001) <BVdd=2.5V, CVdd=1.8V, LVdd=1.8V>, "&
         "B18_C33_L33  (0b10010) <BVdd=1.8V, CVdd=3.3V, LVdd=3.3V>, "&
         "B18_C33_L25  (0b10011) <BVdd=1.8V, CVdd=3.3V, LVdd=2.5V>, "&
         "B18_C33_L18  (0b10100) <BVdd=1.8V, CVdd=3.3V, LVdd=1.8V>, "&
         "B18_C25_L33  (0b10101) <BVdd=1.8V, CVdd=2.5V, LVdd=3.3V>, "&
         "B18_C25_L25  (0b10110) <BVdd=1.8V, CVdd=2.5V, LVdd=2.5V>, "&
         "B18_C25_L18  (0b10111) <BVdd=1.8V, CVdd=2.5V, LVdd=1.8V>, "&
         "B18_C18_L33  (0b11000) <BVdd=1.8V, CVdd=1.8V, LVdd=3.3V>, "&
         "B18_C18_L25  (0b11001) <BVdd=1.8V, CVdd=1.8V, LVdd=2.5V>, "&
         "B18_C18_L18  (0b11010) <BVdd=1.8V, CVdd=1.8V, LVdd=1.8V>, "&
         "Reserved     (others) <Reserved -- Do Not Use!> "&
         " ) " ;

attribute REGISTER_FIELDS of INIT_Example : entity is

   -- IP configuration register, see chip release documentation.
   --   Done in a register field because of non-contiguous bits;
   --   The PLL enable bits are used and the rest are defaulted in JTAG test.
   "configuration [15] ( " &
      "(IP_reg [12] IS (14 DOWNTO 6, 4, 2, 0) " &
         "DEFAULT (0x0DB) NoPI), " &  -- Required value for 1149 test.
      "(PLL_Enable [3] IS (5,3,1) SAFE (PLLConfigValues(PLLsoff)) NoPI) " &
             " )" ;

attribute REGISTER_ASSEMBLY of INIT_EXAMPLE : entity is
   -- Register Assembly of INIT_DATA register
   "init_data ( "&
   -- TDI
      -- First 36 bits are unused.
      "(reserved1[36] SAFE(0b0) NoPI NoUpd ), "&

      -- Observed in init_data because must be set at power-up.
      --   Deferred value must be specified for board.
      "(VSEL_bits [5] Captures(IO_VSEL_Decodes(*)) NoPO), "&

      -- Configuration register, 15 bits
      "(IP_Config IS configuration), "&

      -- Point to the IP package
      "(USING MyCorp_SERDES_1_2_3), "&
```

```
         -- First four bytes of channels are excludable
         --   Powered down in small package, power and segsel
         --   controlled by input 'Large_Package'
         --   1 bit
         "(IO_En IS SegSel "&
            "Domain_External(LargePkg) Segment(HiChan) TRSTreset), "&
         --  4*50 bits = 200 bits, excludable.
         "(array Databus(7 DOWNTO 4) IS byte_init_data), "&
            -- Values for protocol and TX_swing are deferred
         "(IO_seg_mux   IS SegMux Segment(HiChan)), "&
         -- Next four bytes of channels are fixed
         --  4*50 bits = 200 bits.
         "(array Databus(3 DOWNTO 0) IS byte_init_data), "&
            -- Values for protocol and TX_swing are deferred

      -- Remove the IP package reference
      "(USING -), "&

      -- Register bits that are SERDES IP inputs, not part of IP itself.
      --   Power level supplied to SerDes internal gates;
      "(SerDesCXVddSel [1] DEFAULT (SerDesCXVddSelLevel(*)) MON), "&
      --   Powerup SerDes Test Receivers during SAMPLE operation
      "(SerDesSamplePowerUp [1] DEFAULT (SerDesSampleOvrd (off)) MON), "&
      -- 2^^10 possible decodes. See Reference Manual.
      "(DataTermSel [10] SAFE(0) NoPI), "&
      -- Reserved Field
      "(reserved2[8]) " &
   -- TDO
            "), " &
   -- The init_data register defined above has a minimum length of 277 bits:
   --   36  (reserved1)
   -- +  5  (VSEL_bits)
   -- + 15  (IP_config)
   -- +  1  (IO_En, SEGSEL is always included.)
   -- + 4*50 (Databus(3 DOWNTO 0))
   -- +  1  (SerDesCXVddSel)
   -- +  1  (SerDesSamplePowerUp)
   -- + 10  (DataTermSel)
   -- +  8  (reserved2)
   -- =277
   -- Plus one excludable 4*50 bit segment (Databus(7 DOWNTO 4))

   -- Register Assembly for INIT_STATUS register - read-only per the standard
   "init_status ( "&
      "(INITErrorStatus [2] " &
         "CAPTURES (InitErrorCode(NoError)) NoPO), " &
      "(INITCompletionStatus [2] " &
         "CAPTURES (InitCompletionCode(Completed)) NoPO) " &
               ") ";

attribute REGISTER_ASSEMBLY of INIT_EXAMPLE : entity is
   -- Register Assembly of BOUNDARY register
   "boundary ( "&
   -- TDI
   ...
   -- LVdd I/O voltage domain
   "(upper_data_bus_En  IS SegSel "&
```

```
            "Domain_External(LargePackage) Segment(upper)), "&
   "(upper_data_bus     IS upper_four_byte_bus), "&  -- 72 cells
   "(upper_data_bus_mux IS SegMux Segment(upper)), "&
   "(lower_data_bus     IS lower_four_byte_bus), "&  -- 72 cells
   ...
   -- TDO
         ")";

attribute Register_Association of INIT_EXAMPLE : entity is
   -- Port ID 'VDD_IO1' is the power source for upper bank of I/O
   --   '1' = large package, '0' = small package.
   "upper_data_bus_En  : PORT (Vdd_IO1), "&
   "IO_En              : PORT (Vdd_IO1), "&
   "VSEL_bits(4)       : PORT (VSEL_pin(4)), "&
   "VSEL_bits(3)       : PORT (VSEL_pin(3)), "&
   "VSEL_bits(2)       : PORT (VSEL_pin(2)), "&
   "VSEL_bits(1)       : PORT (VSEL_pin(1)), "&
   "VSEL_bits(0)       : PORT (VSEL_pin(0)) ";

attribute Power_Port_Association of INIT_EXAMPLE : entity is
   -- Port ID 'VDD_IO1' is the power source for upper bank of I/O
   "VDD_IO1 : (DBus4(0), DBus4(1), DBus4(2), DBus4(3), DBus4(4), "&
             " DBus4(5), DBus4(6), DBus4(7), DBus4(8), "&
             " DBus5(0), DBus5(1), DBus5(2), DBus5(3), DBus5(4), "&
             " DBus5(5), DBus5(6), DBus5(7), DBus5(8), "&
             " DBus6(0), DBus6(1), DBus6(2), DBus6(3), DBus6(4), "&
             " DBus6(5), DBus6(6), DBus6(7), DBus6(8), "&
             " DBus7(0), DBus7(1), DBus7(2), DBus7(3), DBus7(4), "&
             " DBus7(5), DBus7(6), DBus7(7), DBus7(8)) ";

...
<EOF>
```

D.1.2 Example PDL for INIT example

This example shows the init_setup and init_run PDL procedures for the component INIT_example. The component is used on a specific board, and on that board, its instance value is U23. Board-level procedures are not shown, just the initialization procedures for this component on that board.

Inspecting the BSDL reveals several register fields that have deferred values. These cannot be set within the PDL procedures supplied by the component designer because the values will vary from instance to instance of use of the component. So two init_setup PDL files are shown, one from the component supplier and the other written for the specific instance of the component.

First is the PDL supplied by the IP supplier.

```
# Supplied by MyCorp for the SerDes product line, version 1.2.3
#   These procedures set up data, but do not perform a scan.
iPDLLevel 0 -version STD_1149_1_2013
iProcGroup MyCorp_SERDES_1_2_3 ; # Code is for BSDL package

# Set up a single channel
iProc setup_channel_init_data { proto swing } {
  iWrite Protocol $proto
  iWrite TX_Swing $swing
  }
```

```
# Set up the hard IP byte (9 channels plus data PLL)
iProc setup_byte_init_data { proto swing freq } {
  iCall parity.setup_channel_init_data $proto $swing
  iCall byte(7:0).setup_channel_init_data $proto $swing
  iWrite byte_pll.Freq $freq
  }
<EOF>
```

Next is the PDL supplied from the component provider (the component provider will pass along the PDL files from the IP provider).

```
# Supplied by Acme Quick-Chip for chip INIT_Example
iSource MyCorp.SerDes_1_2_3.pdl
iPDLLevel 0 -version STD_1149_1_2013
iProcGroup INIT_Example ; # Code is for chip BSDL

# The init_setup standard procedure.
#   This procedure may be used as a template for manually setting up
#   deferred values for a given instance, or as-is in conjunction with
#   an instance specific init_setup procedure.

#####################################################################
# NOTE:  Lines commented out with data fields in chevrons: <data>,
#        must be specified for each instance of this component,
#        or another procedure provided for the instance that replaces
#        these lines.
#####################################################################

iProc init_setup -export { } {
# Set up the data queues for all read and write fields.
  iWrite reserved1 -safe
# iRead  VSEL_bits <decode>
  iWrite IP_Config.IP_reg -default
  iWrite IP_Config.PLL_Enable -safe
  iWrite IO_En Exclude
# Databus(7:4) are initially excluded, but setting up the data doesn't hurt.
# iCall  Databus(7).setup_byte_init_data <proto> <swing> <freq>
# iCall  Databus(6).setup_byte_init_data <proto> <swing> <freq>
# iCall  Databus(5).setup_byte_init_data <proto> <swing> <freq>
# iCall  Databus(4).setup_byte_init_data <proto> <swing> <freq>
# iCall  Databus(3).setup_byte_init_data <proto> <swing> <freq>
# iCall  Databus(2).setup_byte_init_data <proto> <swing> <freq>
# iCall  Databus(1).setup_byte_init_data <proto> <swing> <freq>
# iCall  Databus(0).setup_byte_init_data <proto> <swing> <freq>
# iWrite SerDesCXVddSel <voltage>
  iWrite SerDesSamplePowerUp on
  iWrite DataTermSel -safe
  iWrite reserved2 0b0

# Check status of VSEL_bits and verify they are set correctly.
  iApply -nofail
  ifFalse
    iSetFail -quit "I/O voltage select pins are set incorrectly; stop test!"
  ifEnd

# Check status of excluded segment (depends on package type)
```

```
  iRead IO_En 0b1
  iApply -nofail
  ifTrue
    iWrite IO_En Include
    iApply ; # Include excludable segment
  ifEnd
  iApply ; # Load full register, possibly including excludable segment
# Ready for init_run
  }

# The init_run standard procedure.
#   This procedure is very simple, really requiring only a wait.
#   While it is expected that the status will be done after the wait,
#   a polling loop is included to allow delayed completion.
iProc init_run -export { } {
  iRunLoop 2500 ; # Wait 2500 TCK for internal init state machine
  iLoop
    iRead INITErrorStatus NoError
    iRead INITCompletionStatus Completed
    iApply -nofail
  iUntil -match -maxcnt 10 "Failure of initialization of INIT_Example."
  iNote -status "Initialization processing for INIT_Example complete.\n"
  }
<EOF>
```

The board test engineer could take the init_setup routine above and treat it as a template by manually substituting the needed values for the U23 instance on the board, uncommenting those lines, and changing the iProcGroup to U23. The test engineer may also have a test setup tool that reads the BSDL, recognizes all of the fields in the init_data register with deferred values, and then asks the test engineer to choose (from mnemonic values, if provided) the appropriate values for the U23 instance. The tool then could write out the following PDL procedure for execution, and this procedure (which does not include any **iApply** commands) would be run before the init_setup procedure for the INIT_Example component above, which was provided by the component supplier. In effect, this generated file simply replaces the lines of the component file that were commented out due to deferred values.

This is a simple situation, and either the template approach or the separate PDL procedures work equally well. If the init_setup procedure for the component were complex, then the board test engineer might prefer to not touch the verified procedure from the component supplier and use the two files.

```
# Generated by Maddox TestFast 1149.1 test generator
#   Run before init_setup procedure associated with chip
iSource MyCorp.SerDes_1_2_3.pdl
iPDLLevel 0 -version STD_1149_1_2013
iProcGroup U23 ; # Code is for instance U23 of chip INIT_Example

iProc init_setup -export { } {
  iRead  VSEL_bits  B25_C25_L18
  iCall  Databus(7).setup_byte_init_data XAUI 75%_Swing 125Mhz
  iCall  Databus(6).setup_byte_init_data XAUI 75%_Swing 125Mhz
  iCall  Databus(5).setup_byte_init_data XAUI 75%_Swing 125Mhz
  iCall  Databus(4).setup_byte_init_data XAUI 75%_Swing 125Mhz
  iCall  Databus(3).setup_byte_init_data XAUI 75%_Swing 125Mhz
  iCall  Databus(2).setup_byte_init_data XAUI 75%_Swing 125Mhz
  iCall  Databus(1).setup_byte_init_data XAUI 75%_Swing 125Mhz
  iCall  Databus(0).setup_byte_init_data XAUI 75%_Swing 125Mhz
  iWrite SerDesCXVddSel 1.8v
  }
<EOF>
```

D.2 Multiple wrapper serial port structure and procedures

D.2.1 Wrapper serial port structural description

This set of examples illustrates the use of the selectable segment capability, including documentation of broadcast scan (scanning data into multiple registers in parallel).

IEEE Std 1500 defines test wrappers for "cores." The IEEE 1500 architecture was designed to allow interface compatibility with the IEEE 1149.1 test access port (TAP) controller. Indeed, the wrapper serial control (WSC) interface (with the exception of the SelectWIR and optional TransferDR signals) for the wrapper serial port (WSP) corresponds to the recommended TDR interface documented in Table 9-1 and Table 9-2. The actual description of the WSP register structure, however, was not compliant and could not be documented in BSDL prior to the 2013 version of this standard. The scannable registers of a WSP are similar to the registers of this standard in that there is a wrapper instruction register (WIR) and two or more wrapper data registers (WDRs), each selected by a value in the WIR.

Single WSP

A typical, if simple, WSP with a single design-specific register in the core is shown in Figure D-2 and illustrates the use of selectable segments to document a WSP.

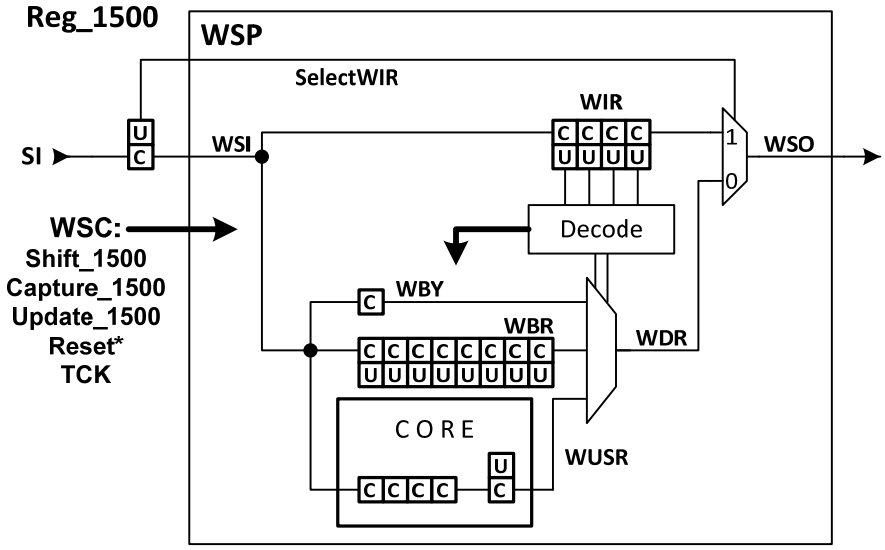


Figure D-2—Simple wrapper serial port

This design contains a single register cell (which is not part of the IEEE 1500-compliant WSP as shown by the box) to generate the SelectWIR signal and a WSP, as defined in IEEE Std 1500. The WSP contains two selectable segment structures, the inner that selects between the available WDRs and the outer that selects between the WDR and the WIR. In this example, there are three WDRs: the required wrapper bypass (WBY) and wrapper boundary (WBR) registers, and one design-specific wrapper user (Wusr) register in the core. For the purposes of illustration, the Wusr register controls and captures status from a self-test capability in the core.

Per IEEE Std 1500, the SelectWIR signal must be provided as part of the WSC, although it is shown separately here. The wrapper scan in (WSI) fans out to all registers, and multiplexing circuits are used to select one register for connection to the wrapper scan out (WSO). The WSP inside the box is what is defined in IEEE Std 1500. Note that gating logic to control scan and update operations in the scan segments are not shown for simplicity.

This structure could be documented in a BSDL user package as follows.

```
-- Supplied by MyCorp for REG_1500 version 1.0
package REG_1500 is
   use STD_1149_1_2013.all;
end REG_1500;

package body REG_1500 is

   use STD_1149_1_2013.all;

Attribute REGISTER_MNEMONICS of REG_1500  : package is
  "WIR_decode  ( "&
    "WS_BYPASS  (0B0000) <Wrapper Bypass Instruction>, "&
    "WS_EXTEST  (0B0001) <Wrapper Serial External Boundary Instruction>, "&
    "WS_INTEST  (0B0010) <Wrapper Serial Internal Boundary Instruction>, "&
    "WS_BIST    (0B0100) <BIST Instruction>, "&
    "WP_ALL     (0B1xxx) <Wrapper Parallel instructions> "&
    " )," &
  "BISTGROUP ( "&
    "Disable  (0B0) < BIST has not been enabled >, "&
    "Enable   (0B1) < BIST enabled > "&
    " ),"&
  "STATGROUP ( "&
    "PASS     (0B1001),  "&
    "FAIL     (0B0111)   "&
    " )," &
  "MODEGROUP ( "&
    "MODE0    (0X0),  "&
    "MODE3    (0X3)   "&
    " )";

Attribute REGISTER_ASSEMBLY of REG_1500  : package IS
  "REG_1500 ( " &  -- The Select WIR bit and the Wrapper Serial Port
    -- Reset to WBY
    "(SELWIR [1] DelayPO ResetVal(0b0) TAPReset ),  "&
    "(WSP IS WSP_MUX) "&
  " ), "&
  "WSP_MUX ( "&   -- The outer selectable segments: WIR and WDR
    "(SelectMUX "&
      -- Reset to WBY
      "(WIR IS WIR_Seg), "&
      "(WDR IS WDR_MUX) "&
      "SelectField (SELWIR) "&
      "SelectValues ((WIR : 0b1) (WDR : 0b0)) "&
    " ) "&
  " ), "&
  "WIR_Seg (  (WIR_field [4] "&
              "ResetVal(WIR_decode(WS_BYPASS)) TAPReset )    ), "&
  "WDR_MUX ( "&  -- The inner selectable segments: WBY, WBR, and Wusr
    "(SelectMUX "&
      "(WBY  IS Reg_WBY CAPTURES(0) ), "&
      "(WBR  IS Reg_WBR), "&
      "(WUSR IS Reg_WUSER) "&
      "SelectField (WIR) "&
      "SelectValues ("&
        "(WBY  : WS_BYPASS, WP_ALL) "&
```

```
        "(WBR  : WS_EXTEST, WS_INTEST) "&
        "(WUSR : WS_BIST) "&
        " ) "&
    " ) "&
  " ), "&
  "REG_WBY   ( (WBY[1] NOPO)), " &
  "REG_WBR   ( (WBR[8] )), " &
  "REG_WUSER ( (  CSR[4] CAPTURES(STATGROUP(-)) " &
                    "DEFAULT(MODEGROUP(MODE0)) NOUPD )," &
  "            (  GO [1] ResetVal(BISTGROUP(Disable)) TapReset )   )" ;

end REG_1500;
```

Multiple selectable and gated WSP

In the more general case, there will be the requirement to scan only one selectable segment at a time, as well as to scan data into different groups of selectable segments. In this example, three of the Reg_1500 structures defined earlier are used, but with some new requirements: to scan each Reg_1500 independently while the others hold their state; to scan the first two in parallel while the third holds its state; and to scan all three in parallel. The selection of which Reg_1500 is connected in the scan chain is to be made independently from the selection of which of the Reg_1500 segments are scanned, although there are obvious combinations of those selections that would lead to a broken scan chain and need to be documented as constraints.

This separation of function (selecting the segments to be scanned and selecting the segment to be connected to the scan chain) may be dictated in part by a “separation” between the controlling circuits (TAP, etc.) and the Reg_1500 instances. This could simply be a long distance across a component, or a separate power-down domain, or an external component that is not compliant with IEEE Std 1149.1 in a “3-D” stack or on a board. In any case, in this example, the component designer has determined that it is highly desirable that only one copy of the WSC be sent across the “separation” and any gating of those control signals needed to control which Reg_1500 instance is scanned be done locally to the IEEE 1500 instances by a local field in the TDR scan chain.

Figure D-3 shows an overall structure achieving these goals and needing to be documented.

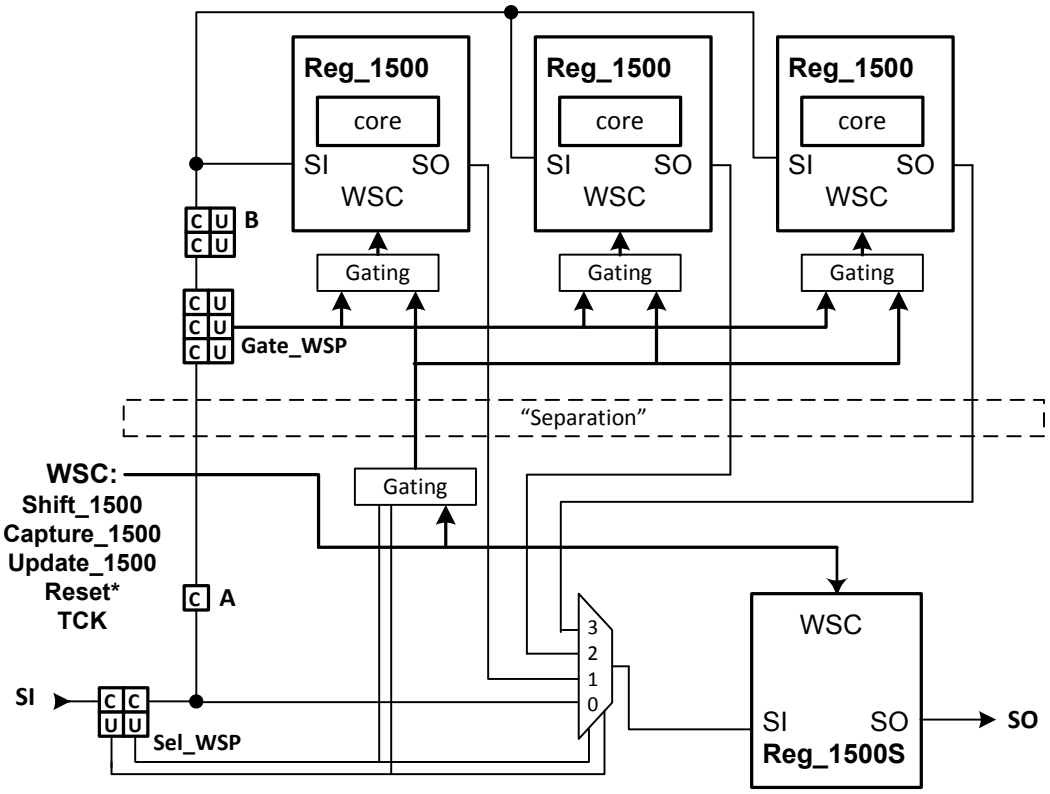


Figure D-3—Three wrappers with WSC gating logic

The three "Gating" blocks above the separation would block Update_1500, Shift_1500, and Capture_1500, as appropriate to ensure that each WSP would hold its state when not selected for scanning by Gate_WSP. The "Gating" block below the separation would block Update_1500, Shift_1500, and Capture_1500, as appropriate to ensure that all three selectable WSPs held their values when none were selected for scan-out by Sel_WSP.

In documenting this structure, the A, Gate_WSP, and B register segments present a challenge. Note the wire from the output of Sel_WSP register segment to the 0 input of the four-way multiplexer. This establishes the "scan fan-out" point for the broadcast at that node, placing the A, Gate_WSP, and B segments in line in all three of the selectable Reg_1500 segments. This can be done by defining the A, Gate_WSP, and B fields, using them in a register segment, and then pointing to that instance with a <instance reference>, but then all three Reg_1500 segments have to be defined in separate register assemblies before they can be instantiated in the selectable segment BSDL structure because there are two elements in each selectable segment.

In addition to the selectable WSP, another WSP with a different configuration of internal registers is added to the scan chain after the selectable REG_1500 segments. It is on the "near" side of the separation, and for this example, it is used to check the connections across the separation by putting the WSP on each side of the separation into external test. This test is illustrated in the procedural code for this example. Figure D-4 shows the configuration of this WSP.

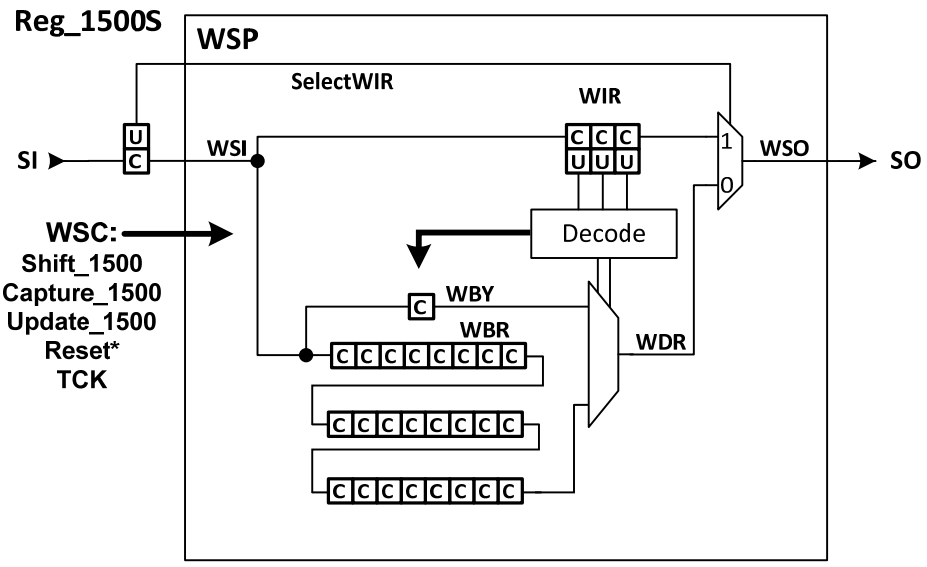


Figure D-4—WSP example for interconnect testing

This additional WSP itself could be documented in another User Package as follows.

```
package REG_1500S is
   use STD_1149_1_2013.all;
end REG_1500S;

package body REG_1500S is

   use STD_1149_1_2013.all;

Attribute REGISTER_MNEMONICS of REG_1500S  : package is
  "WIR_decode  ( "&
    "WS_BYPASS  (0B000) <Wrapper Bypass Instruction>, "&
    "WS_EXTEST  (0B001) <Wrapper Serial External Boundary Instruction>, "&
    "WS_INTEST  (0B010) <Wrapper Serial Internal Boundary Instruction>  "&
      " )";

Attribute REGISTER_ASSEMBLY of REG_1500S  : package IS
  "REG_1500S ( " &  -- The Select WIR bit and the Wrapper Serial Port
    -- Reset to WBY
    "(SELWIR [1] DelayPO ResetVal(0b0) TAPReset ),  "&
    "(WSP IS WSP_MUX) "&
  " ), "&
  "WSP_MUX ( "&   -- The outer selectable segments: WIR and WDR
    "(SelectMUX "&
      -- Reset to WBY
      "(WIR IS WIR_Seg), "&
      "(WDR IS WDR_MUX) "&
      "SelectField (SELWIR) "&
      "SelectValues ((WIR : 0b1) (WDR : 0b0)) "&
    " ) "&
  " ), "&
  "WIR_Seg (  (WIR_field [3] "&
               "ResetVal(WIR_decode(WS_BYPASS)) TAPReset )   ), "&
```

```
  "WDR_MUX ( "&  -- The inner selectable segments: WBY, WBR, and Wusr
    "(SelectMUX "&
      "(WBY  IS Reg_WBY), "&
      "(WBR  IS Reg_WBR)  "&
      "SelectField (WIR) "&
      "SelectValues ("&
        "(WBY  : WS_BYPASS ) "&
        "(WBR  : WS_EXTEST, WS_INTEST) "&
        " ) "&
    " ) "&
  " ), "&
  "REG_WBY   (( WBY[1]  NOPO )), " &
  "REG_WBR   (( WBR[24] NOPO )) ";

end REG_1500S;
```

The multiple gated WSP structure of Figure D-3 could then be documented as follows.

```
package REG_1500_ASSM is
   use STD_1149_1_2013.all;
end REG_1500_ASSM ;

package body REG_1500_ASSM is
   use STD_1149_1_2013.all;
   use REG_1500.all;
   use REG_1500S.all;

attribute REGISTER_MNEMONICS of REG_1500_ASSM  : package is
   "WSP  ( "&
      "   None   (0B00) <Bypass all WSPs>, "&
      "   WSP1   (0B01) <Observe WSP(1)>, "&
      "   WSP2   (0B10) <Observe WSP(2)>, "&
      "   WSP3   (0B11) <Observe WSP(3)>  "&
     " )," &
   "BRDCST ( "&
      "   None      (0B000) <All WSP held>, "&
      "   WSP1      (0B001) <Scan WSP(1) only>, "&
      "   WSP2      (0B010) <Scan WSP(2) only>, "&
      "   WSP3      (0B011) <Scan WSP(3) only>, "&
      "   1AND2     (0B110) <Scan just WSP(1) and WSP(2)>, "&
      "   ALLWSP    (0B111) <Scan all WSPs >  "&
     "  )";

Attribute REGISTER_ASSEMBLY of REG_1500_ASSM : package IS
  "Reg_1500_MUX ( " &
    "(Sel_WSP[2]  ResetVal(WSP(None))    TAPReset  ) ,"&
    "(SELECTMUX " &
      "(WIRE1 is WIRE)," &
      "(ARRAY WSP(1 TO 3) IS WSP_inst) " &
      "SELECTFIELD (Sel_WSP) "&   -- 4:1 selection
      "SELECTVALUES ( "& --   Decode logic for connecting a WSP to Scan-Out
        "(WIRE1:None) (WSP(1):WSP1) (WSP(2):WSP2) (WSP(3):WSP3) )"&
      "BROADCASTFIELD (Gate_WSP) "&  -- Could use WSP_common.Gate_WSP
      "BROADCASTVALUES ( "&    --  Decode  logic for gating WSC
        "(WSP(1),WSP(2),WSP(3) : ALLWSP) "&
        "(WSP(1),WSP(2)        : 1AND2 ) "&
        "(WSP(1)               : WSP1) "&
```

```
        "(WSP(2)                : WSP2) "&
        "(WSP(3)                : WSP3) "&
                        ")"&
    "),"&
    "( WSP_1500S is Reg_1500S)" &    -- Reg_1500S comes after MUX
                ")," &                -- end REG_1500_MUX
  "WIRE ( ( WIRE[0] ) ), "&
  "WSP_inst ( "&
      "(WSP_common), "&
      "(WSP_1500 IS Reg_1500) " &
  "), "&
  "common_seg ( (WSP_common IS common) ), "&
  "common ("&
    "(A [1] NOUPD), "&
    "(Gate_WSP[3] ResetVal(BRDCST(None)) TAPReset ), "&
    "(B [2] ) "&
  ") " ;

attribute REGISTER_CONSTRAINTS of REG_1500_ASSM : package is
  "REG_1500_MUX (" &

    "( Gate_WSP ==  BRDCST{1AND2}   &&   Sel_WSP ==  WSP{WSP3} ) "&
    "ERROR < Sel_WSP of WSP3 not possible with Gate_WSP of 1AND2 >, "&

    "(( (Gate_WSP == BRDCST{WSP2} ) || (Gate_WSP == BRDCST{WSP3}) ) "&
    "     && (Sel_WSP == WSP{WSP1}) ) "&
    "ERROR < Sel_WSP of WSP1 not possible with Gate_WSP of WSP2 or 3 >, "&

    "(( (Gate_WSP == BRDCST{WSP1} ) || (Gate_WSP == BRDCST{WSP3}) ) "&
    "     && (Sel_WSP == WSP{WSP2}) ) "&
    "ERROR < Sel_WSP of WSP2 not possible with Gate_WSP of WSP1 or 3 >, "&

    "(( (Gate_WSP == BRDCST{WSP1} ) || (Gate_WSP == BRDCST{WSP2}) ) "&
    "     && (Sel_WSP == WSP{WSP3}) ) "&
    "ERROR < Sel_WSP of WSP3 not possible with Gate_WSP of WSP1 or 2 > "&
  ")";

end REG_1500_ASSM;
```

These constraint statements document the conflicting combinations of values in the two segment selection fields (Gate_WSP and Sel_WSP), which need to be coordinated.

If the WSPs on the other side of the separation are in a power-down domain, or on separate components that may be missing during test, then the entire structure should be an excludable segment. This provides a second way of dealing with the “wire” or bypass by using a **SEGSEL** and **SEGMUX** combination to exclude or include the entire structure. The **SEGSEL** would capture the “ready to scan” signal indicating that the three IEEE 1500 WSPs may be included in the scan chain. Where the “separator” is a boundary between die in a component and configuration data are available to indicate whether or how many IEEE 1500 WSPs are present, these data could again be used as static “ready to scan” signals and could simplify the problem of multiple configurations in manufacturing.

Figure D-5 shows a structure achieving these modified goals and needing to be documented. In this case, assuming that the additional IEEE 1500 WSP is used only for testing connections across the separator, it is also in the excluded segment, but it need not be.

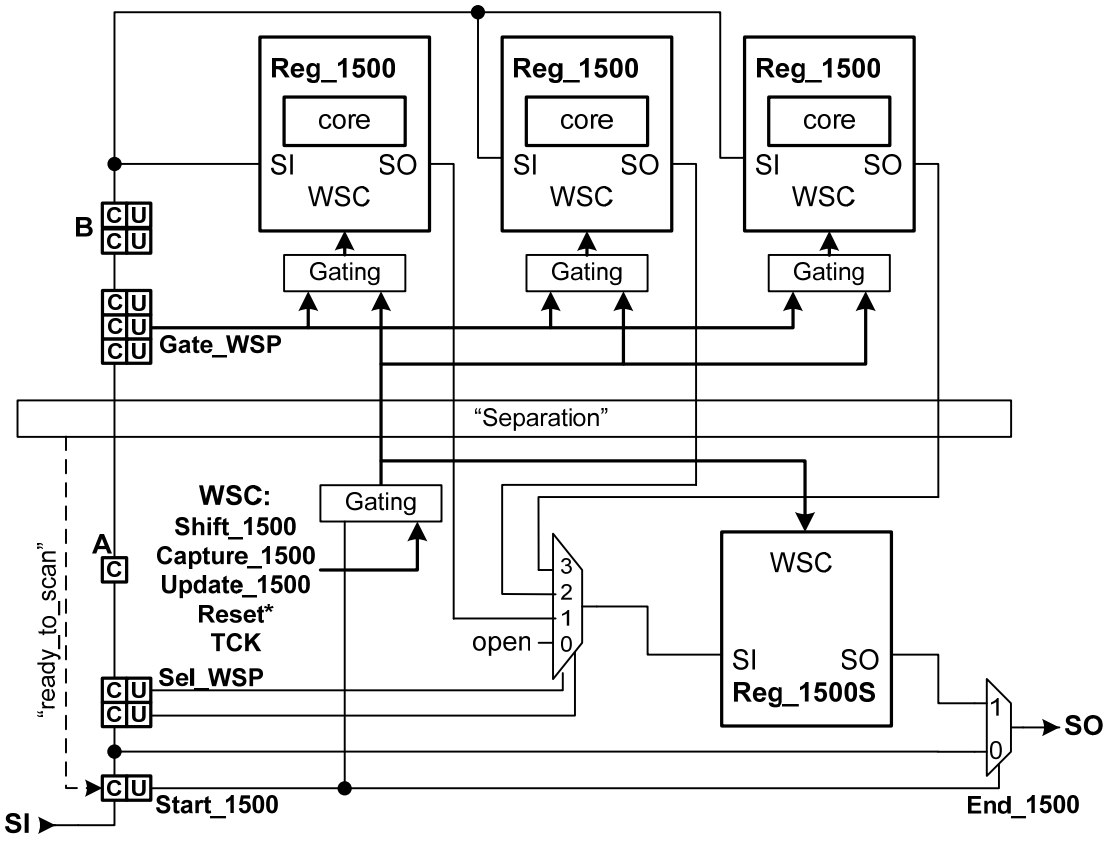


Figure D-5—Three wrappers with WSC gating logic and SEGSEL

Because the **SEGSEL** (Start_1500) and **SEGMUX** (End_1500) handle excluding the four IEEE 1500 WSPs, input 0 to the selection multiplexer is not used and the scan fan-out point is after the three registers that are in series with the WSPs (after the "B" segment). This structure could be documented in BSDL as follows, and the same names are used as in the previous package so that the same PDL will apply to both versions. One or the other would be used, but not both.

```
package REG_1500_ASSM is
   use STD_1149_1_2013.all;
end REG_1500_ASSM ;

package body REG_1500_ASSM is
   use STD_1149_1_2013.all;
   use REG_1500.all;
   use REG_1500S.all;

attribute REGISTER_MNEMONICS of REG_1500_ASSM : package is
   "WSP  ( "&
   "   WSP1    (0B01)   <Observe WSP(1)>, "&
   "   WSP2    (0B10)   <Observe WSP(2)>, "&
   "   WSP3    (0B11)   <Observe WSP(3)>, "&
   "   Open    (others) <DO NOT USE, broken scan chain> "&
   "  ), "&
   "BRDCST  ( "&
   "   WSP1    (0B001)   <Scan WSP(1) only>, "&
   "   WSP2    (0B010)   <Scan WSP(2) only>, "&
```

```
    "   WSP3    (0B011)   <Scan WSP(3) only>, "&
    "   1AND2   (0B110)   <Scan just WSP(1) and WSP(2)>, "&
    "   ALLWSP  (0B111)   <Scan all WSPs >, "&
    "   None    (others)  < DO NOT USE > "&
    "  )";

Attribute REGISTER_ASSEMBLY of REG_1500_ASSM : package IS
  "Reg_1500_MUX ( "&
    -- Other possible segments
    "(Start_1500 IS SegSel TAPReset SEGMENT(Seg_1500)), "&
    "(Sel_WSP[2]  DelayPO ResetVal(WSP(WSP1)) TAPReset ), "&
    "(A [1] NOUPD), "&
    "(Gate_WSP[3] DelayPO ResetVal(BRDCST(ALLWSP)) TAPReset ), "&
    "(B [2]), "&
    "(SELECTMUX "&
      "(ARRAY WSP(1 TO 3) IS REG_1500) "&
      "SELECTFIELD (Sel_WSP) "&   -- 3:1 selection
      "SELECTVALUES ( (WSP(1):WSP1) (WSP(2):WSP2) (WSP(3):WSP3) )"&
      "BROADCASTFIELD (Gate_WSP) "&
      "BROADCASTVALUES ( "&    -- Decode logic for WSC
        "(WSP(1),WSP(2),WSP(3) : ALLWSP) "&
        "(WSP(1),WSP(2)        : 1AND2) "&
        "(WSP(1)               : WSP1) "&
        "(WSP(2)               : WSP2) "&
        "(WSP(3)               : WSP3) "&
      ")"&
    "), "&
    "(WSP_1500S is REG_1500S), "&
    "(End_1500 IS SegMux SEGMENT(Seg_1500)) "&
    -- Other possible segments
  ")";

attribute REGISTER_CONSTRAINTS of REG_1500_ASSM : package is
  "REG_1500_MUX (" &

     "( Gate_WSP == BRDCST{1AND2} && Sel_WSP == WSP{WSP3} ) "&
     "ERROR < Sel_WSP of WSP3 not possible with Gate_WSP of 1AND2>, "&

     "(( (Gate_WSP == BRDCST{WSP2} ) || (Gate_WSP == BRDCST{WSP3}) ) "&
     "     && (Sel_WSP == WSP{WSP1}) ) "&
     "ERROR < Sel_WSP of WSP1 invalid with Gate_WSP of WSP2 or WSP3 >, "&

     "(( (Gate_WSP == BRDCST{WSP1} ) || (Gate_WSP == BRDCST{WSP3}) ) "&
     "     && (Sel_WSP == WSP{WSP2}) ) "&
     "ERROR < Sel_WSP of WSP2 invalid with Gate_WSP of WSP1 or WSP3 >, "&

     "(( (Gate_WSP == BRDCST{WSP1} ) || (Gate_WSP == BRDCST{WSP2}) ) "&
     "     && (Sel_WSP == WSP{WSP3}) ) "&
     "ERROR < Sel_WSP of WSP3 invalid with Gate_WSP of WSP1 or WSP2 > "&
   ")";

end REG_1500_ASSM;
```

These constraint statements document the conflicting combinations of values in the selection fields that need to be coordinated.

D.2.2 Wrapper serial port example

The BSDL for this example is in D.2.1. This example shows PDL for the two types of IEEE 1500 WSP blocks, plus code using the multiple WSPs.

Reg_1500.pdl

First is the PDL provided with the package for the IEEE 1500 WSPs and core shown in Figure D-2.

```
# Supplied by MyCorp for REG_1500 version 1.0

iPDLLevel 0 -version STD_1149_1_2013
iProcGroup REG_1500

#  check that bypass register can be scanned
iProc check_bypass { } {
  iWrite WIR WS_BYPASS        ;# Use WS_BYPASS and not WP_ALL
  iRead WBY 0                 ;# = WDR.WBY, only one WBY in package
  iApply
  }

# Set up mode and execute BIST in 1 scan operation
iProc start_bist { mode } {
  # CSR is documented to be a capture and shift register only.
  # GO has capture, shift, and update.
  iWrite CSR $mode            ;# = WDR.Wusr.CSR, only one CSR in package
  iWrite GO  Enable           ;# = WDR.Wusr.GO, only one GO in package
  iApply
  iRunLoop  100000
  }

# Check BIST results
iProc check_bist { instance mode } {
  iRead  CSR PASS             ;# = WDR.Wusr.CSR, only one CSR in package
  iApply -nofail
  ifFalse
     iSetFail "$instance REG_1500 BIST test with mode = $mode failed\n"
  ifEnd
  }

<EOF>
```

Reg_1500S.pdl

```
# Supplied by MyCorp for 1500S version 1.0
iPDLLevel 0 -version STD_1149_1_2013
iProcGroup REG_1500S ;
# check that bypass register can be scanned
iProc check_bypass { } {
  iRead WBY 0
  }
```

Reg_1500_Assm.pdl

The PDL supplied for the multiple selectable and gated WSPs shown in Figure D-3. This uses the PDL for the REG_1500 above.

When the Gate_WSP field is set to a value that enables broadcast, the Sel_WSP is still set by the **iApply** command to the WSP containing the register field being addressed.

```
# Supplied by MyCorp for 1500_ASSM version 1.0

iSource REG_1500.pdl
iSource REG_1500S.pdl
iPDLLevel 0 -version STD_1149_1_2013
iProcGroup REG_1500_ASSM ;

#  check that the REG_1500S bypass register can be scanned
iProc check_bypass { } {
  iRead WBY 0                       ;# = WDR.WBY, unambiguous
  }

iProcGroup REG_1500_ASSM ;

#  check that bypass registers can be scanned
iProc check_bypass { } {
  iCall WSP_1500S.check_bypass
  # scan occurs in each of the 3 lines after this
  iCall WSP(1).WSP_1500.check_bypass
  iCall WSP(2).WSP_1500.check_bypass
  iCall WSP(3).WSP_1500.check_bypass
  }

# start and check BIST for each WSP_1500

iProc bist_test { } {

  # Enable broadcast to save wait time.
  # Two modes of broadcast exist: ALLWSP and 1AND2,
  #   specify the mode explicitly to avoid ambiguity.
  # WSP(1) is implicitly seleted for scan out by being named.
  iWrite WSP(1).WSP_common.Gate_WSP ALLWSP
  iApply                 ;# Select parallel scan mode, WSP(1) in scan path.

  # writing in broadcast mode, all WSPs are getting BIST mode and start
  iCall WSP(1).WSP_1500.start_bist MODE0

  # turn off broadcast, WSP(1) is currently set for scan-out
  iWrite WSP(1).WSP_common.Gate_WSP WSP1
  iApply

  # check BIST status of each, pass in instance name for error message
  iCall WSP(1).WSP_1500.check_bist WSP(1) MODE0
  iCall WSP(2).WSP_1500.check_bist WSP(2) MODE0
  iCall WSP(3).WSP_1500.check_bist WSP(3) MODE0

  # turn broadcast back on
  iWrite WSP(1).WSP_common.Gate_WSP ALLWSP
  iApply

  # all WSPs are getting BIST mode and start
  iCall  WSP(1).WSP_1500.start_bist MODE1

  # turn off broadcast; WSP(1) is currently set for scan-out
  iWrite WSP(1).WSP_common.Gate_WSP WSP1
```

```
  iApply

  # check BIST status of each, pass in instance name for error message
  iCall WSP(1).WSP_1500.check_bist WSP(1) MODE1
  iCall WSP(2).WSP_1500.check_bist WSP(2) MODE1
  iCall WSP(3).WSP_1500.check_bist WSP(3) MODE1

}

iPDLLevel 1 -version STD_1149_1_2013

iProc 1500_interconnect { } {

  # Connections exist 1:1 between WSP(3:1) and WSP_1500S

  iWrite WSP(1).WSP_common.Gate_WSP ALLWSP
  iApply

  # The WBR access is ambiguous, there are
  # two paths for accessing the WBR, WS_EXTEST and WS_INTEST
  # all four WSPs get WS_EXTEST in the WIR
  iWrite WSP(1).WSP_1500.WIR  WS_EXTEST   ;# Broadcast mode
  iWrite WSP_1500S.WIR  WS_EXTEST
  iApply                                  ;# 4 WSPs in WS_EXTEST mode

  iWrite WSP(1).WSP_1500.WDR.WBR 0        ;# Transmit all 0
  iApply
  iRead  WSP_1500S.WDR.WBR 0              ;# Receive all 0
  iWrite WSP(1).WSP_1500.WDR.WBR(0) 0b1   ;# Broadcast single 1
  iApply

  set i 1
  while {$i < 8} {
    iRead  WSP_1500S.WDR.WBR  0           ;# Expect background of all 0
    set pos [expr {$i - 1}]
    iRead  WSP_1500S.WDR.WBR($pos) 1      ;# Expect 1 from WSP(1)
    set pos [expr {$pos +8}]
    iRead  WSP_1500S.WDR.WBR($pos) 1      ;# Expect 1 from WSP(2)
    set pos [expr {$pos +8}]
    iRead  WSP_1500S.WDR.WBR($pos) 1      ;# Expect 1 from WSP(3)
    iWrite WSP(1).WSP_1500.WDR.WBR 0      ;# Set write background
    iWrite WSP(1).WSP_1500.WDR.WBR($i) 0b1      ;# Write Walking 1
    iApply
    set i [expr {$i + 1}]
    }
  # read last driven values
  iRead  WSP_1500S.WDR.WBR  0             ;# Expect background of all 0
  set pos [expr {$i - 1}]
  iRead  WSP_1500S.WDR.WBR($pos) 1        ;# Expect 1 from WSP(1)
  set pos [expr {$pos +8}]
  iRead  WSP_1500S.WDR.WBR($pos) 1        ;# Expect 1 from WSP(2)
  set pos [expr {$pos +8}]
  iRead  WSP_1500S.WDR.WBR($pos) 1        ;# Expect 1 from WSP(3)
  iApply

}

<EOF>
```

Annex E

(informative)

Example iApply execution flow

The following pseudo-code illustrates the execution flow of the PDL **iApply** command. This does not show any of the processing necessary to verify the syntax and semantics of the command words or other housekeeping. It is strictly informative, and any apparent discrepancy between this pseudo-code and the rules of C.3.7.2 must be resolved in favor of the rules.

This pseudo-code assumes that when **iRead** and **iWrite** (or **iScan**) commands update the write and expect data, then each field that has been updated is known. It also assumes that a mapping is maintained from a TDR to the currently preferred instruction for scanning that TDR, called the **iSetInstruction** table, below.

NOTE 1—*previous* TDR, instruction, and **iApply** arguments are initialized to the appropriate values (null for **iApply** arguments) after a reset and before the first **iApply** command. Rule references are all from C.3.7.2.

```
-------------------------
PROCEDURE iApply

(Clear the local fail flag) (Rule d1)
IF (there are no iRead/iWrite/iScan commands since last iApply)
THEN
  CALL Scan-DR (using accumulated write and default expect data) (Rule c)
ELSE
  IF (there is NOT a single TDR containing all fields to be scanned)
  THEN
    ERROR: multiple TDRs selected. (Rule e);
    EXIT
  END IF
END IF
(Determine current TDR from fields to be scanned)
IF (all fields to be scanned are currently included/selected)
THEN
  CALL Scan-DR (using accumulated write and expect data) (Rule d)
ELSE
  FOR (each mutually exclusive segment -- at least once) (Rule a) (See NOTE)
    WHILE (segment(s) are not yet selected)
      IF (all segment(s) not yet selected are controlled (domain, selection)
          in the same TDR)
      THEN
        CALL Scan-DR (using domain and selection values) (Rule a)
        IF (excludable segment)
        THEN
          UNTIL (all SEGSEL are "ready-to-scan")
            CALL Scan-DR (read "ready-to-scan" bits)
          END UNTIL
        END IF
      ELSE
        Save current TDR name (Rule b)
        FOR (each TDR containing selection fields controlling a segment
             to be scanned)
          (Set current TDR name to TDR containing segment control(s))
          CALL Scan-DR (using domain and selection values,
                        expect only defaults)
```

```
          IF (excludable segment)
          THEN
            UNTIL (all SEGSEL are "ready-to-scan")
              CALL Scan-DR (read "ready-to-scan" bits)
            END UNTIL
          END IF
        END FOR
        (Restore current TDR name to saved TDR name)
      END IF
    END WHILE
    CALL Scan-DR (using accumulated write and expect data)
  END FOR
END IF
(Reset write data for any PULSE0 or PULSE1 cells to '0') (Rule i)
(Reset Expect data to register defaults) [Rule c) of C.3.7.1]
(Copy current TDR and instruction names, and iApply arguments to
      previous TDR and instruction names, and iApply arguments)
END PROCEDURE

-------------------------------------------

PROCEDURE Scan-DR

(Determine current instruction from current TDR using iSetInstruction
      Table) (Rule p)
IF (current instruction is different than previous instruction)
THEN
  IF (previous -shiftPause is set)
  THEN
    ERROR: changing instructions not allowed in paused shift (Rule m)
    EXIT
  END IF
  IF (current -skipRTI is set)
  THEN
    (Perform IR scan using current instruction to select TDR (Rule p);
        do not pass through the Run-Test/Idle TAP controller state (Rule h))
  ELSE
    (Perform IR scan using current instruction to select TDR (Rule p))
  END IF
  (Copy current TDR and instruction names to previous TDR and
        instruction names)
END IF
(Evaluate all constraints that apply to the current TDR and take
      appropriate action) (Rule f)
IF (previous -shiftPause is set)
THEN
  (Go from Pause-DR to Exit2-DR to Shift-DR) (Rule l)
ELSE
  IF (current -skipRTI is set)
  THEN
    (Do not pass through the Run-Test/Idle TAP controller state en route
          to the Shift-DR state) (Rule h)
  ELSE
    (Go to the Shift-DR state)
  END IF
END IF
BEGIN (single operation)
```

```
  (Perform DR scan, writing accumulated write data and comparing expect data,
      setting both fail flags as required) (Rule d2, d3, d4)
  IF (PDL1)
  THEN
    (Capture returned scan data and fail data) (Rule d5)
  END IF
END BEGIN
IF (current -shiftPause is set)
THEN
  (Go from Shift-DR to Exit1-DR to Pause-DR) (Rule k)
ELSE
  IF (current -skipRTI is set)
  THEN
    (Do not pass through the Run-Test/Idle TAP controller state en route
            to the ending state) (Rule h)
  ELSE
    (Go to ending state)
  END IF
END IF
END PROCEDURE
```

NOTE 2—This description does not attempt to deal with the ambiguity in results that can occur when there are multiple mutually exclusive but also mutually dependent segments to be written and/or read by a single **iApply** command. In this situation, the captured values may change depending on the order of writing and reading the segments, which is arbitrary by rule. To use this pseudo-code as shown, the user would have to code the PDL to eliminate the order dependency. Alternatively, a tool provider could enhance this pseudo-code to eliminate the order dependency by repeating the selection and scan sequence twice. The first scan sequence would write the previous data (without the **iWrite** updates) and read the data with the new **iRead** values (and storing the captured values in PDL-1); and the second scan sequence would write the new **iWrite** values and discard the captured values. This ensures that all register segments are read (and the captured values stored) before any segments are written with new data, and it provides exactly the same results as if all the mutually exclusive and dependent segments had instead been serially connected in a single TDR and scanned once, but at the cost of twice as many scans where mutually exclusive selectable segments are involved. A third possibility is for a tool provider to issue a warning upon detecting when multiple mutually exclusive segments are read and written by a single **iApply** command.