



IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data—Core Test Language (CTL)

IEEE Computer Society

Sponsored by the
Test Technology Standards Committee

IEEE
3 Park Avenue
New York, NY 10016-5997, USA
5 April 2006

IEEE Std 1450.6™-2005

IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data—Core Test Language (CTL)

Sponsor

Test Technology Standards Committee
of the
IEEE Computer Society

Approved 17 November 2005
IEEE-SA Standards Board

Reaffirmed 16 June 2011
IEEE-SA Standards Board

Approved 29 December 2005
American National Standards Institute

Reaffirmed 26 July 2012
American National Standards Institute

Abstract: The Core Test Language (CTL) is a language created for a System-on-Chip flow (or SoC flow), where a design created by one group is reused as a sub-design of a design created by another group. In an SoC flow, the smaller design embedded in the larger design is commonly called a core and the larger design is commonly called the SoC. The core is a design provided by a core provider, and the task of incorporating the sub-design into the SoC is called Core System Integration.

Keywords: Core Test Language (CTL), Standard Test Interface Language (STIL), System-on-Chip (SoC), wrapped core, unwrapped core

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

Copyright © 2006 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 5 April 2006. Printed in the United States of America.

IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by the Institute of Electrical and Electronics Engineers, Incorporated.

Print: ISBN 0-7381-4804-0 SH95373
PDF: ISBN 0-7381-4805-9 SS95373

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 or the soundness of any judgments 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, or every ten years for stabilization. When a document is more than five years old and has not been reaffirmed, or more than ten years old and has not been stabilized, 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 his or her independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

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. A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered the official position of IEEE or any of its committees and shall not be considered to be, nor be relied upon as, a formal interpretation of the IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE.

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. Recommendations to change the status of a stabilized standard should include a rationale as to why a revision or withdrawal is required. Comments and recommendations on standards, and requests for interpretations should be addressed to:

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

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 1450.6-2005, IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data—Core Test Language (CTL).

CTL started as a language in the IEEE Std 1500™-2005 standardization activity for core test. This activity provided a representation mechanism for test information that exchanges hands between a core provider and the system integrator. Thus, the language had a charter to provide a mechanism for reuse of test patterns and information that allows for successful design for test and automatic test pattern generator activities on the SoC. As part of IEEE Std 1500-2005, CTL was designed to represent details about the IEEE 1500 wrapper. As CTL and the wrapper technology matured, it became apparent that the two activities should be separated into two standard documents. As a result of this decision, CTL, the language, became IEEE Std 1450.6 activity, and the information model for cores that uses CTL remained in IEEE Std 1500-2005.

Notice to users

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: <http://standards.ieee.org/reading/ieee/updates/errata/index.html>. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: <http://standards.ieee.org/reading/ieee/interp/index.html>.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents or patent applications for which a license may be required to implement an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Participants

The following is a list of participants in the CTL Working Group:

Rohit Kapur, *Chair*

Mike Collins
Douglas Kay

Brion Keller

Maurice Lousberg
Paul Reuter

When the CTL Working Group approved this standard, it had the following short-term membership:

Bill Chown

Yuhai Ma

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

Ken-ichi Anzou	Jake Karrfalt	Benoit Nadeau-Dostie
Luis Basto	Douglas Kay	Charles Ngethe
Sudipta Bhawmik	Brion Keller	Jim O'Reilly
Dwayne Burek	Adam Ley	Adam Osseiran
Chen-Huan Chiang	Dennis Lia	Klaus Rapf
Keith Chow	Maurice Lousberg	Paul Reuter
Bill Chown	Yuhai Ma	Mike Ricchetti
Antonio M. Cicu	Ryan Madron	Gordon Robinson
Luis Cordova	Erik Jan Marinissen	Gil Shultz
Jason Doege	Denis Martin	Douglas E. Sprague
Geir Eide	Gregory Maston	Tony Taylor
Grady Giles	Yinghua Min	Tom Waayers
Alan Hales	Mehdi Mohtashemi	Gregg Wilder
Peter Harrod	James Monzel	T. W. Williams
Mitsuaki Ishikawa	Narayanan Murugesan	Li Zhang
Rohit Kapur		

When the IEEE-SA Standards Board approved this standard on 17 November 2005, it had the following membership:

Steve M. Mills, *Chair*

Richard H. Hulett, *Vice Chair*

Don Wright, *Past Chair*

Judith Gorman, *Secretary*

Mark D. Bowman	William B. Hopf	T. W. Olsen
Dennis B. Brophy	Lowell G. Johnson	Glenn Parsons
Joseph Bruder	Herman Koch	Ronald C. Petersen
Richard Cox	Joseph L. Koepfinger*	Gary S. Robinson
Bob Davis	David J. Law	Frank Stone
Julian Forster*	Daleep C. Mohla	Malcolm V. Thaden
Joanna N. Guenin	Paul Nikolic	Richard L. Townsend
Mark S. Halpin		Joe D. Watson
Raymond Hapeman		Howard L. Wolfman

*Member Emeritus

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

Satish K. Aggarwal, *NRC Representative*

Richard DeBlasio, *DOE Representative*

Alan H. Cookson, *NIST Representative*

Jennie M. Steinhagen

IEEE Standards Project Editor

Contents

1.	Overview.....	1
1.1	General.....	1
1.2	SoC flow	2
1.3	Scope.....	4
1.4	Purpose.....	4
1.5	Limitations of this standard	4
1.6	Structure of this standard	5
2.	Normative references.....	5
3.	Definitions, acronyms, and abbreviations.....	6
3.1	Definitions	6
3.2	Acronyms and abbreviations	6
4.	CTL orientation and capabilities tutorial	7
4.1	Introduction.....	7
4.2	CTL for design configurations.....	8
4.3	CTL for structural information	16
4.4	CTL for test pattern information.....	21
4.5	Beyond the examples	26
5.	Extensions to IEEE Std 1450-1999 and IEEE Std 1450.1-2005	27
5.1	STIL name spaces and name resolution	27
5.2	Optional statements of IEEE Std 1450-1999	27
5.3	Restricting the usage of SignalGroup and variable names	27
5.4	Additional reserved words	28
5.5	STIL statement—extensions to IEEE Std 1450-1999, Clause 8	28
5.6	Extensions to IEEE Std 1450-1999, 17.1 and 23.1.....	29
5.7	Extensions associated with the LockStep construct of Clause 13 of IEEE Std 1450.1-2005....	32
6.	Design hierarchy—cores.....	36
6.1	CoreType block and CoreInstance statement	36
6.2	CoreType block syntax descriptions.....	36
6.3	CoreType block code example	37
7.	Cell expression (cellref_expr).....	38
8.	Environment block—extensions to IEEE Std 1450.1-2005, Clause 17	39
8.1	General.....	39
8.2	Definition of FileReference keywords.....	39
8.3	Example of Environment block FileReference syntax	43
8.4	Extension to NameMaps.....	43
8.5	Extension to the inheritance of environment statements	44
9.	CTLMode block.....	45

9.1	General	45
9.2	CTLMode syntax	45
9.3	CTLMode block—syntax descriptions	47
9.4	CTLMode block syntax example	53
10.	CTLMode—Internal block	56
10.1	General	56
10.2	Internal syntax	56
10.3	Internal block syntax descriptions	60
10.4	Internal BlockSyntax examples	76
11.	CTLMode—ScanInternal block	79
11.1	General	79
11.2	ScanInternal syntax	79
11.3	ScanInternal block syntax descriptions	81
11.4	ScanInternal block syntax example	81
12.	CTLMode—CoreInternal block	83
12.1	General	83
12.2	CoreInternal syntax	83
12.3	CoreInternal block syntax descriptions	84
12.4	CoreInternal block syntax examples	84
13.	CTLMode—Relation Block	85
13.1	General	85
13.2	Relation syntax	85
13.3	Relation block syntax descriptions	85
13.4	Relation block syntax example	87
14.	CTLMode—ScanRelation block	89
14.1	General	89
14.2	ScanRelation syntax	89
14.3	ScanRelation block syntax descriptions	89
15.	CTLMode—External block	89
15.1	General	89
15.2	External statement syntax	90
15.3	External block syntax descriptions	90
15.4	External block syntax example	93
16.	CTLMode—PatternInformation block	93
16.1	PatternInformation syntax	93
16.2	PatternInformation block syntax descriptions	96
16.3	PatternInformation block syntax example	104
	Index	109

IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data—Core Test Language (CTL)

1. Overview

1.1 General

The Core Test Language (CTL) is a language created for a System-on-Chip flow (or SoC flow), where a design created by one group is reused as a sub-design of a design created by another group. In an SoC flow, the smaller design embedded in the larger design is commonly called a core and the larger design is commonly called the SoC. The core is a design provided by a core provider, and the task of incorporating the sub-design into the SoC is called Core System Integration.

CTL is a language designed to be the transfer mechanism of test knowledge between a core provider and a system integrator to allow for interoperability between the producer and the consumer of the information. It facilitates the reuse of test patterns provided for a core for application from the SoC boundary. Although the language is general (the limitations of CTL are listed in 1.5) and can be used in many different ways, this standard is focused on the use of CTL for SoC designs. Thus, CTL allows for

- a) Representation of design constructs and characteristics that are needed to be made visible by the core provider.
- b) Representation of test patterns that are to be reused for cores in an SoC test flow.

CTL provides the language for communication of test information. An adjacent IEEE standard activity (IEEE Std 1500™-2005)¹ defines the information requirements of the core provider that are required to be represented in CTL. As a result of this relationship with IEEE Std 1500-2005, CTL has some constructs that

¹For information on references, see Clause 2.

are there to support IEEE Std 1500-2005's informational requirements for wrapped and unwrapped cores. In an SoC flow (where cores are reused in SoCs), CTL represents all of the test information about the core such that the core can be successfully embedded in the SoC from a test perspective. CTL is a language that provides the information about the structures in the core that are to be reused at the next level of integration. Test reuse is also about the reuse of test patterns. As a result, CTL allows for the description of reusable test patterns (portable tests). CTL leverages existing standard representations. The CTL language is the union of the syntax defined in IEEE Std 1450TM-1999, IEEE Std 1450.1TM-2005, IEEE Std 1450.2TM-2002, and this standard. CTL extends the definitions defined in IEEE Std 1450-1999 and IEEE Std 1450.1-2005 with extensions and exceptions defined in this standard. The reader is assumed to have knowledge of these standards, and their associated content is not repeated or explained in this standard.

CTL has the following characteristics to support the SoC test:

- a) CTL can describe constructs to handle a wide variety of cores.
- b) CTL does not limit the test methodologies of the core provider.
- c) CTL can describe IEEE Std 1500-2005 wrapped and unwrapped cores. (Refer to the associated standard for details.)
 - 1) Unwrapped cores can be described in CTL to aid in the creation of the wrapper.
 - 2) Wrapped cores can be described in CTL to allow for the reuse of the wrapper in the integrated SoC.
 - 2) CTL describes the patterns of the wrapped or unwrapped core such that modifications can be made to the core for application from the SoC boundary.

CTL handles this variety of needs by providing a test mode structure within which primitive concepts are pieced together to describe the test information. Several sets of keywords are provided that form the vocabulary of CTL. The keywords are put together to represent information that are sentences of test information. CTL's ability to handle different designs and their associated test methodologies comes from its reliance on sequences. Predefined sequence types (EstablishMode and TerminateMode) are used to treat the different design configurations (test modes) in a uniform way. Sequence information is leveraged from IEEE Std 1450-1999 and IEEE Std 1450.1-2005 syntax, with the difference that the information is driven from the CTL Environment and test modes.

1.2 SoC flow

As mentioned, an SoC flow is the process in which a sub-design (core) is embedded in a larger design (SoC). The major steps of this flow are shown in Figure 1. Let us assume that CTL exists and it can describe everything needed for the core. Figure 1 shows three major steps in the process of designing the testing mechanism and the test patterns for embedded cores. The first step shows the "Core Design" work. This work is performed in such a manner that the resultant core can be reused in multiple designs with NO change required to either the core design or to the bulk of the test patterns information (data portion of the test patterns) that are developed for testing the core.

The first section in the diagram shows the core design process where the "IP core model" is represented (e.g., a Verilog or VHDL design). Along with the design, a set of "Core Test Patterns" in CTL syntax is generated (syntax from IEEE Std 1450-1999, IEEE Std 1450.1-2005, IEEE Std 1450.2-2002, and this standard). A complete CTL description encompasses test patterns and its associated constructs and a description of the various test modes of the core. Typical cores support multiple configurations that serve different purposes such as internal testing of the core and external testing of the logic outside the core. CTL information is partitioned across these configurations (test modes). The entire description of CTL is centered around the Environment (this is the top-level block of statements in the language), which is shown in Figure 1. The information requirements for cores is defined by IEEE Std 1500-2005. The requirements ensure that enough information is present to create the test for the finished SoC.

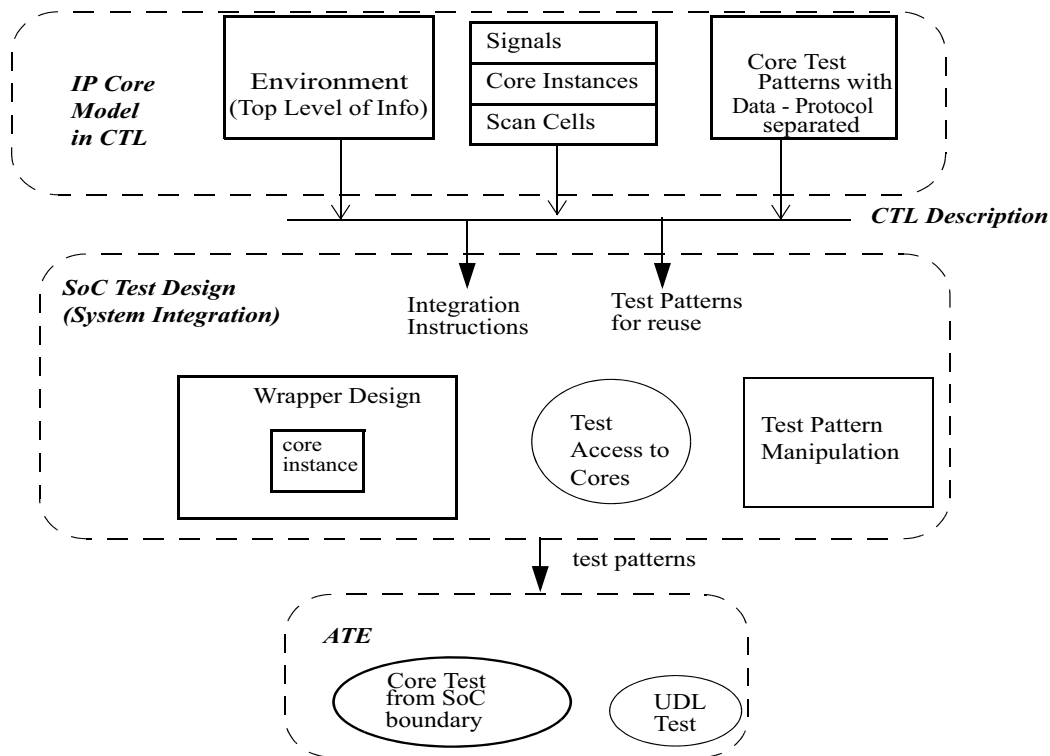


Figure 1—SoC flow showing the CTL use model

The second section of Figure 1 shows the System Integration process. In this process, the SoC integrator makes use of the CTL description of the core design to create the tests for the SoC. This process could involve multiple activities depending on the design and test methodologies in use. The following are some activities that the system integrator may perform:

- Wrapper Design:** Wrapper technology represents isolation hardware at the boundary of cores for test purposes. The wrapper isolates the testing of the core logic from the testing of the user-defined logic on the SoC. IEEE Std 1500-2005 defines such technology. Cores may or may not be provided with wrappers. If the test methodology relies on wrapping cores, then the system integration step would look for information in CTL of the core to determine whether a wrapper is provided with the core. If the wrapper is not provided, the CTL information of the core would be used to determine the wrapper cells and other details of the wrapper technology.
- Test Access to Cores:** The connections from the core boundaries to the top level of the SoC depends on the different cores embedded in the SoC. The design of the associated design entities depends on the scheduling of the tests of the different cores, which is determined by looking at the CTL that comes with each core.
- Test Pattern Manipulation:** The patterns that come with the CTL of every core are written to the boundary of the core. The process of converting these patterns to be applicable at the boundary of the SoC is called test pattern migration or test data porting. The process uses the wrapper technology and test access to cores as relevant to the SoC design. Patterns supplied in CTL have the data and protocol (event sequencing) portion separated. The data represent the 1s and 0s of the test patterns, which represent the bulk of the information. The protocol represents the sequence of the test patterns to apply the data. Test pattern migration is the process of modifying the protocols of the test patterns

such that the test data are oriented to the SoC boundary. CTL information assists in test scheduling tasks on the SoC.

The last section shows the process of testing the manufactured SoC. The core-level tests are combined with the top-level logic tests to provide for complete testing of the SoC on an ATE. The creation of the top-level patterns of the SoC (UDL test) was not shown in the system integration step of Figure 1, but this task typically uses CTL information for external test or low leakage configurations of the cores on the SoC. If the information in last section is packaged in new CTL, then the SoC can be a core for another level of the design.

1.3 Scope

Unless the logic inside embedded cores can be merged with the surrounding user-defined logic (UDL), the SoC test requires reuse of test data and test structures specific to individual cores (designs) when integrated into larger systems. This standard defines language constructs sufficient to represent the context of a core and of the integration of that core into a system, to facilitate reuse of test data previously developed for that core. The SoC test also requires that the core be embedded in the SoC to allow for efficient testing of the logic external to the core. To that effect, this standard defines constructs that represent the test structures internal to the core for reuse in the creation of the tests for the logic outside the core. This provides constructs that will allow for the wrapping operation of an unwrapped core and the necessary wrapper specific information for a wrapped core. In particular, CTL shall support IEEE Std 1500-2005 for the information needs for wrapped and unwrapped cores. Semantic rules will be defined for the language to facilitate interoperability between the different entities (the core provider, the system integrator, and the automation tools) involved in the creation of an SoC. This standard is limited to SoC testing with multiple and/or hierarchical cores through digital interfaces.

All constructs defined in the CTL shall be consistent with IEEE Std 1450-1999 and extensions (STIL) to support the complete description of the test for cores integrated into SoC environments. Although the preferred syntax for the bulk of the test data is STIL, this language provides constructs for linking other test data representations to incorporate legacy cores. The constructs in the language shall support a vast variety of cores and different test methodologies with particular support for the IEEE 1500 standard for embedded core testing. These constructs shall facilitate the transportation of test information from the core provider to the system integrator and support test automation by providing a consistent and uniform definition of the constructs such that the information provided by a core provider is understood in the same way by the system integrator and the tools developed by EDA.

1.4 Purpose

To develop a language that will provide a sufficient description of a core to support reuse of test data developed for that core after integration into SoC environments, and to enable the creation of test patterns for the logic on the SoC external to the core.

1.5 Limitations of this standard

As the development of the language for complete functionality is a huge task, the first version of this standard is limited in its functionality. CTL, as described in this standard, has limited or has no support for the following areas:

- a) Analog testing
- b) Memory testing
- c) Diagnostics and debug applications
- d) ATE interfacing of DFT information

All of these areas are considered important and should be addressed in future extensions of this language.

1.6 Structure of this standard

This standard is to be used with IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002. The conventions established and defined in IEEE Std 1450-1999 are used in this standard and are included verbatim below.

Many clauses in this document add additional constructs to existing clauses in IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002 and are so identified in the title. Most constructs defined in this standard are limited to the Environment block, which is defined by IEEE Std 1450.1-2005.

The following is a copy of the conventions as defined in IEEE Std 1450-1999 and followed by this standards.

Different fonts are used as follows:

- a) SMALL CAP TEXT is used to indicate user data.
- b) `courier text` is used to indicate code examples.

In the syntax definitions

- a) SMALL CAP TEXT is used to indicate user data.
- b) **bold text** is used to indicate keywords.
- c) *italic text* is used to reference metatypes.
- d) () indicates optional syntax that may be used zero or one time.
- e) ()+ indicates syntax that may be used one or more times.
- f) ()* indicates optional syntax that may be used zero or more times.
- g) <> indicates multiple choice arguments or syntax.
- h) Defaults where appropriate are underlined in the syntax.

In the syntax explanations, the verb “shall” is used to indicate mandatory requirements. The meaning of a mandatory requirement varies for different readers of the standard:

- To developers of tools that process CTL (readers), “shall” denotes a requirement that the standard imposes. The resulting implementation is required to enforce this requirement and issue an error if the requirement is not met by the input.
- To developers of CTL files (writers), “shall” denotes mandatory characteristics of the language. The resulting output must conform to these characteristics.
- To the users of CTL, “shall” denotes mandatory characteristics of the language. Users may depend on these characteristics for interpretation of the CTL source.

The language definition clauses contain statements that use the phrase “it is an error”. This phrase indicates improperly defined CTL information.

2. Normative references

CTL encompasses syntax defined in IEEE Std 1450-1999, IEEE Std 1450.1-2005, IEEE Std 1450.2-2002, and this standard. The language extends syntax defined by IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002. Exceptions, if any, to these standards are defined in this standard. The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEEE Std 1450TM-1999, IEEE Standard Test Interface Language (STIL) for Digital Test Vectors.^{2, 3}

IEEE Std 1450.1TM-2005, Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450-1999) for Semiconductor Design Environments.

IEEE Std 1450.2TM-2002, IEEE Standard for Extensions to Standard Test Interface Language (STIL) for DC Level Specification.

IEEE Std 1500TM-2005, Testability Method for Embedded Core-based Integrated Circuits.

3. Definitions, acronyms, and abbreviations

3.1 Definitions

For the purposes of this standard, the following terms and definitions apply. As CTL is the union of the syntax defined in four documents, the terminology defined in the other documents (IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002) is applicable in this standard. *The Authoritative Dictionary of IEEE Standards Terms* should be referenced for terms not defined in this clause.

3.1.4 black box: A condition where the netlist of the design referred to is not available. Commonly used in association with a core, namely, a black box core.

3.1.5 core: Embedded hierarchy of the design. This hierarchy could be created by an imaginary boundary that isolates a portion of the logic in a design.

3.1.6 test mode: It is a configuration of the design that is defined in CTL as a CTLMode. This type of configuration in CTL is identified by the TestMode statement within the CTLMode block of statements.

3.1.7 white box: A condition where the netlist of the design referred to is available. Commonly used in association with a core, namely, a white box core.

3.2 Acronyms and abbreviations

1450.0	IEEE Std 1450-1999
1450.1	IEEE Std 1450.1-2005 extension to STIL
1450.2	IEEE Std 1450.2-2002 extension to STIL
ATE	automated test equipment
ATPG	automatic test pattern generator
BIST	built-in self-test
CTL	Core Test Language (aggregate syntax of 1450.6, 1450.1, 1450.2, and 1450.0)
DFT	design for test (it represents circuit modifications such as scan chains that are created for test)
IC	integrated circuit
SoC	System on Chip
STIL	Standard Test Interface Language (combination of 1450.0, 1450.1, and 1450.2)
TAM	test access mechanism used to connect to the (wrapped) cores being tested
UDL	user-defined logic (a moniker for any logic that may surround an embedded core)
WFC	WaveformCharacter

²IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (<http://standards.ieee.org/>).

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

4. CTL orientation and capabilities tutorial

4.1 Introduction

Design reuse methodologies have allowed for the partitioning of effort needed to create a complete design at the expense of communication between the teams creating the design. Tasks when partitioned are performed by teams that are in close proximity to each other. In such situations, the teams can get together with frequent formal meetings, share common documentation, or discuss problems informally. This is a normal business process for any company that has a reasonably sized project under way. This breaks down when the design teams are separated in time and space. To avoid unreasonable communication problems, the process needs to be formalized and the interface between the core providers and the integrators needs to be standardized. This is where CTL as an industry-wide standard steps in. CTL is developed to address the information transfer needs of the SoC test.

Through CTL, all test aspects of cores can be described such that a system integrator can integrate a core as a black box into a SoC and perform all of the usual test tasks as though the core was a white box with test patterns to be reused. As shown in Figure 2, CTL would describe all information about the core needed by the system integrator. The language is designed to be manually written and created and/or consumed by test automation tools.

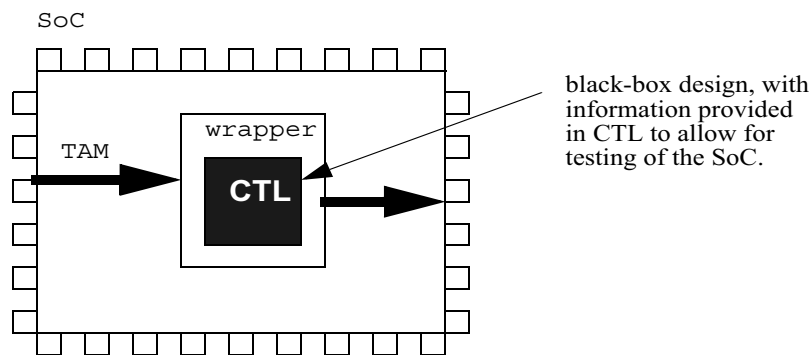


Figure 2—CTL used to represent the test information of a core instead of a netlist

Figure 2 shows a usage of CTL where the information about the design that is to be embedded as a core is represented in CTL. Using the CTL of the embedded design (the core of the SoC), a wrapper can be constructed, and the appropriate TAM, which creates the pathway for the test data of the core, can be determined based on the test constraints in the CTL of the core. The figure depicts the core represented in CTL as a black box as the netlist is not provided for the core. A box around the black box represents a scan wrapper implemented outside the core, and the arrows represent the access mechanism to the wrapper. Once all structures are in place, the test patterns that are also a part of CTL can be retargeted to the boundary of the SoC. CTL is the language to support all information that the core provider needs to give the system integrator such that the integrator can successfully test the embedded core and any UDL around the core. All language constructs defined in CTL would work with all types of digital cores, their different test methodologies, and the different ways in which they are integrated in the design.

Information in CTL is partitioned by the modes of operation of the design being described. Each test mode has its information as relevant for the design and communication needs. Not all DFT used in a design needs to be described to the system integrator of the design.

In the examples, we show how aspects of the information that describes the test details of a design are presented in CTL. This information is part of the content that passes between the core provider and the system integrator in an SoC flow. The information includes

- Design configuration information
- Structural information
- Test pattern information

These pieces of information are critical to the information needs of the design for the successful testing of the SoC. CTL provides a mechanism to represent test information for the test needs in an SoC flow. The information model is not in the scope of this standard and is addressed in IEEE Std 1500-2005 with complete examples. In this standard, the mechanics to represent information in CTL are described.

NOTE—CTL is part of the STIL series of standards, and the language encompasses the syntax defined by IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002. The reader is assumed to be familiar with the content of 1450.0, 1450.1, and 1450.2, and thus, they are not reintroduced here. Examples of IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002 portion of the syntax are given in their corresponding documents.⁴

4.2 CTL for design configurations

When writing the language CTL, one is essentially describing test modes in the Environment block of statements. Within the Environment, each test mode represents a design configuration described in *CTLMode* blocks (*CTLMode {}*). Each test mode contains information relevant to it. When the information gets bulky (such as test pattern information), the information is only referenced in the appropriate test mode block (*CTLMode {}*) of the Environment. The referenced construct is defined outside the Environment.

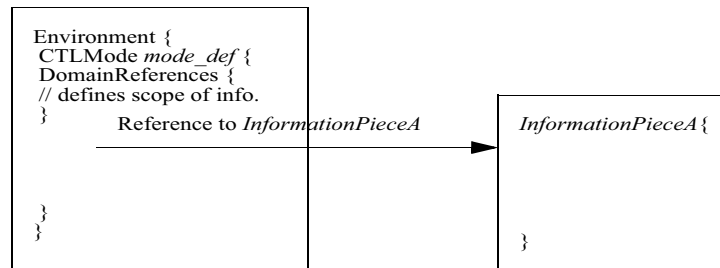


Figure 3—CTL information written outside the Environment and its relationship to the Environment

Figure 3 depicts the informational link between InformationPieceA and the test configuration of the design (*CTLMode*). InformationPieceA is a construct defined outside the Environment and is linked in to the test mode through a statement in the Environment. This mechanism is used for test patterns. The patterns are referenced in the Environment block of statements but defined outside the Environment. The following example shows the reference mechanism as it pertains to patterns:

⁴Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement the standard.

```

Environment {
  CTLMode mymode {
    PatternInformation {
      // reference to pattern P1 as a pattern used to establish the
      // test mode named mymode.
      Pattern P1 { Purpose EstablishMode; Protocol Macro macro_name; }
    }
  }
}
Pattern P1 {
  // definition of pattern P1. All constructs used by P1
  // are required to be in the scope of mymode as defined by
  // DomainReferences.
}

```

The construct being referenced and its supporting constructs should be within the scope of the test mode. CTL uses the concept of local and global entities for grouped constructs defined outside the Environment. Container blocks such as MacroDefs, Procedures, ScanStructures, Timing, Variables, and SignalGroups are defined to allow for the definition of their associated entities. These container blocks can be named with a DOMAIN_NAME to create domains. Domained entities are brought into the scope of the test mode in the Environment block through a construct called DomainReferences. Containers with no domain names are considered to be global pieces of information. Once within the scope of the test mode, the entities within the containers can be referenced or used by referenced constructs as shown in Figure 3. The following example shows the reference mechanism just described:

```

// Nameless container block for signals
Signals {
  // signals defined here (global for all modes)
}
// Nameless container block for Macros
MacroDefs {
  // global macros
}
// Named container block for Macros
MacroDefs A {
  // domain A's macros
}
// Named container block for Macros
MacroDefs B {
  // domain B's macros
}
Environment {
  // global environment
  CTLMode mymode {
    DomainReferences {
      MacroDefs A;
    }
    // Now macros defined in MacroDefs A can be referenced here as they
    // are in the scope of mymode. See reference mechanism defined
    // in the paragraphs before this example. Macros written in the
    // nameless MacroDefs block are also in the scope of mymode.
  }
}

```

This example shows a test mode named *mymode* identified by the CTLMode block within the Environment block of statements. (This will be described in the following text in this subclause.) The information of *mymode* uses the global Macros and the Macros in the MacroDefs domain *A*.

As mentioned, information in CTL is organized around a test mode (configuration) of the core being represented. Some of the information in each test mode pertains to the externally accessible signals of the core. These signals, defined in the signals block of statements (1450.0), give a point of reference for test-mode-related information. It should be noted that the Signals block is an unnamed block, and hence, its information is global and already included in every test mode.

Designs that are described in CTL typically support multiple configurations or test modes.

Figure 4 shows a design supporting multiple modes of operation. The design has nine input signals and three output signals. The design is implemented with two scan chains both of length 2. One scan chain goes across the boundary of the design capturing values on *signal a* and launching values on *signal b*. The boundary scan chain has a scan input BSI and a scan output BSO and is clocked by BCK. Its scan configuration is achieved by putting a logic-0 on the scan enable BSE. The second scan chain is an internal scan chain with scanin SI, scanout SO, scan-enable SE = logic-1, and clock CLK. The signals *y* and *z* are test mode signals that define the configuration of the design. In this case, $y=0, z=0$ is the functional configuration of the design; $y=0, z=1$ puts the design in a test mode where the boundary scan chain can be used to capture and launch values for the purposes of testing the logic outside the design when the design is embedded in a larger design. $y=1, z=0$ sets the design in a test mode where the logic internal to the design can be tested.

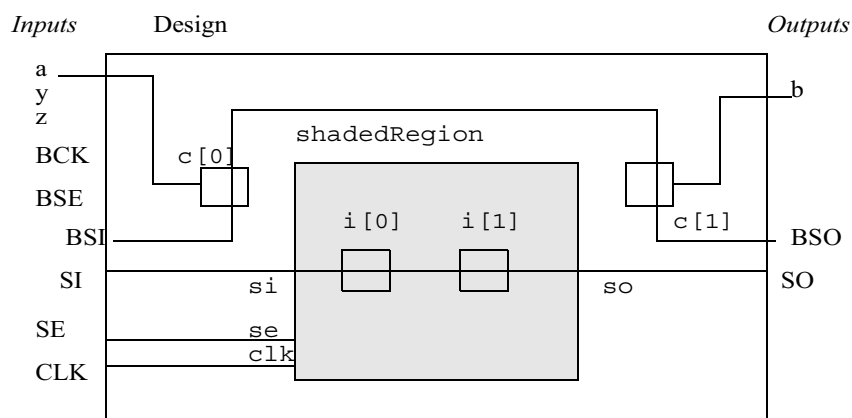


Figure 4—A design for which CTL is written to describe its test-related information

In this subclause on design configurations, the goal is to only describe the configurations of the design. That is, we are interested in only describing how to establish every test mode and not everything that we can describe about the design. The complete CTL-code for the associated modes of Figure 4 is shown as follows. **Bold words** highlight the keywords defined by the standards, which in this case are CTL (1450.6, 1450.0, 1450.1, and 1450.2) constructs. Comments and explanations are embedded in the example:

```
STIL 1.0 {
    Design 2005;
    CTL 2005;
}
```

```
// The information in the environment requires the recognition of the
// boundary of the design. This is described in the Signals block outside
// the environment. Since Signals is a nameless block the signals can
// be then referred to in the Environment. Signals in this example are
// either Inputs (In) or Outputs (Out). Details of Signals are defined
// in 1450.0. Each signal is assigned a default state to denote that if
// no state is specified for that signal on a specific pattern, the
// specified default state would be applied to that signal.
```

```
Signals {
  a In { DefaultState Z;}
  b Out;
  y In { DefaultState Z;}
  z In { DefaultState Z;}
  SE In { DefaultState Z;}
  BSE In { DefaultState Z;}
  CLK In { DefaultState D;}
  BCK In { DefaultState D;}
  SI In { DefaultState Z;}
  SO Out;
  BSI In { DefaultState Z;}
  BSO Out;
} // end Signals
```

```
// Values in CTL are waveform characters that are defined in the timing
// block as events (D, U, N, L, H, X) associated with timing within
// a clock period. The syntax specification of the Timing block can be
// found in 1450.0. Since the Timing block is a named block it needs to
// be explicitly brought into the scope of the test modes. This would
// be done through a statement in the DomainReferences. The timing
// block is needed to support the definition of the sequences that
// configure the design into the test mode. The design is configured
// into a test mode by a sequence of clock periods as defined by the
// Macros, and the values are waveform characters that are supplied by
// the patterns to the Macros.
```

```
Timing T1 {
  WaveformTable W1 {
    Period '100ns';
    Waveforms {
      'a+y+z+SE+BSE+CLK+BCK+SI+BSI' { 01x {'0ns' D/U/N;}}
      'b+SO+BSO' { 01x {'0ns' L/H/X;}}
    }
  }
} // end Timing
```

```
// The MacroDefs block represents the protocol or sequence information
// in CTL. The following is an unnamed MacroDefs block; hence, its macros
// are globally available in every test mode. CTL restrictions on
// pattern syntax as defined in 1450.6 forces the Macros to refer to the
// waveforms within it. The syntax and semantics used are defined
// in 1450.0. These macros defined here is needed to support the
// definition of the mechanism used to establish the test modes. The test
// mode is to be established by patterns that call the macros in this
// example.
```

```
MacroDefs {
```

```
// Macro setupseq takes in two parameters y and z and assigns them to
// the signals y and z. The following uses syntax from 1450.0
setupseq {
    W W1;
    V { y=#; z=#; }
}
} // end MacroDefs

// This is the primary block of information being written in CTL. The
// ordering rules of 1450.0 require language blocks to follow the rule
// that blocks must be defined before they are referenced (with some
// exceptions around pattern blocks). One named Environment block is
// expected to have information pertaining to the test needs of the
// design after inheritance of environments is resolved.
Environment design {

    // A nameless CTLMode block may be used to represent global
    // information that is common to all named CTLMode blocks.
    // In this example the Timing used in all the test modes is the same.
    CTLMode {

        // The named timing block is brought into the scope of the global
        // or nameless CTLMode block of statements. Hence it is also brought
        // into the scope of every other named CTLMode block of statements.
        // In this example the timing block is brought into the scope of the
        // test modes allowing for its information to be used by other
        // constructs in the test mode. As a result of the following
        // statement the Macro setupseq can use waveform table W1.
        DomainReferences {
            Timing T1;
        }
    } // end nameless CTLMode

    // One of the configurations to be described is the normal operation
    // of the design. This is achieved by setting y=0 and z=0. This is
    // achieved in CTLMode through a Pattern that establishes the test
    // mode. The pattern is defined outside the environment and referred
    // to in the PatternInformation block of this test mode. The pattern
    // is accordingly identified by an EstablishMode keyword. After the
    // test mode is established, the signals y and z are to be held
    // constant to maintain the configuration. This is defined in the
    // Internal block.
    CTLMode myN {

        // Test mode is used to define the functional operation of the
        // design, and the function is a non-test function.
        TestMode Normal;

        // The internal block of statements contains information on the
        // signals of the design looking inward. This block contains
        // information on the characteristics of the signals as they are
        // used in the test mode (myN in this case).
        Internal {
```

```

// Signals y and z both are of type TestMode. That is they are
// to remain constant after the configuration is established
// until termination of the test mode. The establish mode
// pattern would have left these signals at a logic-0 state.
'y+z' {
  DataType TestMode {
    // Both signals y and z are to be held to a Logic-0
    ActiveState ForceDown;
  }
} // end 'y+z'
} // end Internal

// Container for Pattern Related Information
PatternInformation {
  // Look for a Pattern P1 outside the Environment. That Pattern
  // is used to Establish this Normal test mode. The Pattern would
  // be written with 1450.6's syntax that restricts patterns to
  // have calls to a sequence (Macro or Procedure) and pass
  // the necessary data to it. If it did not follow the CTL rules
  // for Patterns, it would be identified as a Foreign pattern.
  // Pattern P1 uses Macro setupseq. The complete execution of
  // P1 takes 1 clock period.
  Pattern P1 {
    Purpose EstablishMode;
    Protocol Macro setupseq;
    CycleCount 1;
  }
} // end PatternInformation
} // end CTLMode myN

// The test mode that makes the boundary scan chain of the design
// configured to be passively involved in the test of logic outside
// the embedded design. y=0, z=1.
CTLMode myE {
  // The design is setup to be outward facing.
  TestMode ExternalTest;

  // Information for the ExternalTest test mode on the signals
  Internal {
    // Signal y is expected to remain constant after the test mode
    // is established until termination of the test mode.
    // y is expected to remain at a logic-0. The patterns that
    // establish this configuration would have set y to a logic-0.
    y {
      DataType TestMode {
        ActiveState ForceDown;
      }
    }
    // Signal z is expected to remain constant after the test mode
    // is established and before the termination of the test mode.
    // z is expected to remain at a logic-1. The patterns that
    // establish this configuration would have set z to a logic-1.
    z {
      DataType TestMode {

```

```

        ActiveState ForceUp;
    }
}
} // end Internal

// The container of Pattern information for the test mode.
PatternInformation {
    // Look for Pattern P2 defined outside this environment. That
    // pattern follows the CTL rules and is labeled
    // as a pattern that is to be used to get the design into the
    // outward-facing test mode. Pattern P2 only uses Macro setupseq.
    // The complete execution of P2 takes 1 clock period.
    Pattern P2 {
        Purpose EstablishMode;
        Protocol Macro setupseq;
        CycleCount 1;
    }
} // end PatternInformation
} // end CTLMode myE

// The test mode that defines the configuration that is to be used
// to test the design. This test mode would typically also include
// test patterns. However, for this example, we do not require them
// to be defined. This configuration requires y=1 and z=0.
CTLMode myI {
    // the test mode to test the internals of this design.
    TestMode InternalTest;

    // the container for information on signals.
    Internal {

        // y is expected to remain constant logic-1 after the test mode
        // has been established to maintain the configuration.
        y {
            DataType TestMode {
                ActiveState ForceUp;
            }
        }
        // z is expected to remain constant logic-0 after the test mode
        // has been established to maintain the configuration.
        z {
            DataType TestMode {
                ActiveState ForceDown;
            }
        }
    } // end Internal

    // the container for Pattern information
    PatternInformation {

        // Look for Pattern P3 defined outside this environment. That
        // pattern follows the CTL rules and is labeled
        // as a pattern that is to be used to get the design into the
        // outward-facing test mode. Pattern P3 only uses Macro setupseq.

```

```

    // The complete execution of P3 takes 1 clock period.
    Pattern P3 {
        Purpose EstablishMode;
        Protocol Macro setupseq;
        CycleCount 1;
    }
} // end PatternInformation
} // end CTLMode myI
} // end Environment design

// Definition of Patterns P1, P2, and P3. Note that the pattern follows
// the definition defined in this document. That is, Patterns must only
// call sequences (Protocols—Macros and Procedures). Patterns
// pass data to the sequences. Patterns must call only one type of
// sequence, perhaps multiple times, as limited by the syntax. The name
// of the sequence is defined by the Protocol statement in the invoking
// PatternBurst or the PatternInformation block of statements. In this
// example there is no invoking PatternBurst; hence, the Protocol
// statement must exist in the PatternInformation. In this example the
// Protocol invoked is a Macro called setupseq for all patterns.

// Note that these patterns are not included in any PatternExec or
// PatternBurst. This is legal CTL. The information did not need the
// PatternExec and PatternBurst constructs. In this example P1, P2, and
// P3 are used to define the patterns that are to be executed to
// establish the test mode.
Pattern P1 {
    P { y=0; z=0; }
}
Pattern P2 {
    P { y=0; z=1; }
}
Pattern P3 {
    P { y=1; z=0; }
}

```

Let us step through the thinking process that led to writing the above CTL. We need to describe test modes for the design. Thus, we need an Environment block of statements (Environment {}) with three CTLMode blocks of statements one for every test mode. Each CTLMode block is labeled with the type of configuration it falls under with a TestMode statement. Each test mode has an initialization sequence that sets *y* and *z* to certain values. This would be written by creating a reference to an initialization pattern in the appropriate CTLMode block and writing the patterns outside. The patterns rely on infrastructure such as Macros, Timing, and Signals. After establishing the test mode, some signals need to remain constant. This is indicated with the DataType-ActiveState statement. To write this statement, the Signals need to be defined in the Signals block of statements. All constructs written outside the CTLMode blocks, if referred by this block, should be in the scope of the CTLMode block of statements. This is done by the DomainReferences statement. Common information across all modes can be collected in the nameless CTLMode block of statements.

Now let us step through the example in the reverse direction and see if what is described says what we expect it to say. Start with the Environment. The Environment has one common CTLMode block and three CTLMode blocks for each test mode defined. Every test mode has some sequence defined to establish its configuration. The test modes are established by executing patterns that are identified in their appropriate

PatternInformation blocks. The configurations are maintained by holding signals y and z to their appropriate values as specified by the ActiveStates of the TestMode data types.

Thus, in this example, the party that created the CTL and the party that interprets the CTL would have the same information about the design.

4.3 CTL for structural information

We use the same design of the previous example to show how scan chains and a subdesign are represented in CTL.

In this subclause, the two scan chains of the example design are described in CTL. One scan chain is a boundary scan chain, and the other scan chain is an internal scan chain of the design. A subdesign (the shaded portion of the design) as an embedded hierarchy is recognized, and the relationship between scan cells and the hierarchy is maintained. The example describes the fact that the subdesign comes with a single scan chain with two scan cells and its construction as part of the scan chains of the complete design.

Although the scan chains may be available in some modes and not others, in this example, the information is made test mode independent:

```
STIL 1.0 {
    Design 2005;
    CTL 2005;
}

// The information requires the recognition of the
// boundary of the design. This is described in the Signals block outside
// the environment. This is the same as that defined in the previous
// example.
Signals {
    a In { DefaultState Z; }
    b Out;
    y In { DefaultState Z; }
    z In { DefaultState Z; }
    SE In { DefaultState Z; }
    BSE In { DefaultState Z; }
    CLK In { DefaultState D; }
    BCK In { DefaultState D; }
    SI In { DefaultState Z; ScanIn 2; }
    SO Out { ScanOut 2; }
    BSI In { DefaultState Z; ScanIn 2; }
    BSO Out { ScanOut 2; }
} // end Signals

// A hierarchy is defined for the shaded area. This hierarchy is
// recognized to have some inputs and outputs and a scan chain. The
// scan chain is of length 2 with scan cells i[0..1] (this expands
// to be a list of cells i[0], i[1]). The first cell in the list i[0] is
// closer to si and i[1] is closer to so. The syntax for CoreType is
// defined in this document. However, it reuses the syntax for Signals
// and ScanStructures from 1450.0 and 1450.1.
CoreType subDesign {
    Signals {
```

```

    si In;
    so Out;
    se In;
    clk In;
  }
  ScanStructures {
    ScanChain twoCellChain {
      ScanLength 2;
      ScanIn si;
      ScanOut so;
      ScanCells i[0..1];
      ScanEnable se;
      ScanMasterClock clk;
    }
  }
} // end CoreType subDesign

// A single instance of the design is defined. This instance is called
// shadedRegion (see Figure 4). As a result, the signals
// of shadedRegion are shadedRegion:si, shadedRegion:so, shadedRegion:se
// and shadedRegion:clk. The scan cells of this hierarchy are hence
// shadedRegion:i[0..1]. If a domain name was used for the scan
// structures in the CoreType definition, the name of the scan cells
// would include the domain-name from the scan structures to look like:
// CORE_INSTANCE_NAME:DOMAIN_NAME::SCAN_CELL_NAMES
CoreInstance subDesign {
  shadedRegion;
}

// The scan chains of the design are defined in the ScanStructures. In
// this example, one of the scan chains is constructed out of scan cells
// of the embedded hierarchy. The syntax used here is defined in 1450.0
// and 1450.1.
ScanStructures {
  ScanChain bc1 {
    ScanIn BSI;
    ScanOut BSO;
    ScanLength 2;
    ScanEnable ~BSE;
    ScanCells c[0..1];
    ScanMasterClock BCK;
  }
  ScanChain ic1 {
    ScanIn SI;
    ScanOut SO;
    ScanLength 2;
    ScanCells shadedRegion:i[0..1];
    ScanEnable SE;
    ScanMasterClock CLK;
  }
} // end nameless ScanStructures

// Values in CTL are waveform characters that are defined in the timing
// block as events (D, U, N, L, H, X) associated with timing within

```

```
// a clock period. Details of the Timing block should be obtained from
// 1450.0. Since the timing block is a named block it needs to be
// explicitly brought into the scope of the test modes. This would
// be done through a statement in the DomainReferences. The timing
// block is needed to support the definition of the sequences that
// configure the design into the test mode. The design is configured
// into a test mode by a sequence of clock periods as defined by the
// Macros and the values are waveform characters that are supplied by
// the patterns to the Macros.
Timing T1 {
  WaveformTable W2 {
    Period '100ns';
    Waveforms {
      'a+y+z+SE+BSE+CLK+BCK+SI+BSI' { 01x {'0ns' D/U/N;}}
      'CLK+BCK' { P {'0ns' D; '50ns' U; '60ns' D;}}
      'b+SO+BSO' { 01x {'0ns' L/H/X;}}
    }
  }
} // end Timing

// Some protocols take in parameters. These parameters can be provided
// as values on signals, signal groups, or variables. In this example
// the variables used are defined below. Note these variables are defined
// in a nameless Variables block. Thus, the variables are global
// across the information in every test mode. If the Variables had a
// domain name associated with it, then the DomainReferences block of
// statements would have to be used in the CTLMode{}.
Variables {
  SignalVariable bsivals[1..0];
  SignalVariable bsovals[1..0];
  SignalVariable sivals[1..0];
  SignalVariable sovals[1..0];
}

// All protocols that need to be described are defined in the MacroDefs
// blocks. In this example there are four protocols that are written in
// an unnamed MacroDefs block (global to all test modes).
MacroDefs {
  // scanbsi takes in variables and applies it as a shift operation to
  // BSI and BSO. The shift register gets bsivals[1..0] and
  // simultaneously bsovals[1..0] are observed at BSO.
  scanbsi {
    W W2;
    C {BSE=0; BCK=0; bsivals=#; bsovals=#;}
    Shift { V { BSI= \W bsivals[1..0]; BSO=\W bsovals[1..0]; BCK=P;}}
  }
  // scansi is similar to scanbsi but defines the sequence through which
  // values in variables sivals[1..0] are applied to scan cells through
  // SI, and values in variables sovals[1..0] are observed from scan
  // cells through signal SO.
  scansi {
    W W2;
    C {SE=1; CLK=0; sivals=#; sovals=#;}
    Shift { V { SI=\W sivals[1..0]; SO= \W sovals[1..0]; CLK=P;}}
  }
}
```

```

    }
    // MCaptureBCK is a sequence that pulses the BCK clock to capture
    // values through the functional path (BSE=1).
    MCaptureBCK {
        W W2;
        V { BSE=1; BCK=P;}
    }
    // MObserveConnection is a sequence that allows for data to be
    // captured and observed through the BSO signal.
    MObserveConnection {
        W W2;
        Macro MCaptureBCK;
        Macro scanbsi;
    }
} // end nameless MacroDefs

// The block of statements that contains the primary information in CTL.
// If one was reading this example, they should start from here.
Environment design {
    // A nameless CTLMode block of statements carries test mode-
    // independent information.
    CTLMode {

        // The named domain for timing information needs to be brought into
        // the scope of the information in this example. If this statement
        // is missing, no information in the scope of this CTLMode block of
        // statements should refer to waveforms in T1. If this statement
        // is missing in this example, an error would occur when the Macros
        // of this CTLMode {} are to be interpreted.
        DomainReferences {
            Timing T1;
        }

        // A container for information on the signals of the design.
        Internal {

            // Information on signal "a" is to follow. Another block of signal
            // "a" cannot occur in this CTLMode{}. Signal "a" can be part of
            // another named group in this CTLMode{}. The same is true for all
            // other named signals and signal-group names in this CTLMode{}.
            a {

                // Values on signal a are captured in scan cell c[0]. The
                // clock that captures values in the state element c[0] is BCK.
                // Values are captured on the LeadingEdge of BCK, and c[0] takes
                // on a value from the Connection when this event occurs. To
                // observe values through the connection, one would have to
                // execute the Macro MObserveConnection. MObserveConnection
                // should be in the scope of information available in this
                // CTLMode{}.
                IsConnected In {
                    StateElement Scan c[0];
                    CaptureClock BCK {
                        LeadingEdge; StateAfterEvent Connection;
                    }
                }
            }
        }
    }
}

```

```
    }  
    TestAccess Observe Macro MObserveConnection;  
  }  
}  
b { // information on b.  
  
  // This is similar to the connection described on signal a.  
  // In this case signal b is on the receiving end of the  
  // connection, and it receives values from scan cell c[1].  
  // The clock that affects the values in c[1] is BCK. After  
  // the leading edge of BCK, c[1] would go to an Unknown state.  
  // Values are put into c[1] by executing the macro scanbsi that  
  // is in the scope of the current CTLMode{ }.  
  IsConnected Out {  
    StateElement Scan c[1];  
    LaunchClock BCK {  
      LeadingEdge; StateAfterEvent ExpectUnknown;  
    }  
    TestAccess Control Macro scanbsi;  
  }  
}  
BSE { // information on BSE  
  // BSE is the test control signal. Specifically it is a  
  // ScanEnable that enables the scan configurations with a  
  // logic-0.  
  DataType TestControl ScanEnable { ActiveState ForceDown; }  
}  
SE { // information on SE  
  // SE is a test control signal. Specifically it is a ScanEnable  
  // that enables the scan configuration with a logic-1.  
  DataType TestControl ScanEnable { ActiveState ForceUp; }  
}  
'BCK+CLK' { // information on BCK and CLK  
  // both BCK and CLK are test control signals. They are clocks  
  // used for Scan Operations and Capture operations. All  
  // sequences in this CTLMode { } assume that these clocks are  
  // at a logic-0 at the beginning of every protocol.  
  DataType TestControl CaptureClock ScanMasterClock  
    { AssumedInitialState ForceDown; }  
}  
BSI { // information on BSI  
  // BSI is a scan input. Since no explicit connection is defined  
  // using the IsConnected statement, the connection from  
  // the BSI to the first scan cell can be obtained from the  
  // scan chain definition.  
  // BSI is connected internally in the design as a scan-in  
  // where the connection is defined in the Scan chains in the  
  // scope of this CTLMode { }. Looking at scan chain bc1, BSI is  
  // connected to scan cell c[0]. The connection is enabled by  
  // setting BSE to a logic-0.  
  // BSI is of the general type TestData and specifically receives  
  // scan data values. BSI is the scan input of the boundary scan  
  // chain.  
  DataType TestData ScanDataIn { ScanDataType Boundary; }
```

```

    }
    BSO { // information on BSO
        // Similar to BSI, BSO's connection is defined by the scan
        // chain. BSO is connected internally to the design to c[1]
        // where BSO is at the receiving end of the connection. There
        // is no gating logic between c[1] and BSO as no Enabling
        // condition exists for the connection. BSO is of type test
        // data and is a scan-out for a boundary scan chain of the
        // design.
        DataType TestData ScanDataOut { ScanDataType Boundary; }
    }
    // Similar to BSI and BSO, the SI and SO are defined as scan
    // signals that are used for internal scan chains of the design.
    SI { DataType TestData ScanDataIn { ScanDataType Internal; } }
    SO { DataType TestData ScanDataOut { ScanDataType Internal; } }
} // end Internal
PatternInformation {

    // If one needs to operate scan chains in this configuration,
    // scanbsi can be used to control and observe values in scan chain
    // bcl and scansi can be used to control and observe values in
    // scan chain icl.
    Macro scanbsi {
        Purpose ControlObserve; ScanChain bcl;
    }
    Macro scansi {
        Purpose ControlObserve; ScanChain icl;
    }
} // end PatternInformation
} // end CTLMode
} // end Environment

```

Several things that needed to be described in this example were associated with the scan chains of the design. The information is provided through several statements. Scan chains can be defined only in the scan structures. Connections from the boundary of the design to the scan cells are defined through the `IsConnected` statement in the environment. Various aspects about the signals and the connections are embedded along with the connection information. From the information there should be no doubt about the relationship between scan cells and the hierarchy called *shadedRegion*.

The reader should perform a similar exercise to that of the previous example and see that the constructs used are the only mechanisms in place for the information that can be described. Reading the CTL should give the reader a crisp view of the information that was described in the paragraph before the example.

4.4 CTL for test pattern information

This example uses the same design as the previous examples.

The intent of this example is to describe test patterns in CTL. The design is to be configured in its internal test mode such that the test patterns can be applied for testing the manufactured product. The details of getting into the internal test configuration are the same as described in the example that described test modes

for the design. That is, signals y and z of the design are to be set to logic-1 and logic-0, respectively. The test patterns are a collection of stimulus and responses that are applied on a cycle-by-cycle basis to the inputs and outputs of the design. The design has 100 stuck-at faults when every gate of the design is faulted. Of these faults, 99 are detected by the tests. Just to show a feature of CTL that would not be highlighted otherwise, the information requires the event during which the test patterns capture values through the functional path of the design be highlighted in the information.

In the explanations of this example, statements will be made to highlight certain features of the test patterns relating to the restrictions CTL puts on STIL syntax.

```
STIL 1.0 {  
    Design 2005;  
    CTL 2005;  
}  
  
// The information requires the recognition of the  
// boundary of the design. This is described in the Signals block outside  
// the environment. This is the same as that defined in the previous  
// example.  
Signals {  
    a In { DefaultState Z; }  
    b Out;  
    y In { DefaultState Z; }  
    z In { DefaultState Z; }  
    SE In { DefaultState Z; }  
    BSE In { DefaultState Z; }  
    CLK In { DefaultState D; }  
    BCK In { DefaultState D; }  
    SI In { DefaultState Z; ScanIn 2; }  
    SO Out { ScanOut 2; }  
    BSI In { DefaultState Z; ScanIn 2; }  
    BSO Out { ScanOut 2; }  
} // end Signals  
  
// Some signal groups are defined for the design for ease of use in  
// writing the patterns.  
SignalGroups {  
    Ins[0..3] = 'BSE+SE+SI+BSI';  
    Clocks[0..1] = 'BCK+CLK';  
    Enables[0..1] = 'BSE+SE';  
    Outs[0..1] = 'SO+BSO';  
} // end SignalGroups  
  
// Some variables are defined for the parameters of the Macros that are  
// written for the test patterns.  
Variables {  
    SignalVariable aval; // to carry only a single value for signal a.  
    SignalVariable bval; // to carry only a single value for signal b.  
    SignalVariable sivals[1..0]; // to carry stimulus for the scan cells  
    SignalVariable sovals[1..0]; // to carry response data for scan cells  
} // end Variables
```

```
// Some default timing data that are referenced by Macros. This is the
// same as those used in the previous examples. For details refer to
// 1450.0 and 1450.1.
```

```
Timing T1 {
  WaveformTable W2 {
    Period '100ns';
    Waveforms {
      Clocks[0..1] { 0P{'0ns' D/D; '60ns' D/U; '65ns' D/D;}}
      Ins[0..3] { 01x {'0ns' D/U/N;}}
      Outs[0..1] { 01x {'0ns' L/H/X;}}
    } // end Waveforms
  } // end WaveformTable W2
  WaveformTable W1 {
    Period '100ns';
    Waveforms {
      'a+y+z+SE+BSE+CLK+BCK+SI+BSI' { 01x {'0ns' D/U/N;}}
      'b+SO+BSO' { 01x {'0ns' L/H/X;}}
    } // end Waveforms
  } // end WaveformTable W1
} // end Timing T1
```

```
// The Macros that are written to support the patterns. CTL syntax
// disallows waveform information in patterns. CTL requires the
// data portion of the tests to be in the patterns and the
// protocols contain the sequencing information. With this restriction
// the Macros would have to define the Waveform tables used. Two macros
// are written in this example, one to support the test pattern used
// to detect the faults and one to support the pattern (seq) that
// is written to establish the internal test mode of the design
// (setupseq).
```

```
MacroDefs forInternalTest {

  // seq is a sequence that takes in the following parameters.
  // aval, sivals[1..0], bval and sovals[1..0]
  // This sequence first sets up the scan configuration and
  // shifts in aval into the boundary scan chain and sivals[1..0]
  // into the internal chain of shadedRegion. Note this information
  // does not require the definition of Scan Cells. Then a labeled
  // statement exists where the clocks are being pulsed. Finally,
  // the scan chains are configured to shift and the values in the
  // wrapper chain and internal chain are observed.
  seq {
    W W2;
    C {aval=#; sivals=#; bval=#; sovals=#;}
    C {Enables[0..1]=01; Clocks[0..1]=00; Ins[2..3]=xx;}
    Shift {V {BSI= \W aval; SI= \W sivals[1..0]; Clocks[0..1]=PP;}}
    mylabel: V { Enables[0..1]=10; Clocks[0..1]=PP;}
    V { Enables[0..1]=01; Clocks[0..1]=00;}
    Shift {V{BSO=\W bval; SO=\W sovals[1..0]; Clocks[0..1]=PP;}}
  } // end seq

  // This macro is used to apply values to y and z.
  setupseq {
    W W1;
```

```
    V { y=#; z=#;}
  } // end setupseq
} // end MacroDefs forInternalTest

// The order in which the Patterns will be applied as it relates to the
// patterns that is defined to establish the test mode. This is the
// schedule of the patterns.
PatternBurst pats {
  // The following MacroDefs must be in the DomainReferences of the
  // associated CTLMode block that invokes the pattern burst.
  MacroDefs forInternalTest;
  PatList {
    // If a Protocol statement exists in the PatternInformation for P3
    // and all_pats the information must be consistent.
    P3 {Protocol Macro setupseq;}
    all_pats {Protocol Macro seq;}
  }
} // end PatternBurst pats

// The pattern exec that contains the test patterns for the design.
PatternExec topPat {
  Timing T1;
  PatternBurst pats;
} // end PatternExec topPat

// The primary block of information that is being written for this
// example. A test mode is defined to be of type internal test. The
// patterns that relate to the test mode are referred to in the test mode
// but the patterns are defined outside. Since CTL follows the same
// rules for "define before referencing" as STIL, the pattern-exec
// and Pattern-Burst blocks are written before the Environment block
// of statements. The patterns are labeled as Production patterns,
// and the fault coverage is specified in the environment. The Macro
// is identified to be used to apply non-overlapped test patterns
// in this test mode. The statement with a label is identified to
// be the time when values are Captured into the memory elements
// through the functional paths.
Environment design {
  // Internal test mode of the design that has its patterns and
  // establishes the testmode sequence. Details of y and z remaining
  // constant for the validity of the test mode are not required
  // in the information that needs to be specified. However, if it
  // does need to be specified, the DataType syntax would have to
  // be used from the first example.
  CTLMode myI {
    // this test mode allows for the internals of the design to be
    // tested.
    TestMode InternalTest;

    // the named domains need to be brought into the scope of the
    // information in this test mode.
    DomainReferences {
      Timing T1;
      MacroDefs forInternalTest;
    }
  }
}
```

```

    }

    // All the patterns to be applied in the test mode are identified
    // in the PatternInformation. There could have been many more
    // patterns written outside, and only some of them are relevant
    // to the test mode.
    PatternInformation {

        // Of all the pattern execs, topPat contains the patterns that
        // are to be used for production testing. Within PatternExec,
        // of all the bursts that may exist, only pats contains all
        // the test patterns. The fault coverage of the tests is
        // 99% single-stuck at coverage of faults.
        PatternExec topPat {
            Purpose Production;
            PatternBurst pats;
            Fault {
                Type StuckAt UnCollapsed;
                FaultCount 100;
                FaultsDetected 99;
            }
        }

        // Macro seq is identified to be used in this test mode to
        // apply test pattern sequences that are not overlapped. mylabel
        // in seq is a Capture event that occurs during the clock period
        // associated with the statement.
        Macro seq {
            // also could have used forInternalTest::seq
            // Since there is no ambiguity in using the shorter name seq,
            // both names resolve to the same Macro. seq is the only macro
            // with that name in all the named MacroDefs blocks active in
            // the current test mode.
            Purpose DoTest;
            Identifiers {
                EventType Capture {
                    Label myLabel { Complete; During; }
                }
            }
        }

        // Pattern P3 is used to establish the test mode. It uses macro
        // setupseq. Note that setupseq is unique in the active
        // MacroDefs. If it was not unique, the macro name would have been
        // forInternalTest::setupseq
        Pattern P3 { Purpose EstablishMode; Protocol Macro setupseq; }

        // Pattern all_pats applies scan patterns and uses macro seq.
        Pattern all_pats { Purpose Scan; Protocol Macro seq; }
    } // end PatternInformation
} // end CTLMode myI
} // end Environment design

// Definition of pattern P3. It supplies data to the Macro setupseq.

```

```
Pattern P3 {  
    P {Y=1; z=0;}  
}  
  
// Definition of pattern all_pats. It supplies data through variables  
// in this example. Note patterns only call protocols and supply the  
// data to the protocols. CTL syntax does not allow for V statements  
// in Patterns. In this example all_pats is defined to use a Protocol  
// that is defined as a Macro named seq.  
Pattern all_pats {  
    P {aval=1; sivals[1..0]=10; bval=0; sovals[1..0]=11;}  
    P {aval=1; sivals[1..0]=10; bval=0; sovals[1..0]=11;}  
    // many more calls to Macro seq (see Protocol statement for name).  
} // end all_pats
```

The example shows test patterns that are provided in CTL. Although no information was provided on Macro setupseq in this example, it could have been referenced in the PatternInformation block as a sequence with a ModeControl purpose and to be used by patterns that establish the test mode.

4.5 Beyond the examples

The examples describe a portion of the constructs in CTL that could be used to convey information to the core user. The keywords in the examples are a limited subset of the keywords available in the language. CTL can be used to describe digital information for many designs and different test methodologies. Combined with the number of use models for the CTL information, it is impossible to describe every scenario. With the examples described, the reader is expected to understand the basic mechanism behind writing CTL. With that understanding and the definitions of the statements allowed, the reader should be able to describe test information for digital ICs.

The reader should be aware that this standard has restricted STIL in certain ways by defining extensions to 1450.0 and 1450.1. As a result, CTL patterns are different from patterns that are written using just the syntax defined in IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002. The syntax and restrictions on patterns defined in this standard are very important for the reusability of test patterns in SoC flows where test patterns are mapped from the core boundary to the SoC boundary. The mechanism supported by CTL allows for protocols to be changed or replaced without having access to or modifying the bulk of the test pattern data, which lies in the Pattern blocks. The reader should also be sensitive to a basic difference between STIL (1450.0, 1450.1, and 1450.2 syntax) and CTL (1450.6, 1450.0, 1450.1, and 1450.2 syntax) that exists:

- In STIL the Environment block is a user defined block that provides some more information about the patterns. In CTL the Environment is the primary block of information, it utilizes constructs outside its environment to complete its information that is partitioned across test modes.
- CTL restricts the grouping of signals and variables to allow for explicitness and bit-indexing capabilities when partial load and unload operations are performed on the scan chains.
- CTL restricts pattern constructs to separate the data and protocol portions of patterns for reusability of tests.
- CTL understands the concept of hierarchy through core-instances.

During the design of CTL, care has been taken to define a single mechanism to describe any given construct. For example, scan cells can only be defined in ScanStructures. Hierarchy can only be represented in CoreTypes and CoreInstances. However, depending on the information to be described, the scan chains can be written to show or not show the hierarchy. The choices will lie in the user flows being constructed.

Similarly the informational needs may spread the information to be described in single or multiple CTL test modes (CTLMode blocks). However, there is only one way to describe a DataType of a signal.

5. Extensions to IEEE Std 1450-1999 and IEEE Std 1450.1-2005

5.1 STIL name spaces and name resolution

The special characters “:” and “::” as defined in 1450.1 are used to concatenate two user-defined names that follow 6.8 of IEEE Std 1450-1999. With this addition, any entity within a domain can optionally be identified across all constructs in CTL (inside or outside the Environment) as D::E, where D is the domain name and E is the entity name. The prior mechanisms allowed by IEEE Std 1450-1999 for identifying the entities without the “:” as defined in 6.16 of IEEE Std 1450-1999 remain when no ambiguity exists in the identification of the entity. Similarly, with this addition, the “::” operator can be used to concatenate the core instance name to the entity name (E or D::E) to identify entities of the core instance. The names in this case take on the form C:E or C:D::E. In each case, C, D, and E are required to follow the naming conventions defined in 6.8 of IEEE Std 1450-1999. The fully expanded name of an entity is required to be unique across all entities of the same type.

Thus, this extension allows for a scan cell name c1 that is defined in CoreInstance i1 to be used as part of the scan chain definition of a ScanStructures outside the Core. This cell would be identified as i1:c1 when used in a ScanStructure as defined by IEEE Std 1450-1999 or IEEE Std 1450.1-2005. This extension allows any signal groupname used in a pattern to be referenced along with its domain name with the “::” separator.

5.2 Optional statements of IEEE Std 1450-1999

IEEE Std 1450-1999 is not specific about the optional and required blocks. CTL’s usage of the constructs requires that all constructs be optional. The statements as identified in Table 7 and Table 8 of IEEE Std 1450-1999 are all optional in CTL. The statements are used when required to support the informational needs of the CTL being written. When the statements are used the define before use ordering as required by STIL still remains for these statements (also see 9.3 of IEEE Std 1450-1999).

Although all statements in CTL are optional, it is very difficult to write a typical CTL file without a Signals block and the Environment block. Furthermore, test patterns that are targeted for execution require the existence of the PatternExec statement and a PatternBurst statement. Thus, there is a requirement that if a Pattern statement exists outside of the EstablishMode or TerminateMode patterns in the CTL mode being described, then a PatternExec and PatternBurst must exist that invoke that Pattern.

5.3 Restricting the usage of SignalGroup and variable names

Values that are passed into Macros and Procedures could be SignalGroups or Variables. When the non-bussed representation (without []) is used to provide scan data, the data are assumed to represent a complete shift operation (full operation of the scan chain). When partial scan operations are performed, the syntax for the parameters (signal-group or variable) is required to use the bussed naming convention in the protocols (Macros and Procedures) to explicitly define the number of shift operations in the load-unload of the scan chain.