

IEEE Std 1532™-2002
(Revision of IEEE Std 1532-2001)

1532™

**IEEE Standard for In-System
Configuration of Programmable
Devices**

IEEE Computer Society

Sponsored by the
Test Technology Technical Council



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

10 January 2003

Print: SH95071
PDF: SS95071

IEEE Standard for In-System Configuration of Programmable Devices

Sponsor

Test Technology Technical Council
of the
IEEE Computer Society

Approved 11 December 2002

IEEE-SA Standards Board

Abstract: The communication protocol described by IEEE Std 1149.1™-2001 (Standard Test Access Port and Boundary-Scan Architecture) has been adopted by this standard for providing standardized programming access and methodology for programmable integrated circuit devices. Devices that implement this standard will first be compliant with IEEE Std 1149.1-2001, which is used for testing purposes. A device, or set of devices, implementing this standard can be programmed (written), read back, erased, and verified, singly or concurrently, with a standardized set of resources. Sample implementation and application details (which are not part of this standard) are included for illustrative purposes.

Keywords: complex programmable logic device (CPLD), erasable programmable read-only memory (EPROM), field programmable gate array (FPGA), Flash RAM, in-system (or in situ) configuration (ISC), programmable device

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

Copyright © 2003 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 10 January 2003. Printed in the United States of America.

Print: ISBN 0-7381-3507-0 SH95071
PDF: ISBN 0-7381-3508-9 SS95071

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The 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 the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an IEEE Standard is wholly voluntary. The 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 this, or any other IEEE Standard document.

The IEEE does not warrant or represent the accuracy or content of the material contained herein, 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 herein 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 five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, 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 this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received 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 interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Note: 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 with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

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.

Introduction

(This introduction is not part of IEEE Std 1532-2002, IEEE Standard for In-System Configuration of Programmable Devices.)

The development of this standard began with a preliminary meeting held 19 April 1996. There, a group of Programmable Device vendors and users discussed the possibility of standardizing the programming process for these devices. Subsequent meetings were held regularly and agreement was developed for basing the programming process upon the IEEE Std 1149.1 communication protocol, because many of these devices were expected to support IEEE Std 1149.1 for testing purposes. This group adopted as its mission:

To define, document, and promote the use of a standardized process and methodology for implementing programming capabilities within programmable integrated circuit devices, utilizing (and compatible with) the IEEE Std 1149.1 communication protocol. This standard would allow the programming of one or more compliant devices concurrently, while mounted on a board or embedded in a system, known as “In-System Configuration.” Concurrent programming may often result in significant programming time efficiencies. The In-System feature would address the need to configure or reconfigure, read back, verify or erase programmable devices after they have been installed by a manufacturing process. This eliminates handling damage and the need for manufacturing steps and inventory management related to preprogrammed devices.

This work received the support of the IEEE Std 1149.1 Working Group which ultimately urged that this effort become its own standard within the Test Technology Technical Committee of the IEEE Computer Society, in July of 1998. The initially approved version of this standard described the necessary hardware elements for compliance. By making this available quickly and early, the development of compliant silicon was facilitated and its availability accelerated. The next revision of this standard described the software elements necessary for full implementation of this standard. The software elements described include extensions to the BSDL file that describe the programming algorithm and hardware implementation of this standard. Additionally a new data file format has been described. This revision further extends the standard by adding support for devices that require adaptive programming algorithms along with some slight modifications to the initial standard that will help to better meet user’s needs.

Following is a list of participants in the In-System Configuration Working Group of the Test Technology Technical Council. Voting members at the time of publication are marked with an asterisk (*).

Neil G. Jacobson, *Chair*
David Bonnet, *Vice-Chair*
Dennis Lia, *Secretary*
Mark Moyer, *Editor*
Kenneth P. Parker, *Editor Emeritus*

Turgut Abacioglu	P. D. Felen	David Paul
John Andrews	Judy Chen Feng	Bruce Peterson
Don Arnold	Dan Gravitz	Etienne Racine
Bill Aronson	Jack Griffin*	Bapcha Ramamurthy
Marco Barahmand	Kurt Guntheroth	Doug Raymond
Jason Baumbach*	Peter Hansen	Jacky Renaux
Glenn Baxter	Steve Harrison	Robert Russell
Menachem Blasberg	Bernie Haverkamp	David Lee Rutledge
Terry Borroz	Alan Herrmann*	Bobbie Saville
Stephen Brown	Brad Ishihara*	Sanjeev Sharma*
Ian Bryant	John Joy	Steve Stark
Steven Butkovich	Raj Kamdar	Chung Sun
Tapan Chakraborty	Doug Kay	Jerry Talen
Sung Chung*	Arthur Khu	Howard Tang*
C. J. Clark*	Peter Larsen	Elke Tiemeyer
Ray Dellecker *	Bertrand Leigh	Douglas Way*
Yadvindra Dhami	Frank Lien	Thomas Wenzel
Wim Driessen*	Keith Loftstrom	Mike Westermeier
Alon Drory	Patrick McHugh	Kevin Wible
Ted Eaton*	G. Meijerink	Bill Wilkie
Rob Eccles	Matthew Murphy	Carmy Yellin
Peter van den Eijnden	Gary O'Brien	Dan Young
Bernardo Elayda		Piotr Zbysinski

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

Morris Balamut	Frans De Jong	Yinghua Min
Roger Bennetts	Heiko Ehrenberg	Mark Moyer
David Bonnett	Peter van den Eijnden	Benoit Nadeau-Dostie
Terry Borroz	Kazumi Hatayama	Bruce Peterson
Tapan Chakraborty	Neil G. Jacobson	Mike Ricchetti
Keith Chow	Adam Ley	Gordon Robinson
C. J. Clark	Dennis Lia	Robert Russell
Luis E. Cordova		Douglas Way

When the IEEE-SA Standards Board approved this standard on 11 December 2002, it had the following membership:

James T. Carlo, *Chair*
James H. Gurney, *Vice Chair*
Judith Gorman, *Secretary*

Sid Bennett
H. Stephen Berger
Clyde R. Camp
Richard DeBlasio
Harold E. Epstein
Julian Forster*
Howard M. Frazier

Toshio Fukuda
Arnold M. Greenspan
Raymond Hapeman
Donald M. Heirman
Richard H. Hulett
Lowell G. Johnson
Joseph L. Koepfinger*
Peter H. Lips

Nader Mehravari
Daleep C. Mohla
William J. Moylan
Malcolm V. Thaden
Geoffrey O. Thompson
Howard L. Wolfman
Don Wright

*Member Emeritus

Also included is the following nonvoting IEEE-SA Standards Board liaison:

Alan Cookson, *NIST Representative*
Satish K. Aggarwal, *NRC Representative*

Noelle D. Humenick
IEEE Standards Project Editor

Contents

1.	Overview	1
1.1	Scope	1
1.2	Organization of the standard	1
1.3	Context	2
1.4	Objectives	2
2.	References	3
3.	Definitions and acronyms	3
3.1	Definitions	3
3.2	Acronyms	9
4.	System modal states	9
4.1	Definition of system modal states	10
4.2	Description of system modal state transitions	12
4.3	System modal state transition timing	15
4.4	System I/O pin behavior	18
5.	ISC architecture	22
5.1	Overview	22
5.2	Specification	35
5.3	Description	38
6.	ISC Instructions	39
6.1	General properties of ISC instructions	39
6.2	Mandatory ISC instructions	46
6.3	Optional ISC programming instructions	52
6.4	Optional ISC program control/security instructions	59
6.5	Optional address/data access instructions	63
6.6	Requirements for IEEE Std 1149.1-2001 masters and ISC programming algorithms	68
6.7	Adding new instructions for ISC programming support	70
7.	Data registers	70
7.1	Device identification register	71
7.2	Bypass register	72
7.3	Boundary scan register	72
7.4	ISC_Default	72
7.5	ISC_Config	73
7.6	ISC_PData	74
7.7	ISC_RData	74
7.8	ISC_Sector	74
7.9	ISC_Info	74
7.10	ISC_Data	75
7.11	ISC_Address	75
7.12	ISC_Inc	75

8.	Conformance and documentation requirements	75
8.1	Conformance	75
8.2	Documentation	76
8.3	BSDL extension for ISC description (STD_1532_2002)	78
8.4	ISC extension structure	79
8.5	ISC BSDL attribute definitions.....	81
9.	Data file format	110
9.1	Data file syntax specification	110
9.2	Description	113
9.3	Mapping data onto a flow	114
9.4	Merging flows for concurrent programming	115
9.5	Data file transmission CRC calculation	117
Annex A	(informative) Use of device features	119
Annex B	(informative) ISC programming flows	123
Annex C	(informative) Example ISC BSDL/data file	134

IEEE Standard for In-System Configuration of Programmable Devices

1. Overview

1.1 Scope

This standard describes a series of mandatory and optional programming instructions and associated data registers that define a standard methodology for accessing and configuring programmable devices. These additional registers and instructions extend the capabilities of devices that comply with IEEE Std 1149.1™-2001 such that the IEEE Std 1149.1-2001 Test Access Port may be used for configuration activities. A data description format and extensions to Boundary-Scan Description Language (BSDL) are also specified that provide for the development of standardized automation tools for device programming.

1.2 Organization of the standard

This standard is divided into nine clauses and three annexes, as follows:

- Clause 1 describes the scope, context, objectives, and organization of this standard.
- Clause 2 provides references to other IEEE documentation needed to understand this document.
- Clause 3 defines terms and acronyms used in this standard.
- Clause 4 defines the variations of system state that are unique to programmable logic devices.
- Clause 5 describes the general architecture of an in-system configurable (ISC) device.
- Clause 6 describes the ISC instruction set.
- Clause 7 describes the ISC register set and the operation of each register.
- Clause 8 describes conformance and documentation requirements.
- Clause 9 provides a data file format to be used to describe data to be downloaded to an ISC device.
- Annex A provides a device designer with a rationale for implementing various features of this standard.
- Annex B describes programming flows for combinations of devices compliant with this standard.
- Annex C gives a coordinated example of a BSDL file and data file for a hypothetical ISC device.

1.3 Context

Figure 1 shows a printed circuit board containing many types of devices. Of these, some could be compliant with IEEE Std 1149.1-2001 for the support of testing activities. Three devices labeled “ISC” are in-system configurable, meaning they are programmable devices that may not have their programming states loaded into them before the board has been assembled. These devices may be programmed after they have been attached to the board, via a physical and logical protocol. The physical protocol for programming utilizes the IEEE Std 1149.1-2001 Test Access Port (TAP) pins, which allows a multiplicity of devices (both ISC and conventional) to be accessed serially with four or five wires. The logical protocol is based upon new instructions added to the IEEE Std 1149.1-2001 instruction set and is the subject of this standard document.

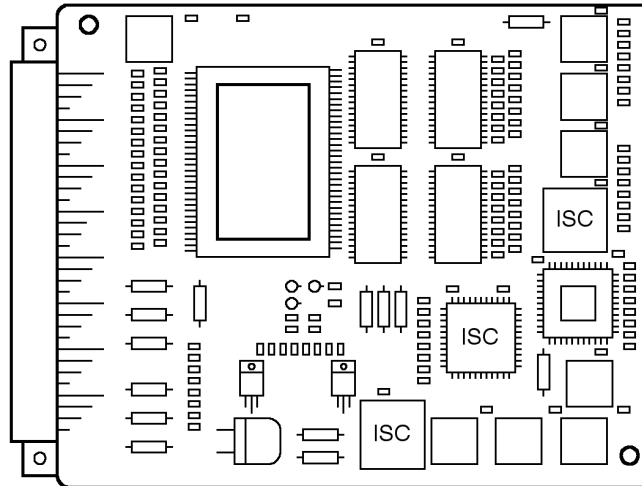


Figure 1—A printed circuit board containing a variety of components, including three ISC integrated circuits

The ISC devices also contain testability circuitry compliant with IEEE Std 1149.1-2001. This allows them to participate in manufacturing tests that detect and diagnose faults such as open solder joints and shorts either before or after they are programmed.

1.4 Objectives

The objective of this standard is to enhance user access to IEEE Std 1149.1-2001-based programmable devices. Since such devices are built on the foundation of the IEEE Std 1149.1-2001 Test Access Port and TAP Controller state machine, they shall already comply with that standard.

Devices that adhere to this standard will realize significant savings in software, product development, and manufacturing costs related to ISC. New tools will not need to be developed to support new vendor parts with ISC features as they arrive in the marketplace. These parts can be integrated immediately into existing systems and tools.

In addition, tools based on this standard may be developed upon the existing IEEE Std 1149.1-2001-based tool sets and knowledge base, rapidly incorporating the features of this standard.

This standard also allows for multiple devices from independent vendors to be programmed concurrently. Development time and manufacturing costs (in terms of programming time) can thereby be significantly reduced, even in the multivendor environment. In the past, one family of devices from a single vendor may have been incompatible with another family from that same vendor, with respect to the programming process.

The standard also serves board and system designers by providing a consistent framework for system initialization, stabilization, and event initiation, which is important for robust system design.

The standard will realize the following additional benefits:

- It will serve as the development model for new entrants to the ISC industry and guarantee their immediate access to all the latest tools, development systems, and test and manufacturing flows and processes.
- It will facilitate innovations potentially including applications in reconfigurable computing, emulation, diagnostics, and field upgrading that would be usable across vendors.
- It will encourage tool vendors targeting the programmable device marketplace that there will be sufficient return on their investment to warrant tool development.

2. References

This standard shall be used in conjunction with the following standards. When the following standards are superseded by an approved revision, the revision shall apply.

IEEE Std 1149.1-2001, IEEE Standard Test Access Port and Boundary-Scan Architecture.^{1,2}

3. Definitions and acronyms

3.1 Definitions

For the purposes of this standard, the following terms and definitions apply. See also IEEE Std 1149.1-2001 for definitions of terms used in this document.

3.1.1 adaptive programming (algorithm): This term may be used when the minimum burn time may vary among identical devices. To optimize the overall programming time, the device can be queried for the status of the programming operation, which will be given by the optional “Operation Success Indication.” If the programming operation has been successfully completed, then the next step in the programming process can commence. If the device has not completed its programming operation to satisfaction, then it will continue until completion or until the maximum time has been met. This process will require the looping of one or more instructions for successful completion. *Contrast: deterministic programming.*

NOTE—See 5.1.10.

3.1.2 addressing: Programmable devices usually contain one or several arrays of memory locations. Each memory location has an address, which must be provided when reading or programming memory content.

3.1.3 asymmetrically writeable device: A device with program memory that can only be written to one state. For example, a “1” may be written into a location containing a “0”, but not vice-versa. Such devices must be erased to the blank state before new program data can be written. *Contrast: symmetrically writeable device.*

¹IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (<http://standards.ieee.org/>).

²The IEEE standards or products referred to in Clause 2 are trademarks owned by the Institute of Electrical and Electronics Engineers, Incorporated.

3.1.4 blank: Programmable devices usually contain one or several arrays of memory locations. The data placed in these locations constitute the device program. Many devices are asymmetrical with respect to writing data into these locations. For example, a “1” may be written into a location containing a “0”, but not vice-versa. Thus, a “blank” device is filled with a field of overwriteable states. For symmetrically writeable devices, the “blank” state is usually defined as a field of states that implement no programmable functionality. *See also:* **device program**.

3.1.5 burn time: An interval of time during which a set (or subset) of device programming bits is presented to on-chip programming functions, which become the definition of the logic functions the device will subsequently implement. In some older technologies, tiny on-chip fuses are vaporized (hence the term *burn*), which requires a minimum time to achieve. Burn time is technology dependent and may be nearly instantaneous in some devices or require many milliseconds in others. Synonyms include bake time, cook time, and soak time.

3.1.6 chain: Two or more IEEE Std 1149.1-2001-compliant devices that share common test clock (TCK) and test mode select (TMS) signals and are linked together with the test data output (TDO) signal of one device connected to the test data input (TDI) signal of a subsequent device. The overall chain may be accessed by a 4 or 5-wire test access port (TAP). *See also:* **test access port (TAP)**.

3.1.7 checksum: This integer is created by a mathematical process that operates on a stream of data. The integer for this data is then published so that it can be compared to subsequently recalculated integers for that same data. If the data have not been changed or disturbed, the published integer should match a newly calculated integer. The mathematical process is often based upon binary addition (to some modulus) or logical Exclusive OR of bytes or words. There are well-known flaws with such processes that can allow damaged data streams to masquerade as good by having the same checksum. *See also:* **CRC cyclic redundant codes**.

3.1.8 completion: In the context of an ISC instruction, this process occurs when the sequence of activities performed during execution has finished. However, the requirements for execution may not be met, or error conditions may be encountered such that successful completion is not possible. These events constitute unsuccessful completion. *See also:* **Execute**.

3.1.9 complex programmable logic device (CPLD): A high-capacity programmable logic device (PLD) usually provided in packages with 40 or more pins, essentially an array of programmable logic blocks interconnected by a routing matrix.

3.1.10 concurrentization: The process of making two or more devices work concurrently.

3.1.11 concurrent operation: The ability to conduct programmable device operations in two or more devices in the same IEEE Std 1149.1-2001 chain concurrently in time. During this process, all devices in a chain operate continuously in the *Run-Test/Idle* TAP Controller state. In particular, the burn times of the devices can be made to elapse concurrently.

3.1.12 CRC cyclic redundant codes: This code can be used as an improved checksum process. Such a code (properly constructed) is much less prone to aliasing a bad data stream with a good stream because it encodes the positional information of 1s and 0s in a stream as well as their frequency. (*See also:* **checksum**.)

3.1.13 deterministic programming (algorithm): A device implements this algorithm when the minimum required burn time to satisfy programming operations is documented in the BSD file and does not change for like devices. *Contrast:* **adaptive programming**.

3.1.14 device program: A sequence of bits that, when loaded into a programmable device via a protocol, subsequently define the functions it will implement. *See also:* **volatile program** and **nonvolatile program**.

3.1.15 displaced: An instruction in the IEEE Std 1149.1-2001 instruction register is displaced when another instruction is shifted in and then a falling edge of TCK in the *Update-IR* TAP Controller state occurs. Note that any instruction besides IDCODE (the default instruction jammed into the IEEE Std 1149.1-2001 instruction register) will be displaced immediately if the TAP Controller enters the *Test-Logic-Reset* TAP Controller state.

3.1.16 duration: In the context of an ISC instruction, a period of time that starts when a TAP Controller first enters the *Run-Test/Idle* TAP Controller state. This period of time is given for each ISC instruction, which specifies how much time (typically a minimum) the action(s) performed by that instruction will take to complete.

3.1.17 dynamic concurrentization: This process merges the programming algorithms of multiple devices at run time and is based on the intermediate results of the programming operations. With this process, decisions regarding when information is shifted into or out of a device or when operations are executed are made immediately before the action takes place. *Contrast:* **static concurrentization**.

3.1.18 electrically erasable programmable read-only memory (EEPROM): Programmable read-only memories have evolved from mask-programmable devices to devices that were erasable by exposure to ultraviolet light (EPROM) to devices that can be electrically erased.

3.1.19 enabled protection mechanism: ISC devices may be implemented with the ability to protect the memory content from erasure, reading or writing, in various combinations. When any of these mechanisms is available, it can be enabled, thus preventing the action it can protect against. *See also:* **protect**.

NOTE—See 5.1.11.

3.1.20 erasable programmable logic device (EPLD): At first an EPROM-based device, the term now includes flash and EEPROM-based devices as well.

3.1.21 erase: An operation that removes any previously installed device program by changing programmed bits to their unprogrammed state. Erasure may be implemented in several ways. For example, there may be a “bulk erase” function that erases the entire device program in one action. There may be selective, or addressed erasure that only erases a specified “word” of the program. Between these two extremes, there may be a “sector” erase feature that erases a group of words. This operation may be inhibited if the device contains a security feature enabled by a Protect operation.

3.1.22 execute: In the context of an ISC instruction, execution occurs after the TAP Controller has proceeded to the *Run-Test/Idle* TAP Controller state. In this state, the instruction performs a specific function that may contain multiple steps or a set of activities. After a minimum amount of time (or clocking of TCK) has occurred, the instruction completes execution. Additional time (or clocking of TCK) beyond this minimum requirement does not affect the results of the instruction’s execution.

3.1.23 explicit addressing: Address information is provided by an external algorithm when operating on memory content.

3.1.24 field-programmable gate array (FPGA): A programmable alternative to a masked gate array using a channeled programmable interconnect matrix to join blocks of configurable logic. Physical characteristics such as line length and number of fuses are not known until routing. This may lead to some timing unpredictability. The advantage comes in logic density and flexibility.

3.1.25 fixed system pin: A system pin with an I/O function that is independent of configuration information programmed into the device and has not been designated as an ISC system pin by the device designer.

3.1.26 flash random access memory (Flash RAM): A memory technology similar to EEPROM but implemented with a single transistor rather than two.

3.1.27 flow: A logical sequence of ISC instructions that performs a basic programming function such as erasing programmable memory, verifying it, or programming it.

3.1.28 generic array logic (GAL): An earlier programmable logic technology still widely used, featuring a fixed OR array and programmable AND array.

3.1.29 implicit addressing: Address information is provided by an internally managed process when operating on memory content. For example, successive memory locations may be sequentially accessed by an “auto-increment” addressing process supported by the device such that an external algorithm does not have to be aware of how memory locations are organized.

3.1.30 in-system configuration (ISC): The ability to load, read back, or erase programming bits in a programmable logic device after it has been attached to a printed circuit board. *See also:* **device program**.

3.1.31 ISC_Disable_Erase signal: An optional signal used to indicate that program data erasure is not allowed.

NOTE—See 5.1.11.3.

3.1.32 ISC_Disable_Program signal: An optional signal used to indicate that program data writing is not allowed.

NOTE—See 5.1.11.2.

3.1.33 ISC_Disable_Read signal: An optional signal used to indicate that program data reading is not allowed.

NOTE—See 5.1.11.1.

3.1.34 ISC_Done signal: A signal that is used to indicate that a device program has been successfully written to the satisfaction of an external algorithm. When this signal is cleared, the device is considered to be unprogrammed, even if physically, the device does contain programmed bits. When this signal is set, the device is considered to be programmed and ready for operation. In volatile devices, this signal is automatically cleared at power-up. In nonvolatile devices, this signal maintains its state during nonpowered intervals. This signal can be set by programming operations described in this standard, and it may be protected by a security function implemented within the device.

3.1.35 ISC_Enabled signal: A signal that is used to indicate that a device has been successfully conditioned for programming activity. This signal is cleared during power-up for both volatile and nonvolatile devices.

NOTE—See Clause 4.

3.1.36 ISC system pin: A system pin with an I/O function that is defined by configuration information programmed into the device, or has been designated as an ISC system pin by the device designer.

3.1.37 load: In the context of an instruction, an instruction is loaded when opcode bits present in the IEEE Std 1149.1-2001 instruction shift register are updated into the parallel-hold portion of the register by passing through the *Update-IR* TAP Controller state. At the time an instruction is loaded, its data register is selected for data shifting. *Contrast:* **execute**.

3.1.38 mission logic: *See:* **system logic**.

3.1.39 mission pin: *See:* **system pin.**

3.1.40 noninvasive instruction: Instructions from IEEE Std 1149.1-2001 (including BYPASS, SAMPLE, PRELOAD, IDCODE, and USERCODE) that maintain the device in system mode rather than in test mode. Thus, these instructions do not interfere with the system behavior of the device. *Contrast:* **pin-permission instruction.**

3.1.41 nontest instruction: Instructions that bring the device out of test mode and put it into unprogrammed, operational, or ISC accessed mode. These instructions can be, but are not limited to, any of the noninvasive instructions or any of the instructions defined in this standard. *Contrast:* **test instruction.**

3.1.42 nonvolatile program: Device programming that is retained by a device even when power is removed and later restored.

3.1.43 nullified: Proper ISC instruction operation is predicated on the device being in the ISC accessed system modal state while it is being executed. If this condition is not met, the instruction's operation is nullified. A nullified ISC instruction will still place a defined register in the shift path and shift bits through it as required by IEEE Std 1149.1-2001, but the ISC actions defined for the instruction by this standard will not be completed and the result of its execution is undefined.

NOTE—See Clause 4.

3.1.44 optimize: ISC programming activities can be optimized for time efficiency. There are two forms of optimization. First, if a device has a specific instruction loaded, it may be redundant to reload this instruction. This is called “IR optimization.” Second, when a register is targeted between TDI and TDO, “DR optimization” can be employed by shifting out captured data while new update data are being shifted in, rather than first shifting out captured data and then shifting in new update data. When a chain of devices is operated, optimization can be performed, but each device in the chain must be considered. For example, if two of three devices in subsequent activities will be executing the same instruction but the third will need a new instruction loaded, then IR optimization is not possible because loading the third device's instruction forces the reload of the first two.

3.1.45 pin-permission instruction: Instructions from IEEE Std 1149.1-2001 (including EXTEST, INTEST, CLAMP, HIGHZ, and RUNBIST) and instructions from this standard that place the device in test or programming modes. These instructions take over control of the system I/O pins. *Contrast:* **noninvasive instruction.**

3.1.46 program: An operation that installs externally supplied device programming bits (or a portion thereof) within a programmable device. This operation may be inhibited if the device contains a security feature enabled by a Protect operation.

3.1.47 programmable array logic (PAL): An earlier programmable logic technology still widely used, featuring a fixed OR array and programmable AND array.

3.1.48 programmable logic devices: This family of devices includes complex programmable logic device (CPLD), electrically erasable programmable read-only memory (EEPROM), erasable programmable logic device (EPLD), field-programmable gate array (FPGA), flash random access memory, and programmable array logic.

3.1.49 protect: Traditionally, an operation that prevents a subsequent read operation from being able to read out a device program. This is a security feature used to prevent copying or deciphering of a device program. In some implementations, writing one or more “security” bits located somewhere in the device program enables protection. Once set, these bits block subsequent read operations. Security bits are usually only

cleared by erasing the device program. More recently, this operation has been expanded to include the prevention of unwanted programming or erasure operations.

3.1.50 read: An operation that allows a device program to be read back out from a programmed device. This operation may be inhibited if the device contains a security feature enabled by a protect operation.

3.1.51 signal: A named signal is used to denote a binary condition used to choose between two outcomes. It is deliberately “implementation-vague” so as not to constrain a designer’s freedom. For example, a signal may be implemented as a bit in a register, or derived as a function of several bits, or as a one-hot decode from an instruction decoder, as fits the designer’s needs. This should not be confused with I/O signals on device pins that may be high, low, or disabled.

3.1.52 simple programmable logic device (SPLD): A family of programmable logic devices, including programmable array logic (PAL), generic array logic (GAL), and the erasable programmable logic device (EPLD).

3.1.53 static concurrentization: This process merges the programming algorithms prior to any operation being executed on the chain of devices. All decisions regarding optimization of the programming algorithms have been made before run time so that the run-time for the flow is completely predictable.
Contrast: **dynamic concurrentization.**

3.1.54 symmetrically writeable device: A device with program memory that can be written to either state. These devices are often Static RAM-based. *Contrast:* **asymmetrically writeable device.**

3.1.55 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. In the case of programmable devices, the system logic may not be defined until after the device is programmed.
See also: **device program.** *Syn.:* **mission logic.**

3.1.56 system modal states: IEEE Std 1149.1-2001 defines two modes, “system” mode and “test” mode. Devices that conform to this standard have a system mode that is divided into four modal states. This reflects the fact that programmable devices, at various times in their operation, may (A) be in “Operational” mode containing system logic, may (B) be in an “Unprogrammed” mode with blank logic configuration memory, may (C) be in “ISC Accessed” mode for programming or accessing logic configuration memory, or may (D) be in “ISC Complete” mode when accessing configuration memory is complete before returning to the operational mode.

NOTE—See Clause 4.

3.1.57 system pin: A component pin that feeds, or is fed from, the on-chip system logic. In the case of programmable devices, the system logic may not be defined until after the device is programmed. In devices that contain conventional logic in addition to programmable logic, system pins may be classified as ISC system pins and fixed system pins. *Syn.:* **mission pin.** *See also:* **device program.**

NOTE—See 3.1.25 and 3.1.36.

3.1.58 TAP Controller: A 16-state finite state machine used to control the activities of the test access port and the test logic within the device. *See also:* **Test Access Port (TAP).**

3.1.59 test access port (TAP): Defined by IEEE Std 1149.1-2001. This consists of four (optionally five) device pins dedicated for use by IEEE Std 1149.1-2001 and programming usage defined by this standard. TAP pins are not system pins.

3.1.60 test instruction: Instructions that include any pin-permission instruction that puts the device into test mode but not into any of the programming modes.

3.1.61 verify: An operation similar to reading that compares an installed device program (or a portion thereof) to a program supplied externally. If a discrepancy is found, an error indication is given. This operation may be inhibited if the device contains a security feature enabled by a protect operation.

3.1.62 volatile program: Device programming that is lost when a device has power removed from it.

3.1.63 write: *See:* **program.**

3.2 Acronyms

The following acronyms are used within this standard.

BSDL	Boundary-Scan Description Language (see IEEE Std 1149.1-2001)
BSR	Boundary-Scan Register
CPLD	Complex Programmable Logic Device
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPLD	Electrically Programmable Logic Device
EPROM	Erasable Programmable Read-Only Memory
FPGA	Field-Programmable Gate Array
GAL	Generic Array Logic
IC	Integrated Circuit
I/O	Input/Output
IOM	Input/Output Module
ISC	In-System Configuration
OST	Operation Success Test
PAL	Programmable Array Logic
PROM	Programmable Read-Only Memory
SRAM	Static Random Access Memory
TAP	IEEE Std 1149.1-2001 Test Access Port
TCK	The Test Clock input pin contained in the IEEE Std 1149.1-2001 TAP
TDI	The Test Data Input pin contained in the IEEE Std 1149.1-2001 TAP
TDO	The Test Data Output pin contained in the IEEE Std 1149.1-2001 TAP
TLR	Test-Logic-Reset, the reset state of the IEEE Std 1149.1-2001 TAP Controller state diagram
TMS	The Test Mode Select input pin contained in the IEEE Std 1149.1-2001 TAP
TRST*	The Test Reset input pin contained in the IEEE Std 1149.1-2001 TAP (this pin is optional)

NOTE—Following the convention used in IEEE Std 1149.1-2001, pins or signals with a trailing asterisk (“*”) in their names are active low true signals.

4. System modal states

A conventional (i.e., nonprogrammable) device compliant with IEEE Std 1149.1-2001 has a “system” mode and a “test” mode. In system mode, the device performs its intended mission even while executing “noninvasive” nontest instructions. In test mode, test instructions control the device’s system I/O pins for testing purposes. A compliant IEEE Std 1149.1-2001 device passes from system mode to test mode when certain IEEE Std 1149.1-2001 instructions become active, as shown in Figure 2.

Devices that are compliant with IEEE Std 1149.1-2001 power-up in system mode. Noninvasive nontest IEEE Std 1149.1-2001 instructions include BYPASS, PRELOAD, SAMPLE, IDCODE, and USERCODE.

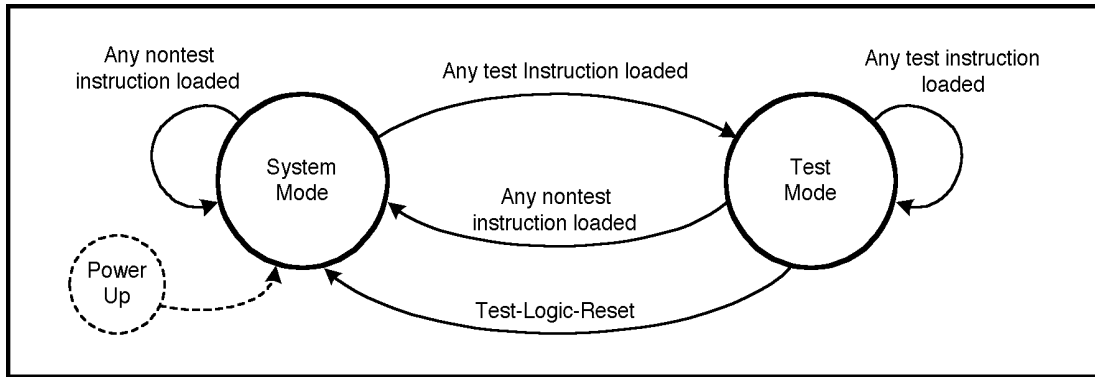


Figure 2—Transitions from system mode to test mode for a conventional device compliant with IEEE Std 1149.1-2001

Test instructions include EXTEST, INTEST, CLAMP, HIGHZ, and RUNBIST. Entering the Test-Logic-Reset TAP Controller state will also force the device into system mode.

NOTES

1—After a device has been in test mode, it may return to system mode but be unable to resume logical operation in concert with other devices it is connected to because while in test mode, all communication with these devices was severed. It will likely be necessary to issue a system reset to restore proper system operation.

2—User-defined instructions may create new modes of operation that are not depicted in Figure 2.

Unlike conventional integrated circuit devices, a programmable logic device may have many different user-defined operation behaviors depending on the pattern that is currently programmed into the device. To precisely define device behavior before, during, and after programming, the system mode of an ISC device has four different “system modal states”, all comprising the “system” mode of the device as defined by IEEE Std 1149.1-2001. (These are defined in 4.1.) The four system modal states define the “system” behavior of system I/O pins when noninvasive instructions from IEEE Std 1149.1-2001 are loaded. (Test instructions already have I/O behavior specified by IEEE Std 1149.1-2001, and ISC instructions defined herein also have prescribed I/O behavior.)

NOTE—See 5.1.1 for a classification of system pins. For the discussion in this clause, assume that all system I/O pins are associated with programmable logic.

Well-defined and predictable I/O behavior is of paramount importance to designers of boards and systems who must ensure that the power-up behavior and system initialization processes of their designs are robust. This is of particular concern where a design is programmable. It must exhibit robust behavior before, during, and after the programming process.

4.1 Definition of system modal states

The four system modal states (see Figure 3) of a programmable device conforming to this standard are as follows:

- **Unprogrammed:** In this system modal state, an ISC device may be blank (see 3.1.4) or incompletely programmed. In this system modal state and while a noninvasive IEEE Std 1149.1-2001 instruction (see 3.1.40) is loaded, the device’s system I/O pins are disabled such that they cannot contend with output drivers that may exist in other devices connected to the ISC device. Any ISC instruction (excepting ISC_ENABLE) will have its ISC operation nullified (see rule b) of 6.1.1) if executed in

- this state. Upon power-up, volatile devices, blank devices, and incompletely programmed nonvolatile devices should reside in this state.
- **ISC Accessed:** In this system modal state, ISC instructions may be used (when permitted) to read, write, verify, protect, or erase the device. In this system modal state and while a noninvasive IEEE Std 1149.1-2001 instruction is loaded, the device's system I/O pins are either disabled or specifically controlled by the content of the device's Boundary-Scan Register as determined in rule d) of 6.1.1.
 - **ISC Complete:** This system modal state exists so an external algorithm can control transitions to the Unprogrammed or Operational system modal state after ISC operations are complete. This system modal state is entered as a result of executing the ISC_DISABLE instruction. This system modal state is temporary; if any instruction other than ISC_DISABLE is loaded into the instruction register (at the falling edge of TCK in the Update-IR TAP Controller state), this system modal state is exited. Thus, this system modal state cannot exist when a noninvasive IEEE Std 1149.1-2001 instruction is present in the instruction register.
 - **Operational:** In this system modal state, the device is ready for its “operational” mode defined by its programming. In this system modal state and while a noninvasive IEEE Std 1149.1-2001 instruction is loaded, the device's system I/O pins take on their programmed system nature of inputs, outputs, bidirectionals, or unused. Any ISC instruction (excepting ISC_ENABLE) will have its operation nullified (see rule b) of 6.1.1) if executed in this system modal state. Successfully programmed nonvolatile devices, upon power-up, should reside in this system modal state.

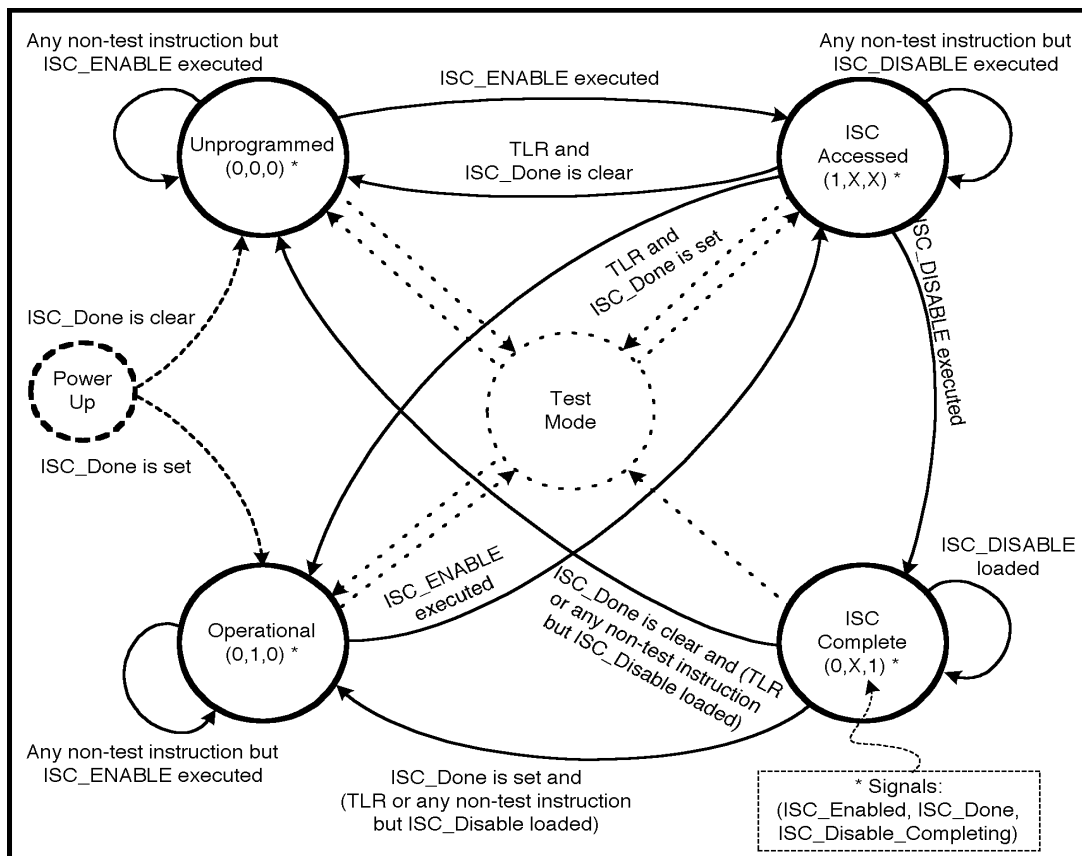


Figure 3—Four system modal states that comprise the “system mode” of operation for an ISC device and transitions. (A description of the transitions to and from test mode appears in Figure 4. The three signals are described in 4.2 and 4.3.)

In all system modal states, all public IEEE Std 1149.1-2001 instructions are available and supported per IEEE Std 1149.1-2001. This means that IEEE Std 1149.1-2001 chain integrity or infrastructure testing can be accomplished at any time regardless of the system modal state of any ISC device.

NOTES

1—Certain operations conducted in test mode (for example, EXTEST-based interconnection tests) may not be electrically compatible with programming activities (or vice versa) conducted in the ISC Accessed system modal state due to noise or power supply considerations. However, it may be quite reasonable to read out a device USERCODE from the ISC Accessed system_modal state and later return to an ISC instruction.

2—There may be restrictions that should be observed for a given ISC device regarding the interruption of a programming flow in order to execute a non-ISC instruction.

It is important to realize that a noninvasive IEEE Std 1149.1-2001 instruction such as BYPASS may demonstrate up to three distinct system I/O behaviors depending on the system modal state of the device, as follows:

- a) All pins disabled (Unprogrammed system modal state)
- b) All pins either disabled or specifically controlled by the content of the device's Boundary-Scan Register as determined in rule d) of 6.1.1 (ISC Accessed system modal state)
- c) All pins behaving in their programmed mission mode (Operational system modal state).

This modal behavior conforms to IEEE Std 1149.1-2001 because of the definition of system modal states given by this standard. The plurality of system I/O pin behaviors allows a device to move from executing an ISC instruction to executing a noninvasive IEEE Std 1149.1-2001 instruction with a seamless and predictable system I/O behavior.

NOTE—It can also lead to a potentially surprising result. For example, when a device is in the ISC Accessed system modal state and its pins are controlled by the Boundary-Scan Register, the PRELOAD instruction may be used to change the content of the Boundary-Scan Register. This change, upon the falling edge of TCK in the Update-DR TAP Controller state, will appear at the device I/O pins, simulating the operation of EXTEST. This is because the system behavior of this device in the ISC Accessed system modal state is defined to have the property of the system I/Os being controlled by the Boundary-Scan Register.

4.2 Description of system modal state transitions

Figure 3 shows the four system modal states for an ISC device and how transitions between these states occur. (Note that Figure 3 also shows transitions to test mode. These are further described in Figure 4.) System modal state transitions are governed by the following three factors:

- The instruction loaded into the IEEE Std 1149.1-2001 instruction register. This instruction becomes effective on the falling edge of TCK in the *Update-IR* TAP Controller state. ISC instructions will define system I/O behavior at this time, but they also require some time spent in the *Run-Test/Idle* TAP Controller state for their operations to complete.
- The state of an internal “ISC_Done” signal. (In volatile devices, this signal is cleared at power-up. In nonvolatile devices, this signal is retained during loss of power.) The ISC_Done signal is set when the device is programmed to the satisfaction of an external ISC programming algorithm. This indicates the device is no longer blank. The ISC_Done signal is set one of three ways; by executing the ISC_DISABLE instruction to completion with a specific data pattern set in the ISC_Config register, by execution of the (optional) ISC_PROGRAM_DONE instruction, or by programming one or more specific bits in the program memory array. The ISC_Done signal also determines the functionality of the device system I/O pins on power-up. If the ISC_Done signal is clear, the device I/O pins are disabled.

NOTE—The ISC_Done signal may be subject to protection by a security feature that may prevent changes to its state.

- The state of an internal volatile signal called the “ISC_Enabled” signal. This signal is set by the execution of the ISC_ENABLE instruction and may be cleared by subsequent events. In both volatile and nonvolatile devices, this signal is cleared at power-up.

Two of these factors (the ISC_DISABLE instruction is loaded and ISC_Enabled is set) are utilized to derive a third signal named “ISC_Disable_Completing.” This signal is used to indicate that the ISC Complete system modal state has been entered legally, that is, from the ISC Accessed system modal state by action of the ISC_DISABLE instruction. The derivation of ISC_Disable_Completing is conceptualized in 4.3, although any number of other derivations are possible.

The Unprogrammed system modal state is entered upon power up if the ISC_Done signal is cleared. The only way to exit the Unprogrammed system modal state is by loading and executing the ISC_ENABLE instruction (see 6.2.1), which sets the ISC_Enabled signal upon successful completion. Setting the ISC_Enabled signal causes a transition to the ISC Accessed system modal state. If the ISC_ENABLE instruction is not successfully completed, the ISC_Enabled signal is not set and the device stays in the Unprogrammed system modal state.

NOTE—If while in the Unprogrammed or Operational system modal state an ISC instruction other than ISC_ENABLE is loaded, it will not operate to fulfill its intended mission (see rule b) of 6.1.1); that is, its operation will be nullified.

Once the ISC Accessed system modal state is entered (that is, the ISC_Enabled signal is set), all other ISC instructions are enabled to operate as documented in Clause 6. The preferred way to exit the ISC Accessed system modal state is by executing the ISC_DISABLE instruction, which clears the ISC_Enabled signal (and potentially sets the ISC_Done signal) and causes a transition to the ISC Complete system modal state. If the Test-Logic-Reset TAP Controller state is entered while in the ISC Accessed system modal state, then the currently loaded instruction is aborted and either the Unprogrammed system modal state is entered if ISC_Done is clear, or the Operational system modal state is entered if ISC_Done is set. In typical ISC programming flows, a sequence of operations, such as erasing, writing, reading, and protecting would all be done successively within the ISC Accessed system modal state.

NOTES

1—If the ISC Accessed system modal state is left abnormally (by traveling to the *Test-Logic-Reset* TAP Controller State) while the ISC_Done signal is set, it is possible that an improperly programmed device could be set to the Operational system modal state. It is the responsibility of external controlling algorithms to avoid this. In some technologies, it is possible that the ISC_Done signal can only be cleared by an erase operation. In such devices, it will be important to construct the erase operation to clear the ISC_Done signal early in the erasure process. This reduces the chance that terminating the erasure process before completion will leave the ISC_Done signal set.

2—The ISC_Disable_Completing signal is also generated by the loading of ISC_DISABLE as seen in 4.3.

In the ISC Complete system modal state, the device is ready to transition to either the Operational or Unprogrammed system modal states, depending on the state of the ISC_Done signal. The transition from ISC Complete is accomplished by loading any instruction other than ISC_DISABLE into the instruction register. One way to do this is to cause the TAP Controller to enter the *Test-Logic-Reset* TAP Controller state, which forces the IDCODE instruction into the instruction register, which in turn forces a transition to either the Unprogrammed or the Operational system modal state depending on the state of the ISC_Done signal. However, if several ISC devices in a chain are in the ISC Complete system modal state, then it may be desirable to sequence their transition to the Operational system modal state in a particular order. Loading one ISC device with BYPASS while reloading the others with ISC_DISABLE, and so progressing until all devices are loaded with BYPASS would accomplish this objective.

Once a device enters the Operational system modal state from the ISC Complete system modal state, it is fully programmed and capable of performing its mission function. System I/O pins are subject to the

instruction that was used to displace the ISC_DISABLE instruction. If it is a test instruction (like EXTEST), then that instruction controls the system I/O pins. If the instruction was an ISC instruction (like ISC_ENABLE), then it will control the system I/O pins per rule d) of 6.1.1. If the new instruction was a noninvasive IEEE Std 1149.1-2001 instruction (like BYPASS), the system I/O pins are governed by the programming of the device and take on their intended system function. The only way to leave the Operational system modal state (excepting a transition into test mode) is to load and execute the ISC_ENABLE instruction, thus returning to the ISC Accessed system modal state. This might be done to reprogram, verify, or erase a device's programming at some future point in time.

NOTES

1—If while in the Operational system modal state an ISC instruction other than ISC_ENABLE is loaded, it will not operate to fulfill its intended mission (see rule b) of 6.1.1); that is, its operation will be nullified.

2—If the Operational system modal state is entered abnormally (in response to the *Test-Logic-Reset* TAP Controller state), the device may not be capable of performing its mission function. If this abnormal transition is performed, it is the responsibility of the programming algorithm to subsequently erase the ISC_Done signal so that the device no longer appears to be properly programmed.

Transitions to and from test mode in Figure 3 were shown with unlabeled dotted lines to avoid clutter. Figure 4 shows how transitions between the system modal states and test mode occur. From any system modal state, the transition to test mode occurs when a test instruction (EXTEST, INTEST, CLAMP, HIGHZ, or RUNBIST) is loaded.

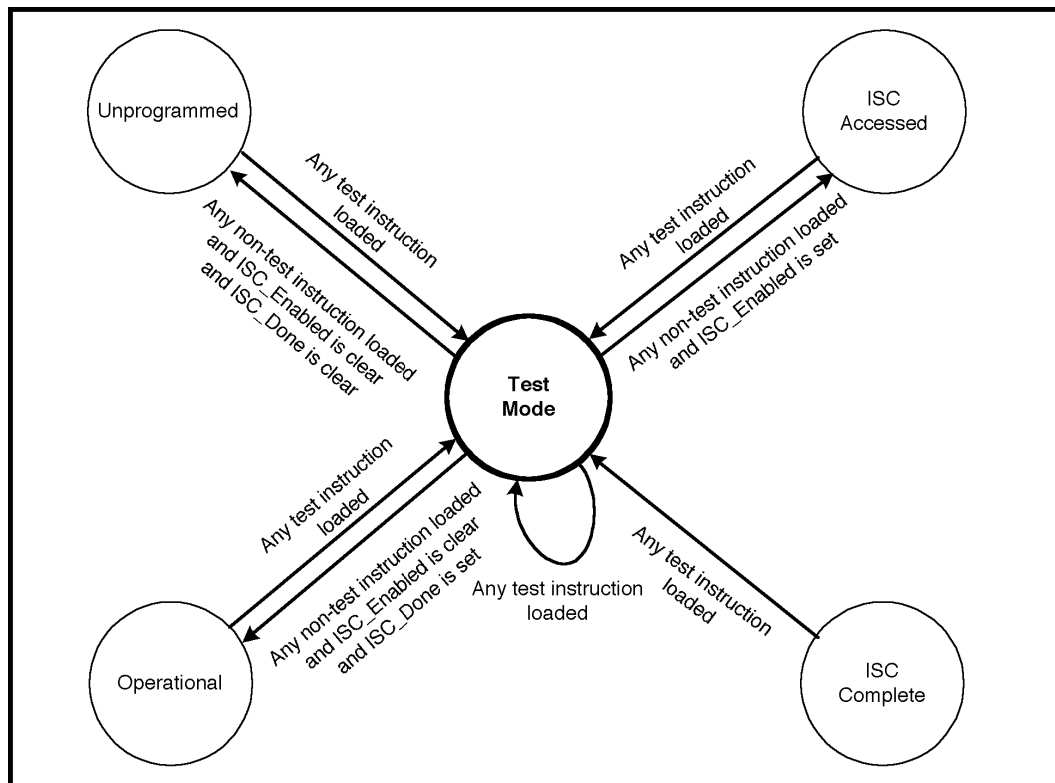


Figure 4—Transitions to and from system modal states and test mode. (A description of the transitions among the four modal states appears in Figure 3.)

There are three ways to return to a system modal state from test mode. All of these require that a nontest instruction (BYPASS, PRELOAD, SAMPLE, IDCODE, USERCODE) or an ISC instruction described by this standard be loaded and are qualified by the states of the ISC_Enabled and ISC_Done signals.

The design of a particular ISC device may not be able to support all the transitions shown in Figure 4. For example, depending on implementation technology, when in the ISC Accessed system modal state, an ISC device may generate high programming voltages internally. While these voltages are present, it may be difficult to ensure correct operation of the device if it also makes transitions on system I/O pins. In all cases, the designer is required to ensure that the IEEE Std 1149.1-2001 instructions operate as specified (see permission s) of 5.2 and related notes) and to document instruction sequences that should be avoided. (See 8.2.)

4.3 System modal state transition timing

Transitions on output pins and between system modal states occur at prescribed times shown in Figure 5. (This drawing does not show time to scale, but represents the ordering of events.)

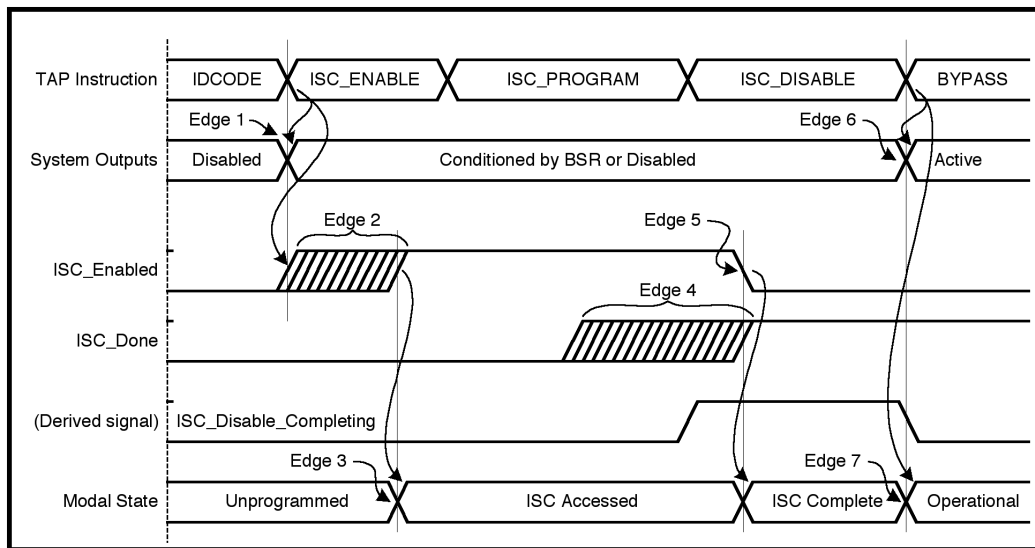


Figure 5—Waveforms (not to scale) for signals that control system outputs and system modal state transitions

NOTE—In Figure 5, the convention is used that “set” is high and “clear” is low.

There are seven edges of interest shown in Figure 5. They are as follows:

- Edge 1: System I/O pins that were disabled take on the state given by rule d) of 6.1.1 when the instruction changes from IDCODE to ISC_ENABLE (triggered by the falling edge of TCK in the *Update-IR* TAP Controller state).
- Edge 2: The ISC_Enabled signal is asserted somewhere between the time the instruction changes from IDCODE to ISC_ENABLE (triggered by the falling edge of TCK in the *Update-IR* TAP Controller state) and the last rising edge of TCK in the *Run-Test/Idle* TAP Controller state that satisfies the minimum duration specification (see rule i) of 6.1.1) for ISC_ENABLE.
- Edge 3: The system modal state changes from Unprogrammed to ISC Accessed when the ISC_Enabled signal is asserted.
- Edge 4: The ISC_Done signal is set either by programming (with the ISC_PROGRAM instruction) one or more bits in the program data area, by the execution of the ISC_DISABLE instruction, or by the execution of the ISC_PROGRAM_DONE instruction. The ISC_Done signal is set no later than

the last rising edge of TCK in the *Run-Test/Idle* Tap Controller state that satisfies the minimum duration specification [see rule i) of 6.1.1 or burn time, rule c) of 6.4.1.1 and rule d) of 6.2.3.1) as appropriate] for the instruction setting *ISC_Done*.

- Edge 5: The *ISC_Enabled* signal is cleared during execution of the *ISC_DISABLE* instruction. This occurs on the last rising edge of TCK in the *Run-Test/Idle* Tap Controller state that satisfies the minimum duration specification (see rule i) of 6.1.1) for *ISC_DISABLE*. This causes the system modal state to change from *ISC Accessed* to *ISC Complete*.

NOTE—In a preferred embodiment, the *ISC_Done* and *ISC_Enabled* signals (at edges 4 and 5) transition at the same time. If *ISC_Done* should transition before *ISC_Enabled*, it is possible for a transition to the *Test-Logic-Reset* TAP Controller state to occur between them, resulting in possible confusion about what *ISC_Done* means. (It will be the responsibility of external programming algorithms to avoid this confusion.) Simultaneous transition of these two signals requires that the *ISC_Done* signal be controlled by the *ISC_DISABLE* instruction.

- Edge 6: When any non-ISC instruction is loaded into the IEEE Std 1149.1-2001 instruction register (upon the falling edge of TCK in the *Update-IR* TAP Controller state), the system outputs take on their programmed nature.
- Edge 7: When any instruction other than *ISC_DISABLE* is loaded into the IEEE Std 1149.1-2001 instruction register (upon the falling edge of TCK in the *Update-IR* TAP Controller state), the system modal state changes from *ISC Complete* to *Operational*.

The case where *ISC_DISABLE* is executed and then *ISC_ENABLE* is immediately executed is shown in Figure 6. When this occurs, the system modal state changes briefly to *Operational*, but because the IEEE Std 1149.1-2001 instruction register still contains an ISC instruction, the system I/O pins remain in the state governed by rule d) of 6.1.1. Once the *ISC_ENABLE* instruction completes, the device is again in the *ISC Accessed* system modal state at Edge 8. Thus, if several ISC operations are executed back-to-back (for example, programming, followed by read-back), the device I/O pins will not leave their quiescent state even when these ISC operations are bounded by *ISC_ENABLE* and *ISC_DISABLE* instructions.

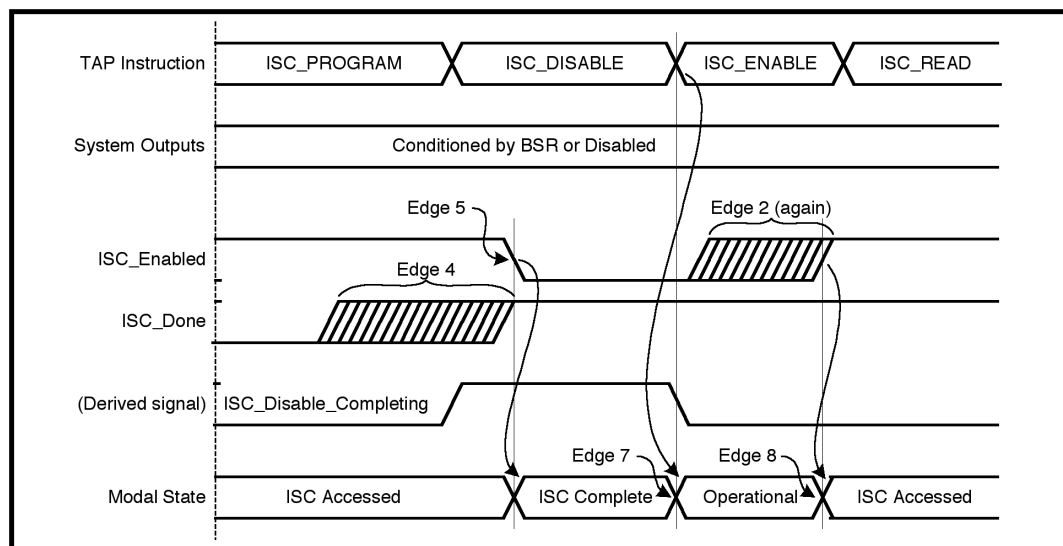


Figure 6—Waveforms observed when *ISC_ENABLE* is executed upon completion of *ISC_DISABLE*

The derived signal *ISC_Disable_Completing*, first referenced as a control signal in Figure 3 and as its waveforms appear in Figure 5 and Figure 6, may at first appear to be a signal from the instruction decoder (call it *ISC_Disable_Decoded*) that is set whenever the *ISC_DISABLE* instruction is loaded in the IEEE Std 1149.1-2001 instruction register. However, it must be a function of both *ISC_Disable_Decoded*

and ISC_Enabled as shown in Figure 7. This reflects the fact that ISC_Disable_Completing indicates that the ISC_DISABLE instruction was loaded properly while the device was in the ISC Accessed system modal state (that is, the ISC_Enabled signal is set).

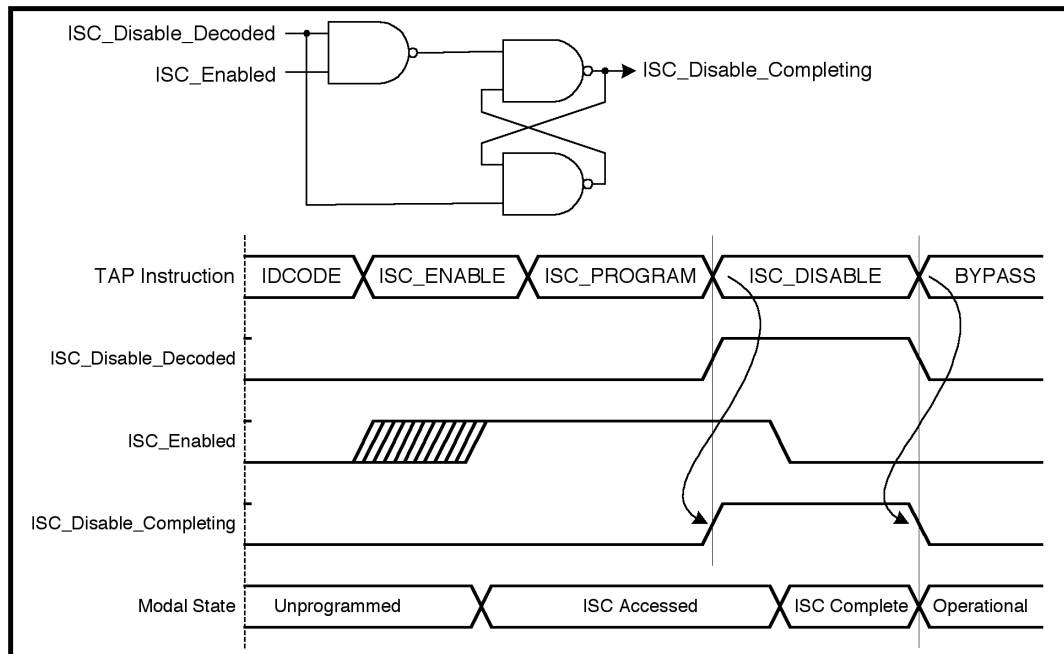


Figure 7—Derivation of the ISC_Disable_Completing signal. The circuit shown is conceptual as there are other implementations with the same result

The case where the ISC_DISABLE instruction is loaded when the device is not in the ISC Accessed system modal state is shown in Figure 8. When this occurs, the system modal state does not change.

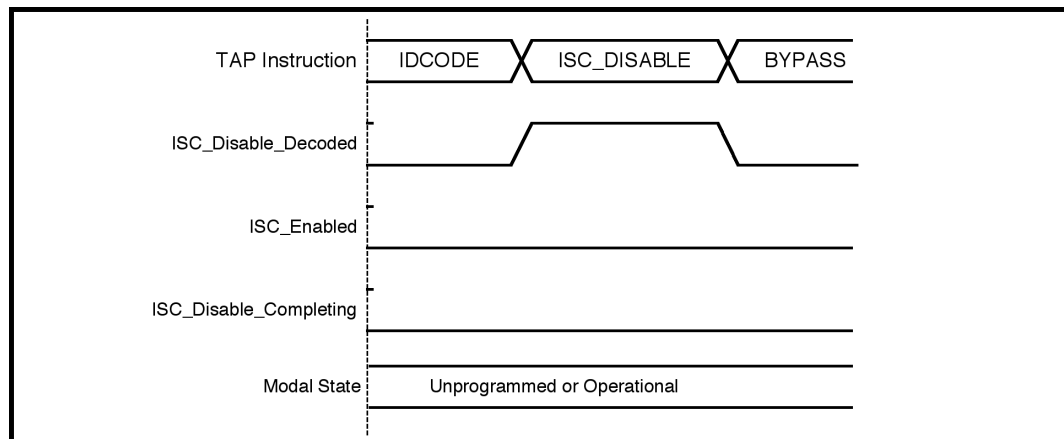


Figure 8—The system modal state remains unchanged, and the ISC_Disable_Completing signal remains cleared when the ISC_DISABLE instruction is not properly prefaced with the ISC_ENABLE instruction

4.4 System I/O pin behavior

System I/O pins of a device compliant with rules given herein have behavior summarized as follows:

- If an ISC instruction defined by this standard (see Clause 6) is loaded in the instruction register, the system I/O pins are controlled per rule d) of 6.1.1 regardless of what system modal state the device may be in.

NOTE—An instruction is loaded in the instruction register on the falling edge of TCK in the *Update-IR* TAP Controller state.

- If an IEEE Std 1149.1-2001 test mode instruction (EXTEST, INTEST, RUNBIST, CLAMP, HIGHZ) is loaded in the instruction register, the system I/O pins are controlled as defined by that standard regardless of what system modal state the device may have been in before the test mode instruction was loaded.
- If an IEEE Std 1149.1-2001 nontest mode instruction (BYPASS, IDCODE, USERCODE, SAMPLE, PRELOAD) is loaded in the instruction register, the system I/O pin behavior is determined by the system modal state of the device when the instruction is loaded.

NOTE—The system modal state of the device cannot be changed by the execution of non-ISC instructions. The system modal state may, however, change from ISC Complete to Operational or Unprogrammed upon loading any instruction other than ISC_DISABLE. The system modal state of the device can be changed by passing to the *Test-Logic-Reset* TAP Controller state, potentially resulting in a change of system I/O pin behavior.

It is when an IEEE Std 1149.1-2001 nontest mode instruction is loaded in an ISC device that there is a choice of behaviors for the system I/O pins. These are summarized as follows:

- If the device is in the Unprogrammed system modal state, the system I/O pins are disabled.
- If the device is in the ISC Accessed system modal state, the system I/O pins are governed by rule d) of 6.1.1.
- If the device is in the Operational system modal state, the system I/O pins operate as determined by the programming of the device.

NOTE—If at the point of time an IEEE Std 1149.1-2001 nontest instruction was loaded into the device it was in the ISC Complete system modal state, the device transitions to the Operational or Unprogrammed system modal state because the ISC_DISABLE instruction has been displaced.

When a device complying with this standard is in the Operational system modal state, it behaves as if it was a conventional device that contains both IEEE Std 1149.1-2001 testability circuitry and mission circuitry as shown in Figure 9. The typical programmable device contains a series of Input-Output Modules (IOMs) that contain Boundary-Scan Register cells and a three-state output driver. The Boundary-Scan Register cells shown have the required Capture Flip-flop (C) and an Update Latch (U). The IOM depicted is fully bidirectional.

NOTE—The IOM shown in Figure 9 can be implemented more efficiently with a single bidirectional data cell structure (see IEEE Std 1149.1-2001) rather than the two-cell data structure shown here for the sake of simplicity. This will eliminate 33% of the cells, saving silicon area as well as test data shift time and test program storage space.

When a device is configured, a given system I/O pin may be fully bidirectional, or limited to being an input, output, or unconnected pin. This limiting may be accomplished by tying the control cell input (the lowest cell in the IOM in Figure 9) to a fixed “0” or “1”. When a pin-permission instruction such as EXTEST is loaded into the instruction register, the driver data and control signals are derived from the Update latches (U) as IEEE Std 1149.1-2001 requires.

NOTE—If a pin’s programming is “unused,” it should at the very least still be monitored by the input Boundary-Scan Register cell of the IOM. The IOM could be treated as fully bidirectional, however, to allow a testing procedure to

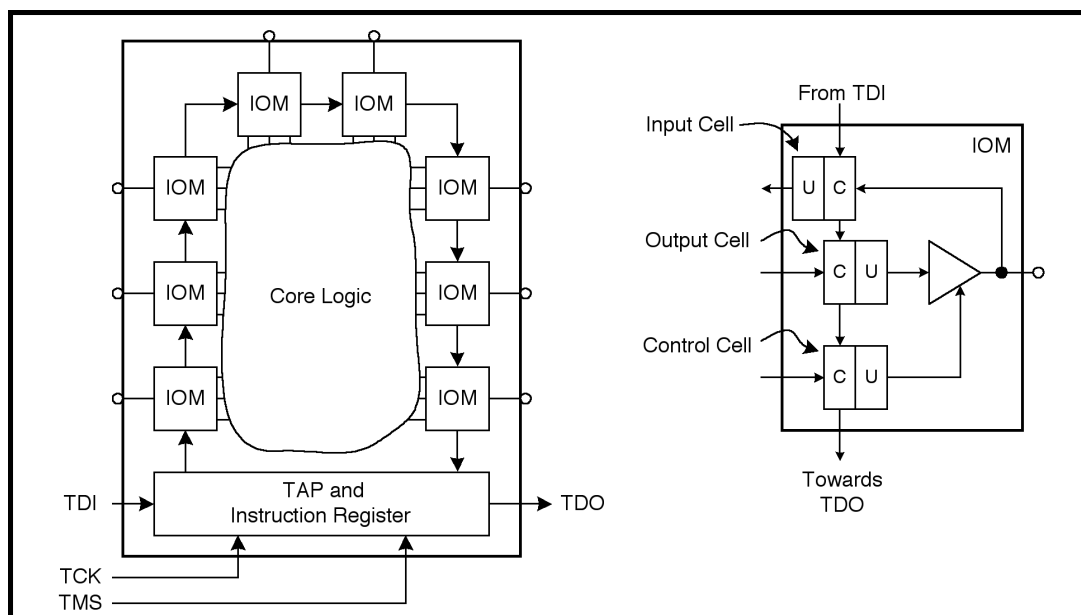


Figure 9—A programmed device and its connection to the Boundary-Scan Register

discover whether shorts exist from “unused” pins to other pins. This would allow the testing procedure to establish an identity (using the IOM driver) for the unused pin that the IOM receiver would verify. Testing even the unused pins for shorts will help discover manufacturing process problems and ensure the integrity of the system in the event the programming is subsequently changed such that unused pins are later used.

When a programmable device is in the Unprogrammed system modal state and an IEEE Std 1149.1-2001 nontest instruction is loaded, then there is, by definition, no core logic as shown in Figure 10. In order to meet the requirement (see rule e) of 6.1.1) that a device in the Unprogrammed system modal state have all of its drivers inactive, the control cell in the IOM should produce a logic level that will disable the driver, a “0” in this case. When a programmable device is in the ISC Accessed or ISC Complete system modal states, then the device drivers should either be controlled by the Boundary-Scan Register or be inactive as governed by rule d) of 6.1.1. Example data cell designs are shown in Figure 11. Two example control cell designs that meet the two options for rule c) of 6.1.1 are shown in Figure 12 and Figure 13.

The data cell design shown in Figure 11(A), used when the device is intended to exhibit the HIGHZ-like behavior (see rule d) of 6.1.1) is identical to a “generic” Boundary-Scan Register data cell design from IEEE Std 1149.1-2001, which may be used for inputs or outputs. It does not make use of any of the specialized ISC signals defined by this standard. The “Mode” signal is generated by the instruction decoder as described in the table within Figure 11. Note that Mode changes on the falling edge of TCK in the *Update-IR* TAP Controller state.

NOTE—The cell shown in Figure 11(A) is known as “BC_1” in BSDL. (See IEEE 1149.1-2001, B.9 “The Standard VHDL Package STD_1149_1_2001” and B.10.2 “Cell description constants” for information on describing Boundary-Scan Register cells.)

The data cell design shown in Figure 11(B) uses ISC_Enabled to ensure that when the device is intended to exhibit the CLAMP-like behavior (see rule d) of 6.1.1), it will pass the Boundary-Scan Register content to the pin.

NOTE—With respect to the BSDL description (see IEEE Std 1149.1-2001) of the data cell design depicted in Figure 11(B), while materially different in the output multiplexing structure from the cell design called BC_1 in BSDL, this design is still equivalent to BC_1. This is because the BC_1 structure describes the capture behavior of the cell and not the multiplexing structure on its output.

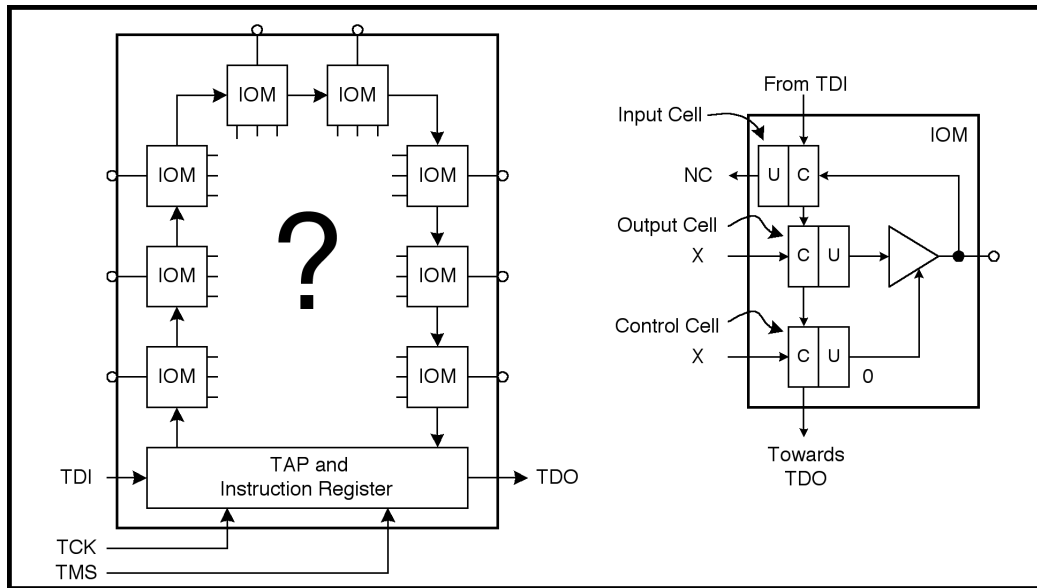


Figure 10—An unprogrammed device and how the unconnected IOM signals should be

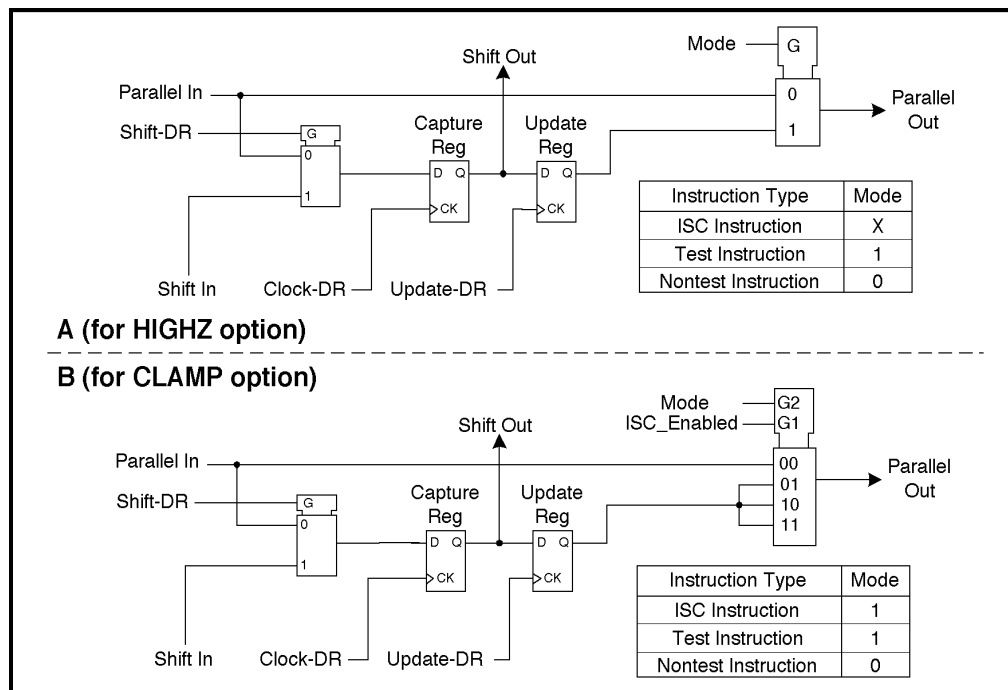


Figure 11—Possible data cell designs that meet the requirements of this standard

The control cell design incorporates ISC signals in its design to govern when an IOM driver is enabled or disabled, and where its source of control data comes from.

Figure 12 shows a possible control cell design that will disable the system I/O pins whenever an ISC instruction is loaded, or whenever a nontest instruction is loaded and the device is in the ISC Accessed system modal state. Loading a test mode instruction will allow the Boundary-Scan Register to control the

state of the driver enable. The “Mode1” and “Mode2” signals are generated by the instruction decoder as described in the table within Figure 12. Note that mode signals change on the falling edge of TCK in the *Update-IR* TAP Controller state, but that the ISC_Enabled and ISC_Done signals change at other times governed by the execution of certain ISC instructions. Thus, these signals cannot change while an ISC instruction is not loaded.

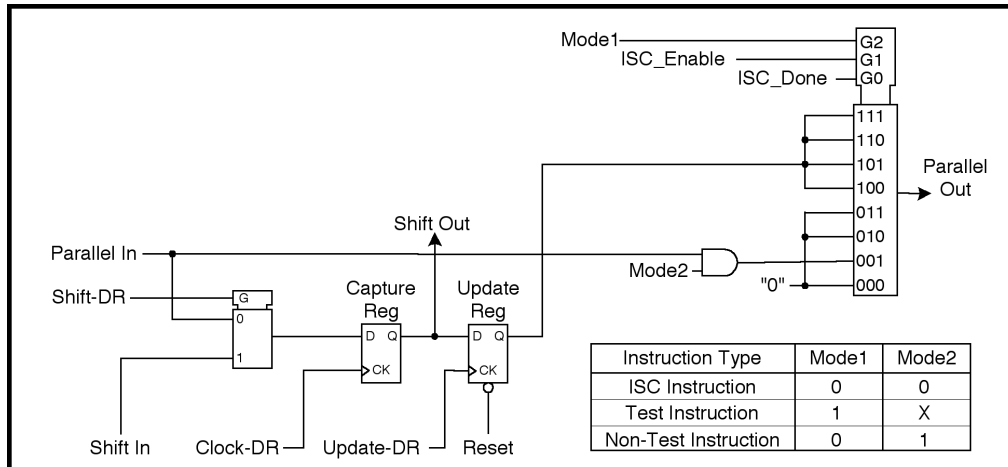


Figure 12—A possible (unoptimized) control cell design that satisfies rule d) of 6.1.1 by disabling the system I/O pins

The control cell design in Figure 13 is similar except that it uses the Boundary-Scan Register to control the output driver enable whenever an ISC (or test mode) instruction is loaded, as well as when a nontest instruction is loaded and the device in the ISC Accessed system modal state. This design has only one “Mode” signal which is derived per the table in Figure 13.

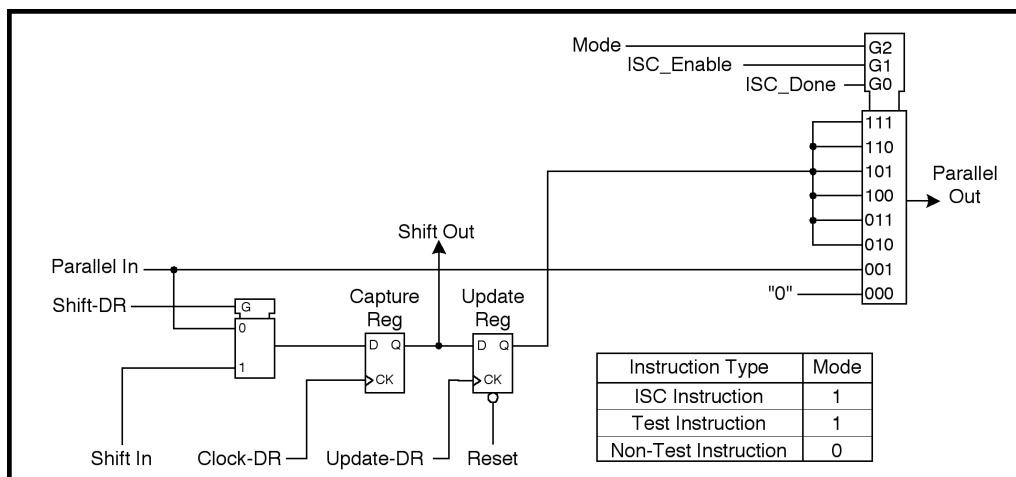


Figure 13—A possible (unoptimized) control cell design that satisfies rule d) of 6.1.1 by using the Boundary-Scan Register to control the system I/O pin state

Both of the designs in Figure 12 and Figure 13 have been shown in an unoptimized form for clarity. A designer would probably spend some effort on optimizing the speed of a single path (Parallel In to Parallel Out). All other paths are low speed and may be compromised accordingly.

NOTE—With respect to the BSDL description (see IEEE Std 1149.1-2001) of the control cell designs depicted in Figure 12 and Figure 13, while materially different in the output multiplexing structure from the cell design called BC_1 in BSDL, these designs are still equivalent to BC_1. This is because the BC_1 structure describes the capture behavior of the cell and not the multiplexing structure on its output.

5. ISC architecture

5.1 Overview

5.1.1 Classification of system pins

This standard recognizes that multiple functions may be placed upon a single integrated circuit die and that some of these may be programmable logic functions addressable by this standard and others may be conventional (nonprogrammable) electronic structures. In some cases, it may be important to differentiate System I/O pins based on recognition that they are connected to programmable functionality or fixed functionality.

The differentiation of System I/O pins is based on whether their I/O functions (input, output, bidirectional) are programmable by the ISC programming process. This excludes other types of pin configuration capability that could exist for pins that may determine parametric factors such as logic voltage levels, drive currents, pull-up or pull-down functions, and so on, that any pin may also possess.

5.1.1.1 In-system configurable system pins

ISC system pins are those pins whose I/O functions (input, output, or bidirectional) are established by configuration information installed by the ISC programming process. Further, additional I/O pins chosen by the device designer may also be designated as ISC system pins. All ISC system pin behavior is determined by the system modal state as discussed in 4.4. Except where specifically noted in this document, the term “system pin” will be synonymous with “ISC system pin.”

5.1.1.2 Fixed system pins

Fixed system pins are those whose I/O functions (input, output, or bidirectional) are not determined by configuration information installed by the ISC programming process, and have not been designated as ISC system pins by the device designer. Fixed system pin behavior is completely specified by IEEE Std 1149.1-2001. This, along with the fact that ISC instructions operate within system modal states (see 4.1) implies that fixed system pins may be fully operational while ISC programming activities are in progress. To reiterate, fixed system pins are in system mode when ISC instructions are being executed.

NOTE—Thus, the designation of fixed and ISC system pins may be predicated solely on their relationship with the programmable portion of a device, or may be arbitrarily designated as ISC system pins at the discretion of the device designer. This allows the designer to determine the I/O behavior of any pin on the device while it is being programmed. This has important consequences as the following example shows.

An example of a device with both fixed and ISC system pins is shown in Figure 14. In this device, a microcontroller and a programmable logic device have been integrated onto the same IC substrate. When first powered up (and with no programming present within the programmable section), the ISC system pins are disabled and the fixed pins are active, allowing the microcontroller to perform its function. Subsequently, through the TAP, the programmable portion of the device can be configured. While this is going on, the ISC system pins are either disabled or fixed by a pattern held in the Boundary-Scan Register. The fixed pins, at the same time, are free to perform system functions. Finally, when the ISC portion of the device is fully configured, the ISC system pins take on their system function when the ISC_DISABLE instruction is

displaced. Notice that the controller that actually programmed the ISC portion of the device may have been the microcontroller on the same substrate, along with other external circuitry not shown.

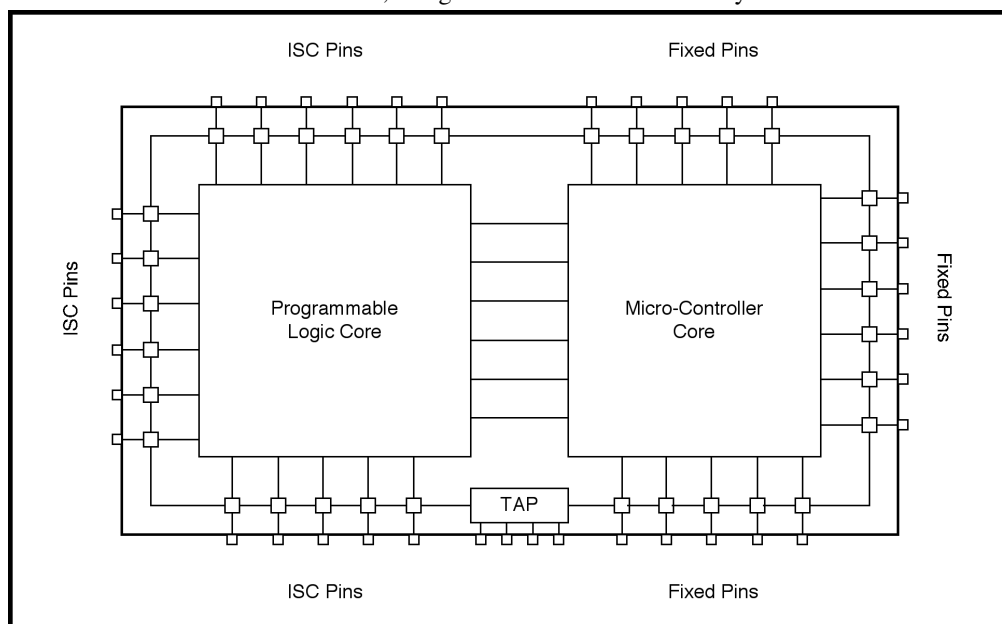


Figure 14—A device containing both programmable and nonprogrammable logic, with associated pins

NOTES

1—In Figure 14, there are connections between the programmable and nonprogrammable cores. These may or may not have Boundary-Scan Register cells at the discretion of the designer, but in either case, when the programmable core is unprogrammed, the nonprogrammable logic should perform appropriately.

2—The designer of a device containing less intelligent nonprogrammable logic could, if desired, design all I/O pins as ISC system pins. Then, all pins would be disabled or fixed at static states during the programming process.

5.1.2 The test access port (TAP)

A device compliant with this standard will possess a Test Access Port containing four (optionally five) pins labeled Test Clock (TCK), Test Mode Select (TMS), Test Data Input (TDI), and Test Data Output (TDO). The optional fifth pin is Test Reset (TRST*), which is active low. These pins are dedicated and may not be shared with other functions.

5.1.3 IEEE Std 1149.1-2001 compliance enable pins

IEEE Std 1149.1-2001 contains, in section 4.8, “Subordination of This Standard Within a Higher Level Test Strategy,” the concept of compliance enable pins. These are one or more pins that must be statically conditioned with a set of logic states before the IEEE Std 1149.1-2001 facilities within the device are guaranteed to be compliant. Further, these pins must be maintained at those states for the duration of any IEEE Std 1149.1-2001-based activities. Such pins may exist on devices that comply with this standard and are thus required to be conditioned before and during all ISC activities.

NOTE—Signals associated with compliance enable pins are not included in the Boundary-Scan Register, per IEEE Std 1149.1-2001.

5.1.4 Programmable voltage pins

For systems requiring overvoltage programming, allowance may be made for one or more static programming voltage pins used to supply energy for the physical programming process (see t) of 5.2). ISC devices shall interpret all instructions in the same manner, and the programming voltage pin(s) may not be used to signal additional data or control functions.

5.1.5 The TAP controller and instruction unit

The TAP is used to drive a 16-state finite state machine called the TAP Controller, completely specified by IEEE Std 1149.1-2001. This state machine controls all aspects of the test logic and the ISC programming process.

Each compliant device contains an instruction register and instruction decoder. This unit, along with signals derived from the TAP, is used to sequence activities within the test and ISC logic.

5.1.6 Mandatory instruction from IEEE Std 1149.1-2001

The mandatory instructions from IEEE Std 1149.1-2001 are required; SAMPLE, PRELOAD, BYPASS, and EXTEST. Programmable devices are also required to implement the IDCODE and USERCODE instructions. See 6.2 for additional mandatory instructions specified by this standard.

5.1.7 Data registers

All devices adhering to this standard are required to implement the Bypass Register, the Boundary-Scan Register, and the Device Identification Register from IEEE Std 1149.1-2001. In addition, specific registers targeted by ISC instructions documented in this standard are also required. (See Clause 7.)

5.1.8 System modal state control signals

As discussed in Clause 4, the system state of a programmable device conforming to this standard has four identified system modal states. Two control signals are required to identify which of these modal states the device currently resides within.

5.1.8.1 ISC_Done

This signal, which is nonvolatile in programmable devices that are nonvolatile, is used to indicate that a device has been programmed to the satisfaction of an external programming algorithm. In volatile devices, this signal is initially clear upon power-up.

5.1.8.2 ISC_Enabled

This volatile signal, initially clear when a programmable device is first powered, is set by the successful execution of the ISC_ENABLE instruction (see j) of 5.2). This signal is cleared by the execution of the ISC_DISABLE instruction or by passage of the TAP Controller to the *Test-Logic-Reset* TAP Controller state (see l) of 5.2). This signal must be set for other ISC instructions to operate correctly.

5.1.9 Optional status indication

An optional status indication scheme is defined by this standard. If this status indication scheme is implemented in a device, it shall adhere to rules given in 5.2.

NOTE—A designer is free to implement a private status indication scheme if desired, but tools designed to work with this standard will not be able to determine status unless specifically modified to recognize this private functionality.

As described in 6.1, each ISC instruction causes events to occur that are initiated by entering the *Run-Test/Idle* TAP Controller state and remaining there for some minimum duration. It is possible that errors could be sensed by the circuitry responding to an instruction that would be useful information for an external programming algorithm. This algorithm could branch to appropriate exception handling routines when it observed errors reported by the status indication.

As all ISC instructions place some data register between TDI and TDO, the data register may be loaded with a status indication field in the *Capture-DR* TAP Controller state. This field is (at minimum) the least two significant bits of the register and indicates the status of the ISC instruction functionality that was last executed in the *Run-Test/Idle* TAP Controller state.

NOTE—If the status indication field is implemented, then all ISC data registers are required to be a minimum of two bits long. As seen in Clause 6 and Clause 7, some ISC data registers may be defaulted to the Bypass Register. This default is not compatible with the implementation of status reporting since the Bypass Register is only one bit long.

Figure 15 gives a structure for the registers that may at some time appear between TDI and TDO. Status reporting can be added by appending a two-bit (minimum) register onto the ISC data registers as shown in Figure 16.

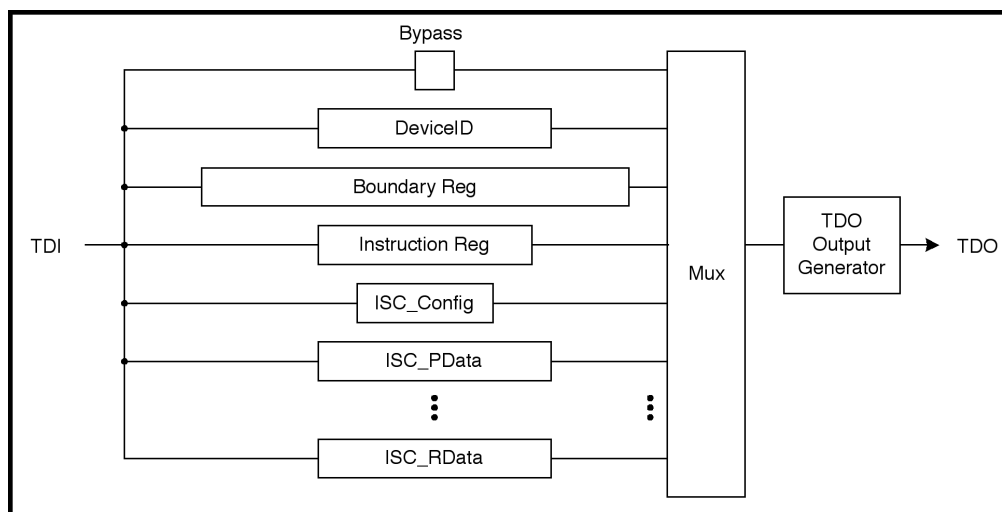


Figure 15—Structure of registers placed between TDI and TDO when there is no status reporting

Simply appending a status field to each register lengthens each by two (or more) bits. This simple model is consistent with the rules in IEEE Std 1149.1-2001 concerning the concatenation of registers: The concatenated register paths (between TDI and TDO) must have a single unique name. Thus, there is no “Status” register addressable by some instruction, only the normal ISC register set extended by two (or more) bits. In a more sophisticated implementation, a designer could reuse the least significant bits of most ISC data registers as status bits that are captured in the *Capture-DR* TAP Controller state. Note that registers, such as ISC_RData (see 7.7), must be extended in length to prevent the status bits from overwriting data that is intended to be read back.

The status field indicates the execution status of the current instruction if the TAP Controller enters the *Capture-DR* TAP Controller state after the *Run-Test/Idle* TAP Controller state. The status of the previously executed ISC instruction can be observed when the TAP Controller enters the *Capture-DR* state after loading a new ISC instruction, but before entering the *Run-Test/Idle* TAP Controller state.

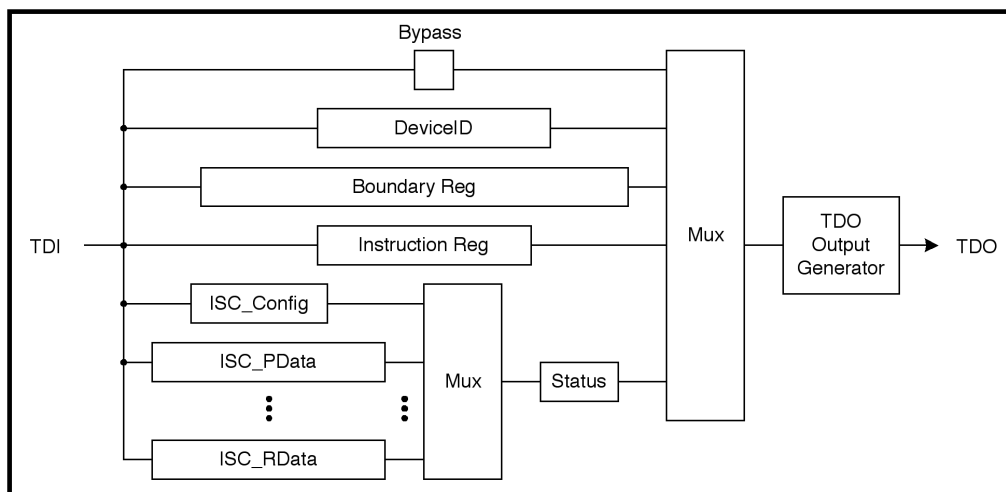


Figure 16—Insertion of a status reporting register at the least significant two bits of the ISC data registers

NOTE—The ISC_NOOP instruction may be used as a status query instruction. However, it will return status of the last executed ISC instruction. If ISC_NOOP is executed a second time, then the status shifted out will be that of the previous ISC_NOOP execution.

Status bits may be manipulated at other times by instructions during their execution. For example, upon first entry into the *Run-Test/Idle* TAP Controller state, instructions may load the status register with an error code that indicates the required duration in the *Run-Test/Idle* TAP Controller state has not expired. After the duration has expired (but before leaving the *Run-Test/Idle* TAP Controller state), the status code may be updated to a non-error state. If the instruction is not allowed to complete its required duration (by leaving the *Run-Test/Idle* TAP Controller state early), the content of the status field subsequently captured will indicate an error.

The ability to observe instruction execution status allows external programming algorithms to respond to errors more closely to the time when they occur. This can allow a programming process to immediately abort or take some remedial action. If status indication is not implemented, an algorithm can still make basic checks, but a substantial amount of time and activity may pass between the occurrence of the error and when it is detected. For example, an ISC_PROGRAM instruction may sense the failure of a data packet to be programmed and present this to an algorithm via the status field. If status is unavailable (or not checked), the algorithm may continue programming all subsequent locations (which may also be failing) and not see any error until a verification process is then executed. Notice that if verification has been made impossible by the activation of a read-protection security mechanism, then the error(s) could not be detected.

The status register may be constructed as shown in Figure 17. At a minimum, it contains a two-bit error code that is used to indicate the status of an executed instruction (see 5.2 and 6.1.1). Optionally, it may also contain an N-bit field containing additional information subcodes defined by the device designer to convey the cause of an error. Two examples are (1) the minimum duration for time spent in the *Run-Test/Idle* TAP Controller state was not met (see rule i) of 6.1.1), and (2) a security violation prevented the instruction from operating (e.g., read security has blocked ISC_READ instruction execution; see 5.1.11.1 and 6.3.1).

5.1.10 Optional operation success indication

An optional operation success indication scheme is defined by this standard. If the operation success indication scheme is implemented it shall conform to the rules set forth in 5.2. This indication scheme is only required for those devices that implement an adaptive programming algorithm.

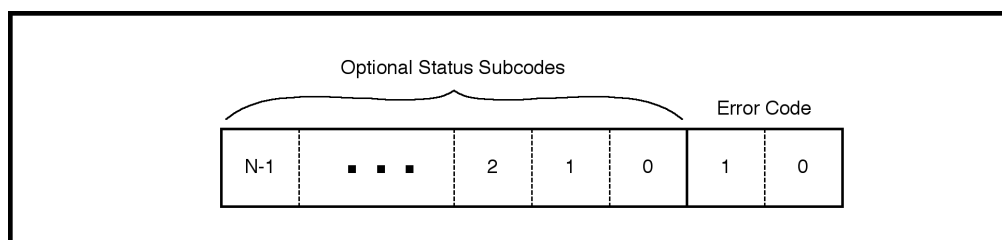


Figure 17—Status register with optional subcode bits

Devices that require the use of adaptive programming algorithms will require a mechanism to indicate the successful completion of certain operations. Some operations do not have a set minimum amount of time, and it will be required that the device notify the programming hardware when the operation is successful and can be terminated. The operation success indication mechanism allows an external programming algorithm to execute an operation for a given amount of time and at the end of that time shift out the operation success indication bit(s) (see 8.5.7.7). If the operation was completed successfully, the external programming algorithm can continue on to the next step in the programming process. If the operation was not successful and the maximum number of iterations has not been met, then the operation will be run again until either it is successful or until the maximum number of iterations is met. There may also be defined a minimum number of iterations the operation shall be executed before the operation success indication is evaluated (see 6.1.1). This will allow tools that implement dynamic concurrentization to maximize the amount of concurrency with devices that implement deterministic programming algorithms. The designer must make a tradeoff between increased concurrency with devices that implement deterministic programming algorithms or potentially improved programming times for dynamic concurrentization algorithms when specifying a minimum number of loops. If only a few loops are required, then a dynamic concurrentization algorithm will be able to exit out of the loop upon success sooner than if a greater number of loops are required. However, the fewer the number of required loops, the more difficult it is for a dynamic concurrentization algorithm to achieve concurrency between adaptive and deterministic devices.

There are no restrictions placed on the number of bits used in the operation success indication nor on the values used to indicate success. The indication that an adaptive programming algorithm has been implemented is given by the use of the loop and OST keywords in the BSD file (see 8.5.7). The “operation success” pattern will be defined as a part of the flow where it will be monitored.

5.1.10.1 Implementing the operation success indication

There are multiple approaches to implementing the operation success indication. These approaches include, but are not limited to, implementing the operation success indication with the status register or implementing it with another register or set of registers. If the optional status register (see 5.1.9) has been implemented, and the operation success indication is included as a part of that register, then the operation success indication shall be included as a part of the subcode bits. If this approach is taken, it will affect the shift length of all ISC data registers, regardless of whether or not they require the operation success indication. An example of this is shown in Figure 18.

Alternatively, the operation success indication could be appended to the register(s) (such as ISC_PData or ISC_RData) that may be targeted by an instruction that would require the operation success indication. In this instance, the only shift operations that would be affected are those that require the operation success indication. This is demonstrated in Figure 19.

Alternatively, the operation success indication may be implemented as a stand alone register with an instruction used to access this register. If this approach is taken, the instruction shall follow the rules set forth in 6.1.1.

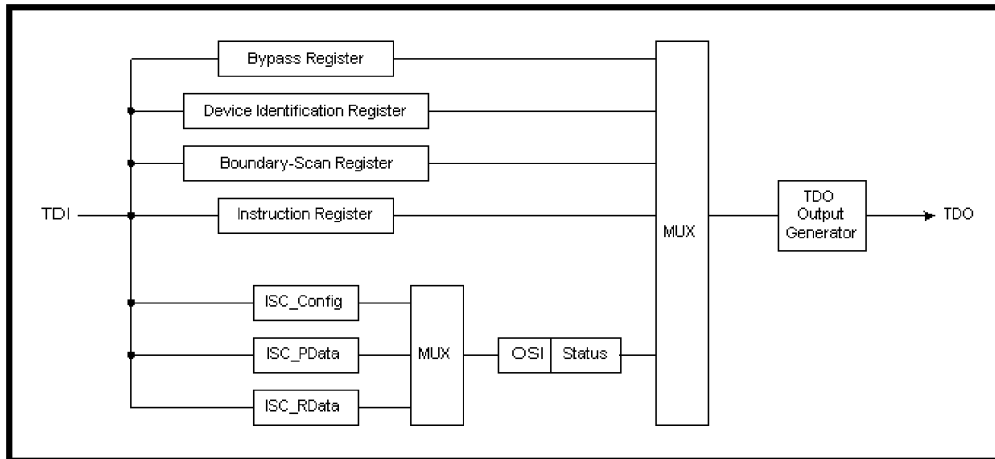


Figure 18—Operation success indication with status

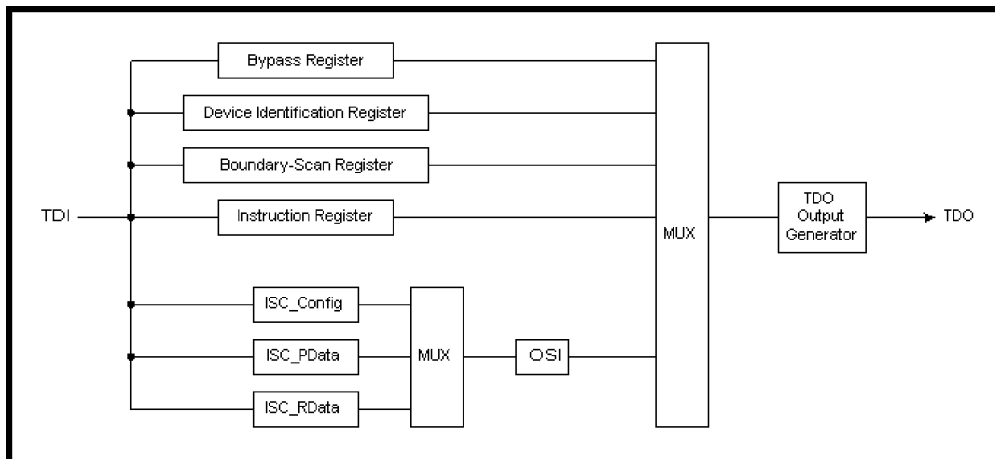


Figure 19—Operation success indication on select registers

5.1.10.2 Monitoring the operation success indication bits

Regardless of how the operation success indication is implemented, it will be evaluated each time an action or sequence of actions is executed. The flow provided in the BSDL file will provide two, and possibly three, pieces of information necessary to evaluate the success of the operation. It will provide the success requirements, a maximum number of times the success requirements should be evaluated before declaring an error, and possibly a minimum number of times the action(s) should be executed before evaluating the success of the operation. If an operation has been evaluated and success has not been indicated, then an error should not be declared unless the maximum number of iterations has been met. If the maximum number of iterations has been met and success has not been indicated, then it should be considered that the operation has failed. When an error is declared, it should be handled based on whether or not the Exit_On_Error tag has been associated with the flow used to describe the adaptive programming loop (see 8.5.7.2). If the tag is associated with the flow, then all programming operations should stop at the end of the data register or instruction register scan that generated the error.

In the case that the operation success indication has been implemented as subcodes in the status bits, it is noted that the functionality of the operation success indication bits is completely independent of the status bits as described in 5.1.9. The operation success indication notifies a programming tool when success has been achieved, whereas the status bits notify the programming tool when there has been an error that needs to be addressed such as an incorrect load of the instruction register. It should not be interpreted as an error if the operation success indication does not indicate that success has been achieved. The only time an error should be declared is when the status bits indicated that there has been an error or when the maximum loop count has been reached and the operation success indication does not report success.

5.1.11 Optional security provisions

An optional security mechanism is provided by this standard. Security features, when activated, prevent unwanted activities such as reading out programming data, changing programming data, or unauthorized erasure. This security mechanism is intended to work in conjunction with the optional `ISC_PROGRAM_SECURITY` instruction (see 6.4.3) or with `ISC_PROGRAM`. This security mechanism is very flexible and contains several options that allow it to implement all security features known to be in use as well as a richer set of capabilities.

The main features of the security mechanism defined herein are as follows:

- a) It allows up to three discrete forms of protection: read-protection, program-protection, and erasure-protection.
- b) It provides for the implementation of a “key” that will allow a user to “unlock” protection. A user can generate the key rather than it being hard-wired by the IC vendor.
- c) A “permanent” lock feature may be added that, when activated, will prevent any subsequent modification of security.
- d) The key may be encrypted by an on-chip function before it is stored such that knowledge of the original key is not sufficient to unlock security.
- e) The architecture of this security mechanism may be reduced in complexity to offer only the most basic read-protection at very low cost, emulating those capabilities used in past devices.

The security mechanism is based upon an Internal Protection Control register (the IPC register) that is nonvolatile in programmable devices that are nonvolatile. This register contains one, two, or three discrete bits used to control the form(s) of security, an additional (optional) N-bit key field and optional space for other functions (such as optional sector selection) as needed by the designer.

Figure 20 shows a general architecture of this security mechanism with all three forms of security implemented independently. The IPC register is divided into a three-bit field for function disable signals and an N-bit key field. This register is programmed by `ISC_PROGRAM_SECURITY` (see 6.4.3), which transfers the contents of the `ISC_PData` register (see 7.6) into the IPC register. This transfer occurs upon the successful completion of the instruction and may be conditioned by some additional factors. In Figure 20, the transfer is allowed when the key bits in the IPC register are all-zero or match those in `ISC_Pdata`, and the key bits in IPC are not all-ones. The “all-ones” key is used as a permanent lock; once the all-ones key is installed (and erasure has been disabled), there is no way to update the IPC register in the future. This permanently establishes the states of the three security signals. (Note that it is assumed that the initial state of the IPC for a blank device is all-zero.) The all-zero code may be thought of as “no security.” The all-zero code is the only code that can be directly overwritten. Once a key (containing both 1s and 0s) has been entered, it cannot be changed directly to another key. Note that `ISC_ERASE` (if not disabled) may be used to zero out the IPC register. Thus, erasure is the only means for changing a nonzero security key.

This architecture may be simplified by removing one or two of the security signals if they are not desired, by sharing any combination of security signals, by removing the all-ones permanent lock feature, or by removing the key field altogether. The architecture may implement stronger security by encoding the key while it is being transferred such that the original key will not match the stored key. This encoding may be

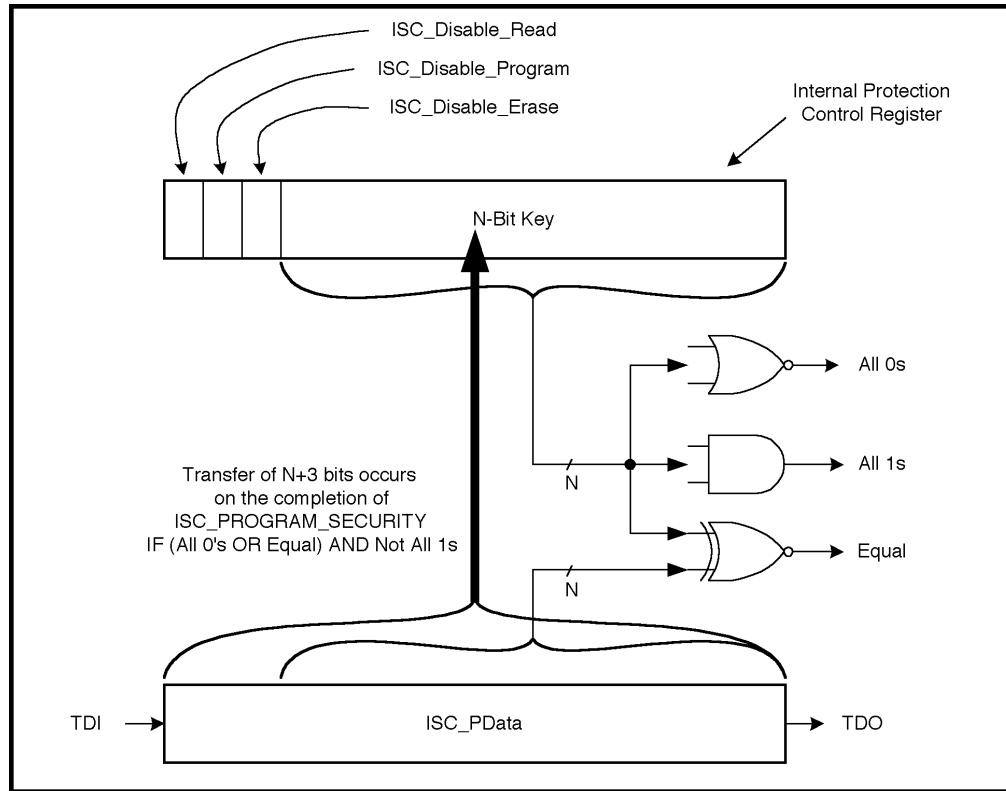


Figure 20—Architecture of the optional security mechanism

made entirely general by making it a function of other bits previously stored elsewhere in program memory. This range of implementation options allows any level of security, from none to arbitrarily strong keyed security to permanent locking. Further, all known security systems previously developed are subsets of this general capability.

5.1.11.1 ISC_Disable_Read

This optional signal, which is nonvolatile in programmable devices that are nonvolatile, is used to indicate to the ISC_READ instruction (see 6.3.1) that the device programming data is secured and should not be read out. This signal may be controlled by an independent bit of memory, or it may share a bit with one or both of the other disable signals. If the optional status indication capability (see 5.1.9) is implemented, then the fact that readback has been inhibited can be indicated. Read-protection is often used to inhibit malicious or inadvertent exposure of system design details.

5.1.11.2 ISC_Disable_Program

This optional signal, which is nonvolatile in programmable devices that are nonvolatile, is used to indicate to the ISC_PROGRAM instruction (see 6.2.3) that the device programming data is secured and should not be overwritten. This signal may be controlled by an independent bit of memory, or it may share a bit with one or both of the other disable signals. If the optional status indication capability (see 5.1.9) is implemented, then the fact that programming has been inhibited can be indicated. Program-protection may be used to inhibit malicious or inadvertent overwriting of existing program data.

5.1.11.3 ISC_Disable_Erase

This optional signal, which is nonvolatile in programmable devices that are nonvolatile, is used to indicate to the ISC_ERASE instruction (see 6.3.2) that the device programming data is secured and should not be erased. This signal may be controlled by an independent bit of memory, or it may share a bit with one or both of the other disable signals. If the optional status indication capability (see 5.1.9) is implemented, then the fact that erasure has been inhibited can be indicated. Erase-protection may be used to inhibit malicious or inadvertent erasure of existing program data.

5.1.12 Programmable memory array access

A programmable logic device contains a memory array that holds the configuration data that determines the programmed function of the device. A basic model is shown in Figure 21. Note that it is composed of a number of rows of data, some of which may contain blank or unused bits. An address and data register are used to access the array. The data register must be wide enough to accommodate the longest row of data. When rows containing unused locations are accessed, corresponding locations in the data register are “don’t care” locations.

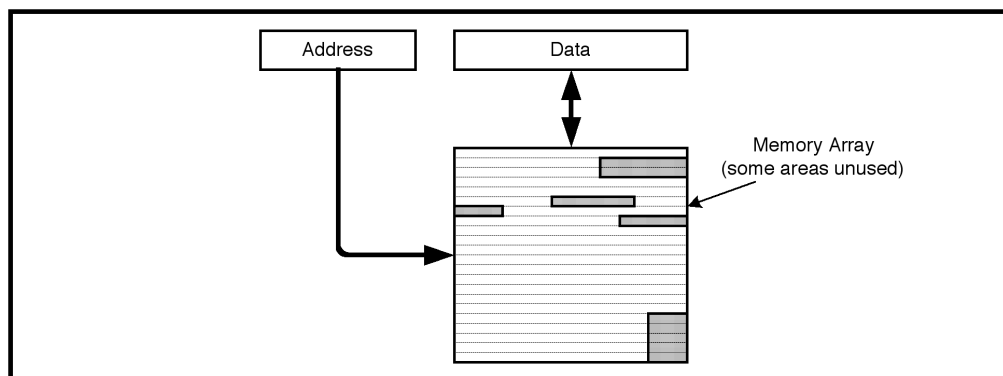


Figure 21—Basic model of a PLD configuration memory

The rules in this standard are designed to precisely and unambiguously define an access methodology for In-System Configuration of programmable logic devices, but at the same time, offer designers a high degree of freedom in the design of the internal programmable structures. Thus, a device designer has many options for delivering address and data information to the internal memory array(s). Figure 22 shows a very basic access concept, where the address and data registers are linked together as one shiftable ISC register. Address and data can be shifted in or out of these two registers via the TDI/TDO shift path, thus setting up a program or read cycle to be executed by ISC instructions. The combination of address and data registers can be labeled the ISC_PData and ISC_RData registers as is allowed by IEEE Std 1149.1-2001. (See Clause 7 for a discussion of ISC data registers.) Whenever an operation on the memory array is done, a full scan of address and data is required.

A different memory access scheme is shown in Figure 23. Here, the address register can be independently accessed (see the ISC_ADDRESS_SHIFT instruction, 6.5.4) to initialize an address. The data register can be accessed directly by the ISC_PROGRAM instruction (6.2.3) or ISC_READ instruction (6.3.1). This scheme still requires that address and data both be shifted into the device to program a single location.

A more sophisticated approach simply adds address generation logic (see Figure 24). This logic may perform an auto-increment function, even skipping unused addresses in a fragmented address space. The triggering of this function may be tied to the programming and reading instructions to coordinate the programming of successive memory locations without the need for loading the address register for each.

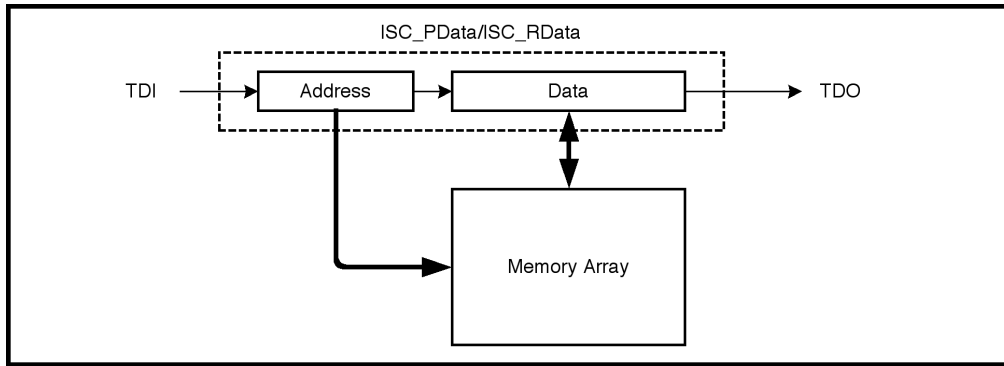


Figure 22—Basic Programmable Memory Array access

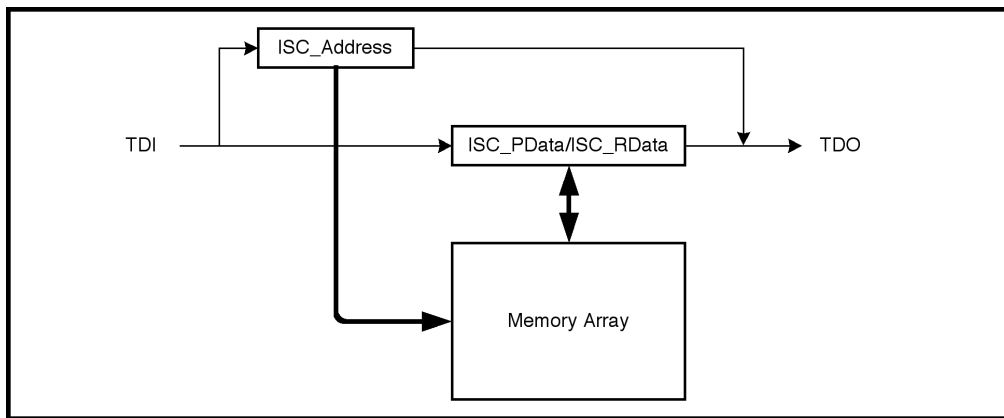


Figure 23—A structure with individual access to address and data

This may greatly reduce the time and amount of data shifting (and storage) needed for linearly accessing long memories.

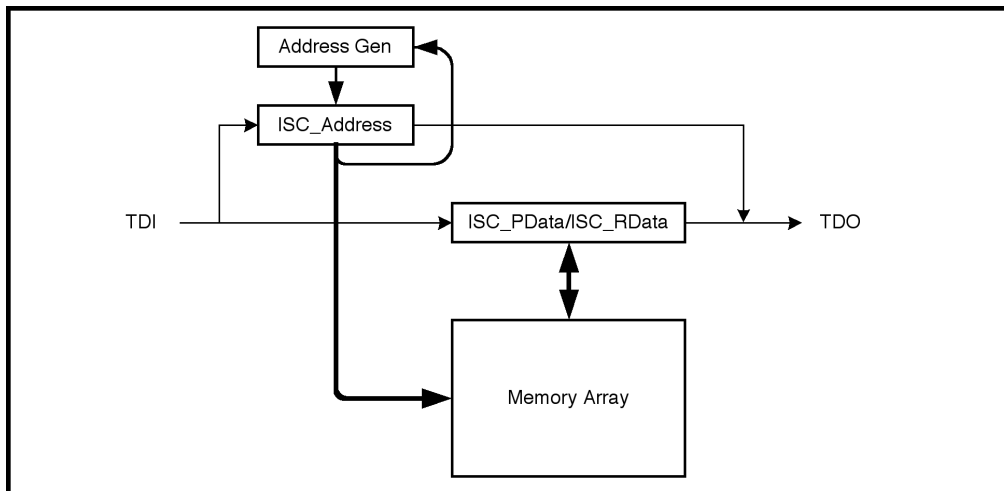


Figure 24—Memory array access with independently controlled address and data registers

It is also possible that the memory data for a programmable device is stored in more than one array, as shown in Figure 25. In this situation, there are two memory arrays with different lengths, widths, and even different burn times. By selecting an address scheme, such as having the high-order address line select between the two arrays, it is possible to use the same structure seen earlier for accessing memory. Memory can be accessed linearly as one block, although a change in duration in the *Run-Test/Idle* TAP Controller state may be needed when the array boundary is crossed.

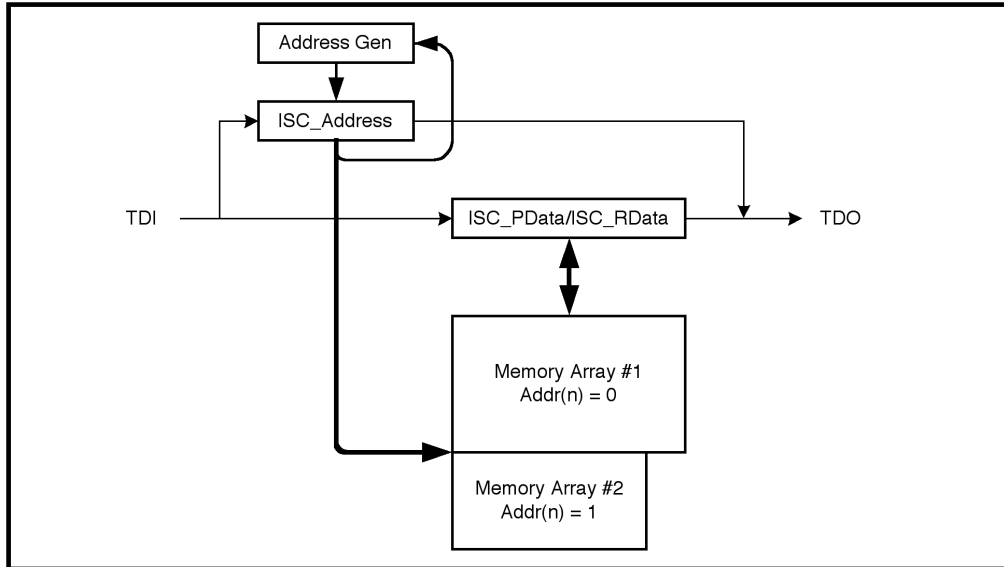


Figure 25—Accessing two memory arrays with different dimensions

A similar situation is shown in Figure 26 where a device contains two quite different arrays of memory, one being volatile and the other nonvolatile. The selection between them is controlled by a bit in the ISC_Config register (which may also control other important parameters as well). In this case, the ISC_ENABLE instruction effectively chooses which array will be accessed. Such a device on a board may have its nonvolatile content supplied once during board manufacture, and the volatile content refreshed while online in the field. Proper execution of ISC_ENABLE will prevent programming of the volatile content from interfering with the nonvolatile content.

Figure 27 shows a scheme where the address register is not directly accessible from the outside, but is controlled by a short (perhaps one-bit) command field appended to the data register. One command could initialize the address register, and a second could increment it. While this again shortens the volume of data that must be scanned into the device, it has less addressing flexibility in that the start of a memory operation cannot be set to an arbitrary location.

The inventiveness of a device designer has few constraints imposed by this standard. At least five access approaches may be utilized, as follows:

- The address may be directly loaded into the ISC_Address register (see 7.11) by previous use of the ISC_ADDRESS_SHIFT instruction (see 6.5.4). This is called explicit addressing.
- The address may be auto-initialized to the start of the array by the act of loading an ISC instruction destined to access memory. Subsequent accesses may automatically increment the addressing mechanism to the next valid entry. This is called implicit addressing.
- An address may be shifted into the ISC_RData or ISC_PData registers in the *Shift-DR* TAP Controller state where, upon passing the *Update-DR* TAP Controller state, it is then loaded into an addressing register. (For example, see permissions e) of 6.3.1.1 and g) of 6.3.1.1.)

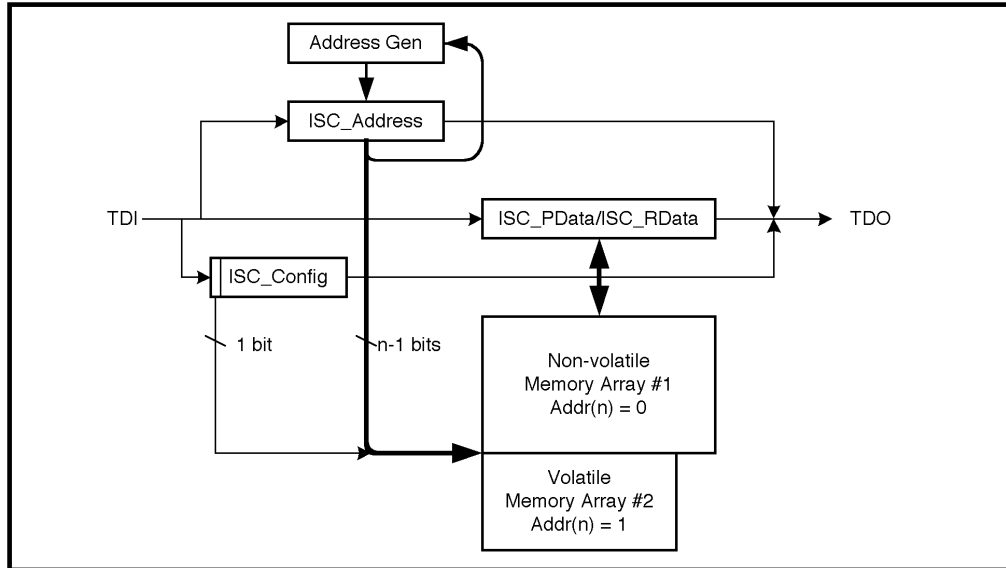


Figure 26—A programmable device with two embedded technologies

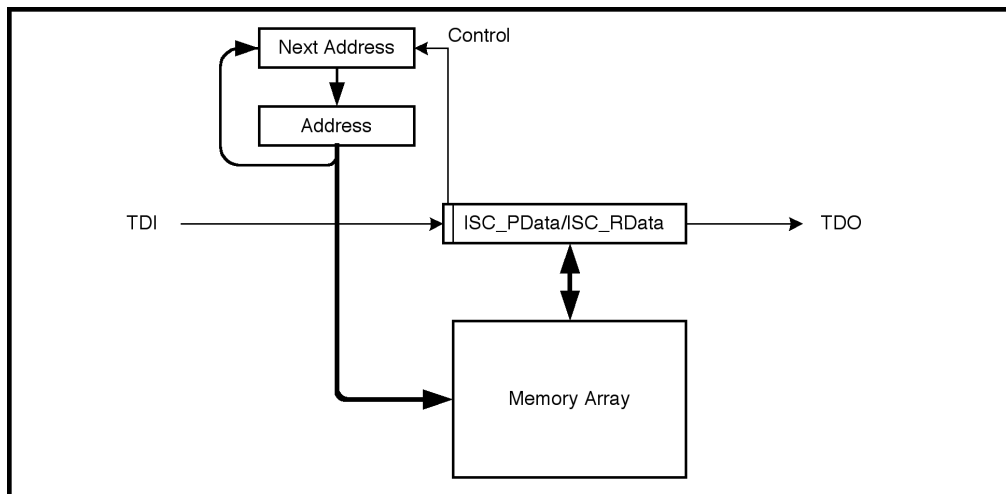


Figure 27—An approach where there is no direct access to the addressing structure

- d) Address control commands may be shifted into the ISC_RData register in the *Shift-DR* TAP Controller state where, upon passing the *Update-DR* TAP Controller state, the commands control an addressing register. (For example, see permissions e) of 6.3.1.1 and g) of 6.3.1.1.) These commands may cause actions, such as initializing the address, incrementing it, decrementing it, shifting it, adding an offset, and so on.
- e) A combination of these methods may be implemented.

The actual technology of the addressing method is not specified by this standard, again allowing the designer freedom to use binary addressing, one-hot addressing, address encoding, and so on.

5.2 Specification

Rules

- a) A component conforming to this standard shall first conform to all mandatory provisions of IEEE Std 1149.1-2001.
- b) The component shall conform to all rules within IEEE Std 1149.1-2001 that are specified for programmable devices.

NOTE—Rule b) assures that the normally optional IDCODE and USERCODE instructions are understood to be mandatory for devices conforming to this standard.

- c) The system mode (described by IEEE Std 1149.1-2001) of a component conforming to this standard shall be organized as four system modal states, Unprogrammed, Operational, ISC Accessed, and ISC Complete, as given in Clause 4.

NOTE—System modal states are determined by the ISC_Done and ISC_Enabled signals (along with a derived signal ISC_Disable_Completing) as shown in Figure 3. Further rules given below govern the derivation and timing of these signals.

- d) The component shall contain an ISC_Done signal that is nonvolatile in nonvolatile programmable devices and that is cleared during power-up in volatile programmable devices.
- e) When the ISC_Done signal is not protected by an enabled protection mechanism, the ISC_Done signal shall be set by exactly one of three choices of implementation:
 - 1) By successful completion of the ISC_DISABLE instruction when a specified data pattern is in place in the ISC_Config register. (See 6.2.2.)
 - 2) By successful completion of the (optional) ISC_PROGRAM_DONE instruction. (See 6.4.1.)
 - 3) By successfully writing a specified bit or set of bits to the program memory using the ISC_PROGRAM instruction. (See 6.2.3.)

NOTES

1—It is intended that ISC_Done should be set upon completion of programming and verifying the program.

2—Choice 1) above is the preferred implementation, with ISC_Done being set as closely to the clearing of the ISC_Enabled signal as possible. (See the discussion in 4.3.)

3—As noted in choice 3) above, the ISC_Done signal may be a function of a set of bits in program memory. While the “AND” of these bits might be one obvious function that derives ISC_Done, this standard does not mandate a function.

- f) When the ISC_Done signal is not protected by an enabled protection mechanism, the ISC_Done signal shall be cleared by one or more of three choices of action:
 - 1) By successful completion of the ISC_ERASE instruction. (See 6.3.2.)
 - 2) By successful completion of the (optional) ISC_ERASE_DONE instruction. (See 6.4.2.)
 - 3) By successfully writing a specified bit or set of bits to the program memory using the ISC_PROGRAM instruction. (See 6.2.3.)

NOTE—This last option 3) can only be supported in programmable devices that have symmetrical programming capability; that is, they can program both “1” and “0” bits in memory locations.

- g) Either the current state of the ISC_Done signal (“1” indicates “set,” “0” indicates “cleared”) or the state that existed prior to entering the ISC Accessed system modal state shall be visible in bit two of the instruction register at the rising edge of TCK in the *Capture-IR* TAP Controller state.

NOTES

1—Bits one and zero of the instruction register are already committed to capturing a “01” pattern by IEEE Std 1149.1-2001. (See IEEE Std 1149.1-2001, Table 7-1 in 7.2, Instruction Register Operation.)

2—This rule guarantees that the ISC_Done signal may be interrogated upon loading any instruction defined by IEEE Std 1149.1-2001 or this standard. The programmed state of the device may then be determined without the need for entering the ISC Accessed system modal state.

3—Because of the new instructions mandated by this standard, the minimum length of the instruction register already exceeds two bits, and hence, this rule does not itself expand the length of this register.

- h) An updated state of the ISC_Done signal (“1” indicates “set,” “0” indicates “cleared”) shall be visible in bit two of the instruction register no later than upon successful completion of ISC_DISABLE. (See 6.2.2.)

NOTE—This rule defines when a recently updated state of the ISC_Done signal must be observable in the instruction register when shifting out captured instruction register bits. In some implementations, there may be unusual voltages present at the time ISC_Done signal is updated that make the immediate transfer of the ISC_Done signal value to the instruction register impractical. Toolmakers must not rely on seeing an updated ISC_Done signal state before the completion of ISC_DISABLE.

- i) The component shall contain an ISC_Enabled signal that is cleared upon device power-up.
- j) The ISC_Enabled signal shall only be set by the successful completion of the ISC_ENABLE instruction. (See 6.2.1.)
- k) The ISC_Enabled signal shall be cleared by either of two events:
 - 1) By successful completion of the ISC_DISABLE instruction. (See 6.2.2.)
 - 2) By the TAP Controller moving to the *Test-Logic-Reset* TAP Controller state.

NOTES

1—The TAP Controller may enter the *Test-Logic-Reset* TAP Controller state asynchronously in response to the assertion of the TRST* TAP signal.

2—The ISC_Enabled signal may also be cleared by an event specified in permission (s) below.

- l) If, while a device is in the ISC Accessed system modal state an IEEE Std 1149.1-2001 pin-permission instruction (EXTEST, INTEST, CLAMP, HIGHZ, or RUNBIST) is loaded, the device shall enter test mode and all rules required for these instructions by IEEE Std 1149.1-2001 take effect immediately.

NOTE—Data residing in the boundary-scan register may be utilized by these instructions.

- m) If a device was in the ISC Accessed system modal state and one or more IEEE Std 1149.1-2001 pin-permission instructions (EXTEST, INTEST, CLAMP, HIGHZ, or RUNBIST) have subsequently been executed, loading an ISC instruction shall (with the exception given in permission s) below) return the device to the ISC Accessed system modal state.

NOTE—Data residing in the boundary-scan register utilized by the IEEE Std 1149.1-2001 instruction(s) may be utilized by an ISC instruction that is loaded per rule d) of 6.1.1.

- n) If recommendation v) below is implemented, the least significant two (or more) bits of all ISC data registers shall capture a code that indicates the status of the most recently loaded ISC instruction that executed in the *Run-Test/Idle* TAP Controller state. These bits include:
 - 1) A mandatory two-bit error code field in bit positions 0 and 1.
 - 2) An optional N-bit status subcode field (starting at bit 2).

NOTES

1—The status error codes for instructions are assigned in Clause 6. ISC data registers are described in Clause 7. Subcodes are discussed in 5.1.9.

2—If no ISC instruction has been executed in the *Run-Test/Idle* TAP Controller state since device power-up, the status bits are not defined. It is expected that an external algorithm will know when status bits are defined.

3—If the execution of non-ISC instructions are intermixed with ISC instructions, this will have no effect on the status field observed the next time an ISC instruction is executed.

4—If the currently loaded ISC instruction has not yet executed in the *Run-Test/Idle* TAP Controller state, then the status captured in the *Capture-DR* TAP Controller state will be that of the most recently loaded ISC instruction that did execute in the *Run-Test/Idle* TAP Controller state.

- o) If recommendation v) is implemented, the status field of the targeted ISC data register shall be loaded no later than the rising edge of TCK in the *Capture-DR* TAP Controller state.

NOTE—This rule explains when the status field of the data register is to be captured in preparation for shifting. The status bits may be manipulated at other times by instructions while they are executing. For example, instructions may load the status field with an error code that indicates the required duration in the *Run-Test/Idle* TAP Controller state has not expired upon first entering this state. After the duration has expired (but before leaving the *Run-Test/Idle* TAP Controller state), the status code may be updated to a nonerror state. If the instruction is not allowed to complete its required duration (by leaving the *Run-Test/Idle* TAP Controller state early), the content of the status field subsequently captured will indicate an error.

- p) No instruction specified by IEEE Std 1149.1-2001 shall have any effect upon ISC data registers specified by this standard.

NOTE—In particular, the address, data, status, and protection mechanisms implemented in accordance with the rules of this standard shall see no side effects as a result of executing any instruction defined by IEEE Std 1149.1-2001.

- q) If the optional security capability described in this standard is implemented, it shall generate one or more of the three signals: *ISC_Disable_Read*, *ISC_Disable_Program*, and *ISC_Disable_Erase*.

NOTES

1—The optional security capability is described in 5.1.11.

2—This rule does not prohibit the implementation of a proprietary security capability, but tools created to support this standard may not be able to access and control this capability without special modification.

- r) The operation success indication shall be implemented on devices that require an adaptive programming algorithm.

NOTES

1—This rule does not specify a specific implementation for the indication scheme, only that the operation success indication exists and is accessible to an external programming algorithm. The operation success indication may be included as a part of the optional status indication, but that is not a requirement. If the operation success indication bits are included as a part of the optional status indication field, then their operation will be completely independent of the status bits.

2—The operation success indication facility is described in 5.1.10.

Permissions

- s) The ISC_Enabled signal may be cleared as a side effect of loading a non-ISC instruction into the IEEE Std 1149.1-2001 instruction register, on the falling edge of TCK in the *Update-IR* TAP Controller state.

NOTES

1—One or more non-ISC instructions may cause this behavior at the option of the device designer. The designer may wish to physically prevent the operation of certain sequences of non-ISC instructions followed by ISC instructions, when unusual programming voltages are present within the device. The generation of these voltages would be deactivated by clearing ISC_Enabled. (See the discussion near the end of 4.2.) The documentation of forbidden instruction sequences (see 8.2) is still required.

2—In any event, any instruction defined by IEEE Std 1149.1-2001 must operate as defined. It is the subsequent return to an ISC instruction after executing an IEEE Std 1149.1-2001 instruction that may find its operation inhibited. It is possible, for example, that PRELOAD is an instruction that clears the ISC_Enabled signal. If so, then PRELOAD cannot be used to establish a new I/O state on system pins in the middle of a programming sequence.

- t) Pins that require the application of voltage levels as a prerequisite for successful programming may exist on the device when these pins provide only energy and do not transmit information.
- u) The operation success indication may be implemented as a subfield of the optional status indication field described in recommendation v).

Recommendations

- v) A status indication field should be implemented in all ISC data registers as defined by these rules.

NOTE—As will become evident in Annex B, status indication allows the detection of anomalies close to their occurrence in time that may be quite useful in debugging problems when concurrently programming multiple devices.

5.3 Description

This standard extends IEEE Std 1149.1-2001 and consists of additional structures and instructions to be added to an IEEE Std 1149.1-2001-conformant device.

A device conforming to this standard will be programmable using a standardized programming process that makes use of these structures and instructions. Further, multiple conformant devices, configured as a chain of devices (as usually configured to support IEEE Std 1149.1-2001-based interconnection tests) may be programmed concurrently, using a programming flow such as discussed in Annex B.

This standard provides for the interrogation of an (optional) status indication facility (see 5.1.9) so that a programming algorithm may know when errors occur within a sequence of programming operations. Since it is envisioned that device programming may take place during the final stages of new board manufacturing, it is likely that failures will occur. These will be much easier to diagnose and correct if this status indication capability is implemented in devices.

This standard also provides for the interrogation of an (optional) operation success indication facility (see 5.1.10) for those devices that require adaptive programming algorithms. These devices can have burn times that vary by multiple orders of magnitude when going from one device to another. Without this mechanism, the devices would always be programmed or erased using the worst case times, when in reality, they may need only a fraction of that time to achieve success.

Nonvolatile programmable devices, by means of an internally maintained ISC_Done signal, will be able to react during power-up as either an operational (programmed) device or as an unprogrammed device that

maintains its outputs in a noncontending state. Volatile devices will always react during power-up as unprogrammed devices.

By means of a volatile, internally maintained signal `ISC_Enabled`, a programmable device will be able to maintain its outputs in either a noncontentious state, or in states determined by first loading the device (using the `PRELOAD` instruction) with a conditioning pattern. This allows programming processes, which may take significant time, to be conducted while maintaining safe I/O conditions among collections of devices on a board.

In some devices, particular sequences of ISC instructions mixed with IEEE Std 1149.1-2001 instructions may not be supported. In such cases, the IEEE Std 1149.1-2001 instructions in question may effectively abort the ISC Accessed system modal state. The IEEE Std 1149.1-2001 instruction behavior must be unaffected, but it will not be possible to return to an interrupted programming sequence. These “forbidden” instruction sequences must be documented (see 8.2) so that applications can avoid them.

6. ISC Instructions

When loaded into the instruction register, all ISC instructions target a data register between TDI and TDO. Clause 7 gives the requirements for these registers. With two exceptions (see `ISC_ENABLE`, 6.2.1 and `ISC_DISABLE`, 6.2.2), all ISC instructions are intended to execute only in the ISC Accessed system modal state (see 4.1). If this expectation is not met, the instruction will still perform as required by IEEE Std 1149.1-2001 (a specific register will be placed between TDI and TDO, capable of shifting data), but the operation of the instruction as specified by this standard is not defined. In this situation, the instruction operation is said to be “nullified.” Tool designers should avoid the use of instructions where their operation will be nullified, and device designers should assure that if an instruction is executed while nullified, the device would not suffer any harm.

6.1 General properties of ISC instructions

All optional and mandatory ISC instructions have a common structure and fundamental operation. All ISC instructions have the following rules in common.

6.1.1 Specifications

Rules

- a) Each ISC instruction shall have a binary instruction code assigned by the device designer.

NOTES

1—IEEE Std 1149.1-2001 contains rules on instruction pattern assignment that shall be honored first.

2—In certain cases, an ISC instruction may share an instruction code with another instruction.

- b) With the exceptions of `ISC_ENABLE` (see 6.2.1) and `ISC_DISABLE` (see 6.2.2), all ISC instructions shall operate correctly in the ISC Accessed system modal state (see 4.1); and when this condition is not met, the instruction is said to be nullified and the following shall be true:
 - 1) The instruction obeys all rules within IEEE Std 1149.1-2001 regarding the operation of the TAP, connection of a register between TDI and TDO and its shift operation.
 - 2) The I/O pin behavior is still governed by rule c) below.
 - 3) The instruction does not operate in such a manner as to cause damage to the device.

NOTES

1—As described later in 6.2, ISC instruction activities shall begin with the proper execution of the ISC_ENABLE instruction, which moves the device to the ISC Accessed system modal state. This rule defines the behavior of ISC instructions when this condition is not met.

2—When instruction operation has been nullified, this can be indicated by the status reporting scheme given by recommendation v) of 5.2 and rule q) below. If status reporting is not supported by a device, then the only way to determine whether nullification has occurred may be to read back and verify programming data, if that has not been precluded by the activation of a security mechanism.

3—It is possible that an individual device could be designed with ISC instructions that operate correctly in other system modal states, but applications utilizing this standard should not make such assumptions and should still use the protocols given by this standard.

- c) When any of the ISC instructions supported in a device is loaded (at the falling edge of TCK in the *Update-IR* TAP Controller state), all ISC system pins (see 5.1.1.1) shall be defined per the options below, and all ISC instructions shall implement one or the other of these options exclusively:
 - 1) All ISC system signals capable of driving out of the device shall be defined by data held in the Boundary-Scan Register.
 - 2) All ISC system signals capable of driving out of the device (including those that may be two-state outputs) shall be placed in the inactive drive state (e.g., high-impedance).

NOTES

1—When option 1) is chosen, the PRELOAD instruction may be used to load the Boundary-Scan Register with a suitable pattern of data bits before the Boundary-Scan Register takes control of the I/O pins.

2—Option 1) is referred to as the “CLAMP-like behavior” option, and option 2) is referred to as the “HIGHZ-like behavior” option because they mimic the I/O behavior of these two IEEE Std 1149.1-2001 instructions.

3—This rule still applies when an ISC instruction is nullified.

- d) When an ISC instruction is loaded, all fixed system pins (see 5.1.1.2) shall perform their system mode of operation.

NOTES

1—This will allow nonprogrammable portions of a device to perform system functions before the programmable logic is configured. This may be crucial to proper initialization of a system, particularly if the nonprogrammable portions of the device are used to control the programming process.

2—The device designer defines and documents (see 8.2) which pins are ISC system pins and which are fixed system pins. (See 5.1.1.)

- e) When IEEE Std 1149.1-2001 defined instructions BYPASS, PRELOAD, SAMPLE, IDCODE, or USERCODE are loaded, all ISC system pins (see 5.1.1.1) shall operate as follows:
 - 1) If the ISC_Done signal and the ISC_Enabled signal are both cleared, then all ISC system signals capable of being driven out of the device shall be placed in the inactive drive state (e.g., high impedance).
 - 2) If the ISC_Done signal is set and the ISC_Enabled signal is cleared, then all ISC system signals will take on their programmed states of input, output, bidirectional, or no-connect.
 - 3) If the ISC_Enabled signal is set, then all ISC system signals shall behave as given by rule d) above, including the option of 1) and 2) that was chosen for ISC instructions.

NOTES

1—The IEEE Std 1149.1-2001 instructions listed are noninvasive and allow the system mode of the device to determine the activity of the ISC system pins. This rule explains how these instructions are to interpret the system modal states of an ISC device.

2—The ISC_Done signal is an identified signal set or cleared by the ISC programming process at the satisfaction of an externally executed programming algorithm. This algorithm is solely responsible for the legitimacy and accuracy of the programming process.

- f) The actions performed by an ISC instruction shall not be affected in any way by signals present at any device system pin.

NOTE—This rule is exclusive of the statically asserted signals that may be necessary on compliance enable pins (see 5.1.3) since these are needed to guarantee conformance with IEEE Std 1149.1-2001. It also excludes pins needed to supply supernormal programming voltages (see 5.1.4).

- g) The actions performed by an ISC instruction shall begin execution only in the *Run-Test/Idle* TAP Controller state.

NOTES

1—These actions include any that manipulate (read, write, or erase) the content of program memory.

2—See 6.6 for rules governing IEEE Std 1149.1-2001 masters.

- h) The actions performed by an ISC instruction shall complete execution no later than by the rising edge of TCK that brings the TAP Controller to the *Update-DR* or *Update-IR* state.

NOTES

1—This rule accommodates device designs that use the exit from the *Run-Test/Idle* TAP Controller state as a trigger for a completion event that terminates within the few TCK cycles that it takes to shift a new data pattern or instruction into the device. It is the responsibility of the device designer to assure that the completion event will finish within this time span, which is a function of the design of the device.

2—While this rule allows a programming activity to progress beyond the time spent in the *Run-Test/Idle* TAP Controller state, it is recommended that all programming activity complete within the time spent in this state. (See rule i) below governing the minimum duration spent in *Run-Test/Idle* TAP Controller state.)

3—If the *Test-Logic-Reset* TAP Controller state is entered before either the *Update-DR* or *Update-IR* state, this will abort the currently operating instruction and may remove the ability to finish the completion event. This sequence of events may produce indeterminate results and should be avoided or specifically handled by programming tools.

4—It is possible to arrive at one of either the *Update-DR* or *Update-IR* TAP Controller states without shifting new instruction or data bits. This path would produce a theoretical minimum number of TCK cycles. However, doing so would have the effect of enabling captured bits to become the new content of a data or instruction register and thus is a nonsensical trajectory.

5—Devices that require the use of an adaptive programming algorithm should refer to permission w) of 6.1.1.

- i) A minimum duration shall be specified by the device designer for the actions performed in the *Run-Test/Idle* TAP Controller state in response to an ISC instruction, and the specification shall be one of
 - 1) A minimum number of TCK cycles
 - 2) A minimum amount of time in seconds
 - 3) A minimum number of TCK cycles followed by a minimum amount of time in seconds

NOTES

1—For any duration specification using a number of TCK cycles, the duration is independent of TCK characteristics such as frequency, duty cycle, or whether TCK is temporarily halted and then restarted.

2—Duration specified in time is needed when the timed event is a physical process such as the storage of electrons on a buried gate. A composite time specification of TCK cycles followed by time may be needed when, upon entering the *Run-Test/Idle* TAP Controller state, a physical process is initiated after some number of TCK cycles have occurred. This guarantees the composite wait time is independent of TCK characteristics such as frequency, duty cycle, or whether TCK is temporarily halted and then restarted.

- j) The component designer shall not specify an absolute maximum duration time for the actions performed in the *Run-Test/Idle* TAP Controller state.

NOTE—A recommended maximum duration is allowed by permission t) below. Exceeding this duration may have some consequences, but rule j) restricts a designer from creating a device that suffers a catastrophic outcome as a result of exceeding a recommended maximum.

- k) For ISC instructions that use an adaptive programming algorithm, the device designer shall specify a maximum possible number of iterations (or loops) that the action shall be executed.
- l) For ISC instructions that use an adaptive programming algorithm, the operation shall be terminated
 - 1) Automatically upon successful completion as shown by the operation success indication
 - 2) By the execution of an instruction that has the function of stopping the operation
 - 3) Upon leaving the ISC_Accessed system modal state
- m) For ISC instructions that produce a result, this result shall be loaded into the register targeted between TDI and TDO by the ISC instruction no later than the rising edge of TCK in the *Capture-DR* TAP Controller state, upon completing the minimum duration.

NOTE—If status indication is implemented (see v) of 5.2), then all ISC instructions produce at least a status result.

- n) Following the specified minimum duration, the result observed by loading and shifting the register targeted between TDI and TDO by an ISC instruction shall be constant regardless of when the *Capture-DR* TAP Controller state is entered.

NOTE—This is true even if permission t) below is followed.

- o) When implementing a device compliant with this standard, there shall be no other timing requirements placed upon
 - 1) The execution of instructions
 - 2) The execution of sequences of instructions
 - 3) The amount of time spent in particular TAP Controller states other than *Run-Test/Idle*

NOTE—For example, it is not allowed to specify a minimum allowable time between the execution of two instructions. This recognizes that systems that program chains of devices may not have “real-time” performance capabilities.

- p) When implementing a device compliant with this standard, there shall be no restrictions placed on TAP Controller state diagram trajectories required for instruction operation.

NOTE—For example, it is not allowed to require that while data are being shifted in the *Shift-DR* TAP Controller state, time must be (or must not be) spent in the *Pause-DR* TAP Controller state.

- q) If status indication is implemented via recommendation v) of 5.2, then the status error codes that are captured shall be
 - 1) A “10” indicating no error occurred
 - 2) A “01” indicating some error did occur (see rule r) below)

and the other codes (“11” and “00”) shall not be captured.

NOTES

1—These assignments (see Figure 17) are encoded redundantly in two bits as extra assurance that a system error does not masquerade as a nonerror condition. The passing pattern (“10”) has deliberately been chosen such that the least significant bit is different from that captured in the instruction register during the *Capture-IR* TAP Controller state or, when it is selected by the current instruction, by the Device Identification Register during the *Capture-DR* TAP Controller state. The all zero/one patterns are restricted, again to reduce the probability of a system error being interpreted as containing information.

2—The least significant bit is the rightmost bit.

- r) If status indication is implemented via recommendation v) of 5.2, then the status error indication shall be given for events that prevent proper instruction execution, including
 - 1) Executing an ISC instruction (excepting ISC_ENABLE and ISC_DISABLE) while not in the ISC Accessed system modal state
 - 2) Failure to observe the minimum instruction duration (rule i) above)
 - 3) An attempt to perform an operation that has been blocked by an enabled protection mechanism
 - 4) Other events unique to the design as provided by the device designer

NOTE—Status subcodes may be used to provide further information about a status error.

- s) During the time between the initiation of the ISC_ENABLE instruction and the termination of the ISC_DISABLE instructions, a transition by the TAP Controller to the *Test-Logic-Reset* state shall abort any current ISC instruction execution. Further, all ISC system pins (see 5.1.1.1) capable of being driven out of the device shall be placed in either
 - 1) The inactive drive state (e.g., high impedance) when the ISC_Done signal is clear
 - 2) Their programmed operational states when the ISC_Done signal is set

NOTES

1—When option 1) occurs, the device returns to the Unprogrammed system modal state (see 4.1). When option 2) occurs, the device moves to the Operational system modal state. In either instance, execution of ISC_ENABLE is required to return to the ISC Accessed system modal state.

2—The abort process is intended to safeguard the device from possible damage as a consequence of failing to complete the actions of the ISC instruction. While it is possible that the aborted instruction could somehow be “restarted” by some other process, this is not required, nor should applications assume it is possible.

3—When option 2) occurs, the fact that ISC_Done is set does not guarantee that the device is properly programmed. External programming algorithms should avoid this situation or take subsequent remedial action if this situation cannot be avoided.

Permissions

- t) Without violating rule j) above, a recommended maximum duration for time spent in the *Run-Test/Idle* TAP Controller state may be specified for an instruction as long as
 - 1) Documentation is provided about the effects of exceeding this duration
 - 2) These effects are not damaging or fundamentally detrimental to the behavior of the device

NOTES

1—This recommended maximum duration specification is intended for use by an analysis routine that determines if two or more programmable devices are suited for concurrent programming. If the minimum duration of one device significantly exceeds the recommended maximum of another device, then they should be programmed separately. This maximum duration, if exceeded, will not result in serious damage to the device, but some secondary effects may be

noted. For example, a device may be sensitive to “overprogramming,” which has the net effect of slightly reducing the number of times it may be reprogrammed.

2—Any recommended maximum duration should be significantly larger (for example, 10 or 20 times larger) than the minimum duration to maximize the opportunity for programming different devices concurrently.

- u) Where specific rules to the contrary are not given, any bits shifted into the data register targeted between TDI and TDO by an ISC instruction before entering the *Run-Test/Idle* TAP Controller state may be utilized (upon receiving a falling edge on TCK in the *Update-DR* TAP Controller state) by subsequent actions of that instruction.

NOTE—Detailed instruction descriptions following this permission may provide rules governing what these bits may symbolize or activate.

- v) For ISC instructions that use an adaptive programming algorithm, a minimum loop count may be specified for the number of times an action should be executed. If a minimum is specified, the operation success indication should be ignored until the minimum number of iterations has been met.

NOTE—By specifying a minimum loop count and by minimizing the duration specified in i) of 6.1.1, a greater amount of concurrentization will be achievable in systems that use dynamic concurrentization because it will be possible to merge n instructions with deterministic burn times with instructions that require multiple iterations where n is the minimum number of iterations specified. If no minimum loop count is specified, then n is one and it will only be possible to perform concurrent operation on the first iteration of the action for which a loop is specified.

- w) Instructions that use an adaptive programming algorithm may continue their physical operation past the time specified in h) of 6.1.1 and continue until the action has been terminated by one of the methods described in l) of 6.1.1.

NOTE—For those instructions that cannot physically be stopped until success has been achieved, this permission gives the ability to continue their action past either the *Update-DR* or *Update-IR* TAP Controller states. This may be a physical requirement of the devices in that charging and discharging the programming circuitry puts undue stress on the memory bits, which can lead to long-term reliability problems.

Recommendations

- x) ISC instructions should be deterministic in nature.
- y) Some form of the generalized optional protection capability described in 5.1.11 should be implemented.

NOTE—Many protection capabilities already in existence use a single bit to perform read-protection only. This case is a subset of the general capability given in 5.1.11.

6.1.2 Description

Several of the general rules for ISC instructions govern the behavior of device ISC system pins (see 5.1.1.1). These reflect the modal states of the “system” behavior of the device and how the ISC system pins should behave while ISC instructions are executing. Fixed system pins (see 5.1.1.2) always perform their system function while ISC instructions are executing.

Rule d) of 6.1.1 gives two possible behaviors for device ISC system pins while an ISC instruction is loaded. The first is the “CLAMP-like” behavior where I/O pins are controlled to states defined by the content of the Boundary-Scan Register previously set up by using the PRELOAD instruction. The second is the “HIGHZ-like” behavior where all system I/O pins are disabled. The CLAMP-like behavior is generally preferred for the reason of flexibility: It can be used to emulate the HIGHZ-like behavior because the preloaded pattern can simply turn off all drivers. (Note that most programmable devices have fully controllable I/O pins.) The CLAMP-like behavior can also be used to establish high or low states on certain board level nodes that can

be used to control other non-ISC devices that may be connected to those being programmed. This may be very important for maintaining system stability while programming is under way.

Rule e) of 6.1.1 states that when noninvasive IEEE Std 1149.1-2001 instructions such as BYPASS or SAMPLE are in effect

- An unprogrammed device shall keep its ISC system outputs disabled
- A successfully programmed device has system-defined states assigned to all pins
- A device undergoing programming will have its ISC system outputs controlled per rule d) of 6.1.1, as selected by the ISC_Done signal and ISC_Enabled signal states.

This rule ensures that the device's system I/O pins will function properly in each of the system modal states, as described in 4.1. The two control cell designs in Figure 12 and Figure 13 show possible Boundary-Scan Register control cell designs (for a structure such as shown in Figure 9) that will meet this requirement.

All ISC instructions perform multistep functions that may take an appreciable amount of time (or TCK cycles) to perform. This timing is performed in the *Run-Test/Idle* TAP Controller state. Just as with RUNBIST, it is intended that several ISC devices could be operated concurrently. When an amount of time for one device has elapsed, the device is expected to stop reacting to additional time or clocking so that other devices that require more time can finish their functions. It is permitted to specify a recommended maximum duration of time spent in the *Run-Test/Idle* TAP Controller state that preserves the optimum functionality of the device. This specification can be used to group compatible devices for concurrent programming.

Rules g) of 6.1.1, h) of 6.1.1, and i) of 6.1.1 govern practical timing issues encountered in device designs. Programming functions for devices with deterministic programming algorithms are to occur in the *Run-Test/Idle* TAP Controller state and the entry into and exit from this state can be used to trigger events. A common design problem comes from the desire to implement an “auto-increment” feature for implicit array addressing, as it is often the case that the address register cannot be changed while there are supernormal voltages present used to actually program a location. Thus, in an address pre-increment scheme, where the next address to program is set up before the burn occurs, some amount of TCK clocking may be required to derive the next address. Once the address is ready, the supernormal programming voltages can be applied for a timed duration. In an address post-increment scheme, the exit from the *Run-Test/Idle* TAP Controller state can be used to indicate that it is now safe to update the address register, but this must be accomplished per rule h) of 6.1.1 such that the new address is ready before a new data or instruction state can be loaded. When a device implements an adaptive programming algorithm that may require the execution of an instruction that could also use the auto-increment feature, the device designer must be extremely cautious in the design so that the register being incremented does not change until the action has been declared a success and the action has been terminated. If the register is incremented every time the instruction is executed before it achieves success, the address will be unknown to the programming software and it will be impossible to achieve programming success. It is strongly recommended not to implement auto-increment features in those instructions that require adaptive programming.

Devices with adaptive programming algorithms have programming functions that begin in the *Run-Test/Idle* TAP controller state but may continue long after that state has been exited and, in fact, may continue after the initiating instruction has been left and supplanted with another as allowed for by permission w) of 6.1.1. When this happens, one instruction may be used to initiate the programming operation and another may be used to check the status of the programming operation. While the status is being checked, the programming operation continues in the background until it has been terminated by one of the methods given in l) of 6.1.1. Status for the execution of these instructions is given by the “operation success indication” (specified in 5.1.10). This status can be evaluated either by the instruction that initiates the action or by a separate instruction. The number of times that status could potentially be checked is specified by a loop count maximum [k] of 6.1.1] and possibly by a loop count minimum. If the minimum is not specified, then only one iteration will be guaranteed and it will only be possible to perform static concurrentization on that one iteration. If a minimum number of iterations is specified, then there is a guaranteed number of times the

action will be repeated before it is even allowable to check for operation success. In this instance, concurrentization can be achieved on each of the iterations of the action up to the minimum number. In any device requiring an adaptive programming algorithm, it is recommended that a minimal number of instructions will require multiple iterations and looping to achieve success. The majority of instructions should be deterministic in nature.

Any ISC device may have its currently executing function aborted by traversing to the *Test-Logic-Reset* TAP Controller state, as given by rule t) of 6.1.1. This may be in response to an error condition sensed by an external algorithm. In this case, there are two responses: If ISC_Done is clear, the device is returned to the Unprogrammed system modal state where all device system pins are disabled; if ISC_Done is set, the device moves to the Operational system modal state where all device pins take on their programmed roles. The internal structures inside the device should be self-protecting in either event to prevent damage.

NOTE—If ISC_Done is set when an instruction is aborted, it is possible that the device is not truly ready for operation. Programming algorithms should avoid this situation if possible, or be ready to take subsequent remedial action.

6.2 Mandatory ISC instructions

The following sections define the set of mandatory ISC instructions that shall be implemented in a device that conforms to this standard. These instructions are ISC_ENABLE, ISC_DISABLE, ISC_PROGRAM, and ISC_NOOP.

6.2.1 The ISC_ENABLE instruction

The mandatory ISC_ENABLE instruction creates and initializes an environment that prepares a device's internal programming structure for programming actions of subsequent instructions (such as ISC_PROGRAM, ISC_READ, and ISC_ERASE). This environment should be properly in place for these other instructions to operate correctly. ISC_DISABLE is used upon completion of programming activities to return the device to a nonprogramming status.

6.2.1.1 Specifications

Rules

- a) Each component conforming to this standard shall provide an ISC_ENABLE instruction.
- b) The ISC_ENABLE instruction shall select either the ISC_Config register or the ISC_Default register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.
- c) The successful execution of ISC_ENABLE, including waiting for the minimum duration to expire, shall set the ISC_Enabled signal.

NOTE—The setting of ISC_Enabled is a prerequisite for any other ISC instruction to operate properly.

- d) The ISC_ENABLE instruction shall be deterministic in nature.

NOTE—This rule ensures that the ISC_ENABLE instruction will not require multiple iterations to achieve success and does not require looping.

- e) The result of executing ISC_ENABLE while the ISC_Enabled signal is set is undefined and shall not cause damage to the device.

NOTES

1—The ISC_ENABLE instruction is intended to be paired with the ISC_DISABLE instruction, creating the environment for ISC programming activities. Passage to the *Test-Logic-Reset* TAP Controller state may abort a sequence of ISC activities, but this will also clear the ISC_Enabled signal.

2—Since the effects of a second execution of `ISC_ENABLE` are undefined, tools should avoid this event.

3—The content of the `ISC_Config` register may be changed while the `ISC_Enabled` signal is set by executing the `ISC_SETUP` instruction (see 6.5.3) if it is implemented.

- f) If the `ISC_Config` register is targeted when the `ISC_ENABLE` instruction is loaded, then upon exiting the *Run-Test/Idle* TAP Controller state, the result of capturing, shifting, and updating new bits in `ISC_Config` is undefined and shall not cause damage to the device.

NOTE—Once `ISC_ENABLE` has been executed, bits in `ISC_Config` may be altered by first executing the `ISC_DISABLE` instruction and then executing `ISC_ENABLE` again, or by executing the `ISC_SETUP` instruction, if implemented. (See 6.5.3.)

- g) If the process performed by the `ISC_ENABLE` instruction senses an error condition during execution, then the `ISC_Enabled` signal shall not be set.

NOTE—Errors that may occur are failure to observe the minimum duration, as well as failure to achieve any additional conditions the designer has included in the design. For example, if charge pumps are used to produce supernormal voltages on-chip, these could fail to perform properly and produce an error signal.

Permissions

- h) When the `ISC_Config` register is targeted, higher order bits of the `ISC_Config` register may capture information specified by the component designer.

NOTES

1—If recommendation v) of 5.2 has not been exercised for this component, then all bits are available.

2—The bits shifted out of `ISC_Config` are those that existed before the execution of `ISC_ENABLE` in the *Run-Test/Idle* TAP Controller state.

- i) When the `ISC_Config` register is targeted, after loading `ISC_ENABLE` but before entering the *Run-Test/Idle* TAP Controller state, a control bit pattern specified by the component designer may be loaded in the *Shift-DR* TAP Controller state to prepare for subsequent programming activity.

6.2.1.2 Description

`ISC_ENABLE` sets up and initializes an environment for subsequent programming activities conducted by other ISC instructions. These programming activities will operate upon the device's program memory.

NOTE—The existence of this environment is represented by the `ISC_Enabled` signal set by the successful execution of the `ISC_ENABLE` instruction. (See 5.1.8.2 and 5.2.)

Parameters for controlling this environment can be loaded into the `ISC_Config` register (see 7.5) if it is implemented. In minimal implementations, the `ISC_ENABLE` instruction takes no parameters. In a slightly larger implementation, `ISC_ENABLE` may target the `ISC_Config` register for shifting parameters, or it may not target the register but simply initialize it to a default. In this second case, the `ISC_SETUP` instruction may be used to alter the content of `ISC_Config`. See 7.5 for examples of how the `ISC_Config` register may control operational parameters used during programming activities.

The internal device logic and programming function should not be affected in any way by input values arriving at the device system pins. Programming initialization can be accomplished by deriving control signals from TCK or an internally generated clock.

The rules relating to the duration of the ISC_ENABLE initialization setup action, rules g) of 6.1.1 through t) of 6.1.1, ensure that sufficient clock edges can be applied to allow completion of the setup activities concurrently in different components on an assembled board. Thus, on a board containing components with ISC_ENABLE duration requirements of 5000, 10 000, and 80 000 rising edges of TCK, the TAP Controllers shall be left in the *Run-Test/Idle* state for 80 000 rising edges of TCK to ensure that all components complete ISC_ENABLE satisfactorily.

Permissions u) of 6.1.1, h) of 6.2.1.1, and i) of 6.2.1.1 govern actions concerning the ISC_Config register in the *Capture-DR* and *Update-DR* TAP Controller states. A given component design may have control bits in the ISC_Config register that control programming options used by later instructions. These values should be loaded into the ISC_Config register before proceeding to the *Run-Test/Idle* TAP Controller state. Bits simultaneously shifted out of ISC_Config may have a meaning assigned by the device designer.

If a device has been configured for programming by ISC_ENABLE, it will maintain this configuration environment for all subsequent ISC instructions until the ISC_DISABLE instruction is executed, the ISC_SETUP instruction is executed, or the TAP controller passes into the *Test-Logic-Reset* TAP Controller state. The results of programming activities are not guaranteed, if the ISC_DISABLE instruction is omitted or incorrectly executed. Passing to the *Test-Logic-Reset* TAP Controller state while in the ISC Accessed system modal state should only be done if irrecoverable error conditions are sensed during programming activities.

6.2.2 The ISC_DISABLE Instruction

The mandatory ISC_DISABLE instruction completes internal device configuration operations. It enables the device's internal programmed structures in the manner defined by their programming. The device I/O pins then take on their programmed nature when the ISC_DISABLE instruction is displaced from the instruction register by a non-ISC instruction. (See rule d) of 6.1.1.) After completion of an ISC_DISABLE instruction, no new ISC instructions will be operable until subsequent completion of another ISC_ENABLE instruction.

The ISC_DISABLE instruction is exceptional because it is allowed to operate in both the ISC Accessed and ISC Complete system modal states.

6.2.2.1 Specifications

Rules

- a) Each component conforming to this standard shall provide an ISC_DISABLE instruction.
- b) The ISC_DISABLE instruction shall select either the ISC_Config register or the ISC_Default register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.
- c) The successful execution of ISC_DISABLE, including waiting for the minimum duration to expire, shall clear the ISC_Enabled signal.

NOTE—When the ISC_Enabled signal is cleared, the system modal state changes from ISC Accessed to ISC Complete.

- d) The ISC_DISABLE instruction shall be deterministic in nature.

NOTE—This rule ensures that the ISC_DISABLE instruction will not require multiple iterations to achieve success and does not require looping.

- e) The execution of ISC_DISABLE shall be preceded by the successful execution of the ISC_ENABLE instruction.
- f) The ISC_DISABLE instruction, when re-executed one or more times in sequence, shall not cause a status error per part 1) of rule r) of 6.1.1.

NOTES

1—This rule allows execution of ISC_DISABLE in the ISC Complete system modal state.

2—No other exceptions are allowed; for example, each re-execution shall observe the same minimum duration requirement for ISC_DISABLE.

Permissions

- g) When the ISC_Config register is targeted, higher order bits of the ISC_Config register may capture information specified by the component designer.

NOTE—If recommendation v) of 5.2 has not been exercised for this component, then all bits are available.

- h) When the ISC_Config register is targeted, after selecting ISC_DISABLE but before entering the *Run-Test/Idle* TAP Controller state, a control bit pattern specified by the component designer may be loaded in the *Shift-DR* TAP Controller state to prepare for subsequent instruction activity.

NOTE—The ISC_DISABLE instruction may set the ISC_Done signal (see e) of 5.2) at the end of the duration in the *Run-Test/Idle* TAP Controller state when a specified data pattern is loaded into the ISC_Config register before entering the *Run-Test/Idle* TAP Controller state. This mechanism for setting ISC_Done may be used in lieu of any other method. It offers the advantage that the ISC_Done signal can be set simultaneously with the clearing of the ISC_Enabled signal. (See 4.3.)

6.2.2.2 Description

The ISC_DISABLE instruction prepares the device to switch to its Operational system modal state in an orderly and well-defined manner. It is important to note that this transition from the ISC Accessed system modal state to the Operational system modal state may be critical to safe system initialization.

The use of a sequence of deliberately applied ISC_DISABLE instructions (and their subsequent displacement) allows a specific initialization sequence of devices to be followed.

If a device has been configured for programming by ISC_ENABLE, it will remain in this configured mode for all subsequent ISC instructions until either the ISC_DISABLE instruction is executed, or the TAP Controller passes into the *Test-Logic-Reset* state. The results of programming activities are not guaranteed, however, if the ISC_DISABLE instruction is omitted. Passing to the *Test-Logic-Reset* TAP Controller state while in the ISC Accessed system modal state should only be done if error conditions are reported during programming activities.

NOTE—Following use of the ISC_DISABLE instruction to restore the environment to nonprogramming status, the now-active 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., nonprogramming) operation.

6.2.3 The ISC_PROGRAM instruction

The mandatory ISC_PROGRAM instruction is used to program the device's internal configuration memory with a specified value.

NOTES

1—If the device has a program security function implemented (see 5.1.11) and this function has been activated, then the ISC_PROGRAM instruction will fail to overwrite the programmed content of the device.

2—The ISC_PROGRAM instruction may be executed concurrently (see 3.1.10 and Annex B) in multiple devices, where all are required to stay in the *Run-Test/Idle* TAP Controller state during the programming burn time.

While the ISC_PROGRAM instruction is selected, the state of all system output pins remains in the control of the mechanism described by rule d) of 6.1.1.

Noncompletion of device programming may result in unpredictable internal device behavior. If the ISC_Done signal is not yet set, these effects will not propagate to the system as a whole since the device's system pins maintain their unprogrammed behavior.

NOTE—Incomplete programming is not allowed to cause device damage, per rule t) of 6.1.1.

6.2.3.1 Specification

Rules

- a) Each component conforming to this standard shall provide an ISC_PROGRAM instruction.
- b) The ISC_PROGRAM instruction shall select either the ISC_PData register or the ISC_Default register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.
- c) When the programmable content of the device is actively program-protected by a security mechanism, the ISC_PROGRAM instruction, after the minimum duration requirement is satisfied, shall leave the existing content of program memory unchanged.

NOTES

1—See the ISC_PROGRAM_SECURITY instruction (6.4.3) for one example of how a security feature may be implemented.

2—Activation of security is controlled by the state of the ISC_Disable_Program signal. (See 5.1.11.2.)

- d) When the programmable content of the device is not actively program-protected by a security mechanism, the ISC_PROGRAM instruction, after the minimum burn time requirement is satisfied, shall overwrite the existing content of program memory.

NOTES

1—The address and data for this write operation may be supplied by some means chosen by the device designer. See 5.1.12.

2—Asymmetrically writeable devices must first be blank before the ISC_PROGRAM instruction can succeed in overwriting memory content.

3—For devices with a deterministic programming algorithm the minimum burn time is specified by i) of 6.1.1. For devices implementing an adaptive programming algorithm, the minimum burn time is satisfied when success has been signified by the operation success indication.

Permissions

- e) When the ISC_PData register is targeted, higher order bits of the ISC_PData register may capture information specified by the component designer, excepting readback of programming information if an optional read-protection feature is activated.

NOTES

1—If recommendation v) of 5.2 has not been exercised for this component, then all bits are available.

2—Activation of read-security is controlled by the state of the ISC_Disable_Read signal. (See 5.1.11.1.)

- f) When the ISC_PData register is targeted, after selecting ISC_PROGRAM but before entering the *Run-Test/Idle* TAP Controller state, a control bit pattern specified by the component designer may be loaded in the *Shift-DR* TAP Controller state to prepare for subsequent programming activity.
- g) The ISC_Done signal, protection bits, or USERCODE bits may be set as a result of programming one or more locations in the program space.

NOTE—This leaves a significant opportunity for ISC_Done to be set when the device in fact has not been properly programmed. (See 4.3.)

6.2.3.2 Description

The ISC_PROGRAM instruction facilitates loading of configuration information into the programmable device via the ISC_PData register, if targeted. The ISC_PROGRAM instruction must address the data to be stored in the memory array. A device designer has several implementation choices as discussed in 5.1.12. For example, if ISC_Default is targeted, then configuration information should be supplied by other means, such as with the ISC_DATA_SHIFT (see 6.5.2) and ISC_ADDRESS_SHIFT (see 6.5.4) instructions.

When the ISC_PData register is targeted, traversal through the *Capture-DR* TAP Controller state may load the ISC_PData register with the configuration memory address and configuration memory data placed by a previous ISC_PROGRAM operation (if read-protection is not activated) at the option of the device designer.

NOTE—Many programmable devices have asymmetrical programming capability. For example, a “1” may overwrite a “0,” but not vice-versa. To rewrite such devices with different data, an erasure process should be executed first.

In the *Shift-DR* TAP Controller state, the ISC_PData register is loaded with new configuration information. This programming information includes data, optional address information, and other optional information. Upon entry to *Update-DR* TAP Controller state, the internal device configuration registers are updated with the program information.

Upon entry to *Run-Test/Idle* TAP Controller state, the programming operation begins and continues for the duration of the period spent in *Run-Test/Idle*. If the *Run-Test/Idle* TAP Controller state is exited before the operation completes or if optional program-protection is activated, the optional status bits will reflect an error condition. If no status bits are implemented, then the failure occurs silently.

ISC_PROGRAM will not overwrite existing program data if an optional program-protection feature is activated.

6.2.4 The ISC_NOOP instruction

The mandatory ISC_NOOP instruction is used, when programming multiple devices in parallel, to place a given device in a “no-operation” mode without the device leaving the ISC Accessed system modal state. This may be done while other devices that are being concurrently programmed are executing a more complex sequence of instructions. (See Annex B.)

NOTE—The abbreviation “NOOP” stands for “No Operation” and is usually pronounced as two syllables “No Op.”

While the ISC_NOOP instruction is selected, the state of all system output pins remains in the control of the mechanism described by rule d) of 6.1.1.

6.2.4.1 Specification

Rules

- a) Each component conforming to this standard shall provide an ISC_NOOP instruction.

- b) The ISC_NOOP instruction shall select only the ISC_Default register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.
- c) The minimum duration in the *Run-Test/Idle* TAP controller state for ISC_NOOP shall be one cycle of TCK.

NOTE—Consequently, there is no way for ISC_NOOP to cause a duration error.

- d) The ISC_NOOP instruction shall be deterministic in nature.

NOTE—This rule ensures that the ISC_NOOP instruction will not require multiple iterations to achieve success and does not require looping.

- e) The content of the ISC_Default register shall have no effect on the device when executing the ISC_NOOP instruction.

NOTE—In particular, the execution of ISC_NOOP has no effect on the addressing mechanism within the device. For example, if the internal addressing mechanism is capable of auto-incrementing, this capability must not be triggered by the execution of ISC_NOOP.

Permissions

- f) Higher order bits of the ISC_Default register may capture information specified by the component designer.

NOTE—If recommendation v) of 5.2 has not been exercised for this component, then all bits are available.

6.2.4.2 Description

The ISC_NOOP instruction is intended for use in one or more devices when they are being programmed concurrently as discussed in Annex B. In scenarios where dissimilar devices (that comply with this standard) are being programmed, it may happen that one device requires fewer operations than another to complete concurrent operations. The simpler device has to perform some instruction while the more complex device is completing extra steps in the programming process. The ISC_NOOP instruction fills this need, by safely doing no operation but maintaining the device in the ISC Accessed system modal state.

The execution of the ISC_NOOP instruction may be indistinguishable from the BYPASS instruction for some devices, except in the case where ISC instruction execution has been nullified (for example, when not preceded by the proper execution of ISC_ENABLE). In this case, the I/O pin behavior is governed by rule c) of 6.1.1 while BYPASS would either disable the pins (in the Unprogrammed system modal state) or place the pins in their operational mode (in the Operational system modal state). For this reason, ISC_NOOP must be a separate instruction from BYPASS.

6.3 Optional ISC programming instructions

The following sections define the set of optional ISC programming instructions that a device may implement. These instructions are ISC_READ, ISC_ERASE, ISC_PROGRAM_USERCODE, and ISC_DISCHARGE.

6.3.1 The ISC_READ instruction

The optional ISC_READ instruction is used to read the programmed values of the device's internal configuration memory. This instruction is active only following an ISC_ENABLE instruction and before issuance of an ISC_DISABLE.

NOTE—If the device has a read security function implemented (see 5.1.11) and this function has been activated, then the ISC_READ instruction will fail to read back the programmed content of the device.

While the ISC_READ instruction is selected, the state of all system output pins remains in the control of the mechanism described by rule d) of 6.1.1.

6.3.1.1 Specification

Rules

- a) The optional ISC_READ instruction shall select either the ISC_RData register (see 7.7) or the ISC_PData register (see 7.6) to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.

NOTE—The appearance of the ISC_PData register in this rule allows for the case where ISC_READ and ISC_PROGRAM both target the same physical register. When both instructions use the same register, programming and verifying data can potentially be done more efficiently by loading and operating the two instructions successively, when the overhead of instruction loading is small compared to data scanning.

- b) The ISC_READ instruction, after the minimum duration requirement is satisfied, shall return valid readback data in the ISC_RData register only when the programmable content of the device is not actively read-protected by a security mechanism.

NOTE—Activation of security is controlled by the state of the ISC_Disable_Read signal. (See 5.1.11.1.)

- c) When the programmable content of the device is actively read-protected by a security mechanism and recommendation v) of 5.2 has not been exercised for this component, the ISC_READ instruction, after the minimum duration requirement is satisfied, shall return readback data in the ISC_RData register that is indistinguishable from data that would be observed if the device was unprogrammed.

NOTES

1—If recommendation v) of 5.2 has been exercised for this component, then no requirement is made upon the form of the data returned other than it should not match the programmed values. Rule r) of 6.1.1 will then allow programming algorithms to determine if protection has been activated.

2—See the ISC_PROGRAM_SECURITY instruction (6.4.3) for one example of how a security feature may be implemented.

3—Activation of security is controlled by the state of the ISC_Disable_Read signal. (See 5.1.11.1.)

Permissions

- d) Higher order bits of the ISC_RData register may capture information specified by the component designer.

NOTES

1—If recommendation v) of 5.2 has not been exercised for this component, then all bits are available.

2—This information represents readback data and potentially other data content in a form specified by the component designer.

- e) After selecting ISC_READ but before entering the *Run-Test/Idle* TAP Controller state, a control bit pattern specified by the component designer may be loaded in the *Shift-DR* TAP Controller state to prepare for subsequent read activity.

NOTE—The control bit pattern may be used to initialize or otherwise control an internal addressing generator.

- f) Upon passage through the *Capture-DR* TAP Controller state (per rule b) above) bits shifted into the ISC_RData register in order to read out captured data may form a control bit pattern specified by the component designer to prepare for subsequent read activity.

NOTE—The control bit pattern may be used to initialize or otherwise control an internal addressing generator on subsequent executions of reading back data.

Recommendations

- g) Rule c) above should be followed even when recommendation v) of 5.2 has been exercised for this component.

NOTE—This proves unambiguously that an activated protection mechanism is working properly.

6.3.1.2 Description

The ISC_READ instruction facilitates readback of previously programmed configuration information from the programmable device via the ISC_RData register.

The ISC_READ instruction must address the data to be read back. A device designer has several implementation choices as discussed in 5.1.12.

After execution of ISC_READ in the *Run-Test/Idle* TAP Controller state, traversal through the *Capture-DR* TAP Controller state loads the ISC_RData register with the program configuration memory data values specified by the configuration memory address bits of the previously loaded ISC_Address register (see 7.11) or an internal address register of the device.

Depending on the choice of addressing implementation, while captured readback data are being shifted out, new addressing information (if any) may be shifted in. In the *Shift-DR* TAP Controller state, the ISC_RData register may be loaded with configuration memory address bits relevant to the ISC_READ operation to be performed. In the case where the configuration memory address bits are implicit, the address need not be specified. Upon entry to the *Update-DR* TAP Controller state, the internal device configuration registers are updated with the configuration memory address location to read.

Upon entry to the *Run-Test/Idle* TAP Controller state, the ISC_READ operation begins and continues for the duration of the period spent in *Run-Test/Idle*.

If the *Run-Test/Idle* TAP Controller state is exited before the operation completes, the optional status bits will reflect an error condition. If no status bits are implemented, then the failure occurs silently. The bits read back may be in error due to insufficient duration timing rather than because they are actually incorrectly programmed.

In the absence of independent status bits, the value of the data word shifted out should reflect the content of the addressed memory. However, if the status bits are present, then they are appended to the readout data in the least significant bit positions.

Rules b) and c) of 6.3.1.1 provide for a readback protection capability that, when enabled, prevents ISC_READ from capturing and shifting out configuration memory values. Rule c) of 6.3.1.1 allows software to verify that protected locations are indeed secure when independent status bits are not implemented. A test can be implemented that expects to see the design-specified pattern that should be read back when programmed locations are secured, rather than simply failing to see the programmed data. Thus, a test for

security can be designed to pass rather than expected to fail (which would mean, “pass”). Rule r) of 6.1.1 allows a simpler scheme that makes use of the optional status register.

6.3.2 The ISC_ERASE instruction

The optional ISC_ERASE instruction is used to erase the device’s internal configuration memory. The instruction may support any or all of individual address, sector, or bulk erase types of operation. When options exist, these are selected by the content of the ISC_Sector register.

NOTE—If the device has an erase security function implemented (see 5.1.11) and this function has been activated, then the ISC_ERASE instruction will fail to erase the programmed content of the device.

While the ISC_ERASE instruction is selected, the state of all system output pins remains in the control of the mechanism described by rule d) of 6.1.1.

6.3.2.1 Specification

Rules

- a) The optional ISC_ERASE instruction shall select either the ISC_Sector register or the ISC_Default register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.
- b) When the programmable content of the device is actively erase-protected by a security mechanism, the ISC_ERASE instruction, after the minimum duration requirement is satisfied, shall leave the existing content of program memory, including the state of the ISC_Done signal, unchanged.

NOTES

1—See the ISC_PROGRAM_SECURITY instruction (6.4.3) for one example of how a security feature may be implemented.

2—Activation of security is controlled by the state of the ISC_Disable_Erase signal. (See 5.1.11.3.)

- c) When the programmable content of the device is not actively erase-protected by a security mechanism, the ISC_ERASE instruction, after the minimum burn time requirement is satisfied, shall erase the existing content of program memory, and clear the state of the ISC_Done signal.

NOTES

1—Erasure returns program memory to the blank (unprogrammed) state.

2—If this instruction targets the ISC_Default register, then no data are used to govern the erasure process and the erasure will be performed on all erasable bits, known as a “bulk erase.”

3—If a device has a sectorized erase feature that allows portions of program memory to be erased, the ISC_Done signal is cleared even when the device may be usable in this partially erased mode. In this event, the process for setting the ISC_Done bit will need to be executed to restore the device to operational capability.

4—For devices with a deterministic programming algorithm, the minimum burn time is specified by i) of 6.1.1. For devices implementing an adaptive programming algorithm, the minimum burn time is satisfied when success has been signified by the operation success indication.

Permissions

- d) When the ISC_Sector register is targeted, higher order bits of the ISC_Sector register may capture information specified by the component designer, excepting readback of programming information if an optional read-protection feature is activated.

NOTES

1—If recommendation v) of 5.2 has not been exercised for this component, then all bits are available.

2—Activation of read-security is controlled by the state of the ISC_Disable_Read signal. (See 5.1.11.1.)

- e) When the ISC_Sector register is targeted, after selecting ISC_ERASE but before entering the *Run-Test/Idle* TAP Controller state, a control bit pattern specified by the component designer may be loaded in the *Shift-DR* TAP Controller state to prepare for subsequent erasure activity.
- f) The ISC_Sector register may comprise sector or configuration memory address data indicating the sector or particular configuration memory address location of the configuration memory to be erased.

Recommendations

- g) If the programmable content of the device is not actively erase-protected, the ISC_ERASE instruction should clear the ISC_Done signal as early as possible in the erasure process, independent of any data pattern intended for control contained in the ISC_Sector register.

NOTE—This reduces the probability that an incompletely erased device still has its ISC_Done signal set.

6.3.2.2 Description

The ISC_ERASE instruction facilitates erasing either individually specified configuration addresses, predetermined sectors, or the entire device in a bulk erase operation. In the case of the first two implementations, the ISC_Sector register defines the exact address or sector to be erased. In the bulk erase implementation, ISC_Default can be targeted. During the ISC_ERASE operation, the device system pins are not disturbed from the state set during the ISC_ENABLE instruction.

The ISC_ERASE instruction clears the ISC_Done signal, regardless of any configuration data that may be provided in the ISC_Sector register, if there is no erasure-protection mechanism enabled. Similarly, the instruction also clears any security protection bits that may exist in the device (provided erase-protection has not been set). The designer is responsible for assuring that security bits are cleared after all other bits intended for protection have been erased, to prevent a partial erasure allowing remaining programming bits to become visible.

After execution of ISC_ERASE for the appropriate duration in the *Run-Test/Idle* TAP Controller state, traversal through the *Capture-DR* TAP Controller state loads the ISC_Sector register with the results from the erase operation. This data may be the erased configuration memory address or sector as well as any status information relevant to the just-executed ISC_ERASE operation.

In the *Shift-DR* TAP Controller state, the ISC_Sector register can be loaded with new erase address or sector information. Upon entry to the *Update-DR* TAP Controller state, the internal device sector registers are updated with erasure information.

Upon entry to the *Run-Test/Idle* TAP Controller state, the erasure operation begins and continues for the duration of the period spent in *Run-Test/Idle*. If the *Run-Test/Idle* TAP Controller state is exited before the operation completes, the optional status bits will reflect an error condition. If no status bits are implemented, then the failure occurs silently but may be detected by reading back the programmed values.

ISC_ERASE will not overwrite existing program data if an optional erase-protection feature is activated.

6.3.3 The ISC_PROGRAM_USERCODE instruction

The optional ISC_PROGRAM_USERCODE instruction is used to set the 32-bit user code information for a device.

NOTE—If the device has a write protection mechanism (see 5.1.11) and this function has been activated, then the ISC_PROGRAM_USERCODE instruction will fail to overwrite existing user code bits.

The ISC_PROGRAM_USERCODE instruction targets the Device Identification Register between TDI and TDO. Bits shifted into the Device Identification Register are transferred into user code memory by passing through the *Update-DR* TAP Controller state and waiting for the specified duration in the *Run-Test/Idle* TAP Controller state. If the required duration is not met, then the state of the user code bits as read out by the USERCODE instruction may be indeterminate.

NOTE—For devices that are asymmetrical with respect to programming data, the user code may not be overwritten without first erasing the device.

While the ISC_PROGRAM_USERCODE instruction is selected, the state of all system output pins remains in the control of the mechanism described by rule d) of 6.1.1.

The following rules apply where the optional ISC_PROGRAM_USERCODE instruction is provided.

6.3.3.1 Specification

Rules

- a) The optional ISC_PROGRAM_USERCODE instruction shall select the Device Identification Register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.
- b) When the programmable content of the device is actively program-protected by a security mechanism, the ISC_PROGRAM_USERCODE instruction, after the minimum duration requirement is satisfied, shall leave the existing content of the user code memory unchanged.

NOTES

1—See the ISC_PROGRAM_SECURITY instruction (6.4.3) for one example of how a security feature may be implemented.

2—Activation of security is controlled by the state of the ISC_Disable_Program signal. (See 5.1.11.2.)

- c) When the programmable content of the device is not actively program-protected by a security mechanism, the ISC_PROGRAM_USERCODE instruction, after the minimum burn time requirement is satisfied, shall update the content of the user code memory.

NOTES

1—For devices that are asymmetrical with respect to programming data, the user code may not be successfully programmed if the device is not blank.

2—For devices with a deterministic programming algorithm, the minimum burn time is specified by i) of 6.1.1. For devices implementing an adaptive programming algorithm, the minimum burn time is satisfied when success has been signified by the operation success indication.

- d) If permission v) of 5.2 has been implemented for this device, then upon passing the *Capture-DR* TAP Controller state, the Device Identification Register shall contain the status result and the higher order bits may be specified by the device designer.

NOTE—This rule clarifies that the bits captured in the Device Identification Register must contain the status result if a status reporting scheme has been implemented. If status has not been implemented, then, at the option of the device designer, the Device Identification Register could capture the current user code.

- e) The updated state of the 32-bit user code pattern installed by the ISC_PROGRAM_USERCODE instruction shall be visible in the Device Identification Register no later than upon successful completion of ISC_DISABLE instruction.

NOTE—Thus, if the USERCODE instruction is executed while still in the ISC Accessed system modal state, the user code pattern read may or may not reflect the newly installed user code.

6.3.3.2 Description

The ISC_PROGRAM_USERCODE instruction complements the USERCODE instruction specified by IEEE Std 1149.1-2001 by targeting the Device Identification Register for the purpose of writing a 32-bit user code pattern into the device. A pattern of user code bits is logically linked to other device programming, so if the device has been program-protected by activating the ISC_Disable_Program signal, then the user code cannot be changed. If status reporting has been implemented, then it can be immediately determined that the instruction cannot change the user code. If not, then the USERCODE instruction can be used to read out and verify the user code.

6.3.4 The ISC_DISCHARGE instruction

Depending on implementation, an ISC device compliant with this standard may require an additional “discharge” step after certain programming functions have been performed. Typically, these are the programming and erasure functions of the ISC_PROGRAM, ISC_ERASE, ISC_PROGRAM_DONE, ISC_ERASE_DONE, and ISC_PROGRAM_SECURITY instructions. These instructions may set up charge pumps and temporarily create supernormal voltages within the device needed for programming functions that should be discharged before completing ISC operations. The ISC_DISCHARGE instruction is intended for this purpose.

While the ISC_DISCHARGE instruction is selected, the state of all system output pins remains in the control of the mechanism described by rule d) of 6.1.1.

The following rules apply where the optional ISC_DISCHARGE instruction is provided.

6.3.4.1 Specification

Rules

- a) The optional ISC_DISCHARGE instruction shall select either the ISC_Sector register or the ISC_Default register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.

Permissions

- b) When the ISC_Sector register is targeted, the content of the ISC_Sector register may be used to control discharging activities when executing the ISC_DISCHARGE instruction.

NOTE—One item that may be controlled is the specification of a minimum duration (see next permission).

- c) The minimum duration may vary depending on what portion of the programming circuitry is being discharged.
- d) The ISC_DISCHARGE instruction shall be deterministic in nature.

NOTE—This rule ensures that the ISC_DISCHARGE instruction will not require multiple iterations to achieve success and does not require looping.

- e) When the ISC_Sector register is targeted, higher order bits of the ISC_Sector register may capture information specified by the component designer.

NOTE—If recommendation v) of 5.2 has not been exercised for this component, then all bits are available.

6.3.4.2 Description

The ISC_DISCHARGE instruction, if implemented, controls a discharging process required after other instructions have created a charged condition within the programming circuitry. The instruction may work homogeneously for all discharge scenarios or it may need to execute with varying minimum duration depending on what is being discharged.

The ISC_DISCHARGE instruction may target the ISC_Default register if no information needs to be set up before entering the *Run-Test/Idle* TAP Controller state for the minimum duration. If some information does need to be provided, for example, to select a portion of the circuitry for discharging (possibly with a unique minimum duration requirement), then this instruction may target the ISC_Sector register to supply that data.

6.4 Optional ISC program control/security instructions

Several optional ISC instructions are defined that fulfill physical requirements of programming, or allow specific features such as the ISC_Done signal or program security bits to be accessed in an independent fashion.

6.4.1 The ISC_PROGRAM_DONE instruction

As noted in e) of 5.2, there are implementation choices for setting the state of the ISC_Done signal. One choice is by implementing an ISC_PROGRAM_DONE instruction that specifically targets the ISC_Done signal.

While the ISC_PROGRAM_DONE instruction is selected, the state of all system output pins remains in the control of the mechanism described by rule d) of 6.1.1.

The following rules apply where the optional ISC_PROGRAM_DONE instruction is provided.

6.4.1.1 Specification

Rules

- a) The optional ISC_PROGRAM_DONE instruction shall select either the ISC_PData register or the ISC_Default register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.
- b) When the programmable content of the device is actively program-protected by a security mechanism, the ISC_PROGRAM_DONE instruction, after the minimum duration requirement is satisfied, shall leave the existing state of the ISC_Done signal unchanged.
- c) When the programmable content of the device is not actively program-protected by a security mechanism, the ISC_PROGRAM_DONE instruction, after the minimum duration requirement is satisfied, shall set the ISC_Done signal.

NOTE—For devices with a deterministic programming algorithm, the minimum burn time is specified by i) of 6.1.1. For devices implementing an adaptive programming algorithm, the minimum burn time is satisfied when success has been signified by the operation success indication.

Permissions

- d) When the ISC_PData register is targeted, higher order bits of the ISC_PData register may capture information specified by the component designer.

NOTE—If recommendation v) of 5.2 has not been exercised for this component, then all bits are available.

6.4.1.2 Description

The ISC_PROGRAM_DONE instruction will set the ISC_Done signal indicating that the device has been programmed to the satisfaction of an external algorithm. It should be noted that if this instruction is implemented, then there is a window of time between the setting of ISC_Done and the clearing of the ISC_Enabled signal (by successfully executing ISC_DISABLE) where the device may appear to be properly programmed when the completion of all necessary activities has not occurred.

6.4.2 The ISC_ERASE_DONE instruction

As noted in f) of 5.2, there are implementation choices for clearing the state of the ISC_Done signal. One choice is by implementing an ISC_ERASE_DONE instruction that specifically targets the ISC_Done signal.

While the ISC_ERASE_DONE instruction is selected, the state of all system output pins remains in the control of the mechanism described by rule d) of 6.1.1.

The following rules apply where the optional ISC_ERASE_DONE instruction is provided.

6.4.2.1 Specification

Rules

- a) The optional ISC_ERASE_DONE instruction shall select the ISC_Default register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.
- b) When the programmable content of the device is actively erase-protected by a security mechanism, the ISC_ERASE_DONE instruction, after the minimum duration requirement is satisfied, shall leave the existing state of the ISC_Done signal unchanged.
- c) When the programmable content of the device is not actively erase-protected by a security mechanism, the ISC_ERASE_DONE instruction, after the minimum burn time requirement is satisfied, shall clear the ISC_Done signal.

NOTE—For devices with a deterministic programming algorithm, the minimum burn time is specified by i) of 6.1.1. For devices implementing an adaptive programming algorithm, the minimum burn time is satisfied when success has been signified by the operation success indication.

6.4.2.2 Description

The ISC_ERASE_DONE instruction is used by an external programming algorithm to clear the ISC_Done signal. As an example, before erasing the programming in a device, the ISC_Done signal can be cleared so that at any time before the successful completion of the erasure process, the device cannot be mistaken for a properly programmed device.

6.4.3 The ISC_PROGRAM_SECURITY Instruction

A programmable device may contain an internally maintained security function that effectively disables the ISC_READ instruction, preventing the readback of program data, and may prevent other operations such as programming and erasure as described in 5.1.11. The ISC_PROGRAM_SECURITY instruction provides

direct access to the security function. Gradations of security may be supported, from permanent security to less security based on a key scheme as described in 5.1.11.

If this optional instruction is omitted, security may optionally be implemented by use of the ISC_PROGRAM instruction writing specific data to one or more locations in program memory.

While the ISC_PROGRAM_SECURITY instruction is selected, the state of all system output pins remains in the control of the mechanism described by rule d) of 6.1.1.

The following rules apply where the optional ISC_PROGRAM_SECURITY instruction is provided.

6.4.3.1 Specification

Rules

- a) The optional ISC_PROGRAM_SECURITY instruction shall select the ISC_PData register, the ISC_Sector register, or the ISC_Default register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.

NOTE—The use of the ISC_Default register is allowed when this instruction is used simply to set a single bit (regardless of what may be shifted into the ISC_Default register) that can only be cleared by erasing the device. Simple read-protection schemes in existing programmable devices work in this manner.

- b) When the security of the device is actively protected by a security keying or locking mechanism, the ISC_PROGRAM_SECURITY instruction, after the minimum duration requirement is satisfied, shall leave the existing security setting unchanged.

NOTE—The changing of security may be prevented by an optional locking or keying concept. (See 5.1.11.)

- c) When the programmable content of the device is not actively program-protected by a security mechanism, the ISC_PROGRAM_SECURITY instruction, after the minimum burn time requirement is satisfied, shall update the content of the security memory.

NOTES

1—The new state of security could take effect immediately upon completion of the duration requirement for ISC_PROGRAM_SECURITY in some implementations, but for others, unusual programming voltages may be present that make this difficult to achieve practically.

2—For devices with a deterministic programming algorithm, the minimum burn time is specified by i) of 6.1.1. For devices implementing an adaptive programming algorithm, the minimum burn time is satisfied when success has been signified by the operation success indication.

- d) The effect of programming a security function shall take effect no later than upon successful completion of the ISC_DISABLE instruction.

NOTES

1—The new state of security could take effect immediately upon completion of the duration requirement for ISC_PROGRAM_SECURITY in some implementations, but for others, unusual programming voltages may be present that make this difficult to achieve practically.

2—If a transition to the *Test-Logic-Reset* TAP Controller state occurs before ISC_DISABLE is successfully completed, then the state of the security function may be undefined. It is the responsibility of the controlling algorithm to either avoid or contend with this situation.

- e) The enabling of a read-protection security mechanism shall not prevent the operation of the USERCODE instruction.

NOTE—The USERCODE instruction may read the actual 32-bit result from the programmable memory, but this may not be inhibited by the setting of a security function.

Permissions

- f) When either the ISC_PData or ISC_Sector register are targeted, higher order bits of the register may capture information specified by the component designer.

NOTE—If recommendation v) of 5.2 has not been exercised for this component, then all bits are available.

- g) The content of the ISC_PData or ISC_Sector register may be used to specify (see 5.1.11)
 - 1) The states of any ISC_Disable_Read, ISC_Disable_Program, and ISC_Disable_Erase signals
 - 2) Any additional “key” bits used to enhance security
 - 3) Other addressing bits (for example, sector addressing) needed to specify security

NOTE—Via permission h) below, if a key system has been implemented, the content of ISC_PData or ISC_Sector may later be used as an unlocking key to change the programmed security options.

- h) If either the ISC_PData register or ISC_Sector register is chosen for placement between TDI and TDO, it may be loaded with a key value that later may be used by a second execution of ISC_PROGRAM_SECURITY that unlocks the security function for changing the programmed security options. (See 5.1.11.)

NOTES

1—A designated key value of, for example, all zeros might be a “no key” value that always allows the circumvention of security. This would allow users to select whether or not they want to leave the possibility of reading protected data open for the future.

2—If a key scheme is implemented, then the device designer should ensure that an attacker cannot break the security feature by executing the ISC_PROGRAM_SECURITY instruction a second time, overwriting an unknown key with a known key.

3—The key could be transformed by a logic function before it is stored. Thus, knowledge of the original key would not allow an attacker to unlock the current state of security. The transforming logic function could also be dependent on other bits previously stored in other portions of memory yielding user-defined encryption.

6.4.3.2 Description

The optional ISC_PROGRAM_SECURITY instruction activates the protection feature that may exist within a device that prevents the readback of device programming, overwriting of data, and erasure of data. A security feature may not inhibit the operation of the USERCODE instruction.

At the option of the designer (see permission h) of 6.4.3.1), a security key system may be implemented. The first execution of the ISC_PROGRAM_SECURITY instruction stores a key value that was shifted into the ISC_PData register and secures the device from performing subsequent ISC_READ operations. Later, a subsequent execution of ISC_PROGRAM_SECURITY with the same key (or some encrypted version of the original key) may be used to unlock the security feature. Designating some key value (for example, all ones) as a perpetual locking value could be used to permanently secure the device’s programming at the option of the programmer. If a security setting has itself been protected with a key or lock, then the ISC_PROGRAM_SECURITY instruction can fail to change security, which may be indicated if the status reporting mechanism is implemented.

6.5 Optional address/data access instructions

These optional instructions allow individual access to specific data or addressing fields contained within the programming structure.

6.5.1 The ISC_READ_INFO instruction

This optional instruction gives access to an ISC data register (ISC_Info) containing information provided by the device vendor that may be used for support purposes. For example, the data register may contain mask identification, foundry location, date code, and lot number, programmed during the production of the device. On request, a user could read out this information and provide it to the device vendor to help diagnose problems.

The length, content, and organization of this data are design specific.

While the ISC_READ_INFO instruction is selected, the state of all system output pins remains in the control of the mechanism described by rule d) of 6.1.1.

The following rules apply where the optional ISC_READ_INFO instruction is provided.

6.5.1.1 Specification

Rules

- a) The optional ISC_READ_INFO instruction shall select the ISC_Info register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.
- b) The content of the ISC_Info register shall have no effect on the device when executing the ISC_READ_INFO instruction.
- c) The ISC_READ_INFO instruction shall be deterministic in nature.

NOTE—This rule ensures that the ISC_READ_INFO instruction will not require multiple iterations to achieve success and does not require looping.

- d) The content of the ISC_Info register shall be read-only and not modifiable by users.

NOTE—The content of the register may be installed during manufacturing by the device vendor using a private process and not subject to erasure by use of publicly available ISC instructions.

- e) The content of ISC_Info shall be readable when read-protection circuitry in a device is activated.
- f) Tools that display the ISC_Info register content shall represent it as a string of either binary or hexadecimal digits with the least significant bit (the first shifted out on TDO) appearing in the least significant bit of the rightmost digit.

NOTES

1—If recommendation v) of 5.2 has been exercised for this component, then the status indication bits are included in this display.

2—The content of the register is not intended for additional processing by tools, particularly since the content of the register is uniquely defined by the device vendor and could be subject to change. The content could be thought of as a “silicon ink stamp.” The reasoning for rule f) is to assure that among tools, there is an unambiguous representation for the data read from devices.

6.5.1.2 Description

The ISC_READ_INFO instruction allows the reading of vendor related data from the device. Status bits (if implemented) may reflect the current operational state (e.g., blank, secured) of the device. Other data may be preprogrammed by the device vendor (e.g., burn time, die information) that may be of use in support situations.

6.5.2 The ISC_DATA_SHIFT instruction

The optional ISC_DATA_SHIFT instruction selects the ISC_Data register to be connected for serial access between TDI and TDO in the *Shift-DR* TAP Controller state. This instruction facilitates loading of program data into the device's programming data register as well as reading values out of the device's internal configuration memory.

While the ISC_DATA_SHIFT instruction is selected, the state of all system output pins remains in the control of the mechanism described by rule d) of 6.1.1.

The following rules apply where the optional ISC_DATA_SHIFT instruction is provided.

6.5.2.1 Specification

Rules

- a) The optional ISC_DATA_SHIFT instruction shall select the ISC_Data register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.
- b) The ISC_DATA_SHIFT instruction shall be deterministic in nature.

NOTE—This rule ensures that the ISC_DATA_SHIFT instruction will not require multiple iterations to achieve success and does not require looping.

Permissions

- c) Higher order bits of the ISC_Data register may capture information specified by the component designer.

NOTE—If recommendation v) of 5.2 has not been exercised for this component, then all bits are available.

- d) The minimum duration in the *Run-Test/Idle* TAP Controller state for ISC_DATA_SHIFT may be one cycle of TCK.

NOTE—Consequently, there is no way to cause a duration error.

6.5.2.2 Description

The ISC_DATA_SHIFT instruction gives access to the data register used in programming or reading data. This register may physically be a portion of the ISC_PData/ISC_RData register. This register may not require any burn time to load, but since this instruction is intended to be executable concurrently with other ISC instructions that may require a duration in the *Run-Test/Idle* TAP Controller state, it should have a minimum duration of at least one cycle of TCK.

6.5.3 The ISC_SETUP instruction

The optional ISC_SETUP instruction selects the ISC_Config register to be connected for serial access between TDI and TDO in the *Shift-DR* TAP Controller state. This instruction facilitates loading or updating

configuration data in a device. This data may be used during subsequent instruction operations to modify or adjust their behavior.

Note that both the `ISC_ENABLE` and `ISC_DISABLE` instructions may also (optionally) target the `ISC_Config` register. Examples showing recommended uses of the `ISC_Config` register are provided in 7.5.

While the `ISC_SETUP` instruction is selected, the state of all system output pins remains in the control of the mechanism described by rule d) of 6.1.1.

The following rules apply where the optional `ISC_SETUP` instruction is provided.

6.5.3.1 Specification

Rules

- a) The optional `ISC_SETUP` instruction shall select the `ISC_Config` register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.
- b) The `ISC_SETUP` instruction shall be deterministic in nature.

NOTE—This rule ensures that the `ISC_SETUP` instruction will not require multiple iterations to achieve success and does not require looping.

Permissions

- c) Higher order bits of the `ISC_Config` register may capture information specified by the component designer.

NOTE—If recommendation v) of 5.2 has not been exercised for this component, then all bits are available.

- d) The minimum duration in the *Run-Test/Idle* TAP Controller state for `ISC_SETUP` may be one cycle of TCK.

NOTE—Consequently, there is no way to cause a duration error.

6.5.3.2 Description

The `ISC_SETUP` instruction gives access to the `ISC_Config` register used to control certain instruction behaviors (see 7.5). This register may not require any burn time to load, but since this instruction is intended to be executable concurrently with other ISC instructions that may require a duration in the *Run-Test/Idle* TAP Controller state, it should have a minimum duration of at least one cycle of TCK.

6.5.4 The `ISC_ADDRESS_SHIFT` Instruction

The optional `ISC_ADDRESS_SHIFT` instruction selects the `ISC_Address` register to be connected for serial access between TDI and TDO in the *Shift-DR* TAP Controller state. This instruction facilitates loading of the programming address into the device's programming address register for programming of bits previously loaded via the `ISC_DATA_SHIFT` instructions. This instruction would also be used to facilitate the reading of previously configured values at the specified address into `ISC_Data` registers. While an address can be shifted into the `ISC_Address` register, the current content of the register can also be read out. This may be used (in some architectures) to allow an interruptible programming flow, where a serial data read or programming process can be stopped and later restarted at the next valid address.

While the `ISC_ADDRESS_SHIFT` instruction is selected, the state of all system output pins remains in the control of the mechanism described by rule c) of 6.1.1.

The following rules apply where the optional ISC_ADDRESS_SHIFT instruction is provided.

6.5.4.1 Specification

Rules

- a) The optional ISC_ADDRESS_SHIFT instruction shall select the ISC_Address register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.
- b) The current state of the internal addressing structure shall be captured in the ISC_Address register on the rising edge of TCK in the *Capture-DR* TAP Controller state.

NOTE—If recommendation v) of 5.2 has been exercised for this component, then the address information is appended to the status information.

- c) A new state for the internal addressing structure shall be loaded from the ISC_Address register on the falling edge of TCK in the *Update-DR* TAP Controller state.

NOTE—These rules can be used by a process to read out and later restore an address.

- d) The ISC_ADDRESS_SHIFT instruction shall be deterministic in nature.

NOTE—This rule ensures that the ISC_ADDRESS_SHIFT instruction will not require multiple iterations to achieve success and does not require looping.

Permissions

- e) The minimum duration in the *Run-Test/Idle* state for ISC_ADDRESS_SHIFT may be one cycle of TCK.

NOTE—Consequently, there is no way to cause a duration error.

6.5.4.2 Description

The ISC_ADDRESS_SHIFT instruction gives access to the address register used in programming or reading data. This register may physically be a portion of the ISC_PData/ISC_RData register. This register may not require any burn time to load, but since this instruction is intended to be executable concurrently with other ISC instructions that may require a duration in the *Run-Test/Idle* TAP Controller state, it should have a minimum duration of at least one cycle of TCK.

6.5.5 The ISC_INCREMENT instruction

The optional ISC_INCREMENT instruction triggers the internal increment of the previously loaded programming address (for example, via the ISC_ADDRESS_SHIFT instruction). The ISC_INCREMENT instruction may select the ISC_Inc register to be connected between TDI and TDO. ISC_Inc may be used to specify an operand or command for use by ISC_INCREMENT.

While the ISC_INCREMENT instruction is selected, the state of all system output pins remains in the control of the mechanism described by rule d) of 6.1.1.

6.5.5.1 Specification

Rules

- a) The optional ISC_INCREMENT instruction shall select either the ISC_Inc register or the ISC_Default register to be connected between TDI and TDO in the *Shift-DR* TAP Controller state.

- b) The execution of ISC_INCREMENT in the *Run-Test/Idle* TAP Controller state shall update the address register to the next valid address in a sequence of addresses, unless the current address is the last address in the sequence.

NOTES

1—If an attempt is made by an algorithm to increment the last address, the resulting state of the address register is undefined. It is the responsibility of an algorithm to know the bounds of the address sequence and to avoid utilizing an undefined address.

2—This rule forbids the increment action from occurring outside of the *Run-Test/Idle* TAP Controller state, for example, at the *Update-DR* TAP Controller state. Thus, if for some reason the *Run-Test/Idle* TAP Controller state is bypassed, the increment will not be performed.

- c) The ISC_INCREMENT instruction shall be deterministic in nature.

NOTE—This rule ensures that the ISC_INCREMENT instruction will not require multiple iterations to achieve success and does not require looping.

Permissions

- d) When the ISC_Inc register is targeted after selecting ISC_INCREMENT but before entering the *Run-Test/Idle* TAP Controller state, a bit pattern specified by the component designer may be loaded in the *Shift-DR* TAP Controller state to control the operation of ISC_INCREMENT.

NOTES

1—Bit patterns supplied by the component designer may be used to skip empty spaces in the addressing sequence, but it is the responsibility of the component designer to provide this data as part of any programming or readback process.

2—If the ISC_INCREMENT instruction does not require data provided in the ISC_Inc register for operation, then a single execution of this instruction will always update the address to the next valid address even when this address space is fragmented with unused areas.

- e) A bit pattern in the ISC_Inc register may be used to specify that the address register is to be initialized to a known state.

NOTE—For example, a 1-bit ISC_Inc register could be defined where a “0” in this register causes the address register to be cleared and a “1” causes the address to increment. This could be the only functional access provided to the address register in an implementation.

- f) When the ISC_Inc register is targeted, higher order bits of the ISC_Inc register may capture information specified by the component designer.

NOTE—If recommendation v) of 5.2 has not been exercised for this component, then all bits are available.

- g) The minimum duration in the *Run-Test/Idle* TAP Controller state for ISC_INCREMENT may be one cycle of TCK.

NOTE—Consequently, there is no way to cause a duration error.

Recommendation

- h) ISC_INCREMENT should not be included in any adaptive programming algorithm loop.

6.5.5.2 Description

The ISC_INCREMENT instruction gives increment access to the address register used in programming or reading data. This register may physically be a portion of the ISC_PData/ISC_RData or a separate register.

Note that internally, the address space may be contiguous, or may contain gaps of unused addresses. This instruction may hide these details such that one operation of this instruction will always move the address to the next valid address, or this instruction may need a control pattern supplied in the ISC_Inc register in order to skip unused addresses. Targeting the ISC_Inc register can also be used to increment the address by integers greater than one, by shifting in the desired increment.

If the ISC_INCREMENT instruction does not place the ISC_Inc register between TDI and TDO but rather targets the ISC_Default register, the content of this register has no effect on the operation of the instruction; that is, the next valid address will be placed in the address register.

When a device implements an adaptive programming algorithm that may require the execution of an adaptive programming loop that could also use the ISC_INCREMENT instruction, the device designer must be extremely cautious in the design so that the register being incremented does not change until the action has been declared a success and the action has been terminated. If the register is incremented every time the programming operation is executed before it achieves success, the address will be unknown to the programming software and it will be impossible to achieve programming success. It is strongly recommended not to implement the ISC_INCREMENT instruction inside of any adaptive programming loops.

6.6 Requirements for IEEE Std 1149.1-2001 masters and ISC programming algorithms

6.6.1 IEEE Std 1149.1-2001 masters

IEEE Std 1149.1-2001 provides rules for slave devices, which then extend to master devices that drive one or more slaves in a chain. For example, knowledge of the TAP Controller state diagram must be embedded within a master device so it can direct slaves in a trajectory through the state diagram that performs useful functions.

Master devices may be loosely categorized into two classes: those that can traverse any path through the TAP Controller state diagram and those that have certain limitations. A particular example of the latter class is a master that cannot make the transition from either *Update-IR* or *Update-DR* TAP Controller states directly to the *Select-DR* TAP Controller state. These masters must pass through the *Run-Test/Idle* TAP Controller state first. An IEEE Std 1149.1-2001 master with this limitation is unsuited for use as an ISC controller because in many instances (for example, the ISC_PROGRAM instruction), passing even momentarily through the *Run-Test/Idle* TAP Controller state before all initializing conditions are set up will cause an ISC instruction to begin operation with an incompletely specified environment.

As an example, consider the ISC_PROGRAM instruction. It is loaded into the instruction register in the *Shift-IR* TAP Controller state. Then, the first word to be programmed is loaded into the ISC_PData register by progressing to the *Shift-DR* TAP Controller state. However, if the master device must move from instruction shifting to data shifting via the *Run-Test/Idle* TAP Controller state, then the ISC_PROGRAM instruction execution is initiated before the data has been shifted into place. (Most likely, the duration requirement also has not been met.) This sequence of events may not produce proper programming results and should thus be avoided. Users of IEEE Std 1149.1-2001 masters should qualify their behavior in this regard. Masters not in compliance will likely need an updated revision of software.

Another requirement of IEEE Std 1149.1-2001 masters is that they be able to control chains of devices with a mix of duration requirements. For example, one device in the chain may require a duration of 800 cycles of TCK while another may require a duration of 700 microseconds of real time. Both requirements must be met (or exceeded) for the duration requirement of the chain to be satisfied. Thus, the master device must be able to deliver the proper amount of TCK cycles and wait an appropriate amount of real time in such cases, which may be complicated somewhat if the TCK cycle time is allowed to be varied by the user.

Rules

- a) For ISC programming, an IEEE Std 1149.1-2001 master device shall be able to load an ISC instruction and then load a data register without passing through the *Run-Test/Idle* TAP Controller state between these two operations.
- b) For ISC programming, an IEEE Std 1149.1-2001 master device shall be able to satisfy the duration requirements of all devices in a chain including delivering at least a minimum number of TCK cycles and concurrently observing the passage of at least a minimum amount of real time.

6.6.2 Deterministic and adaptive programming algorithms

An ISC programming algorithm is a process that loads configuration data patterns into the memories of ISC devices. When all of the devices use deterministic programming algorithms, a high degree of concurrency should be possible because the burn times for all of the devices are executed simultaneously, which will, in turn, minimize the overall programming time.

For technology reasons, it may not be possible for a device to implement a fully deterministic programming algorithm and it may be necessary for a part of the device's overall ISC programming algorithm to be adaptive. When this happens, a portion of the overall programming algorithm will have to be executed multiple times until the device indicates that programming success has been achieved. It will not be known how many times will be required for the device to achieve programming success, so it is difficult to statically concurrentize an adaptive programming flow with a deterministic programming flow. Indicating a minimum number of iterations will increase the amount of static concurrency possible because that portion of the programming algorithm is, for all practical purposes, deterministic. Programming tools will know exactly how many iterations of a programming loop will be executed before the operation success indication should be evaluated.

The difficulties of providing concurrent programming for a chain of devices when some of the devices in the chain use an adaptive programming algorithm lead to the conclusion that devices should use deterministic programming algorithms when possible and the use of adaptive programming algorithms should be minimized. Doing so both eases the development of the programming tools and makes it easier for manufacturing engineers to determine a set time required to program and test a board. If an adaptive programming algorithm is used, it becomes difficult for the manufacturing engineer to determine how many boards he can get through the system in a given amount of time. The calculations would be based on worst case times, which may be much different from the typical times the devices would program in.

An adaptive programming algorithm can be converted into a deterministic programming algorithm by “unrolling” it such that any loop constructs in a flow are always executed the maximum number of iterations and the evaluation of the operation success indication is only done after the last iteration to determine if an error occurred and the programming flow should be halted. This allows tools to execute adaptive programming algorithms without having to make decisions based on intermediate results. It has the negative effect of lengthening programming times for a system or device as it is likely that any programming operation inside of a loop will complete long before the maximum number of iterations is reached.

6.7 Adding new instructions for ISC programming support

All ISC instructions have a similar nature because they obey a set of common rules provided in 6.1.1. Thus, all ISC instructions operate in a common way with differences usually tied to whether or not they are governed by enabled internal security mechanisms. Because of this, a device designer may contemplate adding a new instruction to support a unique feature of a device implementation. This is allowed provided the rules in 6.1.1 are all observed.

6.7.1 Specifications

Rules

- a) Additional public ISC instructions invented by a device designer shall observe all the rules provided in 6.1.1.
- b) Additional public ISC instructions invented by a device designer shall be documented with the facilities provided for instruction description within the BSDL language; further, such instructions shall be identified in descriptive prose for the device, informing users of their operation and intent.

NOTES

1—IEEE Std 1149.1-2001 allows a device designer to invent private instructions that do not need description beyond an indication in BSDL that they exist and are private. These are to be avoided by tools not specifically designed to recognize and utilize their functions.

2—A device designer must invent names for new instructions. These should be unique, not matching any given by this standard (as this would likely lead to a rule or operational violation with the ISC instructions given herein), and preferably, these names would not start with “ISC” so as to avoid confusion.

Permissions

- c) Additional public ISC instructions invented by a device designer may be utilized in a programming process.

6.7.2 Description

Many ISC instructions specified in this standard (for example, ISC_PROGRAM_USERCODE, ISC_INCREMENT, or ISC_SETUP) have almost identical characteristics, but have been included to provide guidance to designers on ways to implement ISC functionality. The ability to specify new instructions given here allows designers additional flexibility for providing ISC functionality tailored to the unique characteristics of their devices.

7. Data registers

All ISC instructions are required to target a register between TDI and TDO to maintain the integrity of the shift path of a chain of devices. If recommendation v) of 5.2 has been implemented, then all ISC registers will load a status indication on the rising edge of TCK in the *Capture-DR* TAP Controller state. Some ISC instructions have the option to target an ISC register or a register defined by IEEE Std 1149.1-2001. If an IEEE Std 1149.1-2001 defined register is targeted, then recommendation v) of 5.2 may not be implementable for any ISC instruction because of it being too short (e.g., the Bypass Register) or because IEEE Std 1149.1-2001 already prescribes the capture behavior for the register.

There may be several uniquely named ISC registers in an implementation of the rules provided by this standard. In some cases, if two registers have the same length, and data transmitted through them is not persistent beyond the execution of the instructions that use those registers, they may actually be

implemented as one physical register. For example, ISC_RData and ISC_PData could be the same physical register in a device. However, their operations with the ISC_PROGRAM and ISC_READ instructions are quite different.

There are interactions between instruction choices. For example, if ISC_PROGRAM is associated with the ISC_Default register, then ISC_DATA_SHIFT (and possibly ISC_ADDRESS_SHIFT) must be implemented to supply configuration data for ISC_PROGRAM.

Instructions that share a register form a logical group. ISC_ENABLE and (possibly) ISC_DISABLE can share the ISC_Config register. No other instruction should target this register since it is likely that the configuration bits set by those instructions must persist for the duration of ISC operations.

Figure 28 lists every ISC instruction and register to show which may be associated. Only one register may be associated with a given instruction, corresponding to the choice of one “X” per row, for the instructions implemented in a given device.

		Target Register									
ISC Instruction		ISC_Default	Device_ID	ISC_PData	ISC_Config	ISC_RData	ISC_Sector	ISC_Info	ISC_Data	ISC_Address	ISC_Inc
Mandatory	ISC_ENABLE	X			X						
	ISC_DISABLE	X			X						
	ISC_PROGRAM	X		X							
	ISC_NOOP	X									
Optional	ISC_PROGRAM_USERCODE		X								
	ISC_SETUP				X						
	ISC_READ			X		X					
	ISC_ERASE	X					X				
	ISC_DISCHARGE	X					X				
	ISC_PROGRAM_DONE	X		X							
	ISC_ERASE_DONE	X		X							
	ISC_PROGRAM_SECURITY	X		X			X				
	ISC_READ_INFO							X			
	ISC_DATA_SHIFT								X		
	ISC_ADDRESS_SHIFT									X	
	ISC_INCREMENT	X									X
		Mandatory				Optional					

Figure 28—List of ISC instruction and their possible target data registers

7.1 Device identification register

This 32-bit register is not an ISC register and is completely defined by IEEE Std 1149.1-2001. This register is targeted by the IDCODE and USERCODE instructions. It may also be targeted by the optional ISC_PROGRAM_USERCODE instruction when it is implemented (see 6.3.3).

When the IDCODE instruction is loaded, then the data loaded into the Device Identification Register at the rising edge of TCK in the *Capture-DR* TAP Controller state is a code that identifies the device itself that is hardcoded into the silicon of the device. This code is static.

When the USERCODE instruction is loaded, then the data loaded into the Device Identification Register at the rising edge of TCK in the *Capture-DR* TAP Controller state is a code that has been programmed into the device to identify the program. If the device has never been programmed or has been erased, then the resulting code is the “blank” code. The blank code must be documented (see d) of 8.2.1) so that programming algorithms can differentiate the blank code from a code containing information.

The optional ISC_PROGRAM_USERCODE instruction can be used to load user code bits into the device, provided the device has not been program-protected by an activated security mechanism (see 6.3.3). If this instruction is not implemented, then the 32-bit user code is loaded as some portion of the general program space.

The codes read out by the IDCODE and USERCODE instructions are never protected by an enabled read-security mechanism as this would violate the operational definition of these instructions as specified by IEEE Std 1149.1-2001.

7.2 Bypass register

This single-bit register is not an ISC register and is completely defined by IEEE Std 1149.1-2001. This register is targeted by the BYPASS, CLAMP, and HIGHZ instructions, and optionally targeted by certain ISC instructions, as noted below in 7.4. Note that this register always captures a “0” in the *Capture-DR* TAP Controller state.

7.3 Boundary scan register

This register, which is used by some instructions to control/observe the system I/O pins of the device, is not an ISC register and is completely defined by IEEE Std 1149.1-2001. It is targeted by the SAMPLE, PRELOAD, EXTEST, and INTTEST instructions, and optionally by the RUNBIST instruction. Depending on the choice made for rule d) of 6.1.1, the Boundary-Scan Register may be utilized by ISC instructions to define the I/O behavior of the system pins similar to the CLAMP instruction. In such cases, the Boundary-Scan Register is not targeted by the instruction and should be initialized by using the PRELOAD instruction.

7.4 ISC_Default

The ISC_Default register is targeted by ISC_NOOP and other instructions that do not need to shift data into a working register to initialize or control an operation. Any data shifted into ISC_Default and later updated in the *Update-DR* TAP Controller state will be completely ignored. This register is intended to be short, and may be only one bit long if recommendation v) of 5.2 has not been implemented in the device. If this recommendation is supported, then ISC_Default will be capable of shifting out the status register that has been implemented.

If ISC_Default is one bit long and it captures a “0” in the *Capture-DR* TAP Controller state, then it is indistinguishable from the Bypass Register and may actually share the same physical register. Even when this is true, it retains its name and it is described in the ISC extension for BSDL (see 8.2.2).

Some of the ISC instruction rules offer the choice of targeting a specific register used for transmitting data, address, or control information, or targeting the ISC_Default register if the embodiment of the instruction does not require data transmission.

Rules

- a) An `ISC_Default` register shall exist in a device conforming to this standard.

NOTE—The mandatory instruction `ISC_NOOP` must target this register.

- b) If recommendation v) of 5.2 has not been implemented in the device, then this register shall be one bit long.
- c) If recommendation v) of 5.2 has been implemented in the device, then this register shall have its length equal to that of the status register implemented for the device.

NOTE—See 5.1.9 for options on the construction of the status register.

- d) Data shifted into `ISC_Default` and updated at the *Update-DR* TAP Controller state shall have no effect on any subsequent programming action.

Permissions

- e) If the `ISC_Default` register is one bit long and it captures a “0” in the *Capture-DR* TAP Controller state, then it is operationally identical to the Bypass Register and may be shared with it.

NOTES

1—It is still referred to as “`ISC_Default`” and is so described in BSDL.

2—The “0” capture behavior is desirable since it increases the places in a programming process where known data are produced that can be checked by programming hardware. This may help flag system errors more quickly if these expected bits fail to appear.

7.5 `ISC_Config`

The `ISC_Config` register is optionally targeted by the `ISC_ENABLE` and `ISC_DISABLE` instructions and must be targeted by `ISC_SETUP` if this instruction is provided. This register may be loaded with data before passing into the *Run-Test/Idle* TAP Controller state for the minimum duration specified for the targeting instruction. The device designer determines the functionality that is controlled by this data. If this data must remain in effect for the entire time that ISC operations are conducted, then it may not be combined with another register.

The `ISC_Config` register may be used to supply configuration parameters that modify or adjust the behavior of subsequent ISC instructions. However, this information may not be used to modify the lengths or definitions of other data registers, or the association between instructions and data registers, as this would violate rules in IEEE Std 1149.1-2001.

Example 1: A device may contain multiple programmable arrays of memory, implemented using different programmable technologies (for example, Static RAM and FLASH memory). The `ISC_Config` register may be used to select one of the available memory arrays at a time for programming. While the `ISC_PROGRAM` instruction must always be used for programming, the information in the `ISC_Config` register may be used to select which array is programmed. Note that the programming duration required for `ISC_PROGRAM` may vary significantly for the different arrays.

Example 2: For some nonvolatile memory technologies, a threshold parameter may be specified when reading the programmed contents of the device. This parameter controls the voltage or current used to read each memory element, and permits verification of the programmed margin, that is, the degree of programming or erasure of each memory element. To ensure adequate margin, such devices may need to be

verified twice, once with a high threshold value, and once with a low threshold value. While the ISC_READ instruction must always be used to read the programmed contents of the device, the ISC_Config register may be used to set the value of the threshold parameter prior to verification.

The ISC_Config register may be set to a default state upon entry into the ISC Accessed system modal state (indeed, the ISC_ENABLE instruction might not target ISC_Config). Later the ISC_SETUP instruction may be used to alter the content of the ISC_Config register to modify the operational environment of subsequent instructions.

7.6 ISC_PData

The ISC_PData register is targeted (optionally) by the ISC_PROGRAM, ISC_PROGRAM_DONE, ISC_ERASE_DONE, and ISC_PROGRAM_SECURITY instructions. It may also be listed as the target for ISC_READ when ISC_READ targets the same register as ISC_PROGRAM.

The ISC_PData register is used to load programming information into a device. The device designer defines the nature of this data. For example, the information may consist of an address/data pair or simply data if some other addressing or auto-addressing process has been implemented.

7.7 ISC_RData

The ISC_RData register is targeted by the ISC_READ instruction only. This register is used to return readback data from the device. If recommendation v) of 5.2 has been implemented, then the least two significant bits of this register shall contain status information, necessitating the lengthening of ISC_RData by two bits to contain both the readback data and status. The readback information may contain data, or perhaps address/data pairs, at the option of the device designer.

Rules

- a) An ISC_RData register shall exist in a device conforming to this standard if ISC_READ is implemented and the ISC_RData register is physically distinct from the ISC_PData register.

NOTE—ISC_RData register name may be omitted if the ISC_RData register is physically the ISC_PData register.

- b) If recommendation v) of 5.2 has been implemented in the device, then ISC_RData shall be long enough to accommodate all readback information (in higher order bits) and status information.

7.8 ISC_Sector

The ISC_Sector register is targeted (optionally) by up to three instructions that may treat the device as a group of areas (sectors) with common properties: they are ISC_ERASE, ISC_DISCHARGE, and ISC_PROGRAM_SECURITY. This register is used to identify what areas of the device program structure to operate upon, where this capability is implemented.

7.9 ISC_Info

The ISC_Info register is targeted by the ISC_READ_INFO instruction. This design-specific register loads both current-state information and factory-preset data on the rising edge of TCK in the *Capture-DR* TAP Controller state. The device designer defines the amount of data, its representation, and organization.

Rules

- a) An ISC_Info register shall exist in a device conforming to this standard if ISC_READ_INFO is implemented.

7.10 ISC_Data

The ISC_Data register is targeted by the ISC_DATA_SHIFT instruction. This register may be an independent register, or it may physically be a subset of either or both the ISC_RData and ISC_PData registers, allowing access to the data portion of an address/data pair.

Rules

- a) An ISC_Data register shall exist in a device conforming to this standard if ISC_DATA_SHIFT is implemented.

7.11 ISC_Address

The ISC_Address register is targeted by the ISC_ADDRESS_SHIFT instruction. This register may be an independent register, or it may physically be a subset of either or both the ISC_RData and ISC_PData registers, allowing access to the address portion of an address/data pair.

Rules

- a) An ISC_Address register shall exist in a device conforming to this standard if ISC_ADDRESS_SHIFT is implemented.

7.12 ISC_Inc

The ISC_Inc register is targeted (optionally) by the ISC_INCREMENT instruction. When this register exists, it is used to control how addressing is modified in preparation for subsequent reading or programming activity.

8. Conformance and documentation requirements

8.1 Conformance

8.1.1 Specification

Rules

- a) A component conforming to this standard shall comply with all rules set out herein.

NOTE—Due to rule a) of 5.2, this implies conformance with the rules set out in IEEE Std 1149.1-2001 as well.

8.1.2 Description

Conformance to the rules set out herein and in IEEE Std 1149.1-2001 is essential for three reasons:

- a) The capabilities given by conformance to IEEE Std 1149.1-2001 are essential for testing boards and other assemblies containing programmable devices before these devices are programmed. Manufacturing defects such as shorted or open solder joints may be found and repaired before the investment in device programming is made. The ability to test boards and other assemblies after programming, using these same capabilities, allows testing to be done independently of the content of programmable devices, which may be application specific or even variable in time.

NOTE—There are some exceptions beginning to appear of devices that have programmable I/O parametric behavior such as the logic family and whether they are single-ended or differential. This configurability of I/O may require programming (at least a partial download) before testing is electrically possible.

- b) Conformance with this standard allows concurrent programming of programmable devices, with the potential for significant time efficiencies, particularly when the devices belong to different device families or come from different vendors.
- c) The standardization of system modal states, their behaviors, and transitions between them (see Clause 4) is crucial for board and system designers who need to assure the orderly behavior of their products before, during, and after programming.

8.2 Documentation

Because adherence to this standard implies adherence to IEEE Std 1149.1-2001, all ISC devices shall have a description supplied in Boundary-Scan Description Language (BSDL) provided with IEEE Std 1149.1-2001.

8.2.1 Specification

Rules

- a) A component conforming to this standard shall be documented with a BSDL description.

NOTE—See IEEE Std 1149.1-2001 for a description of IEEE BSDL. The precursor to IEEE BSDL developed in 1990 cannot be used for documentation because it does not support the concept of “BSDL Extensions.” (See 8.4.)

- b) A component shall have all the ISC instructions, register relationships, and optional provisions documented via the syntax provided by BSDL and the BSDL extension provided by this standard.

NOTE—New BSDL syntax (contained within a “BSDL extension”) for describing concepts and structures introduced by this standard are given in 8.3. A BSDL extension is a mechanism provided by IEEE Std 1149.1-2001 (see “User extensions to BSDL”), which allows proprietary syntax to be provided that will allow tools to work that are unaware of this syntax.

- c) Instruction and register names shall be documented in BSDL exactly as given herein.

NOTE—This includes spelling and the use of the underscore (“_”) character, but not casing (BSDL is case-insensitive).

- d) The “blank” code observed when the USERCODE instruction is executed on an unprogrammed or erased device shall be documented.
- e) Other properties of an ISC device not described in BSDL shall be documented by the manufacturer.

NOTE—For example, this includes static programming voltages that may need to be applied to the device (see permission t) of 5.2), TAP signal voltage level requirements, and any additional power requirements of the device caused by ISC activities.

Recommendations

- f) If additional status indication bits are defined in a device, the meaning of these should be documented.

8.2.2 Description

Consider a device with a full set of ISC instructions and registers. Here are fragments of BSDL needed to document this device.

First, an “Instruction_Opcode” attribute used to define instruction names and binary code assignments should be given:

```
Attribute INSTRUCTION_OPCODE of PLD:entity is
  "BYPASS          (111111), " &
  "EXTTEST         (000000), " &
  "SAMPLE          (000001), " &
  "IDCODE          (010001), " &
  "USERCODE        (010010), " &
  "ISC_NOOP         (000010), " &
  "ISC_READ        (000011), " &
  "ISC_PROGRAM     (000100), " &
  "ISC_ERASE       (000101), " &
  "ISC_ENABLE      (000110), " &
  "ISC_DISABLE     (000111), " &
  "ISC_PROGRAM_DONE (001000), " &
  "ISC_ERASE_DONE  (001001), " &
  "ISC_DISCHARGE   (001010), " &
  "ISC_PROGRAM_SECURITY (001011), " &
  "ISC_READ_INFO   (001100), " &
  "ISC_DATA_SHIFT  (001101), " &
  "ISC_SETUP       (001110), " &
  "ISC_ADDRESS_SHIFT (001111), " &
  "ISC_INCREMENT   (010000)";
```

```
attribute INSTRUCTION_CAPTURE of PLD:entity is "000x01";-- Note 'x'
```

NOTES

1—The entity name in these examples is shown as “PLD.” This would be replaced with a name unique to the device.

2—The “x” in the INSTRUCTION_CAPTURE string corresponds to the location of ISC_Done (see h) of 5.2). The state of this bit may change as programming activities progress, so it must be declared unknown here in this statement.

Since both IDCODE and USERCODE are implemented, these are described.

```
Attribute IDCODE_REGISTER of PLD:entity is
  "0011" & -- Version
  "1111000011110000" & -- Part number
  "00001010100" & -- Manufacturer ID
  "1"; -- Required by IEEE Std 1149.1

Attribute USERCODE_REGISTER of PLD:entity is
  "00000000000000000000000000000000"; -- Blank pattern
```

Later, a Register_Access attribute is given:

```
Attribute REGISTER_ACCESS of PLD:entity is
  "ISC_Default[2] (ISC_NOOP Captures 10), " &
  "ISC_Data[16]   (ISC_DATA_SHIFT Captures xxxxxxxxxxxxxx10), " &
  "ISC_Config[2]  (ISC_ENABLE Captures 10), " &
  "ISC_Config     (ISC_DISABLE Captures 10), " &
  "ISC_Sector[2]  (ISC_ERASE Captures 10), " &
  "ISC_PData[16]  (ISC_PROGRAM Captures xxxxxxxxxxxxxx10), " &
  "ISC_RData[18]  (ISC_READ), " &
  "ISC_Address[16] (ISC_ADDRESS_SHIFT Captures xxxxxxxxxxxxxx10), " &
  "ISC_Info[258]  (ISC_READ_INFO)";
```

The Register_Access attribute defines the existence of optional registers, their length, the instruction(s) that target them, and any consistent capture data they may be loaded with in the *Capture-DR* TAP Controller state.

Note in the Register_Access attribute that some “Captures” clauses are included for most register associations that define what that register will capture in the *Capture-DR* TAP Controller state for a given instruction. This reflects that this device has implemented the optional status indication scheme. Also note that the ISC_RData and ISC_Info registers do not document a “Captures” clause because, even though they both capture a “10” in their least significant positions, they also capture a large amount of variable data that has to be determined by an external algorithm. While any ISC-oriented algorithm should be able to provide a prediction of what is captured anyway, including the “Captures” clauses offers completeness and good form.

NOTE—If a test is generated from this BSDL by an algorithm that is not ISC-aware, and if this algorithm attempts to verify the “Captures” bits for ISC instructions, it will most likely fail since the status bits are sensitive to the proper execution and ordering of ISC instructions.

8.3 BSDL extension for ISC description (STD_1532_2002)

The information provided above in 8.2.2 showed how to document features of an ISC device using the existing syntax of BSDL. This information is incomplete. BSDL has been defined (see IEEE Std 1149.1-2001) to be extensible using a mechanism called a “BSDL Extension.” A BSDL Extension for describing additional features of ISC devices is given here.

NOTE—Refer to IEEE Std 1149.1-2001, B.8.17, titled “User extensions to BSDL,” for precise information regarding BSDL extensions.

The extension mechanism chosen for describing ISC devices is based on the definition of a VHDL package with the name “STD_1532_2002,” which contains the definitions of attributes that will be used to supply ISC-relevant data. Therefore, an ISC-compliant device BSDL will contain an additional “use” statement appearing just after the “standard use statement,” as in this excerpt of a BSDL file:

```
...
use STD_1149_1_2001.all;      -- Standard 'use' statement
use STD_1532_2002.all;      -- BSDL Extension for ISC devices
...
```

Utilization of the extension mechanism of BSDL guarantees that ISC information can be supplied to applications that are cognizant of ISC functionality without hindering other applications that may not be aware of this functionality. Noncognizant applications will simply ignore the extension.

The VHDL package STD_1532_2002 contains the definition of additional attributes used to complete the description of the ISC features of a device. The content of this VHDL package is given here.

Syntax

```
Package STD_1532_2002 is -- Attribute definitions for ISC
description
use STD_1149_1_2001.all; -- Refer to BSDL definitions

attribute ISC_Conformance:    BSDL_Extension;
attribute ISC_Pin_Behavior:    BSDL_Extension;
attribute ISC_Fixed_System_Pins: BSDL_Extension; -- Optional
attribute ISC_Status:          BSDL_Extension;
attribute ISC_Blank_Usercode:  BSDL_Extension;
attribute ISC_Security:        BSDL_Extension; -- Optional
attribute ISC_Flow:            BSDL_Extension;
attribute ISC_Procedure:       BSDL_Extension;
attribute ISC_Action:          BSDL_Extension;
attribute ISC_Illegal_Exit:    BSDL_Extension; -- Optional
attribute ISC_Design_Warning:  BSDL_Extension; -- Optional
end STD_1532_2002;

Package Body STD_1532_2002 is
-- No content, this package body is required by BSDL syntax
end STD_1532_2002;
```

This VHDL package is “read-only” and may be maintained within a given system in the same location as the standard package STD_1149_1_1994 or STD_1149_1_2001.

8.4 ISC extension structure

The following subclauses define the syntax and applicable semantics for the BSDL attributes defined by the ISC extension. The form of the syntax used is defined in IEEE Std 1149.1-2001.

NOTES

1—See IEEE Std 1149.1-2001, “Lexical elements of BSDL” and “Notes on syntax definitions,” for the conventions used herein to describe the parsing requirements for BSDL.

2—Syntactic items are shown surrounded in “<>” brackets when referenced in text. Many of these items will be defined here, but some will be adopted from IEEE Std 1149.1-2001. Those that are adopted will be underlined to indicate their source.

When an ISC extension exists in the extension area of a BSDL description, it must have the structure shown here. The various attributes, both mandatory and optional, must appear in a prescribed order and not be intermixed with other statements.

NOTE—These other statements include BSDL attributes, attributes for other BSDL extensions, and general VHDL constructs.

8.4.1 Specification

Syntax

```

<ISC Extension> ::=
  <ISC conformance description>           (see 8.5.2.1)
  <ISC pin behavior description>          (see 8.5.3)
  [ <ISC fixed pin description> ]         (see 8.5.4)
  <ISC status description>               (see 8.5.5)
  <ISC blank usercode description>        (see 8.5.6)
  [ <ISC security description> ]         (see 8.5.1)
  <ISC flow description>                 (see 8.5.7.1)
  <ISC procedure description>            (see 8.5.8.1)
  <ISC action description>               (see 8.5.9.1)
  [ <ISC illegal exit description> ]     (see 8.5.10.1)
  [ <ISC design warning> ]              (see 8.5.11.1)

```

Rules

- a) The syntax for an ISC Extension as it appears in the extension area of a BSDL description shall be that shown above.

NOTE—The attributes must appear in the order shown above.

Permissions

- b) An ISC Extension may appear anywhere in the extension area of a BSDL description, subject to any rules defined by those extensions as to ordering.

NOTE—If more than one extension exists, an ISC Extension may appear before or after any of them within the limits of their definitions.

8.4.2 Description

The six mandatory and four optional attributes of an ISC Extension must appear in the order shown. No other statements may be intermixed within the extension, as the syntax above specifies. An ISC Extension may coexist with a number of other extensions.

8.4.3 Keywords for ISC BSDL

This subclause lists the keywords of ISC BSDL. These keywords are in addition to the reserved words of BSDL and VHDL. (See “BSDL reserved words” and “VHDL reserved and predefined words” in IEEE Std 1149.1-2001.)

array	min
blank_check	not
CRC	optional
erase	OST
exit_on_error	proc_blank_check
implemented	proc_disable
initialize	proc_enable

ISC_Action	proc_erase
ISC_Blank_Usercode	proc_error_exit
ISC_Conformance	proc_preload
ISC_Design_Warning	proc_program
ISC_Disable_Erase	proc_program_done
ISC_Disable_Key	proc_read
ISC_Disable_Program	proc_verify
ISC_Disable_Read	program
ISC_Flow	proprietary
ISC_Illegal_Exit	read
ISC_Security	recommended
ISC_Fixed_System_Pins	repeat
ISC_Pin_Behavior	security
ISC_Procedure	terminate
ISC_Status	unprocessed
loop	verify
max	wait

NOTE—There is no significance in the casing of the reserved words.

8.5 ISC BSDL attribute definitions

The mandatory and optional attributes needed to describe the ISC properties of a device are given in the remainder of this section. They must appear in the order given in 8.4.1.

8.5.1 Attribute ISC_Conformance

The ISC conformance statement shall identify the edition of this standard to which the configuration circuitry of a physical component conforms.

It is possible for a component designed in 2002 to be described by the BSDL extensions for ISC defined in this edition of the standard but not in the 2001 edition of this standard. Additions such as the **LOOP** construct and the ability to have multiple **Repeat** blocks make it necessary to have the <ISC conformance statement> so that programming tools may easily associate features implemented in a device with the edition of the standard that describes those features.

8.5.1.1 Specification

Syntax

```
<ISC conformance statement> ::= attribute ISC_Conformance of <component name> : entity is
    <conformance string> ;
<conformance string> ::= " <conformance identification> "
<conformance identification> ::= STD_1532_2001|STD_1532_2002
```

Rules and semantic checks:

- a) The syntax for the mandatory ISC_Conformance attribute shall be that shown above.

NOTE—The ISC_Conformance attribute was not included as a part of IEEE Std . As a result, BSDL files written to that edition of the standard may not have the ISC_Conformance attribute. A programming tool using a BSDL file written for a 1532-compliant device that does not have the ISC_Conformance attribute should assume the device is compliant to IEEE Std 1532-2001.

8.5.1.2 Description

The symbol STD_1532_2001 refers to IEEE Std 1532-2001 and STD_1532_2002 refers to IEEE Std 1532-2002. Subsequent editions of this standard may add new values to the <conformance identification> element.

NOTE—The ISC_Conformance attribute was not included as a part of IEEE Std 1532-2001. As a result, BSDL files written to that edition of the standard may not have the ISC_Conformance attribute. A programming tool using a BSDL file written for a 1532-compliant device that does not have the ISC_Conformance attribute should assume the device is compliant to IEEE Std 1532-2001.

Examples

```
attribute ISC_Conformance of PLD1:entity is "STD_1532_2002";

attribute ISC_Conformance of PLD2:entity is "STD_1532_2001";
```

8.5.2 Attribute ISC_Pin_Behavior

There are two choices for system I/O pin behavior (see 4.4). The ISC_Pin_Behavior attribute communicates the choice implemented within an ISC device.

8.5.2.1 Specification

Syntax

```
<ISC pin behavior description> ::= attribute ISC_Pin_Behavior of <component name> : entity is
    <behavior string> ;
<behavior string> ::= " <pin behavior option> "
<pin behavior option> ::= HIGHZ | CLAMP
```

Rules

- a) The syntax for the mandatory ISC_Pin_Behavior attribute shall be that shown above.

8.5.2.2 Description

Examples

Two devices are described. The first implements the CLAMP-like system I/O behavior. The second device exhibits the HIGHZ-like system I/O behavior.

```
attribute ISC_Pin_Behavior of PLD1:entity is "CLAMP";

attribute ISC_Pin_Behavior of PLD2:entity is "HIGHZ";
```

8.5.3 Attribute ISC_Fixed_System_Pins

The ISC_Fixed_System_Pins attribute documents fixed system pins (see 5.1.1.2) when they exist. This attribute may be omitted when a device has no fixed system pins, but must be given for devices that do have such pins.

8.5.3.1 Specification

Syntax

```
<ISC fixed pin description> ::= attribute ISC_Fixed_System_Pins of <component name> : entity is
    <fixed pin string> ;
<fixed pin string> ::= " <fixed pin list> "
<fixed pin list> ::= <fixed pin> { , <fixed pin> }
<fixed pin> ::= <port ID>
```

Rules

- The syntax for the optional ISC_Fixed_System_Pin attribute shall be that shown above.
- The value of <port ID> shall contain a <port name> declared in the logical port description for the device.

NOTE—A <port ID> may consist of a <port name> or a <subscripted port name> as described in “Commonly used syntactic elements” in IEEE Std 1149.1-2001. See also “Logical port description statement” in that standard.

- The value of <port ID> shall refer to a system pin only and not to any pin that is a scan port signal or a compliance port pin; further, the <pin type> of the <port name> shall not be “linkage.”

8.5.3.2 Description

The ISC_Fixed_System_Pin attribute can refer to individual system pins as <subscripted port name> elements or as <port name> elements that refer to individual system pins or collections of system pins.

Example

```
attribute ISC_Fixed_System_Pins of pld : entity is
    "Addr_bus, IRQ(2), IRQ(1), Sysclock" ;
```

8.5.4 Attribute ISC_Status

The ISC_Status attribute communicates whether the status option (see 5.1.9) has been implemented within an ISC device.

8.5.4.1 Specification

Syntax

```
<ISC status description> ::= attribute ISC_Status of <component name> : entity is
    <status string> ;
<status string> ::= " <status option> "
<status option> ::= [ <modifier> ] Implemented
<modifier> ::= Not
```

Rules

- a) The syntax for the mandatory ISC_Status attribute shall be that shown above.

NOTE—If a proprietary status indication scheme has been implemented, this cannot be described by the “Implemented” phrase because the operations needed to read out this status are unknown. In this case, the phrase should be “Not Implemented.”

8.5.4.2 Description

Examples

Two devices are described. The first implements a status register. The second device has no status register.

```
attribute ISC_Status of PLD1:entity is "Implemented";  
  
attribute ISC_Status of PLD2:entity is "Not Implemented";
```

8.5.5 Attribute ISC_Blank_Usercode

The ISC_Blank_Usercode attribute documents the 32-bit value of the Device Identification Register that will be observed when the IEEE Std 1149.1-2001 USERCODE instruction is executed in a blank or erased device.

8.5.5.1 Specification

Syntax

```
<ISC blank usercode description> ::= attribute ISC_Blank_Usercode of <component name> : entity  
is  
    <blank usercode string> ;  
<blank usercode string> ::= " <32-bit pattern> "
```

Rules

- a) The syntax for the mandatory ISC_Blank_Usercode attribute shall be that shown above.

8.5.5.2 Description

Example

```
attribute ISC_Blank_Usercode of PLD1:entity is  
    "11111111111111111111111111111111";
```

8.5.6 Attribute ISC_Security

The amount and types of security provided by an implementation is described by the ISC_Security attribute. Some devices may provide extensive protection while others may have minimal or no such features. This attribute is only used to describe the standardized security feature given in 5.1.11. If no security mechanism, or a proprietary security mechanism, is implemented (which means the device might not even possess an ISC_PROGRAM_SECURITY instruction), this attribute is omitted.

When this attribute exists, a device programming tool may program security data into a device that was extracted from an ISC data file (see Clause 9). However, if this data were tagged as “Security” data, the tool

can choose to display an interface that portrays the options available in this device that the user can then override. The data supplied by this attribute facilitate this capability.

8.5.6.1 Specification

Syntax

```

<ISC security description> ::= attribute ISC_Security of <component name> : entity is
    <protection string> ;
<protection string> ::= " <protection spec> "
<protection spec> ::= <read spec> , <program spec> , <erase spec> , <key spec>
<read spec> ::= ISC_Disable_Read <bit spec>
<program spec> ::= ISC_Disable_Program <bit spec>
<erase spec> ::= ISC_Disable_Erase <bit spec>
<key spec> ::= ISC_Disable_Key <bit range>
<bit spec> ::= <no bit> | <integer>
<bit range> ::= <no bit> | <msb> - <lsb>
<no bit> ::= *
<msb> ::= <integer>
<lsb> ::= <integer>

```

Rules and semantic checks:

- The syntax for the optional ISC_Security attribute shall be that shown above.
- The ISC_Security attribute shall appear only when the ISC_PROGRAM_SECURITY instruction is implemented in the device and it targets the ISC_PData register between TDI and TDO.

NOTE—In the case where the ISC_PROGRAM_SECURITY instruction targets the ISC_Default register, no data are used by the instruction so this attribute provides no information and should be omitted.

- When the ISC_PROGRAM_SECURITY instruction is implemented in the device and it targets the ISC_PData register, then at least one of the <bit spec> elements and/or the <bit range> element shall not be described as <no bit>.
- The numerical value of <msb> shall be greater than or equal to the value of <lsb>.
- The numerical values of <msb>, <lsb>, and any numerical <bit spec> shall be between 0 (the least significant bit) and N-1 (the most significant bit) of the N-bit ISC_PData register.

8.5.6.2 Description

The ISC_Security attribute identifies four pieces of information about the optional security mechanisms (see 5.1.11) implemented in a device, but only when the ISC_PData register is used to provide this information. The first three fields identify a bit number within ISC_PData for each of three protection signals ISC_Disable_Read (see 5.1.11.1), ISC_Disable_Program (see 5.1.11.2), and ISC_Disable_Erase (see 5.1.11.3) if any exist. The fourth field identifies the bits (if any exist) that form a security key. In <bit range>, the <msb> integer shall be equal to or greater than the <lsb> integer, describing a field of one or more bits.

NOTE—This also reflects the bit ordering of IEEE Std 1149.1-2001, where the rightmost bit of a register, closest to TDO, is bit 0.

Examples

A device with a full-featured security implementation could be described as follows.

```
attribute ISC_Security of PLD2:entity is
  "ISC_Disable_Read  27, " & -- Bit 27 controls read security
  "ISC_Disable_Program 26, " & -- Bit 26 controls program security
  "ISC_Disable_Erase  25, " & -- Bit 25 controls erasure security
  "ISC_Disable_Key    24-0 " ; -- Bits 24 down to 0 are a key
```

Another example shows how a very simple two-bit read/program security system would be described.

```
attribute ISC_Security of PLD3:entity is
  "ISC_Disable_Read  0, " & -- Bit 0 controls read security
  "ISC_Disable_Program 1, " & -- Bit 1 controls program security
  "ISC_Disable_Erase  *, " & -- No erase security
  "ISC_Disable_Key    * " ; -- No key
```

In the case where ISC_PROGRAM_SECURITY is implemented but targets the ISC_Default register, and then simply executing the instruction with no data instantiates whatever security feature has been provided by the device designer. No ISC_Security attribute is needed.

8.5.7 Attribute ISC_Flow

A flow is a logical sequence of ISC instructions that performs a basic programming function such as erasing programmable memory, programming it, or reading it. Certain ISC instructions are used repetitively in repeat blocks to perform these basic functions while others set up or terminate the function. When multiple devices are chained together, programming activities may be done concurrently (see Annex B), but the process of issuing instructions and data must be done serially through the TAP of the chain of devices.

A flow also specifies how programming data are to be provided to a programming process. The flow may contain hard-coded data, may calculate data (typically address data) algorithmically, or may read from or write to an external data file. (See Clause 9 for data format specifications designed to work with flows.)

NOTE—The word “file” is used here but is intended to include other data sources such as a graphical interface or telecommunication network links. In any case, the format of data received or transmitted is specified in Clause 9.)

Programmable logic devices are supported by vendor-supplied software that take a design description and derive the programmable memory download image needed to implement the desired design. For a single device, it is relatively straightforward to create a sequence of ISC instructions that will program this download image into the device. It is more challenging to assemble a concurrent programming stream of ISC instructions and data for a chain of devices, where some (but not necessarily all) devices have ISC capability and where ISC devices from several vendors could exist. What is needed is a description of how a given device’s ISC instructions may be utilized to program, erase, or read data. This description is called an “ISC Flow.” Then, application software that understands ISC Flows can examine these devices and, taking into account the overall structure of the chain, assemble the instruction/data streams needed to program multiple devices concurrently.

In general, ISC Flows have three phases of operation, as follows:

- a) An initialization phase that sets up a repeated operation
- b) A “body” that is repeated, each time delivering data to/from the device
- c) A termination phase that finishes up details

For example (assuming an ISC device has already entered the ISC Accessed system modal state), the phases of programming the device could be as follows:

- Empty (no initialization needed).

- A repeated portion that loads the ISC_PROGRAM instruction into the IEEE Std 1149.1-2001 instruction register and scans in data while (optionally) scanning out status and then waiting in the *Run-Test/Idle* TAP Controller state for a specified duration. A decision is made when to exit this portion based on the number of items to program.
- Load the ISC_DISCHARGE instruction into the IEEE Std 1149.1-2001 instruction register, scan out status, and go to the *Run-Test/Idle* TAP Controller state for a specified duration.

A different process might be required for a device that needs its internal addressing structure initialized but does not require a discharge cycle:

- First, load the ISC_ADDRESS_SHIFT instruction into the IEEE Std 1149.1-2001 instruction register. Next, scan starting address information into the device and go to the *Run-Test/Idle* TAP Controller state for a specified duration.
- A repeated portion that loads the ISC_PROGRAM instruction into the IEEE Std 1149.1-2001 instruction register and scans in data while (optionally) scanning out status and then waiting in the *Run-Test/Idle* TAP Controller state for a specified duration. A decision is made when to exit this portion based on the number of items to program.
- Empty. (Note the status of the final programming step can be observed in the data scan of a subsequent operation, such as ISC_NOOP.)

The attribute ISC_Flow gives information on how to execute a repeated instruction. However, this information may later be processed and optimized. Note that in the foregoing examples, the repeated portion shows ISC_PROGRAM being reloaded into the instruction register on every pass through the repetition. This guarantees that the repeat block will execute properly. However, optimizing application software may notice that once ISC_PROGRAM is loaded, it remains in place for the duration of the repetition. Thus, this software, similar to optimizing compilers, may schedule certain activities ahead of the point in the repeat block that are essentially redundant within the repeat block. When considering a concurrent programming flow of multiple devices, it is not known ahead of time what instructions will be placed inside the loop and which can be moved outside. (For example, the ISC_NOOP instruction may be placed inside a loop for one device to accommodate the activities of another device being programmed concurrently.) Thus, the BSDL description of the repeated operation must be “unoptimized” such that if additional instructions (e.g., ISC_NOOP) are inserted in the repeat block, the operation will still proceed correctly.

NOTE—It is important to understand that the ISC_Flow attribute does not describe the vectors to be applied to a device during programming. Rather, ISC_Flow describes what data are supplied to and expected from the device, what instructions are executed, and in what order. Software processes the flow to produce vectors for a device or chain of devices.

The attribute ISC_Flow also gives information on blocks of instructions that may be looped in adaptive programming algorithms. A maximum loop value must be provided so that if application software does not support adaptive programming, all of the information necessary to fully program the device is available. This will result in longer burn times than may be needed but will result in software that should be easier to develop. If a minimum loop value is provided, a greater amount of optimization and concurrentization should be possible when programming multiple devices. At any given time, for a given device, there will only be one loop as nested loops for adaptive programming are expressly disallowed. Any time a loop is encountered, there will be one or more test conditions required to terminate the loop. In a simple case, there will be a single bit that is observed to notify the software that the operation success factors have been met and the loop can be terminated. In a more complex case, there can be multiple test conditions that must all be satisfied for the loop to be terminated.

8.5.7.1 Specification

Syntax

```

<ISC flow description> ::= attribute ISC_Flow of <component name> : entity is <flow definition string>;
<flow definition string> ::= " <flow definition list> "
<flow definition list> ::= <flow definition> { , <flow definition> }
<flow definition> ::= <flow descriptor> [ <initialize block> ] [ { <repeat block> } ] [ <terminate block> ]
<flow descriptor> ::= <flow name> [ ( <data name> ) ] [ Unprocessed ] [ Exit_on_Error ]
<flow name> ::= <VHDL identifier>
<data name> ::= <standard data name> | <data identifier>
<standard data name> ::= Array | Usercode | Security | Idcode | Preload
<data identifier> ::= <VHDL identifier>
<initialize block> ::= Initialize <activity list>
<repeat block> ::= Repeat <repeat count> <activity list>
<terminate block> ::= Terminate <activity list>
<activity list> ::= <activity list element> { <activity list element> }
<activity list element> ::= <activity> | <loop block>
<loop block> ::= loop [ min <min loop count> ] max <loop count> ( <activity> { <activity> } )
<activity> ::= ( <instruction name> [ <update field list> ] <wait specification> [ <capture field list> ] )

<update field list> ::= <update field> { , <update field> }
<update field> ::= <field width> [ : <data expression> ]
<data expression> ::= <hex string> | <input specifier> | <variable expression>
<variable expression> ::= <variable> | <variable assignment> | <variable update>
<variable assignment> ::= <variable> = <hex string> | <variable> <input specifier>
<variable update> ::= <variable> <complement operator> | <variable> <binary operator> <operand>
<input specifier> ::= <input operator> | <IO operator>

<capture field list> ::= <capture field> { , <capture field> }
<capture field> ::= <field width> : <capture specification> [ : <CRC tag> ] [ : <OST tag> ]
<capture specification> ::= <expected data> [ <compare mask> ]
<expected data> ::= [ <TDO output> ] [ <data expression> ]
<compare mask> ::= * [ <compared output> ] [ <data expression> ]
<TDO output> ::= <output operator>
<compared output> ::= <output operator>

<wait specification> ::= wait <duration specification> [ min [ , <duration specification> max ] ]
<duration specification> ::= <clock cycles> | <time> | <clock cycles> , <time>
<clock cycles> ::= <port ID> <cycle count>
<cycle count> ::= <integer>
<variable> ::= $ <VHDL identifier>
<operand> ::= <integer>
<binary operator> ::= + | - | >> | <<
<complement operator> ::= ~
<input operator> ::= ?
<output operator> ::= !
<IO operator> ::= ?!
<field width> ::= <integer>
<loop count> ::= <integer>
<hex string> ::= <hex digit> { <hex digit> }
<hex digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | a | b | c | d | e | f
<CRC tag> ::= CRC
<OST tag> ::= OST

```

Rules and semantic checks:

- a) The syntax for the mandatory `ISC_Flow` attribute shall be that shown above.
- b) The value of `<min loop count>` shall be greater than 0 and less than or equal to `<loop count>`.

NOTE—This value is a decimal integer.

- c) The value of `<loop count>` shall be greater than 0 and greater than or equal to `<min loop count>`.

NOTE—This value is a decimal integer.

- d) The value of `<field width>` shall be greater than 0.

NOTE—This value is a decimal integer.

- e) The value of `<repeat count>` shall be greater than 0.

NOTE—This value is a decimal integer.

- f) At least one `<activity list>` shall exist in each `<flow definition>` element.
- g) For any `<activity>` element, the sum of all the values of `<field width>` elements within the `<update field list>` element (when it exists) shall equal the length of the register targeted by the instruction given by the `<instruction name>` element.
- h) For any `<activity>` element, the sum of all the values of `<field width>` elements within the `<capture field list>` element (when it exists) shall equal the length of the register targeted by the instruction given by the `<instruction name>` element.
- i) All significant (i.e., nonzero) bits in a `<hex string>` used within any `<data expression>` shall have bit positions between 0 and N-1 only, where N is the value of `<field width>` associated with the `<data expression>`.
- j) All significant (i.e., nonzero) bits in the binary representation of the value of an `<operand>` used within any `<data expression>` with the addition or subtraction `<binary operator>` elements shall have bit positions between 0 and N-1 only, where N is the value of `<field width>` associated with the `<data expression>`.

NOTE—The `<operand>` value is a decimal integer.

- k) The addition and subtraction operations shall be performed with 2s complement arithmetic, and carries beyond the field width shall be discarded.

NOTE—There is no way that carry bits generated in one `<data expression>` can be propagated to another `<data expression>`.

- l) The value of an `<operand>` used within any `<data expression>` with the left or right shift `<binary operator>` elements shall be between 1 and N-1 only, where N is the value of `<field width>` associated with the `<data expression>`.

NOTE—The `<operand>` value is a decimal integer.

- m) The left and right shift operators shall logically shift data, and any significant bits shifted beyond the field width shall be discarded.

NOTES

1—This disallows arithmetic shifting or any kind of wrap-around of shifted data. Nor is there any way that bits from one `<data expression>` can be propagated into an adjacent `<data expression>`.

2—In a logical shift operation, “0” bits are shifted into the register at the end opposite to the direction of the shift instruction. For example, a left shift operation shifts 0s in (and leftward) from the right side.

- n) The value of <cycle count> shall be greater than 0.

NOTE—This value is a decimal integer.

- o) The value of the real number <time> in a <duration specification> shall denominate time in units of seconds and shall be greater than 0.0.
- p) The value of <port ID> in a <clock cycles> element shall be the name of the TCK port as identified in the <TCK stmt> that appears earlier in the BSDL description.

NOTE—See “Scan Port Identification” in IEEE Std 1149.1-2001 for the <TCK stmt> definition.

- q) Each value of <flow descriptor>, excluding any **Unprocessed** or **Exit_on_Error** flags, in the <flow definition list> shall be unique.
- r) Each value of <instruction name> in an <activity> shall exist in the <instruction opcode stmt> that appears earlier in the BSDL description.

NOTE—See “Instruction Register Description” in IEEE Std 1149.1-2001 for the <instruction opcode stmt> definition.

- s) A <variable> or <variable update> element shall always be preceded by a <variable assignment> for the same variable name within a <flow definition>.

NOTE—This limits the scope of a variable and states that it must be initialized before being referenced for use. This requirement can be checked while parsing the definition.

- t) When a <variable> appears in an <update field> or a <capture field> element, the value of the <field width> for the <variable> shall be the same as that of the <field width> used in the <variable assignment> that defined the <variable> element.

NOTE—This requirement can be checked while parsing the definition. It prevents incompatible field widths between definition and use of a variable.

- u) When one or more <input operator> or <IO operator> elements exist in a <flow definition>, the associated <flow descriptor> shall contain a <data name>.

8.5.7.2 Description

A <flow description> is a list of one or more uniquely named <flow definition> elements, each containing up to three blocks to perform initialization, repetitive actions, and termination. A <flow definition> is named by a unique <flow descriptor> element consisting of a <flow name> and optionally referencing a <data name> element. The <data name> element may be one of five standard names (**Array**, **Usercode**, **Security**, **Preload**, and **Idcode**) or an identifier. A <data name> in the BSDL file must correspond to a <data name> that will be found in the data file for the device (see Clause 9). A run-time error will occur if a <data name> in a flow does not match a <data name> in the data file. A standard name may be recognized by tools as a special case that, for example, may cause the invocation of a user interface that interacts with the user.

In some devices with simple internal organizations, the **Array** <data name> indicates the principal programming area of the device. However, in devices where the internal organization may be composed of architecturally distinct regions, the <data identifier> may be used to name these regions. Examples might be “Logic” and “IO_Config” for a device with a region that configures the internal logic and a second that controls the parametric behavior of device pins.

The **Usercode** and **Security** <data name> elements are used for accessing usercode and security data patterns. This allows software to specifically target these areas when applications need to vary their content, such as when usercodes are customized. For example, a flow may be constructed to read a static usercode from a data file. A tool that processes this flow can see the **Usercode** data name and intervene, replacing the static file data with customized data as appropriate to the application. Similar processing can be done for setting security.

The **Preload** <data name> element can be used to identify data that are used by a PRELOAD instruction to initialize the Boundary-Scan Register. This data may be calculated to set up board-level conditions that are needed to avoid bus fights or other potentially damaging problems.

The **Unprocessed** tag on a <flow descriptor> is used to identify a flow that writes out data that cannot later be read in by a complementary flow without additional preprocessing. (This flow could be used to “copy” a programmed device into an unprogrammed device if the original data file is unavailable.) The tag signifies that the data need to be processed by a vendor-supplied routine before it can be written into a new device. This can happen when reading and writing a device are not symmetrical activities. For example, the processing routine might need to add addresses, rearrange bit fields, provide masking information, etc. If the two flows supplied for copying are symmetrical, then no post-processing is required and the tag need not be supplied in the read-out flow. (The **Unprocessed** tag also appears in the data file syntax given in Clause 9.)

The **Exit_On_Error** tag on a <flow descriptor> is used to identify a flow that could generate an error that should cause all programming activity to come to a halt. Halting programming, for this device and any being concurrently programmed with it, may be desirable to eliminate wasted time—for example, it may be pointless to continue with the process that may be lengthy since it will ultimately be redone in the future. The actual halt occurs at the end of the instruction register or data register scan that generated the error. (See 8.5.8.2 for more discussion on error processing.)

Any <initialize block>, <repeat block>, or <terminate block> in a <flow definition> element (but typically the initialization or termination block) may be omitted, although at least one must exist. Multiple <repeat block> elements are permitted. A block is an <activity list> of one or more <activity> elements that may, or may not, be included inside of a loop. If it is inside of a loop, then there will be a <loop count> that provides the maximum number of iterations that activity list can be executed and possibly a <min loop count> that provides a minimum number of iterations the loop should be executed before the test condition, indicated by one or more “:OST” tags, is evaluated. An <activity> consists of

- An ISC instruction <instruction name> to be shifted in on TDI and updated into the instruction register
- An optional <update field list> giving the instruction arguments to be shifted in on TDI and updated into the data register prior to execution
- A <wait specification> that gives the duration spent in the *Run-Test/Idle* TAP Controller state
- An optional <capture field list> to specify what (if anything) to compare with the instruction results captured by the data register after execution as they are shifted out on TDO.

NOTES

1—A <capture field list> may also specify the shifting of raw (uncompared) data and data output. See 8.5.7.3.

2—The lengths of data specified by both the <update field list> and <capture field list> are the same length as the register targeted by the instruction. It is possible to overlap the shifting out of data from one execution of an instruction with the shifting in of new data needed by a new execution of that same instruction.

The field lists that are provided depend upon how the instruction is implemented in a specific device. If a field list is omitted in an activity, the corresponding data register scan need not be performed for the specified instruction, or if it is performed, the data are “don’t care” and have no effect on the instruction’s execution.

The <repeat block> contains a (decimal) <repeat count>. When data for a repeat block are generated by the flow itself, the repeat count tells how many times to repeat. When data come from a data file, the repeat count is a maximum. The repeat block terminates without error if the data, coming from a data file, are fully exhausted at the end of a repetition, even if the repeat count has not been reached. It is an error if there is data left over when the repeat count is reached. It is an error if there is not enough data to complete a full iteration of the repeat block.

The field list for an activity comprises a single complete data register scan. The total <field width> of the field list must exactly equal the data register size declared in the Register_Access attribute. A field list is interpreted with the least significant field on the right and the most significant field on the left.

An <update field> consists of a (decimal) <field width> followed (optionally) by a <data expression> giving data to shift and update into the device in the *Update-DR* TAP Controller state.

Capture field data are used to describe the relevant bits captured by a data register in the *Capture-DR* TAP Controller state. A <capture field> consists of a (decimal) <field width> followed by (optional) <expected data> and an (optional) <compare mask> specification. The <expected data> is a <data expression> preceded by an optional <TDO output> operator that specifies that the raw captured data are to be output to a data file. If the <data expression> exists, this implies comparing raw captured data with the data from the expression. If the <data expression> is omitted, the default comparison with all-zeros is performed. These data are optionally masked with data specified by the <compare mask> element. The <compare mask> also consists of a <data expression> preceded by an optional <compared output> operator that specifies that the masked data are to be output to a data file. When no explicit <compare mask> is provided, the mask defaults to all 1s, meaning all bits are to be compared or output unchanged. The length of the mask must equal the length of the <expected data>. (See 8.5.7.3 for more detail on how data are compared, masked, and output.)

A <data expression> may be (among other things) a hexadecimal <hex string> constant. The <hex string> constant is the conventional numeric representation, with the least significant bit on the right. Nonsignificant leading zeros can be omitted. It is an error if there are more significant bits in the constant than the number of bits specified by the field width.

A <data expression> may assign a value to a <variable> that may be operated upon by various unary or binary operators and used again later in the flow. If this is the first assignment via the “=,” “?,” or “?!” operators, a new <variable> is created. The <variable> remains in scope until the last <activity> of the <flow definition> completes; at which time, it becomes undefined. The <variable> is visible in every block within a <flow definition>. The size of the <variable> is the width of the field in which it is first assigned a value. If this is not the first assignment, the previously created <variable> is used. It is an error if an assignment of an existing <variable> happens in a <data expression> of different <field width> than the <field width> of the original assignment. Such errors can be determined statically when the flow is read.

A <data expression> may reference a <variable>. It is an error if the <variable> has never been assigned a value at the time it is first referenced. Such errors can be determined statically when the flow is read.

NOTE—The “\$” character that precedes a <VHDL identifier> in a <variable> allows software to unambiguously differentiate a <variable> from a <hex string>. For example, “\$BAD1” is a <variable> while “BAD1” is a valid hexadecimal constant.

A <data expression> may update a <variable> by applying an operator to it.

- The logical inversion “~” <unary operator> inverts the state of each bit in the <variable>.
- The addition “+” and subtraction “-” <binary operator> elements perform arbitrary length unsigned addition and subtraction of a decimal <operand> with unchecked overflow on the <variable>. The result of overflow is undefined and should be avoided. Both operators take an unsigned <operand>, a decimal <integer> constant argument used to increment or decrement the variable. Legal values for

- the increment or decrement <operand> are 1 through 2^N-1 , where N is the field width of the variable.
- The shift toward most significant bit (<<) and shift toward least significant bit (>>) operators perform logical shift with zero-fill on the variable. Both operators take an unsigned <operand>, a decimal <integer> constant argument giving a number of bits to shift the field. Legal values for the shift count <operand> are 1 through N-1, where N is the field width of the variable.

NOTE—Because an <operand> is decimal, a <data expression> may contain a mixed-base operation of a hexadecimal <variable> with a <decimal> operand. This should seem natural when writing shift expressions, but may surprise writers of addition or subtraction expressions. However, it is likely that the values of operands will be small (for example, “1”) in most cases, avoiding confusion.

The operator definitions imply an understanding that the least significant bit of the field is on the right; that is, the least significant bit is shifted into TDI first. The addition and subtraction operators will not perform as intended for fields that have the least arithmetically significant bit closest to TDI (or on the left).

When an <input operator> is encountered inside of a loop construct, special care must be taken so that only one set of data is used until the operation inside of the loop is successful. The data should be read in from the data file, saved, and used with every iteration of the loop until success is achieved and the programming flow advances to the next <activity>. When an <output operator> is encountered, data should only be written to a data file on the last iteration of the loop. Data should not be written on any iteration of the loop other than the last iteration.

The <wait specification> gives the minimum and (optional) maximum wait times. A wait specification with a single <time> value (a <real number>) implies a minimum, with no specified maximum. A <duration specification> within a <wait specification> may list a (decimal) number of <clock cycles> on a <port ID>, a real <time> value, or both in succession.

8.5.7.3 Data input and output formatting

A <data expression> may contain “?” or “?! ” <input operator> or <IO operator> elements. When either is encountered, the value for the field is read from the data input source, and in the case of the “?! ” <IO operator>, the data are subsequently written to the output file. Bytes containing hexadecimal data are read from the input, mapped to binary values, and used to fill in the field starting with the least significant (closest to TDO) byte of the field. Enough bytes are read to fill the field. If the <field width> is not an exact multiple of 4 bits, then there are unused bits in the more significant part of the first (i.e., leftmost) hex character read in. These bits must be zero and are not used. As an example, for a field width of 5, the <hex string> “1F” represents all five bits set to one. The <hex string> “1E” represents the top four bits set to one and the LSB, nearest TDO, set to zero. The <hex string> “70” has two unused bits set to one and is an error. Figure 29 diagrams the conversion process for another example.

NOTE—Execution software that reads an <update field list> (comprising 1 or more <update field> elements) from a flow must read the entire expression list before it can assemble the actual bit pattern to be shifted into a device. This is because the least significant bit of the expression list is the first to be shifted into TDI, but is the last bit read in from the flow.

When an <update field list> expression is omitted and data are shifted out of the data register for that activity, the default data shifted into the register may be chosen by the software application.

A <capture field> expression may contain constant data that are to be compared with capture data shifted from the data register of the device. Optionally, this data may be masked with <compare mask> information also read from the flow. Figure 31 shows the organization of <capture field> data and <compare mask> bits.

NOTE—Execution software that reads a <capture field list> (comprising 1 or more <capture field> elements) from a flow must read the entire expression list before it can assemble the actual bit pattern to be compared with data shifted out

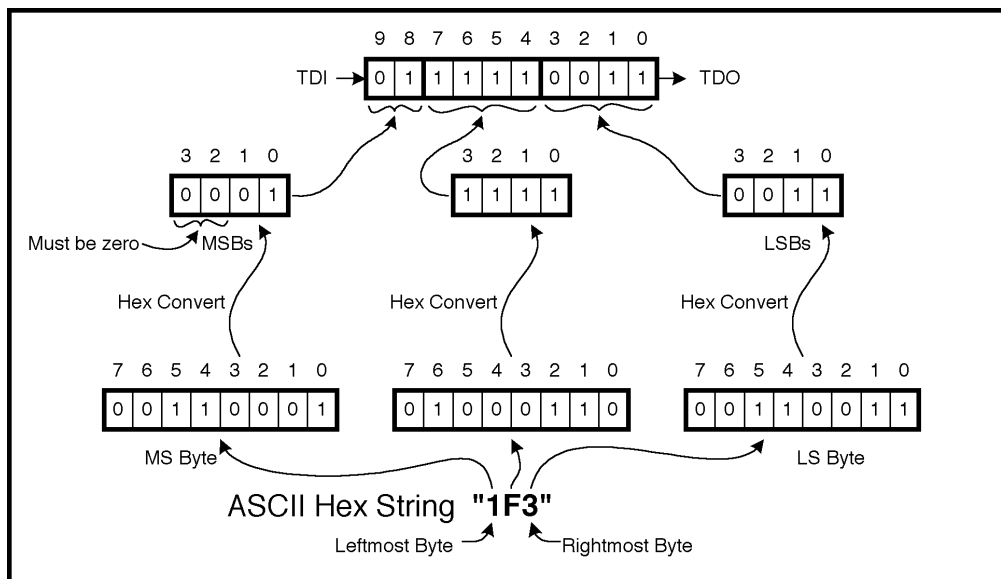


Figure 29—The “?” <input operator> conversion of three ASCII characters (24 bits) to three hexadecimal values (12 bits) to describe a 10-bit data register’s content

of a device. This is because the least significant bit of the expression list is the first to be shifted out of TDO, but is the last bit read in from the flow. The same is true for mask data when it exists.

A <capture field> expression may contain one or more “!” <output operator> elements. When this operator is encountered, register data are converted to ASCII as depicted in Figure 30. Enough ASCII bytes are written to express all the data. The most significant hexadecimal digit of the field becomes the leftmost byte. If the field is not an exact multiple of 4 bits, zeros are inserted for the non-field bits in the most significant part of the most significant hexadecimal digit. Thus, data written by the “!” output operator are compatible for reading with the “?” input operator of the same field width.

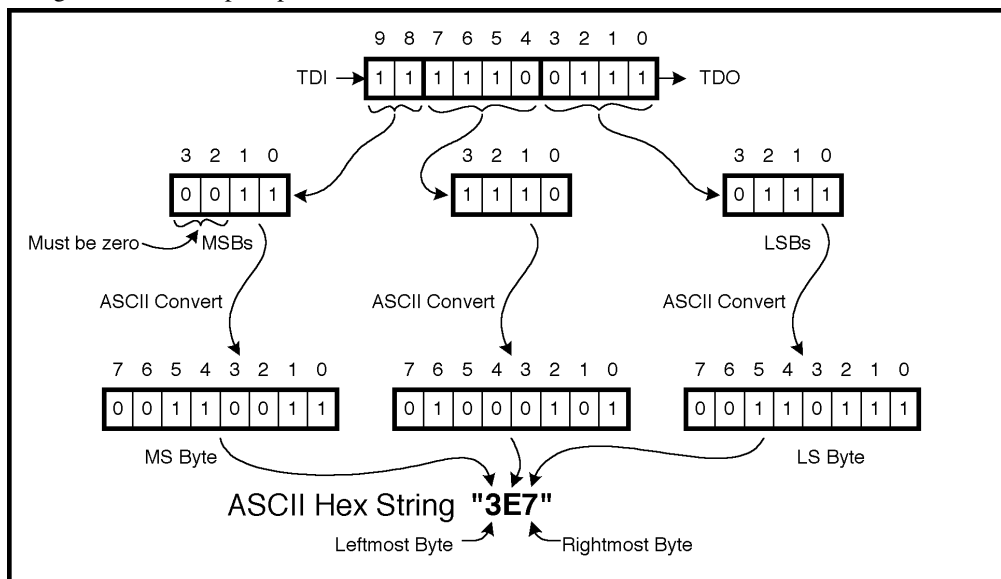


Figure 30—The “!” <output operator> causes shift data to be written to the output in ASCII hexadecimal text. Here, a 10-bit register is converted to three hex digits that are then converted to their ASCII representation

NOTE—Execution software that writes capture data to files is expected to write the data in the image shown in Figure 30. This means the entire register must be shifted out before it can be written to a file because the first bit shifted out of TDO forms the last to be written to the file.

When a <capture field list> is omitted, and the data register for the associated activity is read out, the content of the register is ignored; i.e., the implicit <compare mask> is all-zeros.

8.5.7.4 Data output source mapping and CRC enabling

There are four sources of data that can be specified by a <capture field> element. They are as follows:

- Raw capture data (the TDO stream unmasked)
- Expected data
- Mask data
- Compared and masked capture data

These four sources are shown in Figure 31 along with a mapping that shows how each “!” operator refers to an associated source. (Note that it is unlikely that all four possible output operators would be used in a <capture field>, but they could.) Such a <capture field> would emit four 4-bit data bytes. From left-to-right, they would be the raw TDO, the expected data, the masked compared TDO, and the mask. There is no provision to emit the CRC enable stream as this is formally encoded in the flow itself and never read from a file.

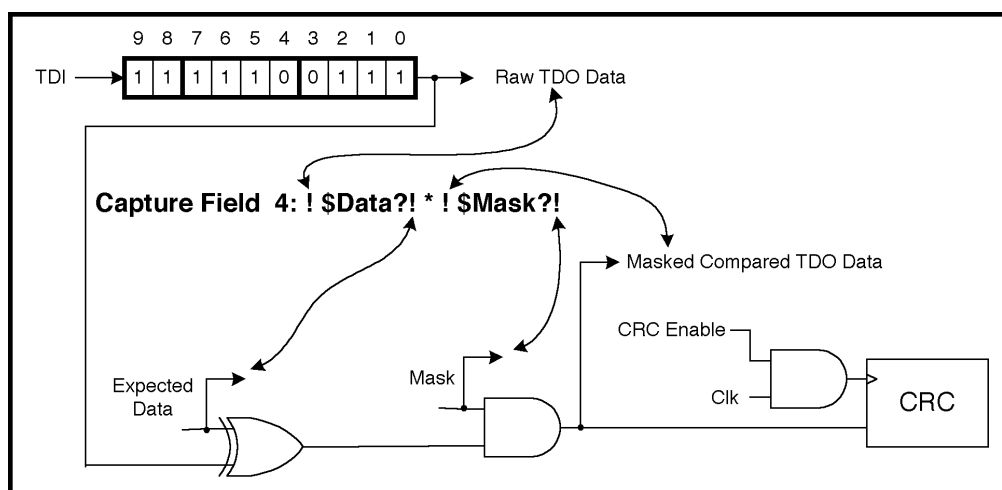


Figure 31 — Mapping of data sources in a <capture field> element

Figure 32 shows an example of a <capture field list> containing three <capture field> elements that together specify the output of 10 bits of raw captured TDO data. The list specifies the following:

- 10 bits of defaulted expected data (defaulted to zeros)
- 10 bits of mask data (2 specified, 4 defaulted to ones, 4 defaulted to ones)
- 10 bits of CRC enable (2 defaulted to zeros, 4 enabled, 4 defaulted to zeros)
- 10 bits of captured TDO data as shown

Figure 33 shows a similar <capture field list> that sends masked raw TDO data to an output file. Again, the expected data are defaulted to zeros and the mask and CRC enable bits play a role in masking TDO and choosing which of the raw TDO bits are accumulated in the CRC.

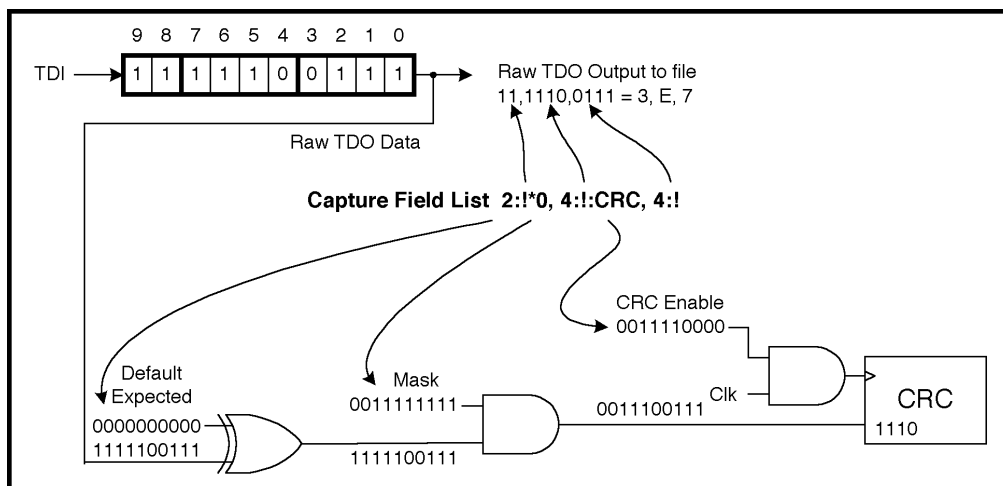


Figure 32—A Capture Field List may specify the output of raw (uncompared) captured data to a file, with output data CRC accumulation. CRC data are accumulated after masking

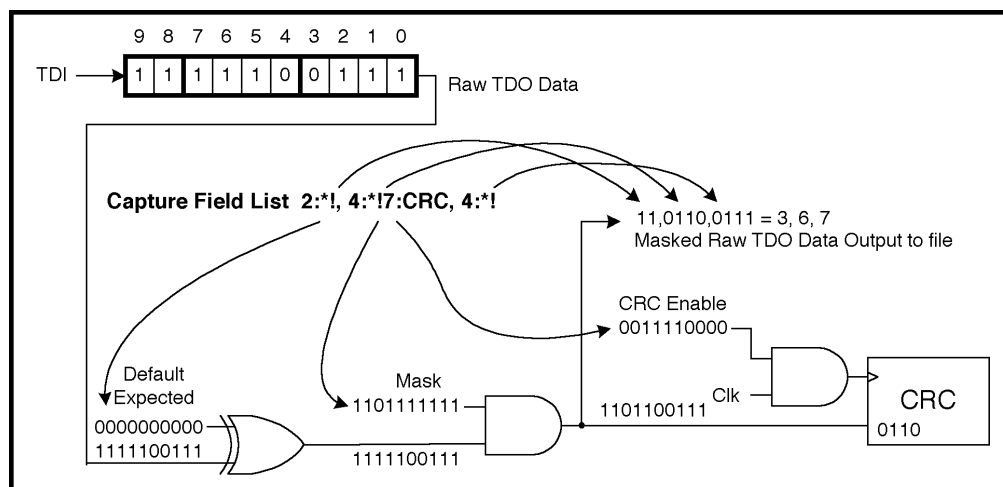


Figure 33—Masked capture data output to a file, with output data CRC accumulation. CRC data are accumulated after masking

Finally, Figure 34 shows a <capture field list> that contains expected data. Now compared and masked data are sent to the output and a portion of it is accumulated in the CRC. These data are no longer raw, but represent failure information. A logic 1 represents a bit that differs from the expected value.

8.5.7.5 CRC accumulation

The <capture field> syntax may have an appended <CRC tag> to indicate that the data from that field is to be processed into a cyclic redundant code (CRC, see 3.1.12). CRC encoding provides superior performance compared to older checksum algorithms in that the possibility of bad data aliasing as good is greatly reduced. The older parallel checksum accumulation methods are incompatible with serial programming streams, especially when multiple devices are programmed concurrently. However, the ability to guard data integrity for programming activities with a simple hexadecimal signature is preserved with CRC generation. The CRC algorithm is found in 8.5.7.6.

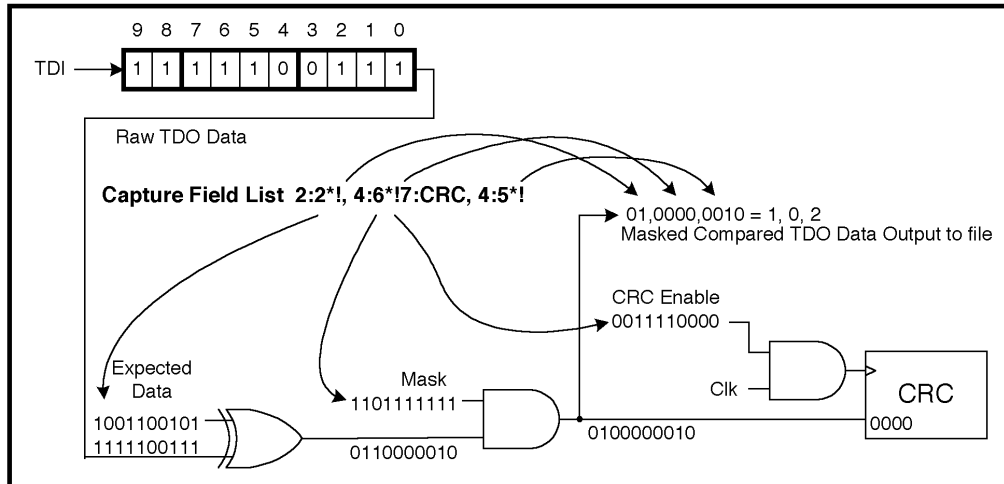


Figure 34—Compared and masked capture data output to file, with compared data CRC accumulation. CRC data are accumulated after comparison and masking

NOTE—Data CRC encoding should not be confused with transmission CRC encoding. See 9.5.

When a <capture field> is tagged for CRC accumulation, the <field width> determines the number of bits that are accumulated. Because a field may be a member of a list of fields that together describe a data register, all or part of the data being shifted out of a register may participate in the CRC calculation. Bits are processed in TDO-to-TDI order.

NOTE—Programming system hardware may or may not choose to stop/abort when a failure is encountered. If a system continues, ignoring errors, the presence of an incorrect CRC can be used to simply determine that an error occurred somewhere in the process.

The data shifted from the device may be masked as it is accumulated. Thus, the shifted output data are ANDed with the mask before it is accumulated in the CRC, as in Figure 32. The <CRC tag> elements can be applied without restriction, but an author of flows should carefully consider the implications of their placement.

NOTE—There is to be a unique output data CRC accumulator per device when multiple devices are programmed concurrently since it not possible to predict in advance what an aggregate data CRC for an unknown collection of devices would be.

A tool vendor, with knowledge of a device's implementation given by BSDI, can simulate the results of a flow and associated data to predict a priori what CRC will result from executing a flow.

If the <CRC tag> is included inside of a loop construct, then the CRC should only be accumulated on the last iteration of the loop, whether the loop is successful or not. Accumulating the CRC on every iteration of the loop will lead to an incorrect CRC value when compared to the CRC value given in the data file.

8.5.7.6 Data CRC algorithm

The algorithm for data CRC calculation is given in the “C” language. The counter is 32 bits long and uses a maximal length feedback polynomial.

```
#define Polynomial 0x04C11DB7 // CRC-32 polynomial (Ethernet)
#define High_Bit 0x80000000 // Bit 31
#define Low_Bit 0x00000001 // Bit 0
```

```

unsigned int CRC_Register;    // 32-bit CRC accumulator

void init_crc()
{CRC_Register = 0;           // Clear CRC value
}

unsigned int crc_result()
{return CRC_Register;        // Return the CRC value
}

void compute_crc(unsigned int Bit)    // LSB has bit to accumulate
{Bit &= Low_Bit;                     // Assure only LSB present
if (CRC_Register && High_Bit)         // if MSB is set
    CRC_Register = (CRC_Register<<1)^Polynomial^Bit;
else                                 // MSB not set
    CRC_Register = (CRC_Register<<1)^Bit;
}

```

The initializer `init_crc()` sets the CRC register to zero. The zero code is a one-step cycle, whereas the nonzero codes are a maximal $2^{32}-1$ length.

8.5.7.7 Loop conditional testing

The <capture field> may have an appended <OST tag> to indicate that this capture field provides an operation success test that must be satisfied to exit out of a loop. If there is more than one <OST tag> inside of a loop, then all of the operation success tests must be satisfied before the loop can be exited.

If the loop reaches the maximum <loop count> and the operation success test conditions have not been met, then this is considered an error. When an error is declared, it should be handled based on whether or not the **Exit_On_Error** tag has been associated with the flow used to describe the adaptive programming loop (see 8.5.7.2). If the tag is associated with the flow, then all programming operation should stop at the end of the data register or instruction register scan that generated the error. The `ISC_Done` signal should remain unprogrammed.

If there is a <min loop count> specified, any operation success test conditions should be ignored until the <min loop count> value has been met. The operations success test conditions should first be evaluated on the same iteration as the <min loop count>. Additionally, if a <min loop count> is provided, then the software shall execute each and every one of the iterations up to the <min loop count> rather than just assume that it would be acceptable to reduce the number of iterations by executing each of the activities in the <loop block> for <min loop count> multiplied by the duration specified in each of the activities. There is no guarantee that the device designer has not implemented circuitry that turns itself off after the minimum duration and that waiting in the *Run-Test/Idle* TAP Controller state for a longer amount of time will have the same effect as looping through *Run-Test/Idle* multiple times.

Rules

- a) If an <min loop count> is provided, the software tool shall execute each iteration of the activities specified in the <loop block> up to the <min loop count>.

8.5.7.8 ISC_Flow examples

Examples appear next. Most examples show a single <flow definition> for a device though multiple definitions can be described in a single `ISC_Flow` attribute.

NOTE—In some examples given below, <capture field list> data shifted out by an action may be shifted out at the same time as data in an <update field list> *for the next action* is being shifted in (assuming both actions target the same ISC register). In other cases, a sequence of actions only specifies shift in or shift out data but not both. In this event, don't care data may be specified for input or output.

Example 1: This device corresponds to the very simple device of Figure 22 of 5.1.12. The 32-bit ISC_PData register contains a 16-bit encoded (counting 0, 1, 2, ...) row address and 16-bit data word. The address is initialized in the initialize block and incremented by the “+” operator in the repeat block. Data come from an external data source (labeled “Array”) using the “?” operator. Since this device does not report status, and the instructions do not capture data, the capture field list is omitted, to indicate that no interesting data are captured. The variable created in the initialize block holds the initial value of the address. This variable is incremented in the repeat block. Since the first location is programmed in the initialize block, there remain $2^{16}-1$ locations to program in the repeat block.

```
attribute ISC_Flow of Example1:entity is
  "program(Array) " &
  "  Initialize    " &
  "    (ISC_PROGRAM 16:$addr=0,16:?      wait 1.0e-2) " &
  "Repeat 65535 " &
  "    (ISC_PROGRAM 16:$addr+1,16:?      wait 1.0e-2) ";
```

Example 2: This device corresponds to the device of Figure 23. The ISC_ADDRESS_SHIFT instruction selects a 16-bit encoded programming address register. ISC_PROGRAM selects a 32-bit data register. The address is initialized to a constant, and then incremented by rule. Data come from the data file. Although the address register is 16 bits, there are only 1024 rows of data.

```
attribute ISC_Flow of Example2:entity is
  "program(Array) " &
  "  Initialize    " &
  "    (ISC_ADDRESS_SHIFT 16:$addr=0      wait TCK 1) " &
  "    (ISC_PROGRAM      32:?              wait 1.0e-2) " &
  "  Repeat 1023 " &
  "    (ISC_ADDRESS_SHIFT 16:$addr+1      wait TCK 1) " &
  "    (ISC_PROGRAM      32:?              wait 1.0e-2) " &
```

Example 3: This device corresponds to the device of Figure 24. In this device, the 96-bit decoded (1-of-n) row address is initialized via ISC_ADDRESS_SHIFT. The address is implicitly shifted within the device by ISC_PROGRAM. Since the address is implicitly shifted, no variable is needed in the flow to perform the increment. Rows are programmed by ISC_PROGRAM, with a 32-bit data word obtained from the data file.

```
attribute ISC_Flow of Example3:entity is
  "program(Array) " &
  "  Initialize    " &
  "    (ISC_ADDRESS_SHIFT 96:1              wait TCK 1) " &
  "  Repeat 96    " &
  "    (ISC_PROGRAM      32:?              wait 1.0e-2)";
```

Example 4: This device corresponds to the device of Figure 25. In this device, the address has two components, a 1-bit array selector and a 96-bit decoded row selector. The address is initialized by the ISC_ADDR_SHIFT instruction and implicitly post-incremented by the ISC_READ instruction. The device has two programmable arrays. One array (labeled “Logic”) uses all 96 rows of the decoded address, and the other array (labeled “IO_Config”) uses only 8 rows. The ISC_RData register captures 64 bits of data. However, the rows of the smaller array read by this flow do not have data in the lowest 16 bits. The values captured for these unused bits are indefinite, so they must be masked out. Because ISC_READ does not

update the ISC_RData register, the update field list is omitted, meaning that no data register scan need be performed before the ISC_READ is executed in the *Run-Test/Idle* TAP Controller state. CRC data are accumulated on the content of each array.

```
attribute ISC_Flow of Example4:entity is
  "CRC_Read(Logic) " &
  " Initialize " &
  " (ISC_ADDRESS_SHIFT 1:0,96:1 wait TCK 1) " &
  " Repeat 96 " &
  " (ISC_READ                               wait TCK 1 64::CRC), "&

  "CRC_Read(IO_Config) " &
  " Initialize " &
  " (ISC_ADDRESS_SHIFT 1:1,96:1 wait TCK 1) " &
  " Repeat 8 " &
  " (ISC_READ                               wait TCK 1 64:*FFFFFFFFFFFFFF0000:CRC) ";
```

Example 5: This device corresponds to the device of Figure 26. In this device, the address has two components, an array selector in the one-bit ISC_Config register and a 96-bit decoded (1-of-n) row address in the ISC_Address register. The address is implicitly shifted by ISC_READ, which captures 32 bits of data. The first array (labeled “Logic”) has 96 rows of 32-bit data words. The second array (labeled “IO_Config”) has 10 rows, which capture 16 valid data bits and 16 indefinite data bits. The two flows in the example blank check the arrays.

```
attribute ISC_Flow of Example5:entity is
  "blankcheck(Logic) " &
  " Initialize " &
  " (ISC_SETUP1:0wait TCK 1) " &
  " (ISC_ADDRESS_SHIFT96:1wait TCK 1) " &
  " Repeat 96 " &
  " (ISC_READwait TCK 1 32:FFFFFFFF), " &

  "blankcheck(IO_Config) " &
  " Initialize " &
  " (ISC_SETUP 1:1 wait TCK 1) " &
  " (ISC_ADDRESS_SHIFT96:1wait TCK 1) " &
  " Repeat 10 " &
  " (ISC_READwait TCK 1 32:FFFFFFFF*FFFF0000)";
```

Example 6: This device corresponds to the device in Figure 27. In this device, the address cannot be explicitly set. Instead, a control bit in the ISC_PData register commands the address generation logic either to reset or to post-increment the address. A total of 212 rows of 400 data bits are programmed into the device. This device requires some time to discharge the row lines after each row is programmed. Normally, this time is hidden by the time spent shifting in the next row of data. However, after the last row is programmed, a delay is added so that the next operation may safely begin.

```
attribute ISC_Flow of Example6:entity is
  "program(Array) " &
  " Initialize " &
  " (ISC_PROGRAM1:0,400:?wait TCK 100) " &
  " Repeat 211 " &
  " (ISC_PROGRAM 1:1,400:?wait TCK 100) " &
  " Terminate " &
  " (ISC_DISCHARGE wait TCK 400)";
```

Example 7: In this device, separate instructions load the address, load the data, command programming, and discharge the device after programming. A device designer might choose to implement programming in several steps to provide events that sequence the internal operation of the device. The address is 16 bits, and the data are 200 bits.

```
attribute ISC_Flow of Example7:entity is
  "Program(Array) " &
  " Initialize      " &
  " (ISC_ADDRESS_SHIFT16:$addr=0wait TCK 1) " &
  " (ISC_DATA_SHIFT200:?wait TCK 1) " &
  " (ISC_PROGRAM      wait 1.0e-2) " &
  " (ISC_DISCHARGE    wait 1.0e-3) " &
  " Repeat 65535      " &
  " (ISC_ADDRESS_SHIFT16:$addr+1wait TCK 1) " &
  " (ISC_DATA_SHIFT200:?wait TCK 1) " &
  " (ISC_PROGRAM      wait 1.0e-2) " &
  " (ISC_DISCHARGE    wait 1.0e-3)";
```

Example 8: This device corresponds to Figure 22, but with some added complexities. Here, 96 bits of address are explicitly updated to the ISC_RData register and 32 bits of data are captured in the least significant part of ISC_Rdata register. Even-numbered rows have valid data in every data bit, but odd-numbered rows have valid data only in the low half of each byte. An indefinite value is captured in the high half byte. This flow blank checks the device. The flow does not read any data from an external data source.

```
attribute ISC_Flow of Example8:entity is
  "Blank_check " &      -- No <data name>
  " Initialize      " &
  " (ISC_READ96:$addr=1,32:0wait TCK 1096:0*0,32:0) " &
  " (ISC_READ 96:$addr<<1,32:0 wait TCK 10  96:0*0,32:0*0F0F0F0F) "&
  " Repeat 47      " &
  " (ISC_READ96:$addr<<1,32:0wait TCK 1096:0*0,32:0) " &
  " (ISC_READ96:$addr<<1,32:0wait TCK 1096:0*0,32:0*0F0F0F0F)";
```

Example 9: Simple verify of a device program. The ISC_RData register captures a 16-bit data word and uses an address set by the ISC_PROGRAM or ISC_ADDRESS_SHIFT instruction. Verification is performed by reading each location and comparing it to data read from a file. Output CRC generation is performed on the data.

```
attribute ISC_Flow of Example9:entity is
  "verify(Array) " &
  " Initialize      " &
  " (ISC_ADDRESS_SHIFT 16:$addr=0 wait 1.0e-3) " &
  " (ISC_READwait tck 1 16:?:CRC) " &
  " Repeat 65535      " &
  " (ISC_ADDRESS_SHIFT 16:$addr+1wait 1.0e-3) " &
  " (ISC_READwait tck 1 16:?:CRC)";
```

Example 10: Incremental Verify after Program. This device has an ISC_PData register that captures a 16-bit address and 16-bit data word. The ISC_RData register captures a 16-bit data word and uses an address set by the ISC_PROGRAM or ISC_ADDRESS_SHIFT instruction. Incremental verify is performed by reading each location after programming it. This process reuses the programming data saved in a variable, so it does not have to appear twice in the data file. Output CRC generation is performed on the data.

```
attribute ISC_Flow of Example10:entity is
```

```

    "verify(Array) " &
    " Initialize " &
    " (ISC_PROGRAM 16:$addr=0,16:$data? wait 1.0e-3) " &
    " (ISC_READ wait tck 1 16:$data:CRC) " &
    " Repeat 65535 " &
    " (ISC_PROGRAM 16:$addr+1,16:$data? wait 1.0e-3) " &

    " (ISC_READ wait tck 116:$data:CRC)";

```

Example 11: Compound Addresses. This device has an address register with two parts. There is a 3-bit encoded array selector and a 96-bit decoded (1-hot) address. The ISC_PData register contains the address, a 32-bit data field, and 2-bit non-overlapping status. A <flow definition> for each selected array is needed even though the array selector number comes from the data file.

```

attribute ISC_Flow of Example11:entity is
"program(Array_A) " &
" Initialize " &
" (ISC_PROGRAM 3:sel?,96:$addr=1,32:?,2:0 wait 1.0e-3 131:0*0,2:2) " &
" Repeat 95 " &
" (ISC_PROGRAM 3:sel,96:$addr<<1,32:?,2:0 wait 1.0e-3 131:0*0,2:2) ," &
"program(Array_B) " &
" Initialize " &
" (ISC_PROGRAM 3:sel?,96:$addr=1,32:?,2:0 wait 1.0e-3 131:0*0,2:2) " &
" Repeat 95 " &
" (ISC_PROGRAM 3:sel,96:$addr<<1,32:?,2:0 wait 1.0e-3 131:0*0,2:2) ," &
"program(Array_C) " &
" Initialize " &
" (ISC_PROGRAM 3:sel?,96:$addr=1,32:?,2:0 wait 1.0e-3 131:0*0,2:2) " &
" Repeat 95 " &

" (ISC_PROGRAM 3:sel,96:$addr<<1,32:?,2:0 wait 1.0e-3 131:0*0,2:2)";

```

In this example, each array was identical in size. They could be different.

Example 12: Writing data to a file. This flow can be used to interrogate the programming of an unknown device to create a data file that can then be used by a complementary programming flow (as in Example 1 above) to replicate the program in other devices. (A tool that expects to read out a device and create a new data file compatible with programming other devices would have to create the appropriate formatting of the file as described in Clause 9.) This device has an ISC_Rdata register with 16 bits of address and 16 bits of data. The address is set up before the wait time, and 16 bits of data is written to the data output after the wait.

```

attribute ISC_Flow of Example12:entity is
"Interrogate(Array) " &
" Initialize " &
" (ISC_READ 16:$addr=0,16:0 wait 1.0e-3 16:0*0, 16:!) " &
" Repeat 65535 " &
" (ISC_READ 16:$addr+1,16:0 wait 1.0e-3 16:0*0, 16:!) ";

```

When the device address has more than one part, such as a row and column address or an address plus array selector, the conceptual programming algorithm has nested repeats or recursive functions. The flow list of ISC_Procedure (see 8.5.8) may command performance of a flow multiple times. The effect is that all levels of outer repetitions are unrolled into ISC_Procedure's flow list. ISC_Flow is a simple linear flow with one implicit repeat block representing the innermost repetition. A device with a complex address structure could have lengthy ISC_Procedure flow lists.

The flow in example 11 reads the array selector from the file, and then programs all the rows in the selected array. ISC_Procedure's flow list for programming would command this flow to be performed once for each array. The data file would have flow IDs for each array, with different values for the array selector and perhaps different numbers of rows. If one array needed a different burn time, the designer would define a similar flow for that array.

The designer must define status bits explicitly for the following reasons:

- There is no way for the software that parses flows to distinguish between a semantically incorrect field list with too few bits specified and a correct field list with implicit status bits.
- If status bits were explicitly provided for some, but not all, activities in the flow, the interpretation would be ambiguous since software would not know whether to use implicit rules to generate the status bits, or to reuse the explicit rules seen elsewhere.
- The software could use explicit status if present, and implicit rules if it was not. But if status overwrote bits instead of adding onto a register, it would be ambiguous to determine by inspecting an activity whether it was correctly written.

Thus, to avoid ambiguity that can lead to differing behaviors among tools, status bits must be defined.

Example 13: Simple program of a device using an adaptive programming algorithm. This flow describes the programming algorithm for a simple device that has a single bit used for the operation success indication. The value for the operation success indication is included in the flow description rather than in the data file for the device. The operation success indication is included as subcodes in the status bits and can be accessed by using the ISC_NOOP instruction. The device does not have an automatic shutoff mechanism, so the ISC_DISCHARGE instruction is used to disable the programming circuitry.

```
attribute ISC_Flow of Example13:entity is
"program (Array)      " &
  "Initialize          " &
  "  (ISC_PROGRAM      16:$addr=0,16:?      wait 1.0e-2) " &
  "  loop min 10 max 100 (      " &
  "    (ISC_NOOP      wait 1.0e-3 1:1:OST, 2:0*0) ) " &
  "  (ISC_DISCHARGE    wait 1.0e-2) " &
  "Repeat 65535        " &
  "  (ISC_PROGRAM      16:$addr+1,16:?      wait 1.0e-2) " &
  "  loop min 10 max 100 (      " &
  "    (ISC_NOOP      wait 1.0e-3 1:1:OST, 2:0*0) ) " &
  "  (ISC_DISCHARGE    wait 1.0e-2) ";
```

In this example, the ISC_NOOP instruction points to the ISC_Default register, which has access to the status register. The program operation is started outside of the loop operation, and the ISC_NOOP instruction is used to evaluate the operation success indication. Once the operation is successful, the program operation is terminated by executing the ISC_DISCHARGE instruction. There are minimum and maximum values given for the loop operation that will give dynamic concurrentization algorithms a greater possibility to achieve concurrency and improved test times.

Example 14: Program of a device using multiple Repeat blocks. This flow describes the programming algorithm for a device that may have two separate memory spaces to be programmed and that share a common address and data register. One of the memory spaces will be significantly smaller than the other, which will allow for a reduced data file size.

```
attribute ISC_Flow of Example14:entity is
"program(Array) " &
"Repeat 4096" &
```

```

"      (ISC_PROGRAM 128:0 wait 1.0e-2) " &
"Repeat 1024" &
"      (ISC_PROGRAM 96:0, 32:? Wait 1.0e-2) ";

```

A second reason for having multiple **Repeat** blocks inside of a flow is the ability to generate a single CRC when working with multiple memory spaces. If the same memory spaces were handled in separate flows, but in the same procedure, there would be a CRC generated for each flow but not for the procedure as a whole as CRC generation is flow based rather than procedure or action based.

8.5.8 Attribute ISC_Procedure

Attribute ISC_Procedure defines the Procedures that can be commanded in terms of flows defined by the device designer. (Attribute ISC_Action, discussed in 8.5.9, defines commonly used sequences of procedures.) Application software reads these attributes to construct a user interface to control device programming procedures.

8.5.8.1 Specification

Syntax

```

<ISC procedure description> ::= attribute ISC_Procedure of <component name> : entity is
    <procedure string> ;
<procedure string> ::= “ <procedure list> ”
<procedure list> ::= <procedure> { , <procedure> }
<procedure> ::= <procedure name> [ ( <data name> ) ] = ( <flow descriptor list> )
<flow descriptor list> ::= <flow descriptor> { , <flow descriptor> }
<procedure name> ::= <VHDL identifier>

```

NOTE—The definition of <data name> was given in 8.5.7.1.

Rules and semantic checks:

- a) The syntax for the mandatory ISC_Procedure attribute shall be that shown above.
- b) The value of each <procedure name> shall be chosen from the list given below in 8.5.8.2.
- c) The <procedure list> shall contain **proc_error_exit** and **proc_program_done** <procedure> values.

NOTE—These two procedures are mandatory. They are always required by programming tools.

- d) When the value of a <procedure name> is a <VHDL identifier>, the first five characters shall not be “**proc_**”.

NOTE—This rule reserves procedure names for use by future revisions of this standard.

- e) The value of each <flow descriptor> shall appear in a <flow definition> (see 8.5.7.1).

8.5.8.2 Description

A <procedure> is a named list of <flow descriptor> elements. The <flow descriptor> elements contain <data name> elements that match <data name> elements in the data file (see Clause 9). A <flow descriptor> may appear more than once in a single <procedure>, and it may appear in multiple <procedure> elements.

To perform a procedure, the flows in that procedure are performed in order from first to last. A flow is performed unconditionally if it does not take data input. If a flow requires data input, the data are scanned for a matching <data name> element. If no data with a matching <data name> is found, an error occurs.

This standard does not address what happens when an error occurs. Not all hardware detects errors or stops applying vectors immediately when an error is detected. It is possible that some or all flows in an operation will be performed even after an error is detected.

Predefined <procedure name> elements have the meanings listed below. Predefined <procedure name> elements all begin with “**proc_**”.

proc_read	Read the device’s memory arrays and output the values, for example, to a file.
proc_verify	Verify the device’s memory arrays against user-supplied data.
proc_program	Program the device’s memory arrays with user-supplied data.
proc_erase	Bulk-erase the device.
proc_blank_check	Compare the device against its blank state.
proc_enable	Enter ISC mode. The minimal content of this procedure is the ISC_ENABLE instruction.
proc_disable	Exit ISC mode. The minimal content of this procedure is the ISC_DISABLE instruction.
proc_preload	Preload the boundary-scan register using the PRELOAD instruction. Used for devices with CLAMP-like I/O pin behavior during programming. This procedure may be executed before proc_enable . By referencing the <standard data name> “Preload,” an interface can determine it is to obtain state information for the boundary-scan register that is application dependent.

NOTE—Some test tools may assign preload values independently of the programming operation, and for this reason, the preload function should be designated as either optional or recommended.

proc_error_exit	This mandatory procedure may be performed by software at run-time when an action is to be aborted (for example, because the user pressed a “break” key) or some error condition is sensed. The minimal content of this procedure is the ISC_DISABLE instruction, but ISC_DISCHARGE or other instructions may be placed here. This procedure is used to terminate programming without the ISC_Done bit being set.
proc_program_done	An action calls this mandatory procedure to set the ISC_Done bit.

<VHDL identifier> Perform some procedure on the device not listed above. Note that this procedure cannot be localized³ and will not be handled in a standard way on user interfaces. Procedure identifiers should only be used for device-specific functions.

There is a <data name> associated with a <procedure name> as well as with <flow name> elements. Those associated with <procedure name> elements are called abstract while those associated with <flow name> elements are called concrete. At the procedure level, the architectural details of a device can be hidden.

Examples

An ISC_Procedure attribute for a device containing two different arrays. The data names used (when needed) on the procedures are abstract, but the underlying architecture represented by the flows shows the detail for the two arrays.

```
attribute ISC_Procedure of Device1:entity is
  "proc_preload = (Preload_flow), "          -- Preload BReg
```

³The term “localize” refers to specific customization of a tool to use the language (for example, French) of the expected users.

```

"proc_read(Array) = (read_flow(Array1),
                    read_flow(Array2)), "& --read two different arrays
"proc_verify(Array) = (verify_flow(Array1),
                    verify_flow(Array2)), "& -- verify two arrays
"proc_program(Array) = (program_flow(Array1),
                    program_flow(Array2)), "& -- program two arrays
"proc_program_done = (program_done_flow), " & -- program done bit
"proc_erase = (erase_flow) " & -- bulk erase
"proc_blank_check = (blank_check_flow1,
                    blank_check_flow2), "& -- check two arrays
"proc_program(Security) = (program_flow(Security)), "& -- set security
"proc_enable = (Enable_flow), " & -- Turn on ISC mode
"proc_disable = (Disable_flow), " & -- Turn off ISC mode
"proc_error_exit = (Disable_flow) ";

```

If the architecture of the device is simple, then the abstract and concrete data names may be the same, as in this example:

```

attribute ISC_Procedure of Device2:entity is
"proc_read(Array) = (read_flow(Array)), "& --read one array
"proc_verify(Array) = (verify_flow(Array)), "& -- verify one array
"proc_program(Array) = (program_flow(Array)), "&-- program one array
"proc_program_done = (program_done_flow), " & -- program done bit
"proc_erase = (erase_flow) " & -- bulk erase
"proc_blank_check = (blank_check_flow) "& -- check two arrays
"proc_preload(Preload) = (preload_flow(Preload)), "--read preload pat-
tern
"proc_enable = (Enable_flow) " & -- Turn on ISC mode
"proc_disable = (Disable_flow) " & -- Turn off ISC mode
"proc_error_exit = (Disable_flow) ";

```

8.5.9 Attribute ISC_Action

A user commands a device programming tool to “program the device,” but has in mind a sequence of procedures like “erase, blank check, program and verify.” Both the action (“program the device”) and the component procedures (“erase, blank check, program and verify”) are entities understood by the user. These procedure sequences and component procedures use keywords with standardized meanings so that user interfaces are consistent for all compliant devices, and so that user interfaces can be localized to display prompts in the user’s natural language without requiring customization for each device.

The exact sequence of procedures required to perform a high-level function varies both by device type (one device may require erase before program, while another does not) and by the user’s situation (one user knows their device is factory-fresh and already erased, while another cannot make that assumption). The procedures that make up an action can thus be marked as mandatory, recommended, or optional. A sophisticated user interface may allow customization of the action by including or excluding component procedures.

8.5.9.1 Specification

Syntax

```

<ISC action description> ::= attribute ISC_Action of <component name> : entity is <action string> ;
<action string> ::= “ <action list> ”
<action list> ::= <action> { , <action> }

```

```

<action> ::= <action name> [ ( <data name> ) ] [ <exclusivity> ] =
    ( <action specification> { , <action specification> } )
<action name> ::= <VHDL identifier>
<action specification> ::= <procedure name> [ ( <data name> ) ] [ <exclusivity> ] [ <option specification> ]
<option specification> ::= optional | recommended
<exclusivity> ::= proprietary

```

NOTE—The definition of <data name> was given in 8.5.7.1.

Rules and semantic checks:

- The syntax for the mandatory ISC_Action attribute shall be that shown above.
- The value of each <action name> shall be chosen from the list given below in 8.5.9.2.
- The value of each <procedure name> shall appear in the <procedure description> (see 8.5.8.1).
- The value of each <data name> associated with a <procedure name> shall appear in the <procedure description> (see 8.5.8.1).

8.5.9.2 Description

Attribute ISC_Action contains a list of <action> elements. Each <action> contains an <action name> and a list of <action specification> elements. An <action specification> gives a <procedure name> and optional <data name> element, along with an optional <option specification> that tells if the procedure's execution is optional or recommended. Procedures are not available to the user for execution except through <action> elements. The <action> elements correspond to major choices (buttons, menu picks, command line arguments) on a user interface.

Predefined <action name> elements can be localized to the users' natural language in the user interface. Predefined <action name> elements are:

read	Read a memory region of a device. The optional <data name> may specify data arrays, icode, usercode, and security bits. Produce a checksum of the read data per the CRC tags found in the flows.
verify	Verify a memory region of a device's memory arrays, icode, usercodes, or security bits against user-supplied data.
program	Program a memory region of the device. This action does all the things needed to install data in a device, whether it is previously programmed or not. Typically, the device is bulk-erased, blank-checked, programmed, and verified.
erase	Bulk-erase the device or erase a region identified by <data name>.
blank_check	Compare the device against its blank state.

In addition to predefined <action name> elements, the following is allowed.

<u><VHDL identifier></u>	Perform some operation on the device. Note that this operation cannot be localized and will not be handled in a standard way on user interfaces. User interfaces are not required to display these options at all. Identifiers should only be used for device-specific functions.
--------------------------------	---

Each <action specification> is a procedure name, an <exclusivity> flag and possible <option specification>. All nonproprietary predefined actions given in this standard may be executed concurrently in more than one device. An <action name> specified by a <VHDL identifier> (that is, nonpredefined) may have restrictions on its use that are communicated by an <exclusivity> keyword if it exists. If the keyword **proprietary** is

given, then the action is *not* to be considered when optimization is performed (see 3.1.44). Further, any proprietary action or procedure is to be executed by itself, meaning other devices in the chain should be executing ISC_NOOP or an IEEE Std 1149.1-2001 instruction. If the keyword is omitted, then the action may be used with full freedom to be optimized concurrently with other actions. Within an action, an individual proprietary procedure may also be similarly tagged.

NOTES

1—Nonpredefined actions may be utilized by general tools, or proprietary tools and routines supplied by the vendor of the device containing them, who defines the applicability of their behavior and use. If the action is not labeled proprietary, then it is fully optimizable.

2—Nonpredefined actions may be used to implement new or experimental capabilities for ISC devices not anticipated by this standard.

3—Nonpredefined actions may be used to “grandfather” legacy programming capabilities of devices invented before this standard was approved. These can be documented for use by that vendor’s proprietary tools.

Action specification <option specification> elements offer control over procedures:

- When no option is specified, the procedure is always performed. It may optionally be displayed on a user interface to inform the user what procedures are performed by the selected action.
- The option **optional** means the procedure need not generally be performed. A user interface that gives a user selection control of the procedures making up an action will disable this procedure by default. A user interface that does not give control of the procedures making up an action does not perform this procedure.
- The option **recommended** means the procedure need not be performed, but is generally needed. A user interface that gives a user selection control of the procedures making up a sequence enables this procedure by default. A user interface that does not give control of the procedures making up a sequence performs this procedure.

The **optional** and **recommended** options can be overridden by a mechanism found in the data file (see 9.1). This allows someone who is preparing the downloadable logic for an individual device to control the choice of options by a different person preparing a programming process. For example, the first person may want to enforce the programming of security bits that the device vendor’s action description has described as “optional.”

Example

```
attribute ISC_Action of Device1:entity is
  "erase = (proc_preload(Preload) optional, proc_enable,
            proc_erase,"&
            proc_blank_check optional, " &
            proc_disable ), " &
  "program = (proc_preload(Preload) optional, proc_enable, " &
            proc_erase recommended, " &
            proc_blank_check recommended, " &
            proc_program(array),"&
            "      proc_program(usercode) recommended,"&
            "      proc_verify(array), proc_verify(usercode) recommended, " &
            "      proc_program(security) optional, " &
            "      proc_verify(security) optional, " &
            "      proc_program_done, proc_disable ), " &
  "verify = (proc_preload(Preload) optional, proc_enable, " &
            "      proc_verify, proc_verify(usercode) recommended, " &
```

```

"      proc_verify(security) recommended, proc_disable), " &
"read = (proc_preload(Preload) optional, proc_enable, " &
"      proc_read(idcode), proc_read(security), " &
"      proc_read(array), proc_read(usercode), proc_disable )," &
"special_read proprietary = (proc_enable, read_action proprietary, "
"      proc_disable), " &
"program(security) = (proc_preload(Preload) optional, proc_enable, " &
"      proc_program(security), proc_disable)";

```

In the above example, a general programming action (with no <data name>) is given as well as a programming action that singles out the security bits. Note that a proprietary action is shown that, if selected, would not be executed concurrently with actions in other devices, nor would any optimization of instruction or data shifting be performed within this action.

8.5.10 Attribute **ISC_Illegal_Exit** (Optional)

Permissions of 5.2 and related notes and 4.2 describe the situation where practical limitations in the design of a programmable logic device prevent the return from a non-ISC instruction to the middle of a programming sequence. The optional **ISC_Illegal_Exit** attribute describes any instruction that, as a side effect of its execution, clears the **ISC_Enabled** signal.

8.5.10.1 Specification

Syntax

```

<ISCillegalexitdescription> ::= attribute ISC_Illegal_Exit of <componentname> : entity is <exit string>;
<exit string> ::= " <exit instruction list> "
<exit instruction list> ::= <exit instruction> { , <exit instruction> }
<exit instruction> ::= <instruction name>

```

Rules and semantic checks:

- The syntax for the optional **ISC_Illegal_Exit** attribute shall be that shown above.
- The value of <exit instruction> shall not be any ISC instruction defined by this standard.
- The value of <exit instruction> shall exist in the <instruction opcode stmt> that appears earlier in the BSDL description.

NOTE—See “Instruction Register Description” in IEEE Std 1149.1-2001 for the <instruction opcode stmt> definition.

8.5.10.2 Description

An <exit instruction> is any instruction name defined by IEEE Std 1149.1-2001, or any user-defined instruction name.

Example

```

attribute ISC_Illegal_Exit of PLD1:entity is
    "EXTEST, INTEST, CLAMP, My_Bist_Instruction";

```

8.5.11 Attribute **ISC_Design_Warning** (Optional)

An optional **ISC_Design_Warning** attribute may be used, in a manner identical to the **Design_Warning** attribute of BSDL, to alert users to special circumstances that may exist in the ISC implementation of a given ISC device.

8.5.11.1 Specification

Syntax

<ISC design warning> ::= **attribute** **ISC_Design_Warning** of <component name> **:entity is** <string> ;

Rules

- a) The syntax for the optional `ISC_Design_Warning` attribute shall be that shown above.

8.5.11.2 Description

An <ISC design warning> attribute associates a <string> with an ISC device description. The text of this string has no syntax, but rather contains a human-readable message. This text allows the designer of the ISC device a means for communicating with users about special ISC features or options that have been implemented in the device. In particular, any option that must be used with care or in specifically limited ways should be described by the designer. ISC tools that consume BSDL are expected to display <string> to users at appropriate times during their use.

Example

```
attribute ISC_Design_Warning of PLD1:entity is
  "This device has a limited lifetime with respect to the " &
  "number of programming/erase cycles it will reliably " &
  "support. This lifetime may be reduced by raising the " &
  "Vdd voltage by 10 percent during erasure which also " &
  "significantly decreases erasure time. See Application " &
  "Note B-103 for more information on how lifetime tradeoffs" &
  "versus programming efficiency may be made";
```

NOTES

1—A BSDL `Design_Warning` attribute may be used to convey specific designer knowledge to users concerning the implementation of IEEE Std 1149.1-2001 features.

2—Tools that ignore the ISC features of a device (described by this BSDL Extension capability) are not expected to recognize the `ISC_Design_Warning` attribute or display the associated text.

9. Data file format

Data to be programmed into a programmable logic device must be generated by software provided by the device vendor. This data can be organized to correspond to a programming flow for a single device. Application software that delivers data to one or more devices in a chain can, by examining the BSDL files for each device and by reading the matching data files containing mission data, produce a download stream that will concurrently program those devices in the chain. (See Figure 35.)

The key is for the data file format to match the structure of a flow identified in an `ISC_Flow` attribute, and for every instruction requiring unique data in that flow to have a corresponding data item in the data file.

9.1 Data file syntax specification

The data file format uses ASCII characters to express data and to divide the file into sections. It is therefore less compact than a binary format but provides for human-readability and editability, and avoids data

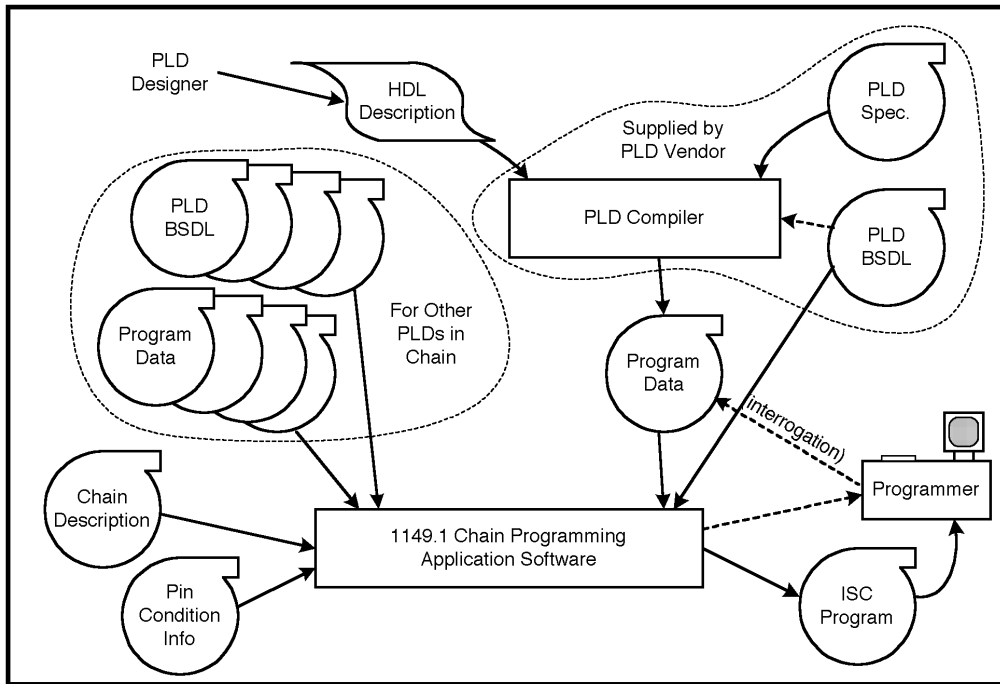


Figure 35—Program development process for a PLD that will be concurrently programmed with other PLDs

representation incompatibilities. Comments are allowed as any text preceded by two dashes "--" through the end of a line.

A data file contains labels and describes programming data to be matched with flows from the ISC_Flow attribute. Several properties of flows reduce the data file size, when compared to a serial vector file. A flow can contain operators that generate the addresses, reducing the need to store addresses in the data file. A flow can contain constant data and masks. A flow has means to reuse recently read data. Thus, a data file can be reduced to only expressing data that represent the unique information required for programming the device.

The byte stream oriented appearance of data in the data file is also amenable to block oriented commercial compression tools such as Lempel/Ziv/Waite stream compression. Thus, although the formal representation of data is in ASCII, storage and transmission of data can approach the efficiency of a binary expression.

Syntax

```

<data file> ::= IEEE_1532_Data ( <header record>[<override list>]<data record list> ) [<transmission CRC>]
<header record> ::= header ( std_version <version name>
                                [ creation_date <date string> ]
                                [ creator <string> ]
                                [ entity <entity name list> ] )
<version name> ::= STD_1532_2001 | STD_1532_2002
<date string> ::= <yyyymmdd.hhmm>
<entity name list> ::= <entity name> { , <entity name> }
<entity name> ::= <VHDL identifier>

<override list> ::= <override specification> [ , <override specification> ]

```

<override specification> ::= **override** <procedure name> [(<data name>)] <action name> <option override>
 <option override> ::= **on** | **off** | <option specification>
 <data record list> ::= <data record> { , <data record> }
 <data record> ::= [<processing flag>] **data** <data name> (
 [**initialize** (<data block>)]
 { **repeat** (<data block>) }
 [**terminate** (<data block>)]
 [**data_CRC** (<hex string>)])
 <data block> ::= <hex string list> { , <hex string list> }
 <hex string list> ::= <hex string> { <hex string> }
 <processing flag> ::= **unprocessed**
 <transmission CRC> ::= <hex string>

NOTE—The definition of <yyyymmdd.hhmm> is given in rule b) below. The definition of <data name> is given in 8.5.7.1. The definition of <procedure name> is given in 8.5.8.1. The definition of <option specification> is given in 8.5.9.1.

Rules and semantic checks:

- a) The syntax for the data file shall be that shown above.
- b) The value of <date string> shall be a contiguous numeric string containing (from right to left) a two-digit minute, a two-digit hour, a decimal point, a two-digit day, a two-digit month, and a four-digit year, where all digit fields are zero-padded on the left (known as <yyyymmdd.hhmm> format).
- c) The <version name> shall be **STD_1532_2001** or **STD_1532_2002** only.

NOTE—Subsequent editions of this standard may add new values for <version name>. This can provide a way for tools to recognize older data files versus newer files that may contain new syntax.

- d) The value of <entity name> shall be the same as that found in a matching BSDL description for the device.
- e) In any <override specification>, the values of <procedure name>, <data name> (if it is given), and <action name> shall exist in the same <action> found in the <ISC action description> for the device. Further, there shall be an <option specification> for that <procedure name>.

NOTE—This allows the creator of the data file to specify an override for an optional or recommended option, but not to override a procedure that was required by the <action>.

- f) At least one <data block> shall exist within each <data record> element.
- g) For each <hex string list> in a <data block>, the corresponding flow that references this <data block> (via <data name>) shall contain an <input specifier> for this <hex string list> that has a compatible <field width> value.

NOTE—Compatibility is defined by the discussion in 8.5.7.2 on the mapping between hex strings and binary data in registers. See also the example and text related to Figure 29.

- h) In a **repeat** <data block>, the number of <hex string list> elements in that <data block> shall be an integral multiple of the number of <input specifier> elements in the corresponding flow that references this <data block> (via <data name>), and this integer shall be less than or equal to the value of <repeat count> in the corresponding flow.

NOTE—Therefore while the amount of data in the repeat data block may be less than that needed to fill the entire array, there must be enough data to satisfy each complete repetition of a repeated <activity list> that is executed, and there must not be any data specified beyond this amount.

- i) In a <data block>, the number of <hex string list> elements in that <data block> shall be greater than zero.

9.2 Description

A <data file> begins with a <header record> that lists, at minimum, the <version name> of this standard that it conforms to. This standard lists one version, STD_1532_2002, where the “2002” suffix represents the year the standard was promulgated by the IEEE. Future revisions of this standard may add new version names, allowing software to differentiate older data files from newer files, which will be important if new types of data are later added. Optionally, a <header record> may also provide a creation date (for example, 20020219.1615 which in *yyyymmdd.hhmm* format represents February 19, 2002 at 4:15 PM) and a creator <string> (for example, “Brand-Z PLD-Writer Release v.03a, Jan 11, 2002”) that are useful for version control and support purposes. The entity name list is correlated to entity names in corresponding BSDL files and is optional. A data file may provide data for more than one packaged device when different die-bonding options are used among related devices (containing the same die) or when a technology improvement occurs that improves (for example) the burn time in a new generation of related devices.

Next is an optional <override list> that gives the creator of the data file a mechanism for constraining choices made for <option specification> elements in <action> elements described in <ISC action description> elements. An <override specification> can be used to change the <option specification> for a procedure in an action. This allows the data file creator to influence or remove the choice of procedures within actions. For example, the data file creator may wish to ensure the programming of security bits that the device vendor’s action description has described as “optional.” If a procedure in an action has not been marked **optional** or **recommended**, then no override is permitted. The keyword **on** forces the procedure to be executed. The keyword **off** forces the procedure to be omitted. Additionally, the <override specification> can be used to change the <option specification> from **optional** to **recommended** or from **recommended** to **optional**. It is expected that a graphical user interface that displays options will respond to override specifications and adjust the displayed options accordingly.

Following the optional <override list> is a <data record list> containing the data supplied by one or more <data record> elements. A <data record> has a form similar to a <flow definition> in the ISC_Flow attribute (see 8.5.7.1). A <data record> has between one and three <data block> elements, for initialize, repeat, and terminate data corresponding to those <activity> elements in the ISC_Flow attribute referencing this <data record>. An optional data CRC is used to specify the CRC that should result when the data is processed by a flow for *all flows that reference this <data record> element*.

NOTES

1—A <flow definition> containing <activity> elements in an ISC_Flow description may not require data from a data file, so a matching <data block> may not appear in a <data record>.

2—If a data CRC is found in a <data record>, then CRC generation in any flow referenced in a <data block> should encode data the same way so that a data CRC generated by all will be the same, matching the data CRC.

An optional <processing flag> (**unprocessed**) may appear in a <data record> element. This flag should be present only when two conditions are true. First, the <data record> is the result of an output operator (“!”) contained in a <flow definition> in a ISC_Flow attribute, and second, the data produced needs to be post-processed by a vendor-supplied routine before it can be read in by another <flow definition> in a ISC_Flow attribute that contains an input operator (“?”). This allows for devices where data read out into a file cannot be immediately read back in because (for example) the bit order expected by the ISC_PROGRAM instruction is different than the bit order produced by the ISC_READ instruction. The vendor of such a device that wishes to support device interrogation and reprogramming would have to supply a data reorganization routine. The <processing flag> provides tools with a flag to signal this condition.

NOTE—Devices that require post-processing on interrogated data may suffer from incomplete support from tools not also supplied by the same vendor. Therefore, it is recommended that the internal architecture of devices be symmetrical with respect to interrogation and reprogramming if this capability is supported.

The <data block> elements are composed of one or more <hex string list> elements, separated by commas. Each <hex string list> is composed of one or more <hex string> elements separated by white space. White space is used to break up a long <hex string> into smaller <hex string> elements. However, the total number of hex digits in a <hex string list> (as if all white space has been removed) shall be commensurate with the total <field width> of the flow activity that reads the hex data. Specifically, if the number of digits is less than required by the field width, then most-significant zero-padding is assumed. It is an error if the number of digits is greater than required by the field width and the higher order bits beyond the field width are nonzero. The next subclause (9.3) describes how these hex data are mapped onto data expressions in a flow.

Finally, an optional <transmission CRC> may appear (not to be confused with data CRC calculation given in 8.5.7.6). It must be four hexadecimal characters at the end of the file. It is calculated by the process given in 9.5. The <transmission CRC> can be used to determine if any changes have occurred to a data file since it was created.

Example

```
IEEE_1532_Data
(Header      (STD_Version STD_1532_2001
              Creation_Date 20021224.2359
              Creator "Build-Bits Version 1.2 20020919"
              Entity CPLD_920
            )
  Override proc_erase program OFF, -- Skip erasure when programming
  Override proc_program(security) program ON -- Ensure security is set

  Data Array (Initialize(5F6) -- most significant zeros omitted
             Repeat      (1EE07988, A0005432, 10985412, 21985421,
                          77400002, 22230045, A00C00DE)
             ) -- No data CRC
) -- end of data, no transmission CRC
```

9.3 Mapping data onto a flow

Every occurrence of the “?” operator in a flow must be matched with a <hex string list> in a <data file> with a matching <flow name> and <activity> elements. Consider the portion of a flow shown below:

```
attribute ISC_Flow of PLD3:entity is
" ..... " -- other flows deleted for brevity
"Program(Array) " &
"Initialize " &
" (ISC_ADDRESS_SHIFT 96:addr=1 wait TCK 1) " &
" (ISC_PROGRAM 32:? wait 1.0e-2 min) " &
"Repeat 7" &
" (ISC_ADDRESS_SHIFT 96:addr<<1 wait TCK 1) " &
" (ISC_PROGRAM 32:? wait 1.0e-2 min) ";
```

Then the matching data record could look like this:

```
..... -- header record deleted for brevity
Data Array(
```

```

Initialize (556)
Repeat    (1EE07988, A0005432, 10985412, 21985421,
          22230045, A00C00DE, ABD19901)),
.....    -- other data records deleted for brevity

```

The initialization <action> of the flow for Program(Array) contains a “32:?” field requesting a single 32-bit word (eight hex digits) from the <data file> used to initialize the repetition that programs data. The <data file> provides “556,” which is equivalent to “00000556.” The repeat <action> reads data from the repeat portion for Program(Array) until all the data are consumed. In all, eight 32-bit words are programmed. The “32:?” field is exactly satisfied by a <hex string> containing eight hexadecimal characters, although leading zero digits can be omitted (as in the initialize portion). Note that all address information is provided algorithmically by the flow itself and no data file space is needed to describe addressing.

NOTE—The ordering of bits is described in 8.5.7.2 and shown in Figure 29.

Consider the following activity from ISC_Flow syntax:

```

" (ISC_PROGRAM 3:sel?,96:$addr=1,200:?,2:0 wait 1.0e-3 299:0*0, 2:2) "&
" (ISC_PROGRAM 3:sel,96:$addr<1,200:?,2:0 wait 1.0e-3 299:0*0, 2:2) "&

```

This activity contains three “?” operators, so three data items are required:

```

(2, A004598760023012203377AA0BC0DDFFF00121100003A90009,
  7AA0BC0DDFFF00121100003A90009A00459876002301220337)

```

This creates the question of how to represent hexadecimal data when it is very long. For example, if a field specification specifies “2048:?” this would necessitate a <hex string list> in the <data file> with 512 characters. This in turn could cause problems with software that has limits on file read buffer sizes and would be difficult to view and edit. There are two solutions. The first choice is to break the <hex string list> into a set of blank-separated <hex string> elements. The second choice could be to break the field specification into smaller input fields. For example, change the “2048:?” field to a number of smaller fields with an equivalent total length, such as “256:?, 256:?, 256:?, 256:?, 256:?, 256:?, 256:?, 256:?” that call for the matching data to contain eight 64-character <hex string lists> that can be placed on multiple lines in the <data file>. In either case, it is recommended that the <hex string> elements that make up <hex string list> elements be limited to common text window sizes, such as 72–80 characters maximum. In the example above, using blank-delimited <hex string> elements the data could be written as:

```

(2, A004598760023012 203377AA0BC0DDFF
  F00121100003A900 09,
  7AA0BC0DDFFF0012 1100003A90009A00
  4598760023012203 37)

```

The commas separate the <hex string list> elements required by each input “?” operator.

9.4 Merging flows for concurrent programming

When two or more devices are to be programmed concurrently, the ISC_Flow attribute for each device must be examined to understand how to merge the required activities. Below are ISC_Flow attributes for two devices; the first device (PLD1) requires a starting address to be set up in the initialization phase, which is then auto-incremented later. The second (PLD2) does not require any address initialization but requires a discharge cycle on every program cycle. Neither device has a termination phase. Here is the ISC_Flow attribute for PLD1:

```

attribute ISC_Flow of PLD1:entity is
"flow1(Array) " &
"  Initialize " &
"    (ISC_ADDRESS_SHIFT16:0000    wait TCK 1) " &
"  Repeat 300" &
"    (ISC_PROGRAM      32:?      wait 1.0e-2) ";

```

This is the ISC_Flow attribute for PLD2:

```

attribute ISC_Flow of PLD2:entity is
"flow2(Array) " &
"  Initialize " &
"    (ISC_PROGRAM      16:$addr=0, 16:?wait 1.0e-2) " &
"    (ISC_DISCHARGE          wait 1.0e-3) " &
"  Repeat 200" &
"    (ISC_PROGRAM      16:$addr+, 16:? wait 1.0e-2) " &
"    (ISC_DISCHARGE          wait 1.0e-3) ";

```

After analysis, software for merging flows may construct new (virtual) attributes that look like this:

```

attribute ISC_Flow of PLD1:entity is
"flow1(Array) " &
"  Initialize " &
"    (ISC_ADDRESS_SHIFT16:0000          wait TCK 1) " &
"    (ISC_NOOP                          wait TCK 1) " &
"  Repeat 300" &
"    (ISC_PROGRAM      32:?      wait 1.0e-2) " &
"    (ISC_NOOP                          wait TCK 1) ";

```

and for PLD2:

```

attribute ISC_Flow of PLD2:entity is
"flow2(Array) " &
"  Initialize " &
"    (ISC_PROGRAM      16:$addr=0, 16:? wait 1.0e-2) " &
"    (ISC_DISCHARGE          wait 1.0e-3) " &
"Repeat 200" &
"    (ISC_PROGRAM      16:$addr+, 16:? wait 1.0e-2) " &
"    (ISC_DISCHARGE          wait 1.0e-3) ";

```

The virtual ISC_Flow attribute for PLD1 has been expanded to include ISC_NOOP instructions, while the attribute for PLD2 is unchanged. This lines up the instruction scans in both flows. Note that no data are specified for ISC_NOOP because no other instruction (in this case, ISC_DISCHARGE) needs data when ISC_NOOP is in place. If ISC_NOOP was paired with another instruction that required data, then a “1:0” data field could have been added to load the ISC_Default register with a zero while the other shift loaded meaningful data. Now the two algorithms can be processed concurrently. Note that a potential optimization for PLD1 possible before the merge (moving the instruction scan for ISC_PROGRAM out of the repeat block) is no longer possible. The data files for these two devices still contain the correct amount of data to satisfy each flow after the merge has been completed. There is no need to modify data files as a result of merging flow. Annex B gives other more complex examples.

In general, it is expected that merging flows will be a heuristic operation with many “correct” merges possible. General guidelines are given here.

At the ISC_Action level, inspect flows for procedures that contain IEEE Std 1149.1-2001 instructions (for example, IDCODE) that are executed either before entering or after completing programming operations. Then synchronize the entry into ISC Accessed system modal states in each device. Also, synchronize the exit into the ISC Complete system modal state as discussed next. When any device needs to be “padded” with a non-ISC operation to accomplish this, use the BYPASS instruction.

Upon entering the ISC Accessed system modal state, we can now use ISC_NOOP instructions to pad activities within flows to cause them to line up for better concurrent operation. Disparities in the number of ISC instructions to be scanned should be found and padded. Repeat blocks should be identified, and those operations with long wait times should also be noted, as synchronizing them together may give best results. A simple approach may be to simply “unroll” each flow into a linear structure now devoid of repeat blocks and then perform synchronization and ISC_NOOP padding to optimize the merged result.

Actions that are marked proprietary should be executed sequentially and one at a time, with other ISC devices either executing ISC_NOOP or IEEE Std 1149.1-2001 instructions.

When all concurrent actions are complete, each device should then enter the ISC Complete system modal state at the same time.

9.5 Data file transmission CRC calculation

The transmission CRC for a data file is a 16-bit CRC computed on all bytes in the data file up to but not including the transmission CRC itself and excluding all carriage return characters. The method for computing the CRC is explained below. Any characters, such as comments and white space located after the CRC statement, will not be included in the CRC computation.

The CRC is a 16-bit convolution code based on a generator polynomial. Transmission CRCs for data files are calculated using a generator polynomial⁴ used by the CCITT for 16-bit CRCs: $G(X) = X^{16} + X^{12} + X^5 + 1$. The C code for implementing this algorithm is shown below.

```
#define CCITT_CRC 0x8408          // bit-mask for CCITT CTC polynomial
#define CHAR_BIT 8                // number of bits per byte
unsigned short crc_register;      // global 16-bit shift register

void init_transmission_crc()
{
    crc_register = 0xFFFF;        //start with all ones in shift register
}

void compute_transmission_crc(unsigned char in_byte)
{
    int bit, feedback;
    if (in_byte != '\r')
    {
        // compute for each bit in in_byte
        for (bit = 0; bit < CHAR_BIT; bit++)
        {
            feedback = (in_byte ^ crc_register) & 0x01;    // XOR LSB
            crc_register >>= 1;    // shift the shift register
            if (feedback)
                crc_register ^= CCITT_CRC; // invert selected bits
            in_byte >>= 1;        // get the next bit of in-byte
        }
    }
}
```

The function `init_transmission_crc()` must be called first to initialize the transmission CRC shift register. Then, `compute_transmission_crc()` is called on each byte in the data file, except carriage return characters

⁴This algorithm is logically identical to that published in the JEDEC JESD-71 standard.

(ASCII 13 or 0D Hex). The characters must be processed in order, from the beginning of the file up to the transmission CRC itself (or to the end of the file if the transmission CRC is absent). Finally, `transmission_crc_value()` is called to obtain the final transmission CRC value, which is the complement of the current value of the transmission CRC shift register after all characters have been processed.

Carriage return characters are excluded from the transmission CRC calculation to allow a data file to have the same CRC when stored in the MS-DOS text file format (with CR-LF characters as line separators) or in the UNIX format (with LF character only). Since a data file is a text file, it may be stored in either format, and the CRC will be the same.

If the transmission CRC statement is found, the calculated transmission CRC value should be compared to the expected CRC value represented in the transmission CRC statement. If these values differ, the transmission CRC check fails and the data file contents may be corrupt.

Annex A

(informative)

Use of deviced features

This informative annex provides a rationale for the use of various ISC features. A major goal of this standard is to facilitate programming of compliant devices by the widest variety of means, including device programmers, ATE systems, and embedded controllers. This annex describes the ways in which device-programming software can make use of device features and BSDL extensions.

Four general programming scenarios are contemplated by the standard, as follows:

- a) A single 1532-compliant device may be programmed with a device programmer.
- b) A single 1532-compliant device may be programmed in-circuit on an assembled printed circuit board.
- c) Multiple 1532-compliant devices may be programmed concurrently in-circuit to reduce programming time.
- d) Nonfacilitated, non-1532-compliant programmable devices (for example, Flash memory devices not compliant with this standard) may be programmed in-circuit concurrently, controlled by the IEEE Std 1149.1-2001 Boundary-Scan Registers of surrounding 1149.1-compliant or 1532-compliant devices.

A.1 Programming features

Any standard-conforming device provides a minimal programming capability (see 6.2) for the first two scenarios. The minimal capability only provides a means to install a program into the device's programmable array. The minimal capability provided by the standard does not provide all the features that users may associate with programmable devices. These features are supported by instructions defined as optional by the standard (see 6.3 through 6.5). The device designer should consider implementing the following capabilities.

A.1.1 Erasure

Especially for nonvolatile programmable devices, the ISC_ERASE instruction (see 6.3.2) should be provided unless there is a requirement for a "program only once" capability. For volatile devices, the act of powering down the device may constitute an erasure capability, although this could be suboptimal behavior from a system design point of view.

A.1.2 Readback

The ISC_READ instruction (see 6.3.1) supports reading the contents of the device's programmable cell array. The ISC_READ instruction is optional in the standard because it is possible to imagine a device that was not readable by design. However, readback is a critical capability that should be implemented unless some overriding user need makes it undesirable.

The ISC_READ instruction supports verification of correct programming in many devices. Unless the programming process itself guarantees correct programming, readback is essential to verify that the values programmed into the device are retained at the end of the programming process. Readback supports

standalone verification of the device, which allows a user to validate the installed program. Readback supports device programmers that use a master device as the source of programming data.

Note that if the data read back from the device is the same as the data programmed into the device, only one copy of the programming data need be provided in the device data file.

A.1.3 Multipass programming

In multipass programming, the device is programmed in two or more stages. There are a number of variants on multipass programming. In one use, the basic program is installed in the device during subassembly, and customization data like serial numbers, IP addresses, etc., are installed during final assembly. In another use, the basic device program is installed during subassembly, and the security bits are programmed after testing. In yet another scenario, a lengthy programming operation is broken down into two or more phases to support a manufacturing line with a high beat rate.

To support multipass programming of the main cell array, the device must support setting the programming address. A very simple device that resets the programming address on entry into ISC Accessed system modal state (see Clause 4) and programs a fixed number of data words does not support multipass programming of the cell array.

To support multipass programming of the security bits, the device must either implement the optional ISC_PROGRAM_SECURITY instruction (see 6.4.3) or must implement multipass programming of the cell array, and implement the security bits in the main cell array.

Multipass programming is a commonly expected feature of memory-like devices. Multipass programming of programmable logic devices is less common. Multipass programming is an important feature when a device might contain customization data, or when programming time exceeds a few seconds, and may need to be divided among two or more programming stations.

A.1.4 Security

This standard provides a generalized security mechanism (see 5.1.11) that is much more flexible than the “read protection” common in the past. The security mechanism allows the prevention of reading, erasing, and overwriting of data, selectively. Further, these capabilities can be password protected, and if a goal of a design, the password can be encrypted within the device. One of the interesting capabilities this can yield is a device that starts out as a blank programmable device, but can then be protected from ever being read out, rewritten, or erased in the future, resembling a custom implementation. This feature may be useful for secure applications in financial, intelligence, or gambling industries, etc.

A.2 Support for concurrent programming

Concurrent programming is the synchronization of the programming commands to several devices so that their programming “burn times” start at the same time. This specification extends IEEE Std 1149.1-2001 BSDL with programming data flows for single ISC devices. These single ISC device programming flows can be modified by program tool vendors to produce chain ISC programming flows. The programming tool vendors develop the ISC programming flows by inserting ISC_NOOP instructions in the single ISC programming flows in order to idle the programming of one device when others on the chain require other operations to be performed. They must also adjust the wait time of all of the devices in the *Run-Test/Idle* TAP Controller state to the maximum programming time of all of the devices that have not yet completed programming. See 6.2.4 for information on ISC_NOOP. Examples of programming flows for concurrent programming are given in Annex B.

The purpose of concurrent programming devices in the chain is to reduce the total time to program all of the devices on the chain. The best reduction in programming time will be seen on chains where the programming times of the individual ISC devices are similar to each other. For example (in the best case where data shift time is negligible), if the five devices in the chain have the same programming time and require the same number of addresses to be programmed, the time can be reduced by 5 to 1 (reduced 80%). However, if one of the device's programming time is 10 times longer than the other four devices and all five require the same number of addresses to be programmed, the total programming time will only be reduced 14 to 10 (reduced 29%). Note that even though all of the devices are programmed in the same amount of time, they all have to remain in the *Run-Test/Idle* TAP Controller state waiting for the device with the longest programming time to complete. Likewise, with everything else being the same, if the difference in programming time was 100 to 1, then the total programming time would be reduced 104 to 100 (reduced less than 4%). In general, the larger the difference between the programming time of the devices on the chain, the less of a reduction that can be achieved by the concurrent programming technique.

A.3 Support for programming noncompliant devices

Many users of 1532-compliant programmable devices build circuits containing non-1532-compliant programmable devices, typically Flash memories. These non-1532-compliant devices accept programming commands on their address and data inputs. When these devices are connected to microprocessors or other devices that implement IEEE Std 1149.1-2001, they can be programmed via the Test Access Port.

A 1532-compliant device may have a role in programming noncompliant devices in circuit if it is any of the following:

- A microprocessor, microcontroller, DSP, or peripheral controller with embedded programmable cells.
- A programmable logic device whose design or typical use includes bus controllers, buffers, or address-select logic for microprocessors or microcontrollers.
- A PLD or FPGA whose design or typical use includes implementing microprocessors, microcontrollers, DSPs, or other stored-program machines that might read program or data information from a flash memory device.

To provide minimal support for programming non-1532-compliant devices, a 1532-compliant device should implement the CLAMP-like behavior option of rule d) of 6.1.1. This behavior allows the 1532-compliant device's Boundary-Scan Register to control pin behavior during all phases of a board's programming process. If the HIGHZ-like behavior option of this rule is implemented instead, transitions from driven states to high-impedance states might affect programming of noncompliant devices.

To support programming noncompliant devices concurrently with compliant devices, one of the following conditions must be met. The conditions are listed in order from most desirable to least desirable, as follows:

- a) The 1532-compliant device design should permit IEEE Std 1149.1-2001 pin-permission instructions (EXTEST and PRELOAD) during programming operations in the ISC Accessed system modal state (see 4.1). A device designed in this way can control the pins of a non-1532-compliant device for programming at any time.
- b) If pin-permission instructions cannot be safely performed during the ISC Accessed system modal state, but the device design uses explicit programming addresses, then the device's Boundary-Scan Register can be changed by briefly exiting the ISC Accessed system modal state. In such a device, pin-permission instructions would be bracketed by the ISC_DISABLE and ISC_ENABLE instructions (see 6.2). To be most effective, the required delay times for the device's ISC_ENABLE and ISC_DISABLE should be short, compared to the Flash device's programming burn time.
- c) If pin-permission instructions cannot be safely performed during the ISC Accessed system modal state and the device implements an implicit addressing scheme, then the "current programming

address” and any other internal registers set during the programming process should retain their values between execution of an ISC_DISABLE instruction and execution of the following ISC_ENABLE instruction. In such a device, pin-permission instructions could be bracketed by the ISC_DISABLE and ISC_ENABLE instructions. To be most effective, the required delay times for the device’s ISC_ENABLE and ISC_DISABLE should be short, compared to the Flash device’s programming burn time.

- d) If pin-permission instructions cannot be safely performed during programming operations, and the device implements implicit addressing, and there is no way to retain programming state across the ISC_DISABLE to ISC_ENABLE bracketing, the 1532-compliant device design could include all of
 - 1) Means to capture and shift out the current programming address
 - 2) Means to shift in and set the programming address
 - 3) Short delay times for ISC_ENABLE and ISC_DISABLE
 - 4) A supporting flow in the ISC_Flow BSDL attribute (see 8.5.1) to control entry and exit from the ISC Accessed system modal state

A device with HIGHZ-like pin behavior probably will not support concurrent programming of non-1532-compliant devices. A device with a long programming burn time probably will not support concurrent programming of non-1532-compliant devices unless it implements pin-permission instructions EXTEST and PRELOAD during ISC Accessed system modal state. A device that cannot support pin-permission instructions during the ISC Accessed system modal state, and that has long ISC_ENABLE and ISC_DISABLE delay times, no way to capture and update the current programming address, or that requires variable arguments to ISC_ENABLE or ISC_DISABLE will not support concurrent programming of non-1532-compliant devices.

Annex B

(informative)

ISC programming flows

This informative annex provides some examples of ISC programming flows to illustrate how ISC devices compliant with this standard may be programmed.

B.1 Programming a single, basic ISC device

The following step-by-step process summarizes the programming process for a single, basic 1532-compliant device. This device has a near-minimum set of ISC instructions, but does require the use of ISC_DISCHARGE to finish the programming process. This device implements the optional status reporting mechanism recommended in v) of 5.2, and the ISC_Done signal is a function of data loaded into memory.

Now consider each step in Programming Flow 1 in Table B.1 in detail. Subsequent flows appearing later have similar elements in their behavior.

Table B.1—Programming flow 1: A single, basic ISC device

Step	Tap activity	Instruction/target/action
1	IR-Scan	Load ISC_ENABLE
2	DR-Scan	Shift ISC_Config Reg
3	<i>Run-Test/Idle</i>	Wait for enable time
<i>Device ready for programming.</i>		
4	IR-Scan	Load ISC_PROGRAM
5	DR-Scan	Shift ISC_PData Reg
6	<i>Run-Test/Idle</i>	Wait for write burn time
7	If more data to program, return to step 5	
<i>Now continue with program readback to verify the programmed data.</i>		
8	IR-Scan	Load ISC_READ
9	DR-Scan	Shift ISC_RData reg
10	<i>Run-Test/Idle</i>	Wait for read time
11	If more data to read back, return to step 9	

Table B.1—Programming flow 1: A single, basic ISC device (continued)

Step	Tap activity	Instruction/target/action
12	DR-Scan	Shift ISC_RData reg
<i>Now program security. Note no data are required.</i>		
13	IR-Scan	Load ISC_PROGRAM_SECURITY
14	<i>Run-Test/Idle</i>	Wait for security set time
<i>Now discharge programming voltages before completion.</i>		
15	IR-Scan	Load ISC_DISCHARGE
16	<i>Run-Test/Idle</i>	Wait for discharge time
<i>Now complete the process and make the device operational.</i>		
17	IR-Scan	Load ISC_DISABLE
18	DR-Scan	Scan out status of discharge
19	<i>Run-Test/Idle</i>	Wait for disable time
20	<i>Test-Logic-Reset</i>	Device operational

- 1) In preparation for ISC programming, first place the device in its ISC Accessed system modal state. This step loads the instruction register with the ISC_ENABLE instruction.
- 2) This step loads data into the ISC_Config data register. Since no ISC instruction has been executed before this, there would be no meaningful status bits to examine if recommendation v) of 5.2 has been implemented. If there is no need to set up data in ISC_Config before proceeding to the *Run-Test/Idle* TAP Controller state, then this step could be skipped. However, if the data register scan is required, then passage from *Update-IR* (which completes step 1 above) to *Shift-DR* must not pass through the *Run-Test/Idle* TAP Controller state, per rule a) of 6.6 because this would prematurely initiate execution of ISC_ENABLE, which is to be done in step 3 next. (This rule must be observed in many places in the following examples.)
- 3) Wait in the *Run-Test/Idle* TAP Controller state for the minimum duration required by the ISC_ENABLE instruction. This takes the device to the ISC Accessed system modal state.
- 4) Load the ISC_PROGRAM instruction into the IEEE Std 1149.1-2001 instruction register.
- 5) Load the ISC_PData register with data to be burned into the device. The data may actually contain address information, or addressing may be intrinsic to the operation of ISC_PROGRAM. Note that if recommendation v) of 5.2 has been implemented the data shifted out of ISC_PData will contain the status of the just-executed instruction operation (either ISC_ENABLE or ISC_PROGRAM). If this status shows a failure, the programming process could be aborted at this point by returning to the *Test-Logic-Reset* TAP Controller state.
- 6) Wait in the *Run-Test/Idle* TAP Controller state for the minimum duration required by the ISC_PROGRAM instruction.
- 7) If there is more data to program into the device, then return to step 5. Note that the ISC_PROGRAM instruction is still in the IEEE Std 1149.1-2001 instruction register, and thus, no instruction register scan is required.
- 8) Now that all data have been programmed, it will be common to read it back out as a verification process, although if the status indication scheme may be trusted to indicate successful program-

ming, this process may be superfluous. Load the IEEE Std 1149.1-2001 instruction register with the ISC_READ instruction.

- 9) Shift out the ISC_RData register. First note that if recommendation v) of 5.2 has been implemented, the data shifted out of ISC_RData will contain the status of the just-executed instruction operation (either ISC_PROGRAM or ISC_READ). If this status shows a failure, the readback process could be aborted at this point by returning to the *Test-Logic-Reset* TAP Controller state. (A more sophisticated algorithm might record the address of the failed read for a subsequent retry at programming, if the device has some probability of programming on a retry.) Second, note that if this is the first time this step is encountered, then the ISC_RData register will not contain readback data. Subsequent visits to this step shift out the readback data. (See step 12.)
- 10) Wait in the *Run-Test/Idle* TAP Controller state for the minimum duration required by the ISC_READ instruction.
- 11) If there are more data to read from the device, then return to step 9. Note that the ISC_READ instruction is still in the IEEE Std 1149.1-2001 instruction register, and thus, no instruction register scan is required.
- 12) The just-completed interval in the *Run-Test/Idle* TAP Controller state has produced a result in the ISC_RData register that needs to be shifted out for examination. Note that any status present will represent the status of the last read operation
- 13) Assuming all readback data are good, it is time to set the security option for this device. Load the IEEE Std 1149.1-2001 instruction register with the ISC_PROGRAM_SECURITY instruction.
- 14) Wait in the *Run-Test/Idle* TAP Controller state for the minimum duration required by the ISC_PROGRAM_SECURITY instruction.
- 15) Load the IEEE Std 1149.1-2001 instruction register with the ISC_DISCHARGE instruction. No data are required for this instruction.
- 16) Wait in the *Run-Test/Idle* TAP Controller state for the minimum duration required by the ISC_DISCHARGE instruction.
- 17) Load the IEEE Std 1149.1-2001 instruction register with the ISC_DISABLE instruction.
- 18) The data shifted out of the ISC_Config register will contain the status of the just-executed ISC_DISCHARGE instruction
- 19) Wait in the *Run-Test/Idle* TAP Controller state for the minimum duration required by the ISC_DISABLE instruction. This takes the device to the ISC Complete system modal state. The ISC_Done signal is set as part of the programming process (steps 5–7 above).
- 20) The *Test-Logic-Reset* TAP Controller state is entered, causing the ISC_DISABLE instruction to be displaced from the IEEE Std 1149.1-2001 instruction register. This in turn causes the device to become operational.

B.2 Programming an identical pair of basic ISC devices concurrently

The simplest example of concurrent programming occurs when you have two identical devices in a chain containing only those devices (see B.4 for detail on handling non-ISC devices in the chain). In this case, shown in the following step-by-step process in Table B.2, each step occurs concurrently for each device. The shifting of bits is serialized, which does add time to the process, but since the various times spent in the *Run-Test/Idle* TAP Controller state are usually much larger than shifting time, the overall time increase seen for programming two devices is small compared to programming a single device. In this example, both devices A and B have implemented their discharge function into ISC_DISABLE, so they do not possess an ISC_DISCHARGE instruction.

Table B.2—Programming flow 2: Concurrent programming for an identical pair of basic ISC devices

		Device A	Device B
Step	TAP activity	Instruction/target/action	Instruction/target/action
1	IR-Scan	Load ISC_ENABLE	Load ISC_ENABLE
2	DR-Scan	Shift ISC_Config Reg	Shift ISC_Config Reg
3	<i>Run-Test/Idle</i>	Wait for enable time	
<i>Both devices ready for programming.</i>			
4	IR-Scan	Load ISC_PROGRAM	Load ISC_PROGRAM
5	DR-Scan	Shift ISC_PData Reg	Shift ISC_PData Reg
6	<i>Run-Test/Idle</i>	Wait for write burn time	
7	If more data to program, return to step 5		
<i>Now continue with program readback of both devices.</i>			
8	IR-Scan	Load ISC_READ	Load ISC_READ
9	DR-Scan	Shift ISC_RData Reg	Shift ISC_RData Reg
10	<i>Run-Test/Idle</i>	Wait for read time	
11	If more data to read back, return to step 9		
12	DR-Scan	Shift ISC_RData Reg	Shift ISC_RData Reg
<i>Now program security in both devices and set them to their operational states.</i>			
13	IR-Scan	Load ISC_PROGRAM_SECURITY	Load ISC_PROGRAM_SECURITY
14	<i>Run-Test/Idle</i>	Wait for security set time	
15	IR-Scan	Load ISC_DISABLE	Load ISC_DISABLE
16	DR-Scan	Scan out status of security	Scan out status of security
17	<i>Run-Test/Idle</i>	Wait for disable time	
18	<i>Test-Logic-Reset</i>	Device operational	Device operational

If devices A and B are identical except for burn times (for example, they are devices from the same family but a later device has efficiency improvements), then in steps 3, 6, 10, 14, and 17, it would be necessary to wait for the maximum of the two wait intervals. Since both devices implement status reporting, every data register scan to shift in data for an upcoming operation also shifts out the concatenated status results for the previous operation.

B.3 Programming two similar devices concurrently

Now consider programming two related devices, perhaps from the same family of devices, where one device is smaller than the other and requires less program data. In Table B.3, there are less repetitions needed for Device A than for Device B.

Table B.3—Programming flow 3: Programming two similar (but not identical) devices concurrently

		Smaller device A	Larger device B
Step	Tap activity	Instruction/target/action	Instruction/target/action
1	IR-Scan	Load ISC_ENABLE	Load ISC_ENABLE
2	DR-Scan	Shift ISC_Config reg	Shift ISC_Config reg
3	Run-Test/Idle	Wait for maximum enable time	
Both devices ready for programming.			

Table B.3—Programming flow 3: Programming two similar (but not identical) devices concurrently (continued)

		Smaller device A	Larger device B
Step	Tap activity	Instruction/target/action	Instruction/target/action
4	IR-Scan	Load ISC_PROGRAM	Load ISC_PROGRAM
5	DR-Scan	Shift ISC_PData Reg	Shift ISC_PData Reg
6	Run-Test/Idle	Wait for maximum write burn time	
7	If more device A data to program, return to step 5		
8	IR-Scan	Load ISC_NOOP	Reload ISC_PROGRAM
9	DR-Scan	Shift ISC_Default Reg	Shift ISC_PData Reg
10	Run-Test/Idle	Wait for device B write burn time	
11	If more device B data to program, return to step 9		
Now readback programming data from both devices.			
12	IR-Scan	Load ISC_READ	Load ISC_READ
13	DR-Scan	Shift ISC_RData Reg	Shift ISC_RData Reg
14	Run-Test/Idle	Wait for maximum read time	
15	If more device A data to read back, return to step 13		
16	DR-Scan	Shift ISC_RData Reg	Shift ISC_RData Reg
17	IR-Scan	Load ISC_NOOP	Load ISC_READ
18	DR-Scan	Shift ISC_Default Reg	Shift ISC_RData Reg
19	Run-Test/Idle	Wait for device B read time	
20	If more device B data to read back, return to step 18		
21	DR-Scan	Shift ISC_Default Reg	Shift ISC_RData Reg
Now program security and set both devices to their operational states.			
22	IR-Scan	Load ISC_PROGRAM_SECURITY	Load ISC_PROGRAM_SECURITY
23	Run-Test/Idle	Wait for maximum security set time	
24	IR-Scan	Load ISC_DISABLE	Load ISC_DISABLE
25	DR-Scan	Scan out status of security	Scan out status of security
26	Run-Test/Idle	Wait for maximum disable time	
27	Test-Logic-Reset	Device operational	

Comments on specific programming steps in Table B.3:

Steps 3, 6, 14, 23, and 26: In these steps, the maximum of the durations required by the two devices is used for the duration of these steps.

Steps 10 and 19: The duration time for device B only is used for the duration of these steps.

Steps 1–7: These steps are the same we saw for programming identical devices, except that we stop looping (step 7) when we run out of locations in device A.

Steps 8–11: These steps load device A with the ISC_NOOP instruction so that it may passively wait while device B finishes its programming. Note that the first time into step 9, the ISC_Default register will record the status of the last programming operation conducted in device A.

Steps 12–15: These steps are the same we saw for reading identical devices, except that we stop looping (step 15) when we run out of locations in device A.

Step 16: Here the last read result (and status if it exists) for both devices is shifted out.

Steps 17–20: These steps load device A with the ISC_NOOP instruction so that it may passively wait while device B finishes its reading. Note that the first time into step 18, the ISC_Default register will record the status of the last read operation conducted in device A, although this was also observed in step 16.

Step 21: Here the last read result (and status if it exists) for device B is shifted out.

Steps 22–26: Set the security feature for both devices and exit ISC Accessed system modal state.

Step 27: Both devices become operational simultaneously. Note that reloading the first one with ISC_DISABLE while loading the second with BYPASS would allow the second to become operational before the first one does.

B.4 Conditioning and programming devices in a larger chain

If you have two ISC devices in a chain coexisting with non-ISC devices, then these non-ISC devices need to be kept in a rational state and maintained in this state while programming continues. This may necessitate conditioning the I/O pins of the non-ISC devices as well as the ISC devices and maintaining that conditioning. In the example shown in Table B.4, steps 1-2 are used to set the Boundary-Scan Registers of all the devices to a state that will properly condition all affected I/O pins. Note that the non-ISC devices should reload the conditioning instruction every time a subsequent instruction register scan is done.

Table B.4—Programming flow 4: Conditioning and programming devices in a larger chain

		Smaller device A	Larger device B	Non-ISC device
Step	TAP activity	Instruction/target/action	Instruction/target/action	Instruction/action
1	IR-Scan	Load PRELOAD	Load PRELOAD	Load PRELOAD
2	DR-Scan	Load I/O clamp pattern	Load I/O clamp pattern	Load I/O clamp pattern
3	IR-Scan	Load ISC_ENABLE	Load ISC_ENABLE	Load CLAMP
4	DR-Scan	Shift ISC_Config Reg	Shift ISC_Config Reg	Shift bypass
5	<i>Run-Test/Idle</i>	Wait for maximum enable time		
6	IR-Scan	Load ISC_PROGRAM	Load ISC_PROGRAM	Load CLAMP
7	DR-Scan	Shift ISC_PData reg	Shift ISC_PData reg	Shift bypass
8	<i>Run-Test/Idle</i>	Wait for maximum write burn time		
etc.				

NOTES

1—The HIGHZ instruction may also be available in the non-ISC devices. If not, then EXTEST may be used to simulate the CLAMP or HIGHZ instruction. However, the Boundary-Scan Register in such devices should be maintained during each data register scan.

2—It is possible that a mixture of CLAMP and HIGHZ behavior may need to be supported in various devices while programming is underway.

Proper I/O conditioning is a matter of concern in the design of a system that will undergo ISC programming. ISC devices have the ability to support I/O conditioning (see 4.4), and this may need to be coordinated with

non-ISC devices that may also exist on the board. These non-ISC devices may not be in rational states because the functionality that keeps them rational may not yet exist in the programmable devices with which they interact.

B.5 Programming three differing devices concurrently

Next, consider a more complex example. Here, three devices with the following characteristics are to be programmed concurrently:

Device 1: Address and data are always presented sequentially, using ISC_ADDRESS_SHIFT for the address and ISC_PROGRAM or ISC_READ for the data. This device also requires the fewest program or read cycles. This device supports ISC_PROGRAM_SECURITY and programs its ISC_Done signal with ISC_PROGRAM_DONE.

Device 2: Once the address has been set up (using ISC_ADDRESS_SHIFT), the device increments future addresses automatically (auto-increment). This device also requires the most program or read cycles. This device supports ISC_PROGRAM_SECURITY. The ISC_Done signal is a function of bits in the program memory.

Device 3: The ISC_PData and ISC_RData registers combine address and data together as two fields within the same register. This device merges its security function within the programming bits. Therefore, this device does not have an ISC_PROGRAM_SECURITY instruction. The ISC_Done signal is a function of bits in the program memory.

The ISC_NOOP instruction will be needed to keep the three devices properly phased so that the various burn times will line up. In the example in Table B.5, there is also an erasure preamble on the assumption that the devices are already programmed. All devices implement a simple bulk erase, including bits that form the ISC_Done signal.

Table B.5—Programming flow 5: Programming three differing devices concurrently

		Device type 1	Device type 2	Device type 3
Step	TAP activity	Instruction/target/action	Instruction/target/action	Instruction/action
1	IR-Scan	Load ISC_ENABLE	Load ISC_ENABLE	Load ISC_ENABLE
2	DR-Scan	Shift ISC_Config Reg	Shift ISC_Config Reg	Shift ISC_Config Reg
3	Run-Test/Idle	Wait for maximum enable time		
Now begin bulk erasure of all three devices.				
4	IR-Scan	Load ISC_ERASE	Load ISC_ERASE	Load ISC_ERASE
5	Run-Test/Idle	Wait for maximum erase time		
Devices ready for programming.				
6	IR-Scan	Load ISC_ADDRESS_SHIFT	Load ISC_ADDRESS_SHIFT	Load ISC_NOOP
7	DR-Scan	Shift ISC_Address reg	Shift ISC_Address reg	Shift ISC_Default reg
8	Run-Test/Idle	Wait for maximum address time		
9	IR-Scan	Load ISC_PROGRAM	Load ISC_PROGRAM	Load ISC_PROGRAM
10	DR-Scan	Shift ISC_PData reg	Shift ISC_PData reg	Shift ISC_PData reg
11	Run-Test/Idle	Wait for maximum program time		
12	IR-Scan	Load ISC_ADDRESS_SHIFT	Load ISC_NOOP	Load ISC_NOOP
13	DR-Scan	Shift ISC_Address reg	Shift ISC_Default reg	Shift ISC_Default reg
14	Run-Test/Idle	Wait for maximum address time		
15		If more device type 1 data to program, return to step 9		

Table B.5—Programming flow 5: Programming three differing devices concurrently (continued)

		Device type 1	Device type 2	Device type 3
Step	TAP activity	Instruction/target/action	Instruction/target/action	Instruction/action
16	IR-Scan	Load ISC_NOOP	Load ISC_PROGRAM	Load ISC_PROGRAM
17	DR-Scan	Shift ISC_Default reg	Shift ISC_PData reg	Shift ISC_PData reg
18	Run-Test/Idle	Wait for maximum program time (device types 2 and 3 only)		
19		If more device type 3 data to program, return to step 17		
20	IR-Scan	Load ISC_NOOP	Load ISC_PROGRAM	Load ISC_NOOP
21	DR-Scan	Shift ISC_Default reg	Shift ISC_PData reg	Shift ISC_Default reg
22	Run-Test/Idle	Wait for program time for device type 2		
23		If more device type 2 data to program, return to step 21		
Now continue with program readback of all three devices.				
24	IR-Scan	Load ISC_ADDRESS_SHIFT	Load ISC_ADDRESS_SHIFT	Load ISC_NOOP
25	DR-Scan	Shift ISC_Address reg	Shift ISC_Address reg	Shift ISC_Default reg
26	Run-Test/Idle	Wait for maximum address time		
27	IR-Scan	Load ISC_READ	Load ISC_READ	Load ISC_READ
28	DR-Scan	Shift ISC_RData reg	Shift ISC_RData reg	Shift ISC_RData reg
29	Run-Test/Idle	Wait for maximum read time		
30	IR-Scan	Load ISC_ADDRESS_SHIFT	Load ISC_NOOP	Load ISC_NOOP
31	DR-Scan	Shift ISC_Address reg	Shift ISC_Default reg	Shift ISC_Default reg
32	Run-Test/Idle	Wait for maximum address time		
33	IR-Scan	Load ISC_READ	Load ISC_READ	Load ISC_READ
34	DR-Scan	Shift ISC_RData reg	Shift ISC_RData reg	Shift ISC_RData reg
35	Run-Test/Idle	Wait for maximum read time		
36		If more device type 1 data to read, return to step 30		
37	DR-Scan	Shift ISC_RData reg	Shift ISC_RData reg	Shift ISC_RData reg
38	IR-Scan	Load ISC_NOOP	Load ISC_READ	Load ISC_READ
39	DR-Scan	Shift ISC_Default reg	Shift ISC_RData reg	Shift ISC_RData reg
40	Run-Test/Idle	Wait for maximum read time (device types 2 and 3 only)		
41		If more device type 3 data to read, return to step 39		
42	DR-Scan	Shift ISC_Default reg	Shift ISC_RData reg	Shift ISC_RData reg
43	IR-Scan	Load ISC_NOOP	Load ISC_READ	Load ISC_NOOP
44	DR-Scan	Shift ISC_Default reg	Shift ISC_RData reg	Shift ISC_Default reg
45	Run-Test/Idle	Wait for maximum read time (device type 2 only)		
46		If more device type 2 data to read, return to step 44		
47	DR-Scan	Shift ISC_Default reg	Shift ISC_RData reg	Shift ISC_Default reg
The ISC_Done signal must be explicitly set in device 1. In devices 2 and 3, it is a function of program bits already installed.				
48	IR-Scan	Load ISC_PROGRAM_DONE	Load ISC_NOOP	Load ISC_NOOP
49	DR-Scan	Shift ISC_Default reg	Shift ISC_Default reg	Shift ISC_Default reg
50	Run-Test/Idle	Wait for duration of ISC_PROGRAM_DONE in device 1		
Now program security in all devices. Device 3 uses ISC_PROGRAM to write security bits in a specific address.				
51	IR-Scan	Load ISC_PROGRAM_SECURITY	Load ISC_PROGRAM_SECURITY	Load ISC_PROGRAM
52	DR-Scan	(Don't care data)	(Don't care data)	Shift ISC_PData address/data
53	Run-Test/Idle	Wait for maximum security time		
Now finish up by disabling the devices and sequence them individually to their operational states.				
54	IR-Scan	Load ISC_DISABLE	Load ISC_DISABLE	Load ISC_DISABLE

Table B.5—Programming flow 5: Programming three differing devices concurrently (continued)

		Device type 1	Device type 2	Device type 3
Step	TAP activity	Instruction/target/action	Instruction/target/action	Instruction/action
55	DR-Scan	Scan out status of security	Scan out status of security	Scan out status of security
56	<i>Run-Test/Idle</i>	Wait for maximum disable time		
57	IR-Scan	Load BYPASS	Load ISC_DISABLE	Load ISC_DISABLE
58	IR-Scan	Load BYPASS	Load BYPASS	Load ISC_DISABLE
59	IR-Scan	Load BYPASS	Load BYPASS	Load BYPASS

Comments on specific programming steps in Table B.5:

Steps 1–5: Enable and erase the three devices. Note that all devices support bulk erase, so no sector information needs to be shifted in.

Steps 6–8: Here Device Types 1 and 2 require an address setup. Device Type 3 does not because the data scanned into it will contain an address field. Note that Device Type 2 will automatically increment its address, so this is the only time (for programming) that the address needs to be set.

Steps 9–15: The repetition of these steps programs all three devices until Device Type 1 is filled.

Step 16: This step removes Device Type 1 from any further programming.

Steps 17–19: The repetition of these steps programs Device Types 2 and 3 until Device Type 3 is filled.

Step 20: This step removes Device Type 3 from any further programming.

Steps 21–23: The repetition of these steps programs the remaining locations in Device Type 2.

Steps 24–26: Now prepare addressing in Device Types 1 and 2 for readback.

Steps 27–29: First data shift is needed to send address info into Device Type 3. All device ISC_RData registers contain unknown data at this point.

Steps 30–36: The repetition of these steps reads data from each device until Device Type 1 is completely read. Note that step 34 scans out readback data from the previous ISC_READ operation.

Step 37: Scan last data from all devices to complete the reading of Device Type 1.

Step 38: Set up to read remaining data from Device Types 2 and 3.

Steps 39–41: The repetition of these steps completes the reading of Device Type 3. Note that the repeat no longer needs to update address fields.

Step 42: Scan last data from Device Types 2 and 3 to complete the reading of Device Type 3.

Step 43: Set up to read remaining data from Device Type 2.

Steps 44–46: The repetition of these steps completes the reading of Device Type 2.

Step 47: Scan last data from Device Type 2.

Steps 48–50: Set the ISC_Done signal in Device Type 1. It uses the ISC_PROGRAM_DONE instruction for this. Device Types 2 and 3 implement ISC_Done as a function of bits installed during programming, so they wait in ISC_NOOP while Device Type 1 is programmed.

Steps 51–53: Set security bits in Device Types 1 and 2 using ISC_PROGRAM_SECURITY and in Device Type 3 with ISC_PROGRAM. Note that only Device Type 3 needs data scanned into ISC_PData to specify the address and data for setting security.

Steps 54–56: Execute ISC_DISABLE instruction in all three devices. Note that the status of the security setting operation just completed is now observable.

Steps 57–59: Sequence the devices into operational mode one at a time by loading each with the BYPASS instruction.

B.6 Programming an ISC device with an adaptive programming algorithm

The following step-by-step process in Table B.6 summarizes the programming process for a single 1532-compliant device that requires an adaptive programming algorithm. This device has a near-minimum set of ISC instructions, but does require the use of ISC_DISCHARGE to finish the programming process. This device implements the optional status reporting mechanism recommended in v) of 5.2, and the ISC_Done signal is a function of data loaded into memory. This flow for programming this device follows Example 13 in 8.5.7.8.

Table B.6—Programming flow 6: An ISC device with an adaptive programming algorithm

Step	TAP activity	Instruction/target/action
1	IR-Scan	Load ISC_ENABLE
2	DR-Scan	Shift ISC_Config Reg
3	<i>Run-Test/Idle</i>	Wait for enable time
<i>Device ready for programming.</i>		
4	IR-Scan	Load ISC_PROGRAM
5	DR-Scan	Shift ISC_PData Reg
6	<i>Run-Test/Idle</i>	Wait for write burn time
<i>Start adaptive programming loop.</i>		
7	IR-Scan	Load ISC_NOOP
8	Wait maximum time in the <i>Run-Test/Idle</i> TAP Controller State	
9	DR-Scan	Shift OST data + 2 bits
<i>Ignore OST data for the first nine loops and return to step 8.</i>		
10	DR-Scan	Shift OST data + 2 bits
<i>Check OST data against expected data. If loop count is less than 100 and data do not match them, return to step 8. If loop count is equal to 100 and data do match them, m, execute proc_error_exit.</i>		
11	IR-Scan	Load ISC_DISCHARGE
12	If more program data are available, return to step 4.	
<i>Now continue with program readback to verify the programmed data.</i>		
13	IR-Scan	Load ISC_READ

Table B.6—Programming flow 6: An ISC device with an adaptive programming algorithm (continued)

Step	TAP activity	Instruction/target/action
14	DR-Scan	Shift ISC_RData Reg
15	<i>Run-Test/Idle</i>	Wait for read time
16	If more data to read back, return to step 14	
17	DR-Scan	Shift ISC_RData Reg
<i>Now complete the process and make the device operational.</i>		
18	IR-Scan	Load ISC_DISABLE
19	DR-Scan	Scan out status of discharge
20	<i>Run-Test/Idle</i>	Wait for disable time
21	<i>Test-Logic-Reset</i>	Device operational

Now consider the loop process in Table B.6 in detail. The remainder of the flow is similar to the flow presented in Table B.1.

Steps 1–3: Enter the ISC Accessed system modal state and prepare the device for programming.

Steps 4–6: These steps load the first set of programming data into the ISC_PData data register. In the case of a device that uses an adaptive programming algorithm, the time spent in the *Run-Test/Idle* TAP Controller state may be used to start the programming circuitry. This example has the programming circuitry started in the *Run/Test-Idle* TAP Controller state and leaves it running after leaving that state. The circuitry is stopped by executing the ISC_DISCHARGE instruction

Steps 7–9: This device has the operation status indication implemented as subcode bits in the optional status register. The status register can be accessed through the ISC_NOOP instruction so that instruction is used to check the status of the programming process. The flow in Example 13 in 8.5.7.8 specifies a minimum loop count of 10 iterations and a maximum loop count of 100 iterations. For each iteration, the programming tool will wait the specified duration in the *Run-Test/Idle* TAP Controller state and, after waiting that time, will shift out the operation status indication bits. The programmer should ignore those bits for the first nine iterations and should check those bits starting at iteration number 10. If the data shifted out of the device do not match the expected data, the programming tool should continue looping until either the data do match or until the loop count reaches 100. If the loop count reaches 100 and the data still do not match, the procedure **proc_error_exit** should be executed. Once the data match, the programming tool can continue on to step 10.

Steps 10–11: Load the ISC_DISCHARGE instruction into the IEEE Std 1149.1-2001 instruction register. Once the *Run-Test/Idle* TAP Controller state is entered, the programming circuitry is stopped and the device is ready for the next set of programming data. If additional programming data are available, return to step 4 where the programming data are loaded into the device. It will be necessary to reload the ISC_PROGRAM instruction before loading the data into the device.

For the description of the steps dealing with reading from the device and leaving the ISC Accessed system modal state, refer to B.1.

Annex C

(informative)

Example ISC BSDL/data file

C.1 Example BSDL for entity SAMPLE_1532_DEVICE

```
entity SAMPLE_1532_DEVICE is

generic (PHYSICAL_PIN_MAP : string := "pqfp32" );

port (
    TDI: in bit; TMS: in bit; Gnd_2: linkage bit; TCK: in bit;
    Vcco_1: linkage bit; OUT1: out bit; OUT2: out bit; INOUT1: inout bit;
    INOUT2: inout bit; INOUT3: inout bit; INOUT4: inout bit;
    OUT3: out bit; IN1: in bit; Vcco_2: linkage bit; Vcc_1: linkage bit;
    Gnd_3: linkage bit; OUT4: out bit; OUT5: out bit; OUT6: out bit;
    Vcco_3: linkage bit; OUT7: out bit; Gnd_4: linkage bit; OUT8: out bit;
    TDO: out bit; Vpp: linkage bit; Vcco_4: linkage bit;
    Vcc_2: linkage bit; OUT9: out bit; Gnd_1: linkage bit; OUT10: out bit;
    CLK: in bit; nc: linkage bit);

use STD_1149_1_2001.all;
use STD_1532_2002.all;

attribute COMPONENT_CONFORMANCE of SAMPLE_1532_DEVICE : entity is
    "STD_1149_1_2001";    -- could also be STD_1149_1_1993

attribute PIN_MAP of SAMPLE_1532_DEVICE : entity is PHYSICAL_PIN_MAP;

constant pqfp32: PIN_MAP_STRING:=
    "TDI:3, TMS:5, Gnd_2:6, TCK:7, Vcco_1:8, OUT1:9, OUT2:10," &
    "INOUT1:13, INOUT2:44, INOUT3:1, INOUT4:20, OUT3:14, IN1:15," &
    "Vcco_2:16, Vcc_1:17, Gnd_3:18, OUT4:19, OUT5:21, OUT6:25," &
    "Vcco_3:26, OUT7:27, Gnd_4:28, OUT8:29, TDO:31, Vpp:11, " &
    "Vcco_4:23, Vcc_2:24, OUT9:30, Gnd_1:2, OUT10:32, CLK:12, nc:22";

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_CLOCK of TCK : signal is (1.00e+07, BOTH);
attribute INSTRUCTION_LENGTH of SAMPLE_1532_DEVICE : entity is 8;

attribute INSTRUCTION_OPCODE of SAMPLE_1532_DEVICE : entity is
    "BYPASS ( 11111111)," &
    "SAMPLE ( 00000001)," &
    "PRELOAD( 00000001)," &
    "EXTEST ( 00000000)," &
    "IDCODE ( 11111110)," &
    "USERCODE ( 11111101)," &
```

```

    "HIGHZ ( 11111100)," &
    "CLAMP ( 11111010)," &
-- ISC Instructions
    "ISC_NOOP ( 10011111)," &
    "ISC_ENABLE ( 11101000)," &
    "ISC_PROGRAM ( 11101010)," &
    "ISC_PROGRAM_DONE ( 11101110)," &
    "ISC_ADDRESS_SHIFT ( 11101011)," &
    "ISC_READ ( 11101111)," &
    "ISC_ERASE ( 11101100)," &
    "ISC_DATA_SHIFT ( 11101101)," &
    "ISC_DISABLE ( 11110000)," &
-- Proprietary ISC Instructions
    "VENDOR_BLANK_CHECK ( 11100101)," &
-- Private Instructions
    "PRIVATE1 ( 11110001)," &
    "PRIVATE2 ( 11100011)";

attribute INSTRUCTION_CAPTURE of SAMPLE_1532_DEVICE : entity is
    "00000x01";

attribute INSTRUCTION_PRIVATE of SAMPLE_1532_DEVICE : entity is
    "PRIVATE1, PRIVATE2";

attribute IDCODE_REGISTER of SAMPLE_1532_DEVICE: entity is
    "0001" &          -- version
    "0101001000110100" & -- part number
    "01011001001" &    -- manufacturer's id
    "1";                -- required by standard

attribute USERCODE_REGISTER of SAMPLE_1532_DEVICE : entity is
    "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";

attribute REGISTER_ACCESS of SAMPLE_1532_DEVICE : entity is
    "DEVICE_ID (IDCODE, USERCODE)," &
    "ISC_DEFAULT[1] ( ISC_DISABLE, ISC_NOOP, ISC_ERASE, " &
    "    ISC_PROGRAM, ISC_PROGRAM_DONE )," &
    "ISC_CONFIG[6] ( ISC_ENABLE )," &
    "ISC_PDATA[2048] ( ISC_READ, ISC_DATA_SHIFT ),"&
    "VENDOR_BLANK[128] ( VENDOR_BLANK_CHECK ),"&
    "ISC_ADDRESS[16] ( ISC_ADDRESS_SHIFT )";

attribute BOUNDARY_LENGTH of SAMPLE_1532_DEVICE : entity is 34;

attribute BOUNDARY_REGISTER of SAMPLE_1532_DEVICE : entity is
    " 0 (BC_1, CLK, input, X)," &
    " 1 (BC_1, *, controlr, 0)," &
    " 2 (BC_1, OUT10, output3, X, 1, 0, Z)," &
    " 3 (BC_1, *, controlr, 0)," &
    " 4 (BC_1, OUT9, output3, X, 3, 0, Z)," &
    " 5 (BC_1, *, controlr, 0)," &
    " 6 (BC_1, OUT8, output3, X, 5, 0, Z)," &
    " 7 (BC_1, *, controlr, 0)," &
    " 8 (BC_1, OUT7, output3, X, 7, 0, Z)," &

```



```

"Repeat 5          " &
    "(ISC_READ          wait 50.0e-6 2048:$data?:CRC)," &

-- Read back using auto-incremented address
"flow_read(array)" &
"initialize        " &
    "(ISC_ADDRESS_SHIFT 16:$addr=0 wait TCK 1)" &
    "(ISC_READ          wait 50.0e-6 2048!:CRC)" &
"Repeat 5          " &
    "(ISC_READ          wait 50.0e-6 2048!:CRC)," &

"flow_enable        " &
"initialize        " &
    "(ISC_ENABLE        6:4          wait TCK 1)," &

"flow_disable       " &
"initialize        " &
    "(ISC_DISABLE       wait 110.0e-3)" &
    "(ISC_NOOP          wait TCK 1)," &

-- Read IDCODE value, mask out version bits
"flow_verify(idcode)" &
"initialize        " &
    "(IDCODE            wait TCK 1 32:15434593*0FFFFFFF)," &

"flow_erase        " &
"initialize        " &
    "(ISC_ADDRESS_SHIFT 16:0001wait TCK 1)" &
    "(ISC_ERASE          wait 100.0e-3)," &

-- Test if device is really blank
"flow_blank_check" &
"initialize        " &
"(VENDOR_BLANK_CHECKwait 50e-3 " &
    "                  128:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)," &

-- Compare USERCODE value against blank value
"flow_verify(usercode)" &
"initialize        " &
    "(USERCODE          wait TCK 1 32:FFFFFFFF)," &

-- Compare USERCODE against value read from INPUT
"flow_read(usercode)" &
"initialize        " &
    "(USERCODE          wait TCK 1 32:!), " &

"flow_program_done  " &
"initialize        " &
    "(ISC_PROGRAM_DONE  wait 14.0e-3)," &

-- On any error, erase the device
"flow_error_exit    " &
"initialize        " &
    "(ISC_ADDRESS_SHIFT 16:0001wait TCK 1)" &

```

```

        "(ISC_ERASE          wait 100.0e-3)," &
        "(ISC_DISABLE       wait 110.0e-3)" &
        "(ISC_NOOP          wait TCK 1)";

attribute ISC_PROCEDURE of SAMPLE_1532_DEVICE : entity is
    "proc_verify(idcode)    = (flow_verify(idcode))," &
    "proc_enable           = (flow_enable)," &
    "proc_disable          = (flow_disable)," &
    "proc_erase            = (flow_erase)," &
    "proc_blank_check      = (flow_blank_check)," &
    "proc_program          = (flow_program(array))," &
    "proc_verify           = (flow_verify(array))," &
    "proc_verify(usercode) = (flow_verify(usercode))," &
    "proc_read             = (flow_read(array))," &
    "proc_read(usercode)   = (flow_read(usercode))," &
    "proc_program_done     = (flow_program_done)," &
    "proc_error_exit       = (flow_error_exit)";

attribute ISC_ACTION of SAMPLE_1532_DEVICE : entity is
    "erase                = (proc_verify(idcode) recommended," &
    "                      proc_enable," &
    "                      proc_erase," &
    "                      proc_blank_check optional, " &
    "                      proc_disable)," &
    "program              = (proc_verify(idcode) recommended," &
    "                      proc_enable," &
    "                      proc_erase," &
    "                      proc_blank_check optional, " &
    "                      proc_program," &
    "                      proc_enable," &
    "                      proc_verify optional," &
    "                      proc_disable)," &
    "verify               = (proc_verify(idcode) recommended," &
    "                      proc_enable," &
    "                      proc_verify," &
    "                      proc_disable)," &
    "read                 = (proc_verify(idcode) recommended," &
    "                      proc_enable," &
    "                      proc_read," &
    "                      proc_disable)";

end SAMPLE_1532_DEVICE;

```

C.2 ISC data file for entity SAMPLE_1532_DEVICE

```

IEEE_1532_Data
(header
  (STD_Version STD_1532_2002
   Creation_Date 20020320.1743
   Creator "gen1532data 1.00"
   Entity SAMPLE_1532_DEVICE)
data array (
  Initialize

```



```
009D0024A0180000000000000000000000800074009D00244009101E44009D01
A44009D00274009D00274009D00274009D00274009D00262001000000
000000000082043C009F0024C00900564C009B2024C009F0027C009F00
27C009F0027C009F0027C009F002708100000000000000000008004BC001
F0007C001F0807C001F0087D001F0007C001F0007C001F0007C001F0)

    data_CRC( 830B089D )
)
1AB0
```

Annex D

(informative)

Bibliography

IEEE 100™, *The Authoritative Dictionary of IEEE Standards Terms*, Seventh Edition.⁵

⁵The IEEE standards or products referred to in Annex D are trademarks owned by the Institute of Electrical and Electronics Engineers, Incorporated.