

IEEE Standard for Verilog® Hardware Description Language

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

1364™

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

7 April 2006

IEEE Std 1364™-2005
(Revision of IEEE Std 1364-2001)

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS

Not for Resale

and Electronics Engineers, Inc.
IEEE
nitted without license from IHS
Not for Resale

IEEE Std 1364™-2005
(Revision of IEEE Std 1364-2001)

IEEE Standard for Verilog® Hardware Description Language

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Abstract: The Verilog hardware description language (HDL) is defined in this standard. Verilog HDL is a formal notation intended for use in all phases of the creation of electronic systems. Because it is both machine-readable and human-readable, it supports the development, verification, synthesis, and testing of hardware designs; the communication of hardware design data; and the maintenance, modification, and procurement of hardware. The primary audiences for this standard are the implementors of tools supporting the language and advanced users of the language.
Keywords: computer, computer languages, digital systems, electronic systems, hardware, hardware description languages, hardware design, HDL, PLI, programming language interface, Verilog, Verilog HDL, Verilog PLI

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 7 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.

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

Print: ISBN 0-7381-4850-4 SH95395
PDF: ISBN 0-7381-4851-2 SS95395

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.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

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. 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. Comments on standards and requests for interpretations should be addressed to:

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

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Introduction

This introduction is not a part of IEEE Std 1364-2005, IEEE Standard for Verilog® Hardware Description Language.

The Verilog hardware description language (HDL) became an IEEE standard in 1995 as IEEE Std 1364-1995. It was designed to be simple, intuitive, and effective at multiple levels of abstraction in a standard textual format for a variety of design tools, including verification simulation, timing analysis, test analysis, and synthesis. It is because of these rich features that Verilog has been accepted to be the language of choice by an overwhelming number of integrated circuit (IC) designers.

Verilog contains a rich set of built-in primitives, including logic gates, user-definable primitives, switches, and wired logic. It also has device pin-to-pin delays and timing checks. The mixing of abstract levels is essentially provided by the semantics of two data types: nets and variables. Continuous assignments, in which expressions of both variables and nets can continuously drive values onto nets, provide the basic structural construct. Procedural assignments, in which the results of calculations involving variable and net values can be stored into variables, provide the basic behavioral construct. A design consists of a set of modules, each of which has an input/output (I/O) interface, and a description of its function, which can be structural, behavioral, or a mix. These modules are formed into a hierarchy and are interconnected with nets.

The Verilog language is extensible via the programming language interface (PLI) and the Verilog procedural interface (VPI) routines. The PLI/VPI is a collection of routines that allows foreign functions to access information contained in a Verilog HDL description of the design and facilitates dynamic interaction with simulation. Applications of PLI/VPI include connecting to a Verilog HDL simulator with other simulation and computer-assisted design (CAD) systems, customized debugging tasks, delay calculators, and annotators.

The language that influenced Verilog HDL the most was HILO-2, which was developed at Brunel University in England under a contract to produce a test generation system for the British Ministry of Defense. HILO-2 successfully combined the gate and register transfer levels of abstraction and supported verification simulation, timing analysis, fault simulation, and test generation.

In 1990, Cadence Design Systems placed the Verilog HDL into the public domain and the independent Open Verilog International (OVI) was formed to manage and promote Verilog HDL. In 1992, the Board of Directors of OVI began an effort to establish Verilog HDL as an IEEE standard. In 1993, the first IEEE working group was formed; and after 18 months of focused efforts, Verilog became an IEEE standard as IEEE Std 1364-1995.

After the standardization process was complete, the IEEE P1364 Working Group started looking for feedback from IEEE 1364 users worldwide so the standard could be enhanced and modified accordingly. This led to a five-year effort to get a much better Verilog standard in IEEE Std 1364-2001.

With the completion of IEEE Std 1364-2001, work continued in the larger Verilog community to identify outstanding issues with the language as well as ideas for possible enhancements. As Accellera began working on standardizing SystemVerilog in 2001, additional issues were identified that could possibly have led to incompatibilities between Verilog 1364 and SystemVerilog. The IEEE P1364 Working Group was established as a subcomittee of the SystemVerilog P1800 Working Group to help ensure consistent resolution of such issues. The result of this collaborative work is this standard, IEEE Std 1364-2005.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

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

At the time this standard was completed, the IEEE P1364 Working Group had the following membership:

Johny Srouji, IBM, *IEEE SystemVerilog Working Group Chair*
Tom Fitzpatrick, Mentor Graphics Corporation, *Chair*
Neil Korpusik, Sun Microsystems, Inc., *Co-chair*
Stuart Sutherland, Sutherland HDL, Inc., *Editor*
Shalom Bresticker, Intel Corporation, *Editor through February 2005*

The Errata Task Force had the following membership:

Karen Pieper, Synopsys, Inc., *Chair*

Kurt Baty, WFSDB Consulting
Stefen Boyd, Boyd Technology
Shalom Bresticker, Intel Corporation
Dennis Brophy, Mentor Graphics Corporation
Cliff Cummings, Sunburst Design, Inc.
Charles Dawson, Cadence Design Systems, Inc.
Tom Fitzpatrick, Mentor Graphics Corporation
Ronald Goodstein, First Shot Logic Simulation and Design
Mark Hartoog, Synopsys, Inc.
James Markevitch, Evergreen Technology Group
Dennis Marsa, Xilinx
Francoise Martinolle, Cadence Design Systems, Inc.
Mike McNamara, Verisity, Ltd.
Don Mills, LCDM Engineering
Anders Nordstrom, Cadence Design Systems, Inc.
Karen Pieper, Synopsys, Inc.
Brad Pierce, Synopsys, Inc.
Steven Sharp, Cadence Design Systems, Inc.
Alec Stanculescu, Fintronic USA, Inc.
Stuart Sutherland, Sutherland HDL, Inc.
Gordon Vreugdenhil, Mentor Graphics Corporation
Jason Woolf, Cadence Design Systems, Inc.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The Behavioral Task Force had the following membership:

Steven Sharp, Cadence Design Systems, Inc., *Chair*

Kurt Baty, WFSDB Consulting
Stefen Boyd, Boyd Technology
Shalom Bresticker, Intel Corporation
Dennis Brophy, Mentor Graphics Corporation
Cliff Cummings, Sunburst Design, Inc.
Steven Dovich, Cadence Design Systems, Inc.
Tom Fitzpatrick, Mentor Graphics Corporation
Ronald Goodstein, First Shot Logic Simulation and Design
Keith Gover, Mentor Graphics Corporation
Mark Hartoog, Synopsys, Inc.
Ennis Hawk, Jeda Technologies
Atsushi Kasuya, Jeda Technologies
Jay Lawrence, Cadence Design Systems, Inc.
Francoise Martinolle, Cadence Design Systems, Inc.
Kathryn McKinley, Cadence Design Systems, Inc.
Michael McNamara, Verisity, Ltd.
Don Mills, LCDM Engineering
Mehdi Mohtashemi, Synopsys, Inc.
Karen Pieper, Synopsys, Inc.
Brad Pierce, Synopsys, Inc.
Dave Rich, Mentor Graphics Corporation
Steven Sharp, Cadence Design Systems, Inc.
Alec Stanculescu, Fintronic, USA
Stuart Sutherland, Sutherland HDL, Inc.
Gordon Vreugdenhil, Mentor Graphics Corporation

The PLI Task Force had the following membership:

Charles Dawson, Cadence Design Systems, Inc., *Chair*
Ghassan Khoory, Synopsys, Inc., *Co-chair*

Tapati Basu, Sysnopsys, Inc.
Steven Dovich, Cadence Design Systems, Inc.
Ralph Duncan, Mentor Graphics Corporation
Jim Garnett, Mentor Graphics Corporation
Joao Geada, CLK Design Automation
Andrzej Litwiniuk, Synopsys, Inc.
Francoise Martinolle, Cadence Design Systems, Inc.
Sachchidananda Patel, Synopsys, Inc.
Michael Rohleder, Freescale Semiconductor, Inc.
Rob Slater, Freescale Semiconductor, Inc.
John Stickley, Mentor Graphics Corporation
Stuart Sutherland, Sutherland HDL, Inc.
Bassam Tabbara, Novas Software, Inc.
Jim Vellenga, Cadence Design Systems, Inc.
Doug Warmke, Mentor Graphics Corporation

In addition, the working group wishes to recognize the substantial efforts of past contributors:

Michael McNamara, Cadence Design Systems, Inc.,
1364 Working Group past chair (through September 2004)
Alec Stanculescu, Fintronic USA, *1364 Working Group past vice-chair (through June 2004)*
Stefen Boyd, Boyd Technology, *ETF past co-chair (through November 2004)*

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

Accellera
Bluespec, Inc.
Cadence Design Systems, Inc.
Fintronic U.S.A.
IBM
Infineon Technologies
Intel Corporation
Mentor Graphics Corporation
Sun Microsystems, Inc.
Sunburst Design, Inc.
Sutherland HDL, Inc.
Synopsys, Inc.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

When the IEEE-SA Standards Board approved this standard on 8 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
Dennis B. Brophy
Joseph Bruder
Richard Cox
Bob Davis
Julian Forster*
Joanna N. Guenin
Mark S. Halpin
Raymond Hapeman
William B. Hopf
Lowell G. Johnson
Herman Koch
Joseph L. Koepfinger*
David J. Law
Daleep C. Mohla
Paul Nikolich
T. W. Olsen
Glenn Parsons
Ronald C. Petersen
Gary S. Robinson
Frank Stone
Malcolm V. Thaden
Richard L. Townsend
Joe D. Watson
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*

Michelle D. Turner
IEEE Standards Project Editor

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Contents

1. Overview 1
   - 1.1 Scope 1
   - 1.2 Conventions used in this standard 1
   - 1.3 Syntactic description 2
   - 1.4 Use of color in this standard 3
   - 1.5 Contents of this standard 3
   - 1.6 Deprecated clauses 5
   - 1.7 Header file listings 5
   - 1.8 Examples 5
   - 1.9 Prerequisites 5

2. Normative references 6

3. Lexical conventions 8
   - 3.1 Lexical tokens 8
   - 3.2 White space 8
   - 3.3 Comments 8
   - 3.4 Operators 8
   - 3.5 Numbers 9
     - 3.5.1 Integer constants 10
     - 3.5.2 Real constants 12
     - 3.5.3 Conversion 12
   - 3.6 Strings 12
     - 3.6.1 String variable declaration 13
     - 3.6.2 String manipulation 13
     - 3.6.3 Special characters in strings 13
   - 3.7 Identifiers, keywords, and system names 14
     - 3.7.1 Escaped identifiers 14
     - 3.7.2 Keywords 15
     - 3.7.3 System tasks and functions 15
     - 3.7.4 Compiler directives 15
   - 3.8 Attributes 16
     - 3.8.1 Examples 16
     - 3.8.2 Syntax 18

4. Data types 21
   - 4.1 Value set 21
   - 4.2 Nets and variables 21
     - 4.2.1 Net declarations 21
     - 4.2.2 Variable declarations 23
   - 4.3 Vectors 24
     - 4.3.1 Specifying vectors 24
     - 4.3.2 Vector net accessibility 24
   - 4.4 Strengths 25
     - 4.4.1 Charge strength 25
     - 4.4.2 Drive strength 25
   - 4.5 Implicit declarations 25
   - 4.6 Net types 26
     - 4.6.1 Wire and tri nets 26
     - 4.6.2 Wired nets 27
     - 4.6.3 Trireg net 28

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

4.6.4 Tri0 and tri1 nets ... 31
4.6.5 Unresolved nets ... 31
4.6.6 Supply nets ... 32
4.7 Regs ... 32
4.8 Integers, reals, times, and realtimes ... 32
4.8.1 Operators and real numbers ... 33
4.8.2 Conversion ... 33
4.9 Arrays ... 34
4.9.1 Net arrays ... 34
4.9.2 reg and variable arrays ... 34
4.9.3 Memories ... 35
4.10 Parameters ... 35
4.10.1 Module parameters ... 36
4.10.2 Local parameters (localparam) ... 37
4.10.3 Specify parameters ... 38
4.11 Name spaces ... 39

5. Expressions ... 41
5.1 Operators ... 41
5.1.1 Operators with real operands ... 42
5.1.2 Operator precedence ... 43
5.1.3 Using integer numbers in expressions ... 44
5.1.4 Expression evaluation order ... 45
5.1.5 Arithmetic operators ... 45
5.1.6 Arithmetic expressions with regs and integers ... 47
5.1.7 Relational operators ... 48
5.1.8 Equality operators ... 49
5.1.9 Logical operators ... 49
5.1.10 Bitwise operators ... 50
5.1.11 Reduction operators ... 51
5.1.12 Shift operators ... 53
5.1.13 Conditional operator ... 53
5.1.14 Concatenations ... 54
5.2 Operands ... 55
5.2.1 Vector bit-select and part-select addressing ... 56
5.2.2 Array and memory addressing ... 57
5.2.3 Strings ... 58
5.3 Minimum, typical, and maximum delay expressions ... 61
5.4 Expression bit lengths ... 62
5.4.1 Rules for expression bit lengths ... 62
5.4.2 Example of expression bit-length problem ... 63
5.4.3 Example of self-determined expressions ... 64
5.5 Signed expressions ... 64
5.5.1 Rules for expression types ... 65
5.5.2 Steps for evaluating an expression ... 65
5.5.3 Steps for evaluating an assignment ... 66
5.5.4 Handling X and Z in signed expressions ... 66
5.6 Assignments and truncation ... 66

6. Assignments ... 68
6.1 Continuous assignments ... 68
6.1.1 The net declaration assignment ... 69
6.1.2 The continuous assignment statement ... 69
6.1.3 Delays ... 71

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

6.1.4 Strength ... 71
6.2 Procedural assignments ... 72
6.2.1 Variable declaration assignment ... 72
6.2.2 Variable declaration syntax ... 73

7. Gate- and switch-level modeling ... 74
7.1 Gate and switch declaration syntax ... 74
7.1.1 The gate type specification ... 76
7.1.2 The drive strength specification ... 76
7.1.3 The delay specification ... 77
7.1.4 The primitive instance identifier ... 77
7.1.5 The range specification ... 77
7.1.6 Primitive instance connection list ... 78
7.2 and, nand, nor, or, xor, and xnor gates ... 80
7.3 buf and not gates ... 81
7.4 bufif1, bufif0, notif1, and notif0 gates ... 82
7.5 MOS switches ... 83
7.6 Bidirectional pass switches ... 84
7.7 CMOS switches ... 85
7.8 pullup and pulldown sources ... 86
7.9 Logic strength modeling ... 86
7.10 Strengths and values of combined signals ... 88
7.10.1 Combined signals of unambiguous strength ... 88
7.10.2 Ambiguous strengths: sources and combinations ... 89
7.10.3 Ambiguous strength signals and unambiguous signals ... 94
7.10.4 Wired logic net types ... 98
7.11 Strength reduction by nonresistive devices ... 100
7.12 Strength reduction by resistive devices ... 100
7.13 Strengths of net types ... 100
7.13.1 tri0 and tri1 net strengths ... 100
7.13.2 trireg strength ... 100
7.13.3 supply0 and supply1 net strengths ... 101
7.14 Gate and net delays ... 101
7.14.1 min:typ:max delays ... 102
7.14.2 trireg net charge decay ... 103

8. User-defined primitives (UDPs) ... 105
8.1 UDP definition ... 105
8.1.1 UDP header ... 107
8.1.2 UDP port declarations ... 107
8.1.3 Sequential UDP initial statement ... 107
8.1.4 UDP state table ... 107
8.1.5 Z values in UDP ... 108
8.1.6 Summary of symbols ... 108
8.2 Combinational UDPs ... 109
8.3 Level-sensitive sequential UDPs ... 110
8.4 Edge-sensitive sequential UDPs ... 110
8.5 Sequential UDP initialization ... 111
8.6 UDP instances ... 113
8.7 Mixing level-sensitive and edge-sensitive descriptions ... 114
8.8 Level-sensitive dominance ... 115

9. Behavioral modeling ... 116
9.1 Behavioral model overview ... 116

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

9.2 Procedural assignments ... 117
9.2.1 Blocking procedural assignments ... 117
9.2.2 The nonblocking procedural assignment ... 118
9.3 Procedural continuous assignments ... 122
9.3.1 The assign and deassign procedural statements ... 123
9.3.2 The force and release procedural statements ... 124
9.4 Conditional statement ... 125
9.4.1 If-else-if construct ... 126
9.5 Case statement ... 127
9.5.1 Case statement with do-not-cares ... 128
9.5.2 Constant expression in case statement ... 129
9.6 Looping statements ... 130
9.7 Procedural timing controls ... 131
9.7.1 Delay control ... 132
9.7.2 Event control ... 132
9.7.3 Named events ... 133
9.7.4 Event or operator ... 134
9.7.5 Implicit event_expression list ... 134
9.7.6 Level-sensitive event control ... 136
9.7.7 Intra-assignment timing controls ... 136
9.8 Block statements ... 139
9.8.1 Sequential blocks ... 140
9.8.2 Parallel blocks ... 141
9.8.3 Block names ... 141
9.8.4 Start and finish times ... 142
9.9 Structured procedures ... 143
9.9.1 Initial construct ... 143
9.9.2 Always construct ... 144

10. Tasks and functions ... 145
10.1 Distinctions between tasks and functions ... 145
10.2 Tasks and task enabling ... 145
10.2.1 Task declarations ... 146
10.2.2 Task enabling and argument passing ... 147
10.2.3 Task memory usage and concurrent activation ... 149
10.3 Disabling of named blocks and tasks ... 150
10.4 Functions and function calling ... 152
10.4.1 Function declarations ... 152
10.4.2 Returning a value from a function ... 154
10.4.3 Calling a function ... 155
10.4.4 Function rules ... 155
10.4.5 Use of constant functions ... 156

11. Scheduling semantics ... 158
11.1 Execution of a model ... 158
11.2 Event simulation ... 158
11.3 The stratified event queue ... 158
11.4 Verilog simulation reference model ... 159
11.4.1 Determinism ... 160
11.4.2 Nondeterminism ... 160
11.5 Race conditions ... 160
11.6 Scheduling implication of assignments ... 161
11.6.1 Continuous assignment ... 161
11.6.2 Procedural continuous assignment ... 161

Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

11.6.3 Blocking assignment ... 161
11.6.4 Nonblocking assignment ... 161
11.6.5 Switch (transistor) processing ... 161
11.6.6 Port connections ... 162
11.6.7 Functions and tasks ... 162

12. Hierarchical structures ... 163
12.1 Modules ... 163
12.1.1 Top-level modules ... 165
12.1.2 Module instantiation ... 165
12.2 Overriding module parameter values ... 167
12.2.1 defparam statement ... 168
12.2.2 Module instance parameter value assignment ... 170
12.2.3 Parameter dependence ... 173
12.3 Ports ... 173
12.3.1 Port definition ... 173
12.3.2 List of ports ... 174
12.3.3 Port declarations ... 174
12.3.4 List of ports declarations ... 176
12.3.5 Connecting module instance ports by ordered list ... 176
12.3.6 Connecting module instance ports by name ... 177
12.3.7 Real numbers in port connections ... 178
12.3.8 Connecting dissimilar ports ... 178
12.3.9 Port connection rules ... 179
12.3.10 Net types resulting from dissimilar port connections ... 179
12.3.11 Connecting signed values via ports ... 181
12.4 Generate constructs ... 181
12.4.1 Loop generate constructs ... 183
12.4.2 Conditional generate constructs ... 186
12.4.3 External names for unnamed generate blocks ... 190
12.5 Hierarchical names ... 191
12.6 Upwards name referencing ... 193
12.7 Scope rules ... 195
12.8 Elaboration ... 197
12.8.1 Order of elaboration ... 197
12.8.2 Early resolution of hierarchical names ... 197

13. Configuring the contents of a design ... 199
13.1 Introduction ... 199
13.1.1 Library notation ... 199
13.1.2 Basic configuration elements ... 200
13.2 Libraries ... 200
13.2.1 Specifying libraries—the library map file ... 200
13.2.2 Using multiple library map files ... 202
13.2.3 Mapping source files to libraries ... 202
13.3 Configurations ... 202
13.3.1 Basic configuration syntax ... 202
13.3.2 Hierarchical configurations ... 205
13.4 Using libraries and configs ... 205
13.4.1 Precompiling in a single-pass use model ... 205
13.4.2 Elaboration-time compiling in a single-pass use model ... 206
13.4.3 Precompiling using a separate compilation tool ... 206
13.4.4 Command line considerations ... 206
13.5 Configuration examples ... 206

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

13.5.1 Default configuration from library map file ... 207
13.5.2 Using default clause ... 207
13.5.3 Using cell clause ... 207
13.5.4 Using instance clause ... 208
13.5.5 Using hierarchical config ... 208
13.6 Displaying library binding information ... 208
13.7 Library mapping examples ... 209
13.7.1 Using the command line to control library searching ... 209
13.7.2 File path specification examples ... 209
13.7.3 Resolving multiple path specifications ... 209

14. Specify blocks ... 211
14.1 Specify block declaration ... 211
14.2 Module path declarations ... 212
14.2.1 Module path restrictions ... 212
14.2.2 Simple module paths ... 213
14.2.3 Edge-sensitive paths ... 214
14.2.4 State-dependent paths ... 215
14.2.5 Full connection and parallel connection paths ... 219
14.2.6 Declaring multiple module paths in a single statement ... 220
14.2.7 Module path polarity ... 220
14.3 Assigning delays to module paths ... 222
14.3.1 Specifying transition delays on module paths ... 222
14.3.2 Specifying x transition delays ... 224
14.3.3 Delay selection ... 225
14.4 Mixing module path delays and distributed delays ... 225
14.5 Driving wired logic ... 226
14.6 Detailed control of pulse filtering behavior ... 228
14.6.1 Specify block control of pulse limit values ... 229
14.6.2 Global control of pulse limit values ... 230
14.6.3 SDF annotation of pulse limit values ... 230
14.6.4 Detailed pulse control capabilities ... 230

15. Timing checks ... 237
15.1 Overview ... 237
15.2 Timing checks using a stability window ... 240
15.2.1 $setup ... 241
15.2.2 $hold ... 242
15.2.3 $setuphold ... 243
15.2.4 $removal ... 245
15.2.5 $recovery ... 246
15.2.6 $recrem ... 247
15.3 Timing checks for clock and control signals ... 248
15.3.1 $skew ... 249
15.3.2 $timeskew ... 250
15.3.3 $fullskew ... 252
15.3.4 $width ... 255
15.3.5 $period ... 256
15.3.6 $nochange ... 257
15.4 Edge-control specifiers ... 258
15.5 Notifiers: user-defined responses to timing violations ... 259
15.5.1 Requirements for accurate simulation ... 261
15.5.2 Conditions in negative timing checks ... 263
15.5.3 Notifiers in negative timing checks ... 264

Copyright © 2006 IEEE. All rights reserved.
--`,,`,,,,`-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

15.5.4 Option behavior 264
15.6 Enabling timing checks with conditioned events 265
15.7 Vector signals in timing checks 266
15.8 Negative timing checks 266

16. Backannotation using the standard delay format (SDF) 269
16.1 The SDF annotator 269
16.2 Mapping of SDF constructs to Verilog 269
16.2.1 Mapping of SDF delay constructs to Verilog declarations 269
16.2.2 Mapping of SDF timing check constructs to Verilog 271
16.2.3 SDF annotation of specparams 272
16.2.4 SDF annotation of interconnect delays 273
16.3 Multiple annotations 274
16.4 Multiple SDF files 275
16.5 Pulse limit annotation 275
16.6 SDF to Verilog delay value mapping 276

17. System tasks and functions 277
17.1 Display system tasks 278
17.1.1 The display and write tasks 278
17.1.2 Strobed monitoring 285
17.1.3 Continuous monitoring 286
17.2 File input-output system tasks and functions 286
17.2.1 Opening and closing files 287
17.2.2 File output system tasks 288
17.2.3 Formatting data to a string 289
17.2.4 Reading data from a file 290
17.2.5 File positioning 294
17.2.6 Flushing output 295
17.2.7 I/O error status 295
17.2.8 Detecting EOF 295
17.2.9 Loading memory data from a file 296
17.2.10 Loading timing data from an SDF file 297
17.3 Timescale system tasks 298
17.3.1 $printtimescale 299
17.3.2 $timeformat 300
17.4 Simulation control system tasks 302
17.4.1 $finish 302
17.4.2 $stop 302
17.5 Programmable logic array (PLA) modeling system tasks 303
17.5.1 Array types 303
17.5.2 Array logic types 304
17.5.3 Logic array personality declaration and loading 304
17.5.4 Logic array personality formats 304
17.6 Stochastic analysis tasks 307
17.6.1 $q_initialize 307
17.6.2 $q_add 307
17.6.3 $q_remove 307
17.6.4 $q_full 308
17.6.5 $q_exam 308
17.6.6 Status codes 308
17.7 Simulation time system functions 309
17.7.1 $time 309
17.7.2 $stime 309

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

17.7.3 $realtime ... 310
17.8 Conversion functions ... 310
17.9 Probabilistic distribution functions ... 311
17.9.1 $random function ... 311
17.9.2 $dist_ functions ... 312
17.9.3 Algorithm for probabilistic distribution functions ... 313
17.10 Command line input ... 320
17.10.1 $test$plusargs (string) ... 320
17.10.2 $value$plusargs (user_string, variable) ... 321
17.11 Math functions ... 323
17.11.1 Integer math functions ... 323
17.11.2 Real math functions ... 323

18. Value change dump (VCD) files ... 325
18.1 Creating four-state VCD file ... 325
18.1.1 Specifying name of dump file ($dumpfile) ... 325
18.1.2 Specifying variables to be dumped ($dumpvars) ... 326
18.1.3 Stopping and resuming the dump ($dumpoff/$dumpon) ... 327
18.1.4 Generating a checkpoint ($dumpall) ... 328
18.1.5 Limiting size of dump file ($dumplimit) ... 328
18.1.6 Reading dump file during simulation ($dumpflush) ... 328
18.2 Format of four-state VCD file ... 329
18.2.1 Syntax of four-state VCD file ... 330
18.2.2 Formats of variable values ... 331
18.2.3 Description of keyword commands ... 332
18.2.4 Four-state VCD file format example ... 337
18.3 Creating extended VCD file ... 338
18.3.1 Specifying dump file name and ports to be dumped ($dumpports) ... 338
18.3.2 Stopping and resuming the dump ($dumpportsoff/$dumpportson) ... 339
18.3.3 Generating a checkpoint ($dumpportsall) ... 340
18.3.4 Limiting size of dump file ($dumpportslimit) ... 340
18.3.5 Reading dump file during simulation ($dumpportsflush) ... 341
18.3.6 Description of keyword commands ... 341
18.3.7 General rules for extended VCD system tasks ... 341
18.4 Format of extended VCD file ... 342
18.4.1 Syntax of extended VCD file ... 342
18.4.2 Extended VCD node information ... 344
18.4.3 Value changes ... 346
18.4.4 Extended VCD file format example ... 347

19. Compiler directives ... 349
19.1 `celldefine and `endcelldefine ... 349
19.2 `default_nettype ... 349
19.3 `define and `undef ... 350
19.3.1 `define ... 350
19.3.2 `undef ... 352
19.4 `ifdef, `else, `elsif, `endif, `ifndef ... 352
19.5 `include ... 356
19.6 `resetall ... 356
19.7 `line ... 357
19.8 `timescale ... 358
19.9 `unconnected_drive and `nounconnected_drive ... 360
19.10 `pragma ... 360
19.10.1 Standard pragmas ... 361

Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

19.11 `begin_keywords, `end_keywords 361

20. Programming language interface (PLI) overview 366
20.1 PLI purpose and history 366
20.2 User-defined system task/function names 367
20.3 User-defined system task/function types 367
20.4 Overriding built-in system task/function names 367
20.5 User-supplied PLI applications 367
20.6 PLI mechanism 368
20.7 User-defined system task/function arguments 368
20.8 PLI include files 368

21. PLI TF and ACC interface mechanism (deprecated) 369

22. Using ACC routines (deprecated) 370

23. ACC routine definitions (deprecated) 371

24. Using TF routines (deprecated) 372

25. TF routine definitions (deprecated) 373

26. Using Verilog procedural interface (VPI) routines 374
26.1 VPI system tasks and functions 374
26.1.1 sizetf VPI application routine 374
26.1.2 compiletf VPI application routine 374
26.1.3 calltf VPI application routine 375
26.1.4 Arguments to sizetf, compiletf, and calltf application routines 375
26.2 VPI mechanism 375
26.2.1 VPI callbacks 375
26.2.2 VPI access to Verilog HDL objects and simulation objects 376
26.2.3 Error handling 376
26.2.4 Function availability 376
26.2.5 Traversing expressions 377
26.3 VPI object classifications 377
26.3.1 Accessing object relationships and properties 378
26.3.2 Object type properties 379
26.3.3 Object file and line properties 380
26.3.4 Delays and values 380
26.3.5 Object protection properties 381
26.4 List of VPI routines by functional category 381
26.5 Key to data model diagrams 383
26.5.1 Diagram key for objects and classes 384
26.5.2 Diagram key for accessing properties 384
26.5.3 Diagram key for traversing relationships 385
26.6 Object data model diagrams 386
26.6.1 Module 387
26.6.2 Instance arrays 388
26.6.3 Scope 389
26.6.4 IO declaration 389
26.6.5 Ports 390
26.6.6 Nets and net arrays 391
26.6.7 Regs and reg arrays 393
26.6.8 Variables 395

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.9 Memory ... 396
26.6.10 Object range ... 396
26.6.11 Named event ... 397
26.6.12 Parameter, specparam ... 398
26.6.13 Primitive, prim term ... 399
26.6.14 UDP ... 400
26.6.15 Module path, path term ... 401
26.6.16 Intermodule path ... 401
26.6.17 Timing check ... 402
26.6.18 Task, function declaration ... 402
26.6.19 Task/function call ... 403
26.6.20 Frames ... 404
26.6.21 Delay terminals ... 405
26.6.22 Net drivers and loads ... 405
26.6.23 Reg drivers and loads ... 406
26.6.24 Continuous assignment ... 406
26.6.25 Simple expressions ... 407
26.6.26 Expressions ... 408
26.6.27 Process, block, statement, event statement ... 409
26.6.28 Assignment ... 410
26.6.29 Delay control ... 410
26.6.30 Event control ... 410
26.6.31 Repeat control ... 411
26.6.32 While, repeat, wait ... 411
26.6.33 For ... 411
26.6.34 Forever ... 411
26.6.35 If, if-else ... 412
26.6.36 Case ... 412
26.6.37 Assign statement, deassign, force, release ... 413
26.6.38 Disable ... 413
26.6.39 Callback ... 414
26.6.40 Time queue ... 414
26.6.41 Active time format ... 414
26.6.42 Attributes ... 415
26.6.43 Iterator ... 416
26.6.44 Generates ... 417

27. VPI routine definitions ... 418
27.1 vpi_chk_error() ... 418
27.2 vpi_compare_objects() ... 420
27.3 vpi_control() ... 420
27.4 vpi_flush() ... 421
27.5 vpi_free_object() ... 421
27.6 vpi_get() ... 422
27.7 vpi_get_cb_info() ... 422
27.8 vpi_get_data() ... 423
27.9 vpi_get_delays() ... 424
27.10 vpi_get_str() ... 426
27.11 vpi_get_systf_info() ... 427
27.12 vpi_get_time() ... 428
27.13 vpi_get_userdata() ... 429
27.14 vpi_get_value() ... 429
27.15 vpi_get_vlog_info() ... 435
27.16 vpi_handle() ... 436

Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

27.17 vpi_handle_by_index() ... 437
27.18 vpi_handle_by_multi_index() ... 438
27.19 vpi_handle_by_name() ... 438
27.20 vpi_handle_multi() ... 439
27.21 vpi_iterate() ... 439
27.22 vpi_mcd_close() ... 440
27.23 vpi_mcd_flush() ... 441
27.24 vpi_mcd_name() ... 441
27.25 vpi_mcd_open() ... 442
27.26 vpi_mcd_printf() ... 443
27.27 vpi_mcd_vprintf() ... 444
27.28 vpi_printf() ... 444
27.29 vpi_put_data() ... 445
27.30 vpi_put_delays() ... 447
27.31 vpi_put_userdata() ... 450
27.32 vpi_put_value() ... 450
27.33 vpi_register_cb() ... 453
27.33.1 Simulation event callbacks ... 454
27.33.2 Simulation time callbacks ... 458
27.33.3 Simulator action or feature callbacks ... 460
27.34 vpi_register_systf() ... 461
27.34.1 System task/function callbacks ... 462
27.34.2 Initializing VPI system task/function callbacks ... 463
27.34.3 Registering multiple system tasks and functions ... 464
27.35 vpi_remove_cb() ... 465
27.36 vpi_scan() ... 465
27.37 vpi_vprintf() ... 466

28. Protected envelopes ... 467
28.1 General ... 467
28.2 Processing protected envelopes ... 467
28.2.1 Encryption ... 468
28.2.2 Decryption ... 469
28.3 Protect pragma directives ... 469
28.4 Protect pragma keywords ... 471
28.4.1 begin ... 471
28.4.2 end ... 471
28.4.3 begin_protected ... 471
28.4.4 end_protected ... 472
28.4.5 author ... 472
28.4.6 author_info ... 473
28.4.7 encrypt_agent ... 473
28.4.8 encrypt_agent_info ... 473
28.4.9 encoding ... 474
28.4.10 data_keyowner ... 475
28.4.11 data_method ... 475
28.4.12 data_keyname ... 476
28.4.13 data_public_key ... 477
28.4.14 data_decrypt_key ... 477
28.4.15 data_block ... 478
28.4.16 digest_keyowner ... 478
28.4.17 digest_key_method ... 478
28.4.18 digest_keyname ... 479
28.4.19 digest_public_key ... 479

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

28.4.20 digest_decrypt_key 480
28.4.21 digest_method 480
28.4.22 digest_block 481
28.4.23 key_keyowner 482
28.4.24 key_method 482
28.4.25 key_keyname 482
28.4.26 key_public_key 483
28.4.27 key_block 483
28.4.28 decrypt_license 484
28.4.29 runtime_license 484
28.4.30 comment 485
28.4.31 reset 485
28.4.32 viewport 486

Annex A (normative) Formal syntax definition 487
A.1 Source text 487
A.1.1 Library source text 487
A.1.2 Verilog source text 487
A.1.3 Module parameters and ports 487
A.1.4 Module items 488
A.1.5 Configuration source text 489
A.2 Declarations 489
A.2.1 Declaration types 489
A.2.2 Declaration data types 490
A.2.3 Declaration lists 491
A.2.4 Declaration assignments 491
A.2.5 Declaration ranges 492
A.2.6 Function declarations 492
A.2.7 Task declarations 492
A.2.8 Block item declarations 493
A.3 Primitive instances 493
A.3.1 Primitive instantiation and instances 493
A.3.2 Primitive strengths 494
A.3.3 Primitive terminals 494
A.3.4 Primitive gate and switch types 494
A.4 Module instantiation and generate construct 495
A.4.1 Module instantiation 495
A.4.2 Generate construct 495
A.5 UDP declaration and instantiation 496
A.5.1 UDP declaration 496
A.5.2 UDP ports 496
A.5.3 UDP body 496
A.5.4 UDP instantiation 497
A.6 Behavioral statements 497
A.6.1 Continuous assignment statements 497
A.6.2 Procedural blocks and assignments 497
A.6.3 Parallel and sequential blocks 497
A.6.4 Statements 498
A.6.5 Timing control statements 498
A.6.6 Conditional statements 499
A.6.7 Case statements 499
A.6.8 Looping statements 499
A.6.9 Task enable statements 499
A.7 Specify section 500

Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

A.7.1 Specify block declaration ... 500
A.7.2 Specify path declarations ... 500
A.7.3 Specify block terminals ... 500
A.7.4 Specify path delays ... 500
A.7.5 System timing checks ... 502
A.8 Expressions ... 504
A.8.1 Concatenations ... 504
A.8.2 Function calls ... 504
A.8.3 Expressions ... 504
A.8.4 Primaries ... 505
A.8.5 Expression left-side values ... 506
A.8.6 Operators ... 506
A.8.7 Numbers ... 506
A.8.8 Strings ... 507
A.9 General ... 507
A.9.1 Attributes ... 507
A.9.2 Comments ... 508
A.9.3 Identifiers ... 508
A.9.4 White space ... 509

Annex B (normative) List of keywords ... 510

Annex C (informative) System tasks and functions ... 511
C.1 $countdrivers ... 511
C.2 $getpattern ... 512
C.3 $input ... 513
C.4 $key and $nokey ... 513
C.5 $list ... 513
C.6 $log and $nolog ... 514
C.7 $reset, $reset_count, and $reset_value ... 514
C.8 $save, $restart, and $incsave ... 515
C.9 $scale ... 516
C.10 $scope ... 516
C.11 $showscopes ... 516
C.12 $showvars ... 516
C.13 $sreadmemb and $sreadmemh ... 517

Annex D (informative) Compiler directives ... 518
D.1 `default_decay_time ... 518
D.2 `default_trireg_strength ... 518
D.3 `delay_mode_distributed ... 519
D.4 `delay_mode_path ... 519
D.5 `delay_mode_unit ... 519
D.6 `delay_mode_zero ... 519

Annex E (normative) acc_user.h (deprecated) ... 520

Annex F (normative) veriuser.h (deprecated) ... 521

Annex G (normative) vpi_user.h ... 522

Annex H (informative) Encryption/decryption flow ... 537
H.1 Tool vendor secret key encryption system ... 537
H.1.1 Encryption input ... 537

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

H.1.2 Encryption output 538
H.2 IP author secret key encryption system 538
H.2.1 Encryption input 538
H.2.2 Encryption output 539
H.3 Digital envelopes 539
H.3.1 Encryption input 540
H.3.2 Encryption output 541

Annex I (informative) Bibliography 542

Index 543

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS

Not for Resale

List of Figures

Figure 4-1—Simulation values of a trireg and its driver 28
Figure 4-2—Simulation results of a capacitive network 29
Figure 4-3—Simulation results of charge sharing 30
Figure 7-1—Schematic diagram of interconnections in array of instances 80
Figure 7-2—Scale of strengths 88
Figure 7-3—Combining unequal strengths 89
Figure 7-4—Combination of signals of equal strength and opposite values 89
Figure 7-5—Weak x signal strength 89
Figure 7-6—Bufifs with control inputs of x 90
Figure 7-7—Strong H range of values 90
Figure 7-8—Strong L range of values 90
Figure 7-9—Combined signals of ambiguous strength 91
Figure 7-10—Range of strengths for an unknown signal 91
Figure 7-11—Ambiguous strengths from switch networks 91
Figure 7-12—Range of two strengths of a defined value 92
Figure 7-13—Range of three strengths of a defined value 92
Figure 7-14—Unknown value with a range of strengths 92
Figure 7-15—Strong X range 93
Figure 7-16—Ambiguous strength from gates 93
Figure 7-17—Ambiguous strength signal from a gate 93
Figure 7-18—Weak 0 94
Figure 7-19—Ambiguous strength in combined gate signals 94
Figure 7-20—Elimination of strength levels 95
Figure 7-21—Result showing a range and the elimination of strength levels of two values 95
Figure 7-22—Result showing a range and the elimination of strength levels of one value 96
Figure 7-23—A range of both values 97
Figure 7-24—Wired logic with unambiguous strength signals 98
Figure 7-25—Wired logic and ambiguous strengths 99
Figure 7-26—Trireg net with capacitance 104
Figure 8-1—Module schematic and simulation times of initial value propagation 113
Figure 9-1—Repeat event control utilizing a clock edge 139
Figure 12-1—Hierarchy in a model 193
Figure 12-2—Hierarchical path names in a model 193
Figure 12-3—Scopes available to upward name referencing 196
Figure 14-1—Module path delays 212
Figure 14-2—Difference between parallel and full connection paths 219
Figure 14-3—Module path delays longer than distributed delays 226
Figure 14-4—Module path delays shorter than distributed delays 226
Figure 14-5—Legal and illegal module paths 226
Figure 14-6—Illegal module paths 227
Figure 14-7—Legal module paths 227
Figure 14-8—Example of pulse filtering 228
Figure 14-9—On-detect versus on-event 231
Figure 14-10—Current event cancellation problem and correction 233
Figure 14-11—NAND gate with nearly simultaneous input switching where one event is scheduled prior to another that has not matured 234
Figure 14-12—NAND gate with nearly simultaneous input switching with output event scheduled at same time 235
Figure 15-1—Sample $timeskew 251
Figure 15-2—Sample $timeskew with remain_active_flag set 252
Figure 15-3—Sample $fullskew 254

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Figure 15-4—Timing check violation windows 264
Figure 15-5—Data constraint interval, positive setup/hold 267
Figure 15-6—Data constraint interval, negative setup/hold 268
Figure 18-1—Creating the four-state VCD file 325
Figure 18-2—Creating the extended VCD file 338
Figure 26-1—Example of object relationships diagram 378
Figure 26-2—Accessing a class of objects using tags 379
Figure 27-1—s_vpi_error_info structure definition 419
Figure 27-2—s_cb_data structure definition 423
Figure 27-3—s_vpi_delay structure definition 424
Figure 27-4—s_vpi_time structure definition 424
Figure 27-5—s_vpi_systf_data structure definition 427
Figure 27-6—s_vpi_time structure definition 428
Figure 27-7—s_vpi_value structure definition 430
Figure 27-8—s_vpi_vecval structure definition 430
Figure 27-9—s_vpi_strengthval structure definition 430
Figure 27-10—s_vpi_vlog_info structure definition 436
Figure 27-11—s_vpi_delay structure definition 448
Figure 27-12—s_vpi_time structure definition 448
Figure 27-13—s_vpi_value structure definition 452
Figure 27-14—s_vpi_time structure definition 453
Figure 27-15—s_vpi_vecval structure definition 453
Figure 27-16—s_vpi_strengthval structure definition 453
Figure 27-17—s_cb_data structure definition 454
Figure 27-18—s_vpi_systf_data structure definition 462

Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

List of Tables

Table 3-1—Specifying special characters in string 14
Table 4-1—Net types 26
Table 4-2—Truth table for wire and tri nets 27
Table 4-3—Truth table for wand and triand nets 27
Table 4-4—Truth table for wor and trior nets 27
Table 4-5—Truth table for tri0 net 31
Table 4-6—Truth table for tri1 net 31
Table 4-7—Differences between specparams and parameters 38
Table 5-1—Operators in Verilog HDL 42
Table 5-2—Legal operators for use in real expressions 43
Table 5-3—Operators not allowed for real expressions 43
Table 5-4—Precedence rules for operators 44
Table 5-5—Arithmetic operators defined 45
Table 5-6—Power operator rule examples 46
Table 5-7—Unary operators defined 46
Table 5-8—Examples of modulus and power operators 46
Table 5-9—Data type interpretation by arithmetic operators 47
Table 5-10—Definitions of relational operators 48
Table 5-11—Definitions of equality operators 49
Table 5-12—Bitwise binary and operator 50
Table 5-13—Bitwise binary or operator 50
Table 5-14—Bitwise binary exclusive or operator 51
Table 5-15—Bitwise binary exclusive nor operator 51
Table 5-16—Bitwise unary negation operator 51
Table 5-17—Reduction unary and operator 52
Table 5-18—Reduction unary or operator 52
Table 5-19—Reduction unary exclusive or operator 52
Table 5-20—Results of unary reduction operations 52
Table 5-21—Ambiguous condition results for conditional operator 54
Table 5-22—Bit lengths resulting from self-determined expressions 63
Table 6-1—Legal left-hand forms in assignment statements 68
Table 7-1—Built-in gates and switches 76
Table 7-2—Valid gate types for strength specifications 76
Table 7-3—Truth tables for multiple input logic gates 81
Table 7-4—Truth tables for multiple output logic gates 82
Table 7-5—Truth tables for three-state logic gates 83
Table 7-6—Truth tables for MOS switches 84
Table 7-7—Strength levels for scalar net signal values 87
Table 7-8—Strength reduction rules 100
Table 7-9—Rules for propagation delays 101
Table 8-1—UDP table symbols 108
Table 8-2—Initial statements in UDPs and modules 111
Table 8-3—Mixing of level-sensitive and edge-sensitive entries 115
Table 9-1—Detecting posedge and negedge 133
Table 9-2—Intra-assignment timing control equivalence 138
Table 12-1—Net types resulting from dissimilar port connections 180
Table 14-1—List of valid operators in state-dependent path delay expression 215
Table 14-2—Associating path delay expressions with transitions 223
Table 14-3—Calculating delays for x transitions 224
Table 15-1—$setup arguments 241
Table 15-2—$hold arguments 242

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 15-3—$setuphold arguments 243
Table 15-4—$removal arguments 245
Table 15-5—$recovery arguments 246
Table 15-6—$recrem arguments 247
Table 15-7—$skew arguments 249
Table 15-8—$timeskew arguments 250
Table 15-9—$fullskew arguments 253
Table 15-10—$width arguments 255
Table 15-11—$period arguments 256
Table 15-12—$nochange arguments 257
Table 15-13—Notifier value responses to timing violations 260
Table 16-1—Mapping of SDF delay constructs to Verilog declarations 269
Table 16-2—Mapping of SDF timing check constructs to Verilog 271
Table 16-3—SDF annotation of interconnect delays 273
Table 16-4—SDF to Verilog delay value mapping 276
Table 17-1—Escape sequences for printing special characters 279
Table 17-2—Escape sequences for format specifications 279
Table 17-3—Format specifications for real numbers 281
Table 17-4—Logic value component of strength format 283
Table 17-5—Mnemonics for strength levels 284
Table 17-6—Explanation of strength formats 285
Table 17-7—Types for file descriptors 287
Table 17-8—mtm spec argument 298
Table 17-9—scale type argument 298
Table 17-10—$timeformat units_number arguments 300
Table 17-11—$timeformat default value for arguments 301
Table 17-12—Diagnostics for $finish 302
Table 17-13—PLA modeling system tasks 303
Table 17-14—Types of queues of $q_type values 307
Table 17-15—Argument values for $q_exam system task 308
Table 17-16—Status code values 308
Table 17-17—Verilog to C function cross-listing 313
Table 17-18—Verilog to C real math function cross-listing 324
Table 18-1—Rules for left-extending vector values 331
Table 18-2—How the VCD can shorten values 331
Table 18-3—Keyword commands 332
Table 19-1—Arguments of time_precision 359
Table 19-2—IEEE 1364-1995 reserved keywords 362
Table 19-3—IEEE 1364-2001 reserved keywords 363
Table 19-4—IEEE 1364-2005 reserved keywords 364
Table 26-1—VPI routines for simulation-related callbacks 381
Table 26-2—VPI routines for system task/function callbacks 381
Table 26-3—VPI routines for traversing Verilog HDL hierarchy 382
Table 26-4—VPI routines for accessing properties of objects 382
Table 26-5—VPI routines for accessing objects from properties 382
Table 26-6—VPI routines for delay processing 382
Table 26-7—VPI routines for logic and strength value processing 382
Table 26-8—VPI routines for simulation time processing 382
Table 26-9—VPI routines for miscellaneous utilities 383
Table 27-1—Return error constants for vpi_chk_error() 419
Table 27-2—Size of the s_vpi_delay->da array 425
Table 27-3—Return value field of the s_vpi_value structure union 431
Table 27-4—Size of the s_vpi_delay->da array 449
Table 27-5—Value format field of cb_data_p->value->format 455

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS

Not for Resale

Table 27-6—cbStmt callbacks 457
Table 28-1—protect pragma keywords 469
Table 28-2—Encoding algorithm identifiers 474
Table 28-3—Encryption algorithm identifiers 476
Table 28-4—Message digest algorithm identifiers 481
Table C.1—Argument return value for $countdriver function 512

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

List of Syntax Boxes

Syntax 3-1—Syntax for integer and real numbers 9
Syntax 3-2—Syntax for system tasks and functions 15
Syntax 3-3—Syntax for attributes 16
Syntax 3-4—Syntax for module declaration attributes 18
Syntax 3-5—Syntax for port declaration attributes 18
Syntax 3-6—Syntax for module item attributes 19
Syntax 3-7—Syntax for function port, task, and block attributes 19
Syntax 3-8—Syntax for port connection attributes 20
Syntax 3-9—Syntax for udp attributes 20
Syntax 4-1—Syntax for net declaration 22
Syntax 4-2—Syntax for variable declaration 23
Syntax 4-3—Syntax for integer, time, real, and realtime declarations 32
Syntax 4-4—Syntax for module parameter declaration 36
Syntax 4-5—Syntax for specparam declaration 38
Syntax 5-1—Syntax for conditional operator 53
Syntax 5-2—Syntax for mintypmax expression 61
Syntax 6-1—Syntax for continuous assignment 69
Syntax 6-2—Syntax for variable declaration assignment 73
Syntax 7-1—Syntax for gate instantiation 75
Syntax 8-1—Syntax for UDPs 106
Syntax 8-2—Syntax for UDP instances 113
Syntax 9-1—Syntax for blocking assignments 118
Syntax 9-2—Syntax for nonblocking assignments 119
Syntax 9-3—Syntax for procedural continuous assignments 123
Syntax 9-4—Syntax for if statement 125
Syntax 9-5—Syntax for if-else-if construct 126
Syntax 9-6—Syntax for case statement 127
Syntax 9-7—Syntax for looping statements 130
Syntax 9-8—Syntax for procedural timing control 132
Syntax 9-9—Syntax for event declaration 133
Syntax 9-10—Syntax for event trigger 134
Syntax 9-11—Syntax for wait statement 136
Syntax 9-12—Syntax for intra-assignment delay and event control 137
Syntax 9-13—Syntax for sequential block 140
Syntax 9-14—Syntax for parallel block 141
Syntax 9-15—Syntax for initial construct 143
Syntax 9-16—Syntax for always construct 144
Syntax 10-1—Syntax for task declaration 146
Syntax 10-2—Syntax for task-enabling statement 147
Syntax 10-3—Syntax for disable statement 150
Syntax 10-4—Syntax for function declaration 153
Syntax 10-5—Syntax for function call 155
Syntax 12-1—Syntax for module 164
Syntax 12-2—Syntax for module instantiation 165
Syntax 12-3—Syntax for port 173
Syntax 12-4—Syntax for port declarations 174
Syntax 12-5—Syntax for generate constructs 182
Syntax 12-6—Syntax for hierarchical path names 192
Syntax 12-7—Syntax for upward name referencing 194
Syntax 13-1—Syntax for cell 199
Syntax 13-2—Syntax for declaring library in library map file 201

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS

Not for Resale

Syntax 13-3—Syntax for include command ... 202
Syntax 13-4—Syntax for configuration ... 203
Syntax 13-5—Syntax for default clause ... 203
Syntax 13-6—Syntax for instance clause ... 203
Syntax 13-7—Syntax for cell clause ... 204
Syntax 13-8—Syntax for liblist clause ... 204
Syntax 13-9—Syntax for use clause ... 204
Syntax 14-1—Syntax for specify block ... 211
Syntax 14-2—Syntax for module path declaration ... 212
Syntax 14-3—Syntax for simple module path ... 213
Syntax 14-4—Syntax for edge-sensitive path declaration ... 214
Syntax 14-5—Syntax for state-dependent paths ... 215
Syntax 14-6—Syntax for path delay value ... 222
Syntax 14-7—Syntax for PATHPULSE$ pulse control ... 229
Syntax 14-8—Syntax for pulse style declarations ... 231
Syntax 14-9—Syntax for showcancelled declarations ... 232
Syntax 15-1—Syntax for system timing checks ... 238
Syntax 15-2—Syntax for check time conditions and timing check events ... 239
Syntax 15-3—Syntax for $setup ... 241
Syntax 15-4—Syntax for $hold ... 242
Syntax 15-5—Syntax for $setuphold ... 243
Syntax 15-6—Syntax for $removal ... 245
Syntax 15-7—Syntax for $recovery ... 246
Syntax 15-8—Syntax for $recrem ... 247
Syntax 15-9—Syntax for $skew ... 249
Syntax 15-10—Syntax for $timeskew ... 250
Syntax 15-11—Syntax for $fullskew ... 252
Syntax 15-12—Syntax for $width ... 255
Syntax 15-13—Syntax for $period ... 256
Syntax 15-14—Syntax for $nochange ... 257
Syntax 15-15—Syntax for edge-control specifier ... 258
Syntax 15-16—Syntax for controlled timing check event ... 265
Syntax 17-1—Syntax for $display and $write system tasks ... 278
Syntax 17-2—Syntax for $strobe system tasks ... 285
Syntax 17-3—Syntax for $monitor system tasks ... 286
Syntax 17-4—Syntax for $fopen and $fclose system tasks ... 287
Syntax 17-5—Syntax for file output system tasks ... 288
Syntax 17-6—Syntax for formatting data tasks ... 289
Syntax 17-7—Syntax for memory load system tasks ... 296
Syntax 17-8—Syntax for $sdf_annotate system task ... 297
Syntax 17-9—Syntax for $printtimescale ... 299
Syntax 17-10—Syntax for $timeformat ... 300
Syntax 17-11—Syntax for $finish ... 302
Syntax 17-12—Syntax for $stop ... 302
Syntax 17-13 —Syntax for PLA modeling system task ... 303
Syntax 17-14—Syntax for $time ... 309
Syntax 17-15—Syntax for $stime ... 309
Syntax 17-16—Syntax for $realtime ... 310
Syntax 17-17—Syntax for $random ... 311
Syntax 17-18—Syntax for probabilistic distribution functions ... 312
Syntax 18-1—Syntax for $dumpfile task ... 325
Syntax 18-2—Syntax for filename ... 326
Syntax 18-3—Syntax for $dumpvars task ... 326
Syntax 18-4—Syntax for $dumpoff and $dumpon tasks ... 327

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Syntax 18-5—Syntax for $dumpall task ... 328
Syntax 18-6—Syntax for $dumplimit task ... 328
Syntax 18-7—Syntax for $dumpflush task ... 328
Syntax 18-8—Syntax for output four-state VCD file ... 330
Syntax 18-9—Syntax for $comment section ... 332
Syntax 18-10—Syntax for $date section ... 332
Syntax 18-11—Syntax for $enddefinitions section ... 333
Syntax 18-12—Syntax for $scope section ... 333
Syntax 18-13—Syntax for $timescale ... 334
Syntax 18-14—Syntax for $upscope section ... 334
Syntax 18-15—Syntax for $var section ... 334
Syntax 18-16—Syntax for $version section ... 335
Syntax 18-17—Syntax for $dumpall keyword ... 335
Syntax 18-18—Syntax for $dumpoff keyword ... 336
Syntax 18-19—Syntax for $dumpon keyword ... 336
Syntax 18-20—Syntax for $dumpvars keyword ... 336
Syntax 18-21—Syntax for $dumpports task ... 338
Syntax 18-22—Syntax for $dumpportsoff and $dumpportson system tasks ... 339
Syntax 18-23—Syntax for $dumpportsall system task ... 340
Syntax 18-24—Syntax for $dumpportslimit system task ... 340
Syntax 18-25—Syntax for $dumpportsflush system task ... 341
Syntax 18-26—Syntax for $vcdclose keyword ... 341
Syntax 18-27—Syntax for output extended VCD file ... 343
Syntax 18-28—Syntax for extended VCD node information ... 344
Syntax 18-29—Syntax for value change section ... 346
Syntax 19-1—Syntax for default_nettype compiler directive ... 350
Syntax 19-2—Syntax for text macro definition ... 350
Syntax 19-3—Syntax for text macro usage ... 351
Syntax 19-4—Syntax for undef compiler directive ... 352
Syntax 19-5—Syntax for conditional compilation directives ... 353
Syntax 19-6—Syntax for include compiler directive ... 356
Syntax 19-7—Syntax for line compiler directive ... 357
Syntax 19-8—Syntax for timescale compiler directive ... 358
Syntax 19-9—Syntax for pragma compiler directive ... 360
Syntax 19-10—Syntax for begin keywords and end keywords compiler directives ... 361

Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

IEEE Standard for Verilog® Hardware Description Language

1. Overview

1.1 Scope

Verilog is a hardware description language (HDL) that was standardized as IEEE Std 1364™-1995 and first revised as IEEE Std 1364-2001. This revision corrects and clarifies features ambiguously described in the 1995 and 2001 editions. It also resolves incompatibilities and inconsistencies of IEEE 1364-2001 with IEEE Std 1800™-2005.

The intent of this standard is to serve as a complete specification of the Verilog HDL. This standard contains the following:

- The formal syntax and semantics of all Verilog HDL constructs
- The formal syntax and semantics of standard delay format (SDF) constructs
- Simulation system tasks and functions, such as text output display commands
- Compiler directives, such as text substitution macros and simulation time scaling
- The programming language interface (PLI) binding mechanism
- The formal syntax and semantics of the Verilog procedural interface (VPI)
- Informative usage examples
- Informative delay model for SDF
- The VPI header file

1.2 Conventions used in this standard

This standard is organized into clauses, each of which focuses on a specific area of the language. There are subclauses within each clause to discuss individual constructs and concepts. The discussion begins with an introduction and an optional rationale for the construct or the concept, followed by syntax and semantic descriptions, followed by some examples and notes.

The term *shall* is used throughout this standard to indicate mandatory requirements, whereas the term *may* is used to indicate optional features. These terms denote different meanings to different readers of this standard:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

a) To the developers of tools that process the Verilog HDL, the term *shall* denotes a requirement that the standard imposes. The resulting implementation is required to enforce the requirements and to issue an error if the requirement is not met by the input.
b) To the Verilog HDL model developer, the term *shall* denotes that the characteristics of the Verilog HDL are natural consequences of the language definition. The model developer is required to adhere to the constraint implied by the characteristic. The term *may* denotes optional features that the model developer can exercise at discretion. If such features are used, however, the model developer is required to follow the requirements set forth by the language definition.
c) To the Verilog HDL model user, the term *shall* denotes that the characteristics of the models are natural consequences of the language definition. The model user can depend on the characteristics of the model implied by its Verilog HDL source text.

1.3 Syntactic description

The formal syntax of the Verilog HDL is described using Backus-Naur Form (BNF). The following conventions are used:

— Lowercase words, some containing embedded underscores, are used to denote syntactic categories. For example:

```
module_declaration
```

— Boldface words are used to denote reserved keywords, operators, and punctuation marks as a required part of the syntax. For example:

```
module     =>     ;
```

— A vertical bar separates alternative items unless it appears in boldface, in which case it stands for itself. For example:

```
unary_operator ::=
        + | - | ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~
```

— Square brackets enclose optional items. For example:

```
input_declaration ::= input [range]  list_of_variables ;
```

— Braces ({}) enclose a repeated item unless it appears in boldface, in which case it stands for itself. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus, the following two rules are equivalent:

```
list_of_param_assignments ::= param_assignment  { , param_assignment }
list_of_param_assignments ::=
        param_assignment
      | list_of_param_assignment , param_assignment
```

— If the name of any category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, "*msb*_index" and "*lsb*_index" are equivalent to "index."

The main text uses *italicized* font when a term is being defined and uses `constant-width` font for examples, file names, and constants, especially `0`, `1`, `x`, and `z` values.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

1.4 Use of color in this standard

This standard uses a minimal amount of color to enhance readability. The coloring is not essential and does not affect the accuracy of this standard when viewed in pure black and white. Color is used to show cross references that are hyperlinked to other portions of this standard. These hyperlinked cross references are shown in underlined-blue text (hyperlinking works when this standard is viewed interactively as a PDF file).

1.5 Contents of this standard

A synopsis of the clauses and annexes is presented as a quick reference. There are 28 clauses and 9 annexes. All clauses, as well as Annex A, Annex B, and Annex G, are normative parts of this standard. Annex C, Annex D, Annex H, and Annex I are included for informative purposes only.

IEEE Std 1364-2005 has deprecated the task/function (TF) and access (ACC) routines, which were specified previously in Clause 21 through Clause 25, Annex E, and Annex F of IEEE Std 1364-2001[1]. Clause 20 has been modified to reflect this change. The text of deprecated clauses and annexes has been removed from this version of the standard, but the clause headings have been retained. See the corresponding clauses in IEEE Std 1364-2001 for the deprecated text.

Clause 1 discusses the conventions used in this standard and its contents.

Clause 2 lists references to other publications that are required in order to implement this standard.

Clause 3 describes the lexical tokens used in Verilog HDL source text and their conventions. It describes how to specify and interpret the lexical tokens.

Clause 4 describes net and variable data types. This clause also discusses the parameter data type for constant values and describes drive and charge strength of the values on nets.

Clause 5 describes the operators and operands that can be used in expressions.

Clause 6 compares the two main types of assignment statements in the Verilog HDL—continuous assignments and procedural assignments. It describes the continuous assignment statement that drives values onto nets.

Clause 7 describes the gate- and switch-level primitives and logic strength modeling.

Clause 8 describes how a primitive can be defined in the Verilog HDL and how these primitives are included in Verilog HDL models.

Clause 9 describes procedural assignments, procedural continuous assignments, and behavioral language statements.

Clause 10 describes tasks and functions—procedures that can be called from more than one place in a behavioral model. It describes how tasks can be used like subroutines and how functions can be used to define new operators. The clause describes how to disable the execution of a task and a named block of statements.

Clause 11 describes the scheduling semantics of the Verilog HDL.

[1]For information on references, see Clause 2.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Clause 12 describes how hierarchies are created in the Verilog HDL and how parameter values declared in a module can be overridden. It describes how generated constructs can be used to do conditional or multiple instantiations in a design.

Clause 13 describes how to configure the contents of a design.

Clause 14 describes how to specify timing relationships between input and output ports of a module.

Clause 15 describes how timing checks are used in specify blocks to determine whether signals obey the timing constraints.

Clause 16 describes syntax and semantics of SDF constructs.

Clause 17 describes the system tasks and functions.

Clause 18 describes the system tasks associated with value change dump (VCD) file and the format of the file.

Clause 19 describes the compiler directives.

Clause 20 previews the C language procedural interface standard (i.e., PLI) and interface mechanisms that are part of the Verilog HDL.

Clause 21 has been deprecated. See IEEE Std 1364-2001 for the contents of this clause.

Clause 22 has been deprecated. See IEEE Std 1364-2001 for the contents of this clause.

Clause 23 has been deprecated. See IEEE Std 1364-2001 for the contents of this clause.

Clause 24 has been deprecated. See IEEE Std 1364-2001 for the contents of this clause.

Clause 25 has been deprecated. See IEEE Std 1364-2001 for the contents of this clause.

Clause 26 provides an overview of the types of operations that are done with the VPI routines.

Clause 27 describes the VPI routines.

Clause 28 describes encryption and decryption of source text regions.

Annex A (normative) describes, using BNF, the syntax of the Verilog HDL.

Annex B (normative) lists the Verilog HDL keywords.

Annex C (informative) describes system tasks and functions that are frequently used, but that are not part of this standard.

Annex D (informative) describes compiler directives that are frequently used, but that are not part of this standard.

Annex E has been deprecated. See IEEE Std 1364-2001 for the contents of this annex.

Annex F has been deprecated. See IEEE Std 1364-2001 for the contents of this annex.

Annex G (normative) provides a listing of the contents of the `vpi_user.h` file.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Annex H (informative) describes the various scenarios that can be used for intellectual property (IP) protection, and it also shows how the relevant pragmas will be used to achieve the desired effect of securely protecting, distributing, and decrypting the model.

Annex I (informative) contains bibliographic entries pertaining to this standard.

1.6 Deprecated clauses

IEEE Std 1364-2005 deprecates the Verilog PLI TF and ACC routines that were contained in previous versions of this standard. These routines were described in Clause 21 through Clause 25, Annex E, and Annex F. The text of these clauses and annexes have been removed from this version of the standard. The text of these deprecated clauses and annexes can be found in IEEE Std 1364-2001.

1.7 Header file listings

The header file listings included in Annex G for `vpi_user.h` are a normative part of this standard. All compliant software tools should use the same function declarations, constant definitions, and structure definitions contained in these header file listings.

1.8 Examples

Several small examples in the Verilog HDL and the C programming language are shown throughout this standard. These examples are informative. They are intended to illustrate the usage of Verilog HDL constructs and PLI functions in a simple context and do not define the full syntax.

1.9 Prerequisites

Clause 20, Clause 26, Clause 27, and Annex G presuppose a working knowledge of the C programming language.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

2. Normative references

The following referenced documents are indispensable for the application of this standard. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

Anderson, R., Biham, E., and Knudsen, L. "Serpent: A Proposal for the Advanced Encryption Standard," NIST AES Proposal, 1998, http://www.cl.cam.ac.uk/ftp/users/rja14/serpent.tar.gz.

ANSI Std X9.52-1998, American National Standard for Financial Services—Triple Data Encryption Algorithm Modes of Operation.[2]

ElGamal, T., "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Transactions on Information Theory*, vol. IT-31, no. 4, pp. 469–472, July 1985.

FIPS 46-3 (October 1999), Data Encryption Standard (DES).[3]

FIPS 180-2 (August 2002), Secure Hash Standard (SHS).

FIPS 197 (November 2001), Advanced Encryption Standard (AES).

IEEE Std 754™-1985, IEEE Standard for Binary Floating-Point Arithmetic.[4, 5]

IEEE Std 1003.1™, IEEE Standard for Information Technology—Portable Operating System Interface (POSIX®).

IEEE Std 1364™-2001, IEEE Standard for Verilog® Hardware Description Language.

IETF RFC 1319 (April 1992), The MD2 Message-Digest Algorithm.[6]

IETF RFC 1321 (April 1992), The MD5 Message-Digest Algorithm.

IETF RFC 2045 (November 1996), Multipurpose Internet Mail Extensions (MIME), Part One: Format of Internet Message Bodies.

IETF RFC 2144 (May 1997), The CAST-128 Encryption Algorithm.

IETF RFC 2437 (October 1998), PKCS #1: RSA Cryptography Specifications, Version 2.0.

IETF RFC 2440 (November 1998), OpenPGP Message Format.

[2]ANSI publications are available from the Sales Department, American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

[3]FIPS publications are available from the National Technical Information Service (NTIS), U. S. Dept. of Commerce, 5285 Port Royal Rd., Springfield, VA 22161 (http://www.ntis.org/).

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

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

[6]IETF requests for comments (RFCs) are available from the Internet Engineering Task Force (http://www.ieft.org).

 Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

ISO/IEC 10118-3:2004, Information technology—Security techniques—Hash-functions—Part 3: Dedicated hash-functions.[7]

Schneier, B., "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," *Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993)*, Springer-Verlag, 1994, pp. 191–204.

Schneier, B., et al, *The Twofish Encryption Algorithm: A 128-Bit Block Cipher*, 1st ed., Wiley, 1999.

[7]ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch/). ISO/IEC publications are also available in the United States from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112, USA (http://global.ihs.com/). Electronic copies are available in the United States from the American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

3. Lexical conventions

This clause describes the lexical tokens used in Verilog HDL source text and their conventions.

3.1 Lexical tokens

Verilog HDL source text files shall be a stream of lexical tokens. A *lexical token* shall consist of one or more characters. The layout of tokens in a source file shall be free format; that is, spaces and newlines shall not be syntactically significant other than being token separators, except for escaped identifiers (see 3.7.1).

The types of lexical tokens in the language are as follows:

— White space
— Comment
— Operator
— Number
— String
— Identifier
— Keyword

3.2 White space

White space shall contain the characters for spaces, tabs, newlines, and formfeeds. These characters shall be ignored except when they serve to separate other lexical tokens. However, blanks and tabs shall be considered significant characters in strings (see 3.6).

3.3 Comments

The Verilog HDL has two forms to introduce comments. A *one-line comment* shall start with the two characters `//` and end with a newline. A *block comment* shall start with `/*` and end with `*/`. Block comments shall not be nested. The one-line comment token `//` shall not have any special meaning in a block comment.

3.4 Operators

Operators are single-, double-, or triple-character sequences and are used in expressions. Clause 5 discusses the use of operators in expressions.

Unary operators shall appear to the left of their operand. *Binary operators* shall appear between their operands. A *conditional operator* shall have two operator characters that separate three operands.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

3.5 Numbers

Constant numbers can be specified as integer constants (defined in 3.5.1) or real constants.

```
number ::=  (From A.8.7)
        decimal_number
      | octal_number
      | binary_number
      | hex_number
      | real_number
real_number[a] ::=
        unsigned_number . unsigned_number
      | unsigned_number [ . unsigned_number ] exp [ sign ] unsigned_number
exp ::= e | E
decimal_number ::=
        unsigned_number
      | [ size ] decimal_base unsigned_number
      | [ size ] decimal_base x_digit { _ }
      | [ size ] decimal_base z_digit { _ }
binary_number ::=
      [ size ] binary_base binary_value
octal_number ::=
      [ size ] octal_base octal_value
hex_number ::=
      [ size ] hex_base hex_value
sign ::= + | -
size ::= non_zero_unsigned_number
non_zero_unsigned_number[a] ::= non_zero_decimal_digit { _ | decimal_digit}
unsigned_number[a] ::= decimal_digit { _ | decimal_digit }
binary_value[a] ::= binary_digit { _ | binary_digit }
octal_value[a] ::= octal_digit { _ | octal_digit }
hex_value[a] ::= hex_digit { _ | hex_digit }
decimal_base[a] ::= '[s|S]d | '[s|S]D
binary_base[a] ::= '[s|S]b |  '[s|S]B
octal_base[a]::= '[s|S]o | '[s|S]O
hex_base[a] ::= '[s|S]h | '[s|S]H
non_zero_decimal_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
decimal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
binary_digit ::= x_digit | z_digit | 0 | 1
octal_digit ::= x_digit | z_digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
hex_digit ::=
        x_digit | z_digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
      | a | b | c | d | e | f | A | B | C | D | E | F
x_digit ::= x | X
z_digit ::= z | Z | ?
```

[a]Embedded spaces are illegal.

Syntax 3-1—Syntax for integer and real numbers

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

3.5.1 Integer constants

Integer constants can be specified in decimal, hexadecimal, octal, or binary format.

There are two forms to express integer constants. The first form is a simple decimal number, which shall be specified as a sequence of digits `0` through `9`, optionally starting with a plus or minus unary operator. The second form specifies a *based constant*, which shall be composed of up to three tokens—an optional size constant, an apostrophe character (**'**, ASCII 0x27) followed by a base format character, and the digits representing the value of the number. It shall be legal to macro-substitute these three tokens.

The first token, a size constant, shall specify the size of the constant in terms of its exact number of bits. It shall be specified as a nonzero unsigned decimal number. For example, the size specification for two hexadecimal digits is 8 because one hexadecimal digit requires 4 bits.

The second token, a `base_format`, shall consist of a case-insensitive letter specifying the base for the number, optionally preceded by the single character `s` (or `S`) to indicate a signed quantity, preceded by the apostrophe character. Legal base specifications are `d`, `D`, `h`, `H`, `o`, `O`, `b`, or `B` for the bases decimal, hexadecimal, octal, and binary, respectively.

The apostrophe character and the base format character shall not be separated by any white space.

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

Simple decimal numbers without the size and the base format shall be treated as *signed integers*, whereas the numbers specified with the base format shall be treated as signed integers if the `s` designator is included or as *unsigned integers* if the base format only is used. The `s` designator does not affect the bit pattern specified, only its interpretation.

A plus or minus operator preceding the size constant is a unary plus or minus operator. A plus or minus operator between the base format and the number is an illegal syntax.

Negative numbers shall be represented in twos-complement form.

An `x` represents the *unknown value* in hexadecimal, octal, and binary constants. A `z` represents the *high-impedance value*. See 4.1 for a discussion of the Verilog HDL value set. An `x` shall set 4 bits to unknown in the hexadecimal base, 3 bits in the octal base, and 1 bit in the binary base. Similarly, a `z` shall set 4 bits, 3 bits, and 1 bit, respectively, to the high-impedance value.

If the size of the unsigned number is smaller than the size specified for the constant, the unsigned number shall be padded to the left with zeros. If the leftmost bit in the unsigned number is an `x` or a `z`, then an `x` or a `z` shall be used to pad to the left, respectively. If the size of the unsigned number is larger than the size specified for the constant, the unsigned number shall be truncated from the left.

The number of bits that make up an unsized number (which is a simple decimal number or a number without the size specification) shall be at least 32. Unsized unsigned constants where the high-order bit is unknown (`X` or `x`) or three-state (`Z` or `z`) shall be extended to the size of the expression containing the constant.

NOTE—In IEEE Std 1364-1995, in unsized constants where the high-order bit is unknown or three-state, the `x` or `z` was only extended to 32 bits.

The use of `x` and `z` in defining the value of a number is case insensitive.

 Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

When used in a number, the question-mark (?) character is a Verilog HDL alternative for the z character. It sets 4 bits to the high-impedance value in hexadecimal numbers, 3 bits in octal, and 1 bit in binary. The question mark can be used to enhance readability in cases where the high-impedance value is a do-not-care condition. See the discussion of **casez** and **casex** in 9.5.1. The question-mark character is also used in user-defined primitive (UDP) state tables. See Table 8-1 in 8.1.6.

In a decimal constant, the unsigned number token shall not contain any x, z, or ? digits, unless there is exactly one digit in the token, indicating that every bit in the decimal constant is x or z.

The underscore character (_) shall be legal anywhere in a number except as the first character. The underscore character is ignored. This feature can be used to break up long numbers for readability purposes.

For example:

Example 1—Unsized constant numbers

```
659         // is a decimal number
'h 837FF    // is a hexadecimal number
'o7460      // is an octal number
4af         // is illegal (hexadecimal format requires 'h)
```

Example 2—Sized constant numbers

```
4'b1001     // is a 4-bit binary number
5 'D 3      // is a 5-bit decimal number
3'b01x      // is a 3-bit number with the least
            // significant bit unknown
12'hx       // is a 12-bit unknown number
16'hz       // is a 16-bit high-impedance number
```

Example 3—Using sign with constant numbers

```
8 'd -6      // this is illegal syntax
-8 'd 6      // this defines the two's complement of 6,
             // held in 8 bits—equivalent to -(8'd 6)
4 'shf       // this denotes the 4-bit number '1111', to
             // be interpreted as a 2's complement number,
             // or '-1'. This is equivalent to -4'h 1
-4 'sd15     // this is equivalent to -(-4'd 1), or '0001'
16'sd?       // the same as 16'sbz
```

Example 4—Automatic left padding

```
reg [11:0] a, b, c, d;
initial begin
        a = 'h x;         // yields xxx
        b = 'h 3x;        // yields 03x
        c = 'h z3;        // yields zz3
        d = 'h 0z3;       // yields 0z3
end
reg [84:0]       e, f, g;

        e = 'h5;         // yields {82{1'b0},3'b101}
        f = 'hx;         // yields {85{1'hx}}
        g = 'hz;         // yields {85{1'hz}}
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Example 5—Using underscore character in numbers

```
27_195_000
16'b0011_0101_0001_1111
32 'h 12ab_f001
```

Sized negative constant numbers and sized signed constant numbers are sign-extended when assigned to a **reg** data type, regardless of whether the **reg** itself is signed.

The default length of `x` and `z` is the same as the default length of an integer.

3.5.2 Real constants

The *real constant numbers* shall be represented as described by IEEE Std 754-1985, an IEEE standard for double-precision floating-point numbers.

Real numbers can be specified in either decimal notation (for example, 14.72) or in scientific notation (for example, 39e8, which indicates 39 multiplied by 10 to the eighth power). Real numbers expressed with a decimal point shall have at least one digit on each side of the decimal point.

For example:

```
1.2
0.1
2394.26331
1.2E12 (the exponent symbol can be e or E)
1.30e-2
0.1e-0
23E10
29E-2
236.123_763_e-12 (underscores are ignored)
```

The following are invalid forms of real numbers because they do not have at least one digit on each side of the decimal point:

```
.12
9.
4.E3
.2e-7
```

3.5.3 Conversion

Real numbers shall be converted to integers by rounding the real number to the nearest integer, rather than by truncating it. Implicit conversion shall take place when a real number is assigned to an integer. The ties shall be rounded away from zero. For example:

— The real numbers 35.7 and 35.5 both become 36 when converted to an integer and 35.2 becomes 35.
— Converting –1.5 to integer yields –2, converting 1.5 to integer yields 2.

3.6 Strings

A *string* is a sequence of characters enclosed by double quotes (**""**) and contained on a single line. Strings used as operands in expressions and assignments shall be treated as unsigned integer constants represented by a sequence of 8-bit ASCII values, with one 8-bit ASCII value representing one character.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

3.6.1 String variable declaration

String variables are variables of **reg** type (see 4.2) with width equal to the number of characters in the string multiplied by 8.

For example:

To store the 12-character string `"Hello world!"` requires a **reg** 8 * 12, or 96 bits wide.

```
reg [8*12:1] stringvar;
initial  begin
  stringvar = "Hello world!";
end
```

3.6.2 String manipulation

Strings can be manipulated using the Verilog HDL operators. The value being manipulated by the operator is the sequence of 8-bit ASCII values.

For example:

```
module string_test;
reg [8*14:1] stringvar;
initial begin
    stringvar = "Hello world";
    $display("%s is stored as %h", stringvar,stringvar);
    stringvar = {stringvar,"!!!"};
    $display("%s is stored as %h", stringvar,stringvar);
end
endmodule
```

The output is as follows:

```
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
```

When a variable is larger than required to hold a string value being assigned, the value is right-justified, and the leftmost bits are .padded with zeros, as is done with nonstring values. If a string is larger than the destination string variable, the string is right-justified, and the leftmost characters are truncated.

3.6.3 Special characters in strings

Certain characters can only be used in strings when preceded by an introductory character called an *escape character*. Table 3-1 lists these characters in the right-hand column, with the escape sequence that represents the character in the left-hand column.

Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 3-1—Specifying special characters in string

Escape string	Character produced by escape string
\n	Newline character
\t	Tab character
\\	\ character
\"	" character
\ddd	A character specified in 1–3 octal digits ($0 \le d \le 7$). If less than three characters are used, the following character shall not be an octal digit. Implementations may issue an error if the character represented is greater than \377.

3.7 Identifiers, keywords, and system names

An *identifier* is used to give an object a unique name so it can be referenced. An identifier is either a simple identifier or an escaped identifier (see 3.7.1). A *simple identifier* shall be any sequence of letters, digits, dollar signs ($), and underscore characters (_).

The first character of a simple identifier shall not be a digit or $; it can be a letter or an underscore. Identifiers shall be case sensitive.

For example:

```
shiftreg_a
busa_index
error_condition
merge_ab
_bus3
n$657
```

Implementations may set a limit on the maximum length of identifiers, but the limit shall be at least 1024 characters. If an identifier exceeds the implementation-specified length limit, an error shall be reported.

3.7.1 Escaped identifiers

Escaped identifiers shall start with the backslash character (\) and end with white space (space, tab, newline). They provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal).

Neither the leading backslash character nor the terminating white space is considered to be part of the identifier. Therefore, an escaped identifier \cpu3 is treated the same as a nonescaped identifier cpu3.

For example:

```
\busa+index
\-clock
\***error-condition***
\net1/\net2
\{a,b}
\a*(b+c)
```

 Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

3.7.2 Keywords

Keywords are predefined nonescaped identifiers that are used to define the language constructs. A Verilog HDL keyword preceded by an escape character is not interpreted as a keyword.

All keywords are defined in lowercase only. Annex B gives a list of all defined keywords.

3.7.3 System tasks and functions

The dollar sign (**$**) introduces a language construct that enables development of user-defined system tasks and functions. System constructs are not design semantics, but refer to simulator functionality. A name following the **$** is interpreted as a *system task* or a *system function*.

The syntax for a system task/function is given in Syntax 3-2.

```
system_task_enable ::= (From A.6.9)
        system_task_identifier [ ( [ expression ] { , [ expression ] } ) ] ;
system_function_call ::= (From A.8.2)
        system_function_identifier [ ( expression { , expression } ) ]
system_function_identifier[a] ::= (From A.9.3)
        $[ a-zA-Z0-9_$ ]{ [ a-zA-Z0-9_$ ] }
system_task_identifier[a] ::=
        $[ a-zA-Z0-9_$ ]{ [ a-zA-Z0-9_$ ] }
```

[a]The dollar sign ($) in a system_function_identifier or system_task_identifier shall not be followed by white space. A system_function_identifier or system_task_identifier shall not be escaped.

Syntax 3-2—Syntax for system tasks and functions

The $identifier system task/function can be defined in three places:

— A standard set of $identifier system tasks and functions, as defined in Clause 17 and Clause 18.
— Additional $identifier system tasks and functions defined using the PLI, as described in Clause 20.
— Additional $identifier system tasks and functions defined by software implementations.

Any valid identifier, including keywords already in use in contexts other than this construct, can be used as a system task/function name. The system tasks and functions described in Clause 17 and Clause 18 are part of this standard. Additional system tasks and functions with the $identifier construct are not part of this standard.

For example:

```
$display ("display a message");
$finish;
```

3.7.4 Compiler directives

The ` character (the ASCII value 0x60, called *grave accent*) introduces a language construct used to implement compiler directives. The compiler behavior dictated by a compiler directive shall take effect as soon as the compiler reads the directive. The directive shall remain in effect for the rest of the compilation unless a different compiler directive specifies otherwise. A compiler directive in one description file can, therefore, control compilation behavior in multiple description files.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The `identifier compiler directive construct can be defined in two places:

— A standard set of `identifier compiler directives defined in Clause 19.
— Additional `identifier compiler directives defined by software implementations.

Any valid identifier, including keywords already in use in contexts other than this construct, can be used as a compiler directive name. The compiler directives described in Clause 19 are part of this standard. Additional compiler directives with the `identifier construct are not part of this standard.

For example:

```
`define wordsize 8
```

3.8 Attributes

With the proliferation of tools other than simulators that use Verilog HDL as their source, a mechanism is included for specifying properties about objects, statements, and groups of statements in the HDL source that can be used by various tools, including simulators, to control the operation or behavior of the tool. These properties shall be referred to as *attributes*. This subclause specifies the syntactic mechanism that shall be used for specifying attributes, without standardizing on any particular attributes.

The syntax for specifying an attribute is shown in Syntax 3-3.

```
attribute_instance ::= (From A.9.1)
        (* attr_spec { , attr_spec } *)
attr_spec ::=
          attr_name [ = constant_expression ]
attr_name ::=
        identifier
```

Syntax 3-3—Syntax for attributes

An `attribute_instance` can appear in the Verilog description as a prefix attached to a declaration, a module item, a statement, or a port connection. It can appear as a suffix to an operator or a Verilog function name in an expression.

If a value is not specifically assigned to the attribute, then its value shall be `1`. If the same attribute name is defined more than once for the same language element, the last attribute value shall be used; and a tool can give a warning that a duplicate attribute specification has occurred.

Nesting of attribute instances is disallowed. It shall be illegal to specify the value of an attribute with a constant expression that contains an attribute instance.

3.8.1 Examples

Example 1—The following example shows how to attach attributes to a case statement:

```
(* full_case, parallel_case *)
case (foo)
<rest_of_case_statement>
```

or

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
(* full_case=1 *)
(* parallel_case=1 *) // Multiple attribute instances also OK
case (foo)
<rest_of_case_statement>
```

or

```
(* full_case, // no value assigned
   parallel_case=1 *)
case (foo)
<rest_of_case_statement>
```

Example 2—To attach the `full_case` attribute, but not the `parallel_case` attribute:

```
(* full_case *) // parallel_case not specified
case (foo)
<rest_of_case_statement>
```

or

```
(* full_case=1, parallel_case = 0 *)
case (foo)
<rest_of_case_statement>
```

Example 3—To attach an attribute to a module definition:

```
(* optimize_power *)
module mod1 (<port_list>);
```

or

```
(* optimize_power=1 *)
module mod1 (<port_list>);
```

Example 4—To attach an attribute to a module instantiation:

```
(* optimize_power=0 *)
mod1 synth1 (<port_list>);
```

Example 5—To attach an attribute to a **reg** declaration:

```
(* fsm_state *) reg [7:0] state1;
(* fsm_state=1 *) reg [3:0] state2, state3;
reg [3:0] reg1;  // this reg does NOT have fsm_state set
(* fsm_state=0 *) reg [3:0] reg2;  // nor does this one
```

Example 6—To attach an attribute to an operator:

```
a = b + (* mode = "cla" *) c;
```

This sets the value for the attribute mode to be the string `cla`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Example 7—To attach an attribute to a Verilog function call:

```
a = add (* mode = "cla" *) (b, c);
```

Example 8—To attach an attribute to a conditional operator:

```
a = b ? (* no_glitch *) c : d;
```

3.8.2 Syntax

The syntax for legal statements with attributes is shown in Syntax 3-4 through Syntax 3-9.

The syntax for module declaration attributes is given in Syntax 3-4.

```
module_declaration ::= (From A.1.2)
        { attribute_instance } module_keyword module_identifier
          [ module_parameter_port_list ] list_of_ports ;
          { module_item }
        endmodule
      | { attribute_instance } module_keyword module_identifier
          [ module_parameter_port_list ] [ list_of_port_declarations ] ;
          { non_port_module_item }
        endmodule
```

Syntax 3-4—Syntax for module declaration attributes

The syntax for port declaration attributes is given in Syntax 3-5.

```
port_declaration ::= (From A.1.3)
        {attribute_instance} inout_declaration
      | {attribute_instance} input_declaration
      | {attribute_instance} output_declaration
```

Syntax 3-5—Syntax for port declaration attributes

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The syntax for module item attributes is given in Syntax 3-6.

```
module_item ::= (From A.1.4)
        port_declaration ;
      | non_port_module_item
module_or_generate_item ::=
        { attribute_instance } module_or_generate_item_declaration
      | { attribute_instance } local_parameter_declaration ;
      | { attribute_instance } parameter_override
      | { attribute_instance } continuous_assign
      | { attribute_instance } gate_instantiation
      | { attribute_instance } udp_instantiation
      | { attribute_instance } module_instantiation
      | { attribute_instance } initial_construct
      | { attribute_instance } always_construct
      | { attribute_instance } loop_generate_construct
      | { attribute_instance } conditional_generate_construct
non_port_module_item ::=
        module_or_generate_item
      | generate_region
      | specify_block
      | { attribute_instance } parameter_declaration ;
      | { attribute_instance } specparam_declaration
```

Syntax 3-6—Syntax for module item attributes

The syntax for function port, task, and block attributes is given in Syntax 3-7.

```
function_port_list ::= (From A.2.6)
      {attribute_instance} input_declaration { , {attribute_instance } input_declaration}
task_item_declaration ::= (From A.2.7)
        block_item_declaration
      | { attribute_instance } input_declaration ;
      | { attribute_instance } output_declaration ;
      | { attribute_instance } inout_declaration ;
task_port_item ::=
        { attribute_instance } input_declaration
      | { attribute_instance } output_declaration
      | { attribute_instance } inout_declaration
block_item_declaration ::= (From A.2.8)
        { attribute_instance } reg [ signed ] [ range ] list_of_block_variable_identifiers ;
      | { attribute_instance } integer list_of_block_variable_identifiers ;
      | { attribute_instance } time list_of_block_variable_identifiers ;
      | { attribute_instance } real list_of_block_real_identifiers ;
      | { attribute_instance } realtime list_of_block_real_identifiers ;
      | { attribute_instance } event_declaration
      | { attribute_instance } local_parameter_declaration ;
      | { attribute_instance } parameter_declaration ;
```

Syntax 3-7—Syntax for function port, task, and block attributes

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The syntax for port connection attributes is given in Syntax 3-8.

```
ordered_port_connection ::= (From A.4.1)
        { attribute_instance } [ expression ]
named_port_connection ::=
        { attribute_instance } . port_identifier ( [ expression ] )
```

Syntax 3-8—Syntax for port connection attributes

The syntax for udp attributes is given in Syntax 3-9.

```
udp_declaration ::= (From A.5.1)
        { attribute_instance } primitive udp_identifier ( udp_port_list ) ;
          udp_port_declaration { udp_port_declaration }
          udp_body
        endprimitive
      | { attribute_instance } primitive udp_identifier ( udp_declaration_port_list ) ;
          udp_body
        endprimitive
udp_output_declaration ::= (From A.5.2)
        { attribute_instance } output port_identifier
      | { attribute_instance } output reg port_identifier [ = constant_expression ]
udp_input_declaration ::=
        { attribute_instance } input list_of_port_identifiers
udp_reg_declaration ::=
        { attribute_instance } reg variable_identifier
```

Syntax 3-9—Syntax for udp attributes

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

4. Data types

The set of Verilog HDL data types is designed to represent the data storage and transmission elements found in digital hardware.

4.1 Value set

The Verilog HDL value set consists of four basic values:

```
0 - represents a logic zero, or a false condition
1 - represents a logic one, or a true condition
x - represents an unknown logic value
z - represents a high-impedance state
```

The values `0` and `1` are logical complements of one another.

When the `z` value is present at the input of a gate or when it is encountered in an expression, the effect is usually the same as an `x` value. Notable exceptions are the metal-oxide semiconductor (MOS) primitives, which can pass the `z` value.

Almost all of the data types in the Verilog HDL store all four basic values. Exceptions are the event type (see 9.7.3), which has no storage, and the real type (see 4.8). All bits of vectors can be independently set to one of the four basic values.

The language includes strength information in addition to the basic value information for net variables. This is described in detail in Clause 7.

4.2 Nets and variables

There are two main groups of data types: the variable data types and the net data types. These two groups differ in the way that they are assigned and hold values. They also represent different hardware structures.

4.2.1 Net declarations

The *net* data types can represent physical connections between structural entities, such as gates. A net shall not store a value (except for the **trireg** net). Instead, its value shall be determined by the values of its drivers, such as a continuous assignment or a gate. See Clause 6 and Clause 7 for definitions of these constructs. If no driver is connected to a net, its value shall be high-impedance (`z`) unless the net is a **trireg**, in which case it shall hold the previously driven value. It is illegal to redeclare a name already declared by a net, parameter, or variable declaration (see 4.11).

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The syntax for net declarations is given in Syntax 4-1.

```
net_declaration ::= (From A.2.1.3)
        net_type [ signed ]
         [ delay3 ] list_of_net_identifiers ;
      | net_type [ drive_strength ] [ signed ]
         [ delay3 ] list_of_net_decl_assignments ;
      | net_type [ vectored | scalared ] [ signed ]
         range [ delay3 ] list_of_net_identifiers ;
      | net_type [ drive_strength ] [ vectored | scalared ] [ signed ]
         range [ delay3 ] list_of_net_decl_assignments ;
      | trireg [ charge_strength ] [ signed ]
         [ delay3 ] list_of_net_identifiers ;
      | trireg [ drive_strength ] [ signed ]
         [ delay3 ] list_of_net_decl_assignments ;
      | trireg [ charge_strength ] [ vectored | scalared ] [ signed ]
         range [ delay3 ] list_of_net_identifiers ;
      | trireg [ drive_strength ] [ vectored | scalared ] [ signed ]
         range [ delay3 ] list_of_net_decl_assignments ;
net_type ::= (From A.2.2.1)
        supply0 | supply1
      | tri | triand | trior | tri0 | tri1 | uwire | wire | wand | wor
drive_strength ::= (From A.2.2.2)
        ( strength0 , strength1 )
      | ( strength1 , strength0 )
      | ( strength0 , highz1 )
      | ( strength1 , highz0 )
      | ( highz0 , strength1 )
      | ( highz1 , strength0 )
strength0 ::= supply0 | strong0 | pull0 | weak0
strength1 ::= supply1 | strong1 | pull1 | weak1
charge_strength ::= ( small ) | ( medium ) | ( large )
delay3 ::= (From A.2.2.3)
        # delay_value
      | # ( mintypmax_expression [ , mintypmax_expression [ , mintypmax_expression ] ] )
delay2 ::=
        # delay_value
      | # ( mintypmax_expression [ , mintypmax_expression ] )
delay_value ::=
        unsigned_number
      | real_number
      | identifier
list_of_net_decl_assignments ::= (From A.2.3)
      net_decl_assignment { , net_decl_assignment }
list_of_net_identifiers ::=
      net_identifier { dimension }
        { , net_identifier { dimension } }
net_decl_assignment ::= (From A.2.4)
      net_identifier = expression
dimension ::= (From A.2.5)
      [ dimension_constant_expression : dimension_constant_expression ]
range ::=
      [ msb_constant_expression : lsb_constant_expression ]
```

Syntax 4-1—Syntax for net declaration

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

The first two forms of net declaration are described in this subclause. The third form, called net assignment, is described in Clause 6.

The default initialization value for a net shall be the value `z`. Nets with drivers shall assume the output value of their drivers. The **trireg** net is an exception. The **trireg** net shall default to the value `x`, with the strength specified in the net declaration (**small**, **medium**, or **large**).

4.2.2 Variable declarations

A *variable* is an abstraction of a data storage element. A variable shall store a value from one assignment to the next. An assignment statement in a procedure acts as a trigger that changes the value in the data storage element. The initialization value for **reg**, **time**, and **integer** data types shall be the unknown value, `x`. The default initialization value for **real** and **realtime** variable data types shall be `0.0`. If a variable declaration assignment is used (see 6.2.1), the variable shall take this value as if the assignment occurred in a blocking assignment in an initial construct. It is illegal to redeclare a name already declared by a net, parameter, or variable declaration.

NOTE—In previous versions of this standard, the term *register* was used to encompass the **reg**, **integer**, **time**, **real**, and **realtime** types, but that term is no longer used as a Verilog data type.[8]

The syntax for variable declarations is given in Syntax 4-2.

```
integer_declaration ::= (From A.2.1.3)
      integer list_of_variable_identifiers ;
real_declaration ::=
      real list_of_real_identifiers ;
realtime_declaration ::=
      realtime list_of_real_identifiers ;
reg_declaration ::=
      reg [ signed ] [ range ] list_of_variable_identifiers ;
time_declaration ::=
      time list_of_variable_identifiers ;
real_type ::= (From A.2.2.1)
        real_identifier { dimension }
      | real_identifier = constant_expression
variable_type ::=
        variable_identifier { dimension }
      | variable_identifier = constant_expression
list_of_real_identifiers ::= (From A.2.3)
      real_type { , real_type }
list_of_variable_identifiers ::=
      variable_type { , variable_type }
dimension ::= (From A.2.5)
      [ dimension_constant_expression : dimension_constant_expression ]
range ::=
      [ msb_constant_expression : lsb_constant_expression ]
```

Syntax 4-2—Syntax for variable declaration

If a set of nets or variables share the same characteristics, they can be declared in the same declaration statement.

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

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

CAUTION

Nets and variables can be assigned negative values, but only integer, real, realtime, and signed reg variables and signed nets shall retain the significance of the sign. Time and unsigned reg variables and unsigned nets shall treat the value assigned to them as an unsigned value. See 5.1.6 for a description of how signed and unsigned nets and variables are treated by certain Verilog operators.

4.3 Vectors

A net or **reg** declaration without a range specification shall be considered 1 bit wide and is known as a *scalar*. Multibit net and **reg** data types shall be declared by specifying a range, which is known as a *vector*.

4.3.1 Specifying vectors

The range specification gives addresses to the individual bits in a multibit net or **reg**. The most significant bit specified by the *msb* constant expression is the left-hand value in the range, and the least significant bit specified by the *lsb* constant expression is the right-hand value in the range.

Both the msb constant expression and the lsb constant expression shall be constant integer expressions. The msb and lsb constant expressions may be any integer value — positive, negative, or zero. The lsb value may be greater than, equal to, or less than the msb value.

Vector nets and regs shall obey laws of arithmetic modulo-2 to the power *n* (2^n), where *n* is the number of bits in the vector. Vector nets and regs shall be treated as unsigned quantities, unless the net or **reg** is declared to be signed or is connected to a port that is declared to be signed (see 12.2.3).

For example:

```
wand w;                  // a scalar net of type "wand"
tri [15:0] busa;         // a three-state 16-bit bus
trireg (small) storeit;  // a charge storage node of strength small
reg a;                   // a scalar reg
reg[3:0] v;              // a 4-bit vector reg made up of (from most to
                         // least significant)v[3], v[2], v[1], and v[0]
reg signed [3:0] signed_reg;     // a 4-bit vector in range -8 to 7
reg [-1:4] b;            // a 6-bit vector reg
wire w1, w2;             // declares two wires
reg [4:0] x, y, z;       // declares three 5-bit regs
```

Implementations may set a limit on the maximum length of a vector, but the limit shall be at least 65536 (2^{16}) bits.

Implementations are not required to detect overflow of integer operations.

4.3.2 Vector net accessibility

Vectored and *scalared* shall be optional advisory keywords to be used in vector net or **reg** declaration. If these keywords are implemented, certain operations on vectors may be restricted. If the keyword **vectored** is used, bit-selects and part-selects and strength specifications may not be permitted, and the PLI may consider the object *unexpanded*. If the keyword **scalared** is used, bit-selects and part-selects of the object shall be permitted, and the PLI shall consider the object *expanded*.

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

For example:

```
tri1 scalared [63:0] bus64;   //a bus that will be expanded
tri vectored [31:0] data;     //a bus that may or may not be expanded
```

4.4 Strengths

Two types of *strengths* can be specified in a net declaration as follows:

— *Charge strength* shall only be used when declaring a net of type **trireg**.
— *Drive strength* shall only be used when placing a continuous assignment on a net in the same statement that declares the net.

Gate declarations can also specify a drive strength. See Clause 7 for more information on gates and for information on strengths.

4.4.1 Charge strength

The charge strength specification shall be used only with trireg nets. A trireg net shall be used to model charge storage; charge strength shall specify the relative size of the capacitance indicated by one of the following keywords:

— **small**
— **medium**
— **large**

The default charge strength of a trireg net shall be **medium**.

A trireg net can model a charge storage node whose charge decays over time. The simulation time of a charge decay shall be specified in the delay specification for the trireg net (see 7.14.2).

For example:

```
trireg a;                          // trireg net of charge strength medium
trireg (large) #(0,0,50) cap1;     // trireg net of charge strength large
                                   // with charge decay time 50 time units
trireg (small)signed [3:0] cap2;   // signed 4-bit trireg vector of
                                   // charge strength small
```

4.4.2 Drive strength

The drive strength specification allows a continuous assignment to be placed on a net in the same statement that declares that net. See Clause 6 for more details. Net strength properties are described in detail in Clause 7.

4.5 Implicit declarations

The syntax shown in 4.2 shall be used to declare nets and variables explicitly. In the absence of an explicit declaration, an implicit net of default net type shall be assumed in the following circumstances:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

— If an identifier is used in a port expression declaration, then an implicit net of default net type shall be assumed, with the vector width of the port expression declaration. See 12.3.3 for a discussion of port expression declarations.

— If an identifier is used in the terminal list of a primitive instance or a module instance, and that identifier has not been declared previously in the scope where the instantiation appears or in any scope whose declarations can be directly referenced from the scope where the instantiation appears (see 12.7), then an implicit scalar net of default net type shall be assumed.

— If an identifier appears on the left-hand side of a continuous assignment statement, and that identifier has not been declared previously in the scope where the continuous assignment statement appears or in any scope whose declarations can be directly referenced from the scope where the continuous assignment statement appears (see 12.7), then an implicit scalar net of default net type shall be assumed. See 6.1.2 for a discussion of continuous assignment statements.

The implicit net declaration belongs to the scope in which the net reference appears. For example, if the implicit net is declared by a reference in a generate block, then the net is implicitly declared only in that generate block. Subsequent references to the net from outside the generate block or in another generate block within the same module either would be illegal or would create another implicit declaration of a different net (depending on whether the reference meets the above criteria). See 12.4 for information about generate blocks.

See 19.2 for a discussion of control of the type for implicitly declared nets with the **`default_nettype** compiler directive.

4.6 Net types

There are several distinct types of nets, as shown in Table 4-1.

Table 4-1—Net types

wire	**tri**	**tri0**	**supply0**
wand	**triand**	**tri1**	**supply1**
wor	**trior**	**trireg**	**uwire**

4.6.1 Wire and tri nets

The *wire* and *tri* nets connect elements. The net types **wire** and **tri** shall be identical in their syntax and functions; two names are provided so that the name of a net can indicate the purpose of the net in that model. A **wire** net can be used for nets that are driven by a single gate or continuous assignment. The **tri** net type can be used where multiple drivers drive a net.

Logical conflicts from multiple sources of the same strength on a **wire** or a **tri** net result in `x` (unknown) values.

Table 4-2 is a truth table for resolving multiple drivers on **wire** and **tri** nets. It assumes equal strengths for both drivers. See 7.9 for a discussion of logic strength modeling.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 4-2—Truth table for wire and tri nets

wire/tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

4.6.2 Wired nets

Wired nets are of type *wor*, *wand*, *trior*, and *triand* and are used to model wired logic configurations. Wired nets use different truth tables to resolve the conflicts that result when multiple drivers drive the same net. The **wor** and **trior** nets shall create *wired or* configurations so that when any of the drivers is `1`, the resulting value of the net is `1`. The **wand** and **triand** nets shall create *wired and* configurations so that if any driver is `0`, the value of the net is `0`.

The net types **wor** and **trior** shall be identical in their syntax and functionality. The net types **wand** and **triand** shall be identical in their syntax and functionality. Table 4-3 and Table 4-4 give the truth tables for wired nets, assuming equal strengths for both drivers. See 7.9 for a discussion of logic strength modeling.

Table 4-3—Truth table for wand and triand nets

wand/triand	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

Table 4-4—Truth table for wor and trior nets

wor/trior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

4.6.3 Trireg net

The *trireg* net stores a value and is used to model charge storage nodes. A **trireg** net can be in one of two states:

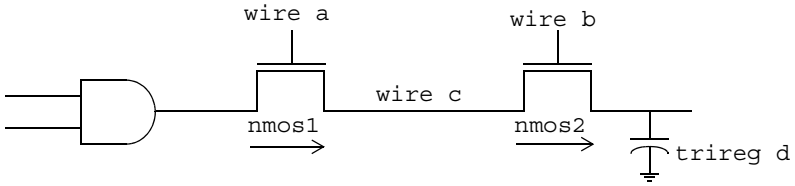
Driven state When at least one driver of a **trireg** net has a value of `1`, `0`, or `x`, the resolved value propagates into the **trireg** net and is the driven value of the **trireg** net.

Capacitive state When all the drivers of a **trireg** net are at the high-impedance value (`z`), the **trireg** net retains its last driven value; the high-impedance value does not propagate from the driver to the **trireg**.

The strength of the value on the **trireg** net in the capacitive state can be **small**, **medium**, or **large**, depending on the size specified in the declaration of the **trireg** net. The strength of a **trireg** net in the driven state can be **supply**, **strong**, **pull**, or **weak**, depending on the strength of the driver.

For example:

Figure 4-1 shows a schematic that includes a trireg net whose size is **medium**, its driver, and the simulation results.



simulation time	wire a	wire b	wire c	trireg d
0	1	1	strong 1	strong 1
10	0	1	HiZ	medium 1

Figure 4-1—Simulation values of a trireg and its driver

a) At simulation time 0, wire `a` and wire `b` have a value of `1`. A value of `1` with a `strong` strength propagates from the **and** gate through the **nmos** switches connected to each other by wire `c` into trireg net `d`.
b) At simulation time 10, wire `a` changes value to `0`, disconnecting wire `c` from the **and** gate. When wire `c` is no longer connected to the **and** gate, the value of wire `c` changes to `HiZ`. The value of wire `b` remains `1` so wire `c` remains connected to trireg net `d` through the `nmos2` switch. The `HiZ` value does not propagate from wire `c` into trireg net `d`. Instead, trireg net `d` enters the capacitive state, storing its last driven value of `1`. It stores the `1` with a **medium** strength.

4.6.3.1 Capacitive networks

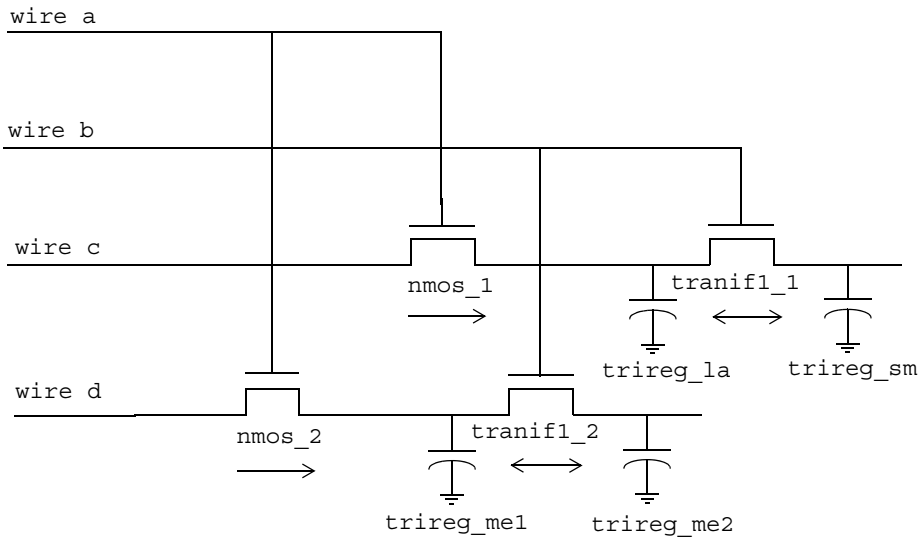
A capacitive network is a connection between two or more trireg nets. In a capacitive network whose trireg nets are in the capacitive state, logic and strength values can propagate between trireg nets.

For example:

Figure 4-2 shows a capacitive network in which the logic value of some trireg nets change the logic value of other trireg nets of equal or smaller size.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale



simulation time	wire a	wire b	wire c	wire d	trireg_la	trireg_sm	trireg_me1	trireg_me2
0	1	1	1	1	1	1	1	1
10	1	[0]	1	1	1	1	1	1
20	1	0	[0]	1	[0]	1	1	1
30	1	0	0	[0]	0	1	[0]	1
40	[0]	0	0	0	0	1	0	1
50	0	[1]	0	0	0	[0]	[x]	[x]

Figure 4-2—Simulation results of a capacitive network

In Figure 4-2, the capacitive strength of `trireg_la` net is **large**, `trireg_me1` and `trireg_me2` are **medium**, and `trireg_sm` is **small**. Simulation reports the following sequence of events:

a) At simulation time 0, wire `a` and wire `b` have a value of `1`. The wire `c` drives a value of `1` into `trireg_la` and `trireg_sm`; wire `d` drives a value of 1 into `trireg_me1` and `trireg_me2`.
b) At simulation time 10, the value of wire `b` changes to `0`, disconnecting `trireg_sm` and `trireg_me2` from their drivers. These trireg nets enter the capacitive state and store the value `1`, their last driven value.
c) At simulation time 20, wire `c` drives a value of `0` into `trireg_la`.
d) At simulation time 30, wire `d` drives a value of `0` into `trireg_me1`.
e) At simulation time 40, the value of wire `a` changes to `0`, disconnecting `trireg_la` and `trireg_me1` from their drivers. These trireg nets enter the capacitive state and store the value `0`.
f) At simulation time 50, the value of wire `b` changes to `1`.

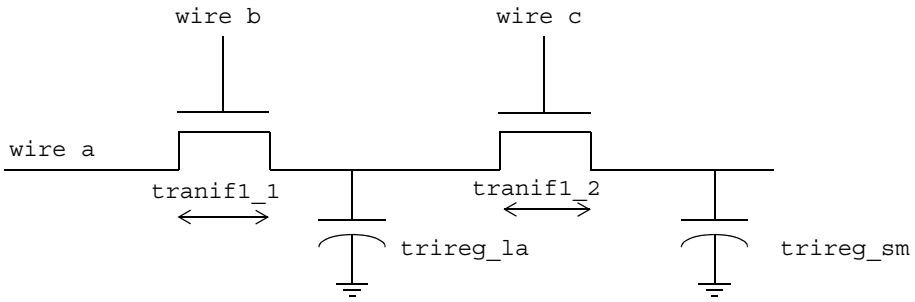
   This change of value in wire `b` connects `trireg_sm` to `trireg_la`; these trireg nets have different sizes and stored different values. This connection causes the smaller trireg net to store the value of the larger trireg net, and `trireg_sm` now stores a value of `0`.

   This change of value in wire `b` also connects `trireg_me1` to `trireg_me2`; these trireg nets have the same size and stored different values. The connection causes both `trireg_me1` and `trireg_me2` to change value to `x`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

In a capacitive network, charge strengths propagate from a larger trireg net to a smaller trireg net. Figure 4-3 shows a capacitive network and its simulation results.



simulation time	wire a	wire b	wire c	trireg_la	trireg_sm
0	strong 1	1	1	strong 1	strong 1
10	strong 1	0	1	large 1	large 1
20	strong 1	0	0	large 1	small 1
30	strong 1	0	1	large 1	large 1
40	strong 1	0	0	large 1	small 1

Figure 4-3—Simulation results of charge sharing

In Figure 4-3, the capacitive strength of `trireg_la` is **large**, and the capacitive strength of `trireg_sm` is **small**. Simulation reports the following results:

a) At simulation time 0, the values of wire `a`, wire `b`, and wire `c` are `1`, and wire `a` drives a `strong 1` into `trireg_la` and `trireg_sm`.
b) At simulation time 10, the value of wire `b` changes to `0`, disconnecting `trireg_la` and `trireg_sm` from wire `a`. The `trireg_la` and `trireg_sm` nets enter the capacitive state. Both trireg nets share the **large** charge of `trireg_la` because they remain connected through `tranif1_2`.
c) At simulation time 20, the value of wire `c` changes to `0`, disconnecting `trireg_sm` from `trireg_la`. The `trireg_sm` no longer shares **large** charge of `trireg_la` and now stores a **small** charge.
d) At simulation time 30, the value of wire `c` changes to `1`, connecting the two trireg nets. These trireg nets now share the same charge.
e) At simulation time 40, the value of wire `c` changes again to `0`, disconnecting `trireg_sm` from `trireg_la`. Once again, `trireg_sm` no longer shares the **large** charge of `trireg_la` and now stores a **small** charge.

4.6.3.2 Ideal capacitive state and charge decay

A *trireg* net can retain its value indefinitely, or its charge can decay over time. The simulation time of charge decay is specified in the delay specification of the **trireg** net. See 7.14.2 for charge decay explanation.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

4.6.4 Tri0 and tri1 nets

The *tri0* and *tri1* nets model nets with resistive *pulldown* and resistive *pullup* devices on them. A **tri0** net is equivalent to a wire net with a continuous `0` value of **pull** strength driving it. A **tri1** net is equivalent to a wire net with a continuous `1` value of **pull** strength driving it.

When no driver drives a **tri0** net, its value is `0` with strength **pull**. When no driver drives a **tri1** net, its value is `1` with strength **pull**. When there are drivers on a **tri0** or **tri1** net, the drivers combine with the strength **pull** value implicitly driven on the net to determine the net's value. See 7.9 for a discussion of logic strength modeling.

Table 4-5 and Table 4-6 are truth tables for modeling multiple drivers of strength **strong** on **tri0** and **tri1** nets. The resulting value on the net has strength **strong**, unless both drivers are `z`, in which case the net has strength **pull**.

Table 4-5—Truth table for tri0 net

tri0	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	0

Table 4-6—Truth table for tri1 net

tri1	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	1

4.6.5 Unresolved nets

The **uwire** net is an unresolved or unidriver wire and is used to model nets that allow only a single driver. The **uwire** type can be used to enforce this restriction. It shall be an error to connect any bit of a **uwire** net to more than one driver. It shall be an error to connect a **uwire** net to a bidirectional terminal of a bidirectional pass switch.

The port connection rule in 12.3.9.3 ensures that an implementation enforces this restriction across the net hierarchy or gives a warning if it does not.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

4.6.6 Supply nets

The *supply0* and *supply1* nets can be used to model the power supplies in a circuit. These nets shall have **supply** strengths.

4.7 Regs

Assignments to a **reg** are made by procedural assignments (see 6.2 and 9.2). Because the **reg** holds a value between assignments, it can be used to model hardware registers. Edge-sensitive (i.e., flip-flops) and level-sensitive (i.e., reset-set and transparent latches) storage elements can be modeled. A **reg** need not represent a hardware storage element because it can also be used to represent combinatorial logic.

4.8 Integers, reals, times, and realtimes

In addition to modeling hardware, there are other uses for variables in an HDL model. Although **reg** variables can be used for general purposes such as counting the number of times a particular net changes value, the integer and time variable data types are provided for convenience and to make the description more self-documenting.

The syntax for declaring **integer**, **time**, **real**, and **realtime** variables is given in Syntax 4-3 (from Syntax 4-2).

```
integer_declaration ::= (From A.2.1.3)
      integer list_of_variable_identifiers ;
real_declaration ::=
      real list_of_real_identifiers ;
realtime_declaration ::=
      realtime list_of_real_identifiers ;
time_declaration ::=
      time list_of_variable_identifiers ;
real_type ::= (From A.2.2.1)
        real_identifier { dimension }
      | real_identifier = constant_expression
variable_type ::=
        variable_identifier { dimension }
      | variable_identifier = constant_expression
list_of_real_identifiers ::= (From A.2.3)
      real_type { , real_type }
list_of_variable_identifiers ::=
      variable_type { , variable_type }
dimension ::= (From A.2.5)
      [ dimension_constant_expression : dimension_constant_expression ]
```

Syntax 4-3—Syntax for integer, time, real, and realtime declarations

The syntax for a list of **reg** variables is defined in 4.2.2.

An **integer** is a general-purpose variable used for manipulating quantities that are not regarded as hardware registers.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

A **time** variable is used for storing and manipulating simulation time quantities in situations where timing checks are required and for diagnostics and debugging purposes. This data type is typically used in conjunction with the **$time** system function (see 17.7.1).

The **integer** and **time** variables shall be assigned values in the same manner as **reg**. Procedural assignments shall be used to trigger their value changes.

The **time** variables shall behave the same as a **reg** of at least 64 bits, with the least significant bit being bit 0. They shall be unsigned quantities, and unsigned arithmetic shall be performed on them. In contrast, **integer** variables shall be treated as signed **regs** with the least significant bit being zero. Arithmetic operations performed on **integer** variables shall produce twos-complement results.

Bit-selects and part-selects of vector **regs**, **integer** variables, and **time** variables shall be allowed (see 5.2).

Implementations may limit the maximum size of **integer** variables, but it shall be at least 32 bits.

The Verilog HDL supports real number constants and real variable data types in addition to integer and time variable data types. Except for the following restrictions, variables declared as **real** can be used in the same places that integer and time variables are used:

— Not all Verilog HDL operators can be used with real number values. See Table 5-2 and Table 5-3 for lists of valid and invalid operators for real numbers and real variables.
— Real variables shall not use range in the declaration.
— Real variables shall default to an initial value of zero.

The realtime declarations shall be treated synonymously with real declarations and can be used interchangeably.

For example:

```
integer a;                // integer value
time last_chng;           // time value
real float ;              // a variable to store a real value
realtime rtime ;          // a variable to store time as a real value
```

4.8.1 Operators and real numbers

The result of using logical or relational operators on real numbers and real variables is a single-bit scalar value. Not all Verilog HDL operators can be used with expressions involving real numbers and real variables. Table 5-2 lists the valid operators for use with real numbers and real variables. Real number constants and real variables are also prohibited in the following cases:

— Edge descriptors (**posedge**, **negedge**) applied to real variables
— Bit-select or part-select references of variables declared as **real**
— Real number index expressions of bit-select or part-select references of vectors

4.8.2 Conversion

Real numbers shall be converted to integers by rounding the real number to the nearest integer, rather than by truncating it. Implicit conversion shall take place when a real number is assigned to an integer. If the fractional part of the real number is exactly 0.5, it shall be rounded away from zero.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Implicit conversion shall take place when an expression is assigned to a real. Individual bits that are `x` or `z` in the net or the variable shall be treated as zero upon conversion.

See 17.8 for a discussion of system tasks that perform explicit conversion.

4.9 Arrays

An array declaration for a net or a variable declares an element type that is either scalar or vector (see 4.3). For example:

Declaration	Element type
reg x[11:0];	scalar **reg**
wire [0:7] y[5:0];	8-bit-wide vector wire indexed from 0 to 7
reg [31:0] x [127:0];	32-bit-wide **reg**

NOTE—Array size does not affect the element size.

Arrays can be used to group elements of the declared element type into multidimensional objects. Arrays shall be declared by specifying the element address range(s) after the declared identifier. Each dimension shall be represented by an address range. See 4.2.1 and 4.2.2 for net and variable declarations. The expressions that specify the indices of the array shall be constant integer expressions. The value of the constant expression can be a positive integer, a negative integer, or zero.

One declaration statement can be used for declaring both arrays and elements of the declared data type. This ability makes it convenient to declare both arrays and elements that match the element vector width in the same declaration statement.

An element can be assigned a value in a single assignment, but complete or partial array dimensions cannot. Nor can complete or partial array dimensions be used to provide a value to an expression. To assign a value to an element of an array, an index for every dimension shall be specified. The index can be an expression. This option provides a mechanism to reference different array elements depending on the value of other variables and nets in the circuit. For example, a program counter **reg** can be used to index into a random access memory (RAM).

Implementations may limit the maximum size of an array, but they shall allow at least 16 777 216 (2^{24}) elements.

4.9.1 Net arrays

Elements of net arrays can be used in the same fashion as a scalar or vector net. They are useful for connecting to ports of module instances inside loop generate constructs (see 12.4.1).

4.9.2 reg and variable arrays

Arrays for all variables types (**reg**, **integer**, **time**, **real**, **realtime**) shall be possible.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

4.9.3 Memories

A one-dimensional array with elements of type **reg** is also called a *memory*. These memories can be used to model read-only memories (ROMs), random access memories (RAMs), and **reg** files. Each **reg** in the array is known as an *element* or *word* and is addressed by a single array index.

An *n*-bit **reg** can be assigned a value in a single assignment, but a complete memory cannot. To assign a value to a memory word, an index shall be specified. The index can be an expression. This option provides a mechanism to reference different memory words, depending on the value of other variables and nets in the circuit. For example, a program counter **reg** could be used to index into a RAM.

4.9.3.1 Array examples

4.9.3.1.1 Array declarations

```
reg [7:0] mema[0:255];      // declares a memory mema of 256 8-bit
                            // registers. The indices are 0 to 255

reg arrayb[7:0][0:255];     // declare a two-dimensional array of
                            // one bit registers
wire w_array[7:0][5:0];     // declare array of wires
integer inta[1:64];         // an array of 64 integer values
time chng_hist[1:1000]      // an array of 1000 time values
integer t_index;
```

4.9.3.1.2 Assignment to array elements

The assignment statements in this subclause assume the presence of the declarations in 4.9.3.1.1.

```
mema = 0;        // Illegal syntax- Attempt to write to entire array
arrayb[1] = 0;   // Illegal Syntax - Attempt to write to elements
                 // [1][0]..[1][255]
arrayb[1][12:31] = 0; // Illegal Syntax - Attempt to write to
                      //  elements [1][12]..[1][31]
mema[1] = 0;      // Assigns 0 to the second element of mema
arrayb[1][0] = 0; // Assigns 0 to the bit referenced by indices
                  // [1][0]
inta[4] = 33559;  // Assign decimal number to integer in array
chng_hist[t_index] = $time;    // Assign current simulation time to
                               //  element addressed by integer index
```

4.9.3.1.3 Memory differences

A memory of n 1-bit regs is different from an n-bit vector **reg**.

```
reg [1:n] rega; // An n-bit register is not the same
reg mema [1:n]; // as a memory of n 1-bit registers
```

4.10 Parameters

Verilog HDL parameters do not belong to either the variable or the net group. Parameters are not variables; they are constants. There are two types of parameters: module parameters and specify parameters. It is illegal to redeclare a name already declared by a net, parameter, or variable declaration.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Both types of parameters accept a range specification. By default, *parameters* and *specparams* shall be as wide as necessary to contain the value of the constant, except when a range specification is present.

4.10.1 Module parameters

The syntax for module parameter declarations is given in Syntax 4-4.

```
local_parameter_declaration ::=  (From A.2.1.1)
        localparam [ signed ] [ range ] list_of_param_assignments
      | localparam parameter_type list_of_param_assignments
parameter_declaration ::=
        parameter [ signed ] [ range ] list_of_param_assignments
      | parameter parameter_type list_of_param_assignments
parameter_type ::=
      integer | real | realtime | time
list_of_param_assignments ::= (From A.2.3)
      param_assignment { , param_assignment }
param_assignment ::= (From A.2.4)
      parameter_identifier = constant_mintypmax_expression
range ::=  (From A.2.5)
      [ msb_constant_expression : lsb_constant_expression ]
```

Syntax 4-4—Syntax for module parameter declaration

The *list_of_param_assignments* shall be a comma-separated list of assignments, where the right-hand side of the assignment shall be a *constant expression*, that is, an expression containing only constant numbers and previously defined parameters (see Clause 5).

The *list_of_param_assignments* can appear in a module as a set of *module_items* or in the module declaration in the *module_parameter_port_list* (see 12.1). If any *param_assignments* appear in a *module_parameter_port_list*, then any *param_assignments* that appear in the module become local parameters and shall not be overridden by any method.

Parameters represent constants; hence, it is illegal to modify their value at run time. However, module parameters can be modified at compilation time to have values that are different from those specified in the declaration assignment. This allows customization of module instances. A parameter can be modified with the **defparam** statement or in the module instance statement. Typical uses of parameters are to specify delays and width of variables. See 12.2 for details on parameter value assignment.

A module parameter can have a *type* specification and a *range* specification. The type and range of module parameters shall be in accordance with the following rules:

— A parameter declaration with no type or range specification shall default to the type and range of the final value assigned to the parameter, after any value overrides have been applied.
— A parameter with a range specification, but with no type specification, shall be the range of the parameter declaration and shall be unsigned. The sign and range shall not be affected by value overrides.
— A parameter with a type specification, but with no range specification, shall be of the type specified. A signed parameter shall default to the range of the final value assigned to the parameter, after any value overrides have been applied.
— A parameter with a signed type specification and with a range specification shall be signed and shall be the range of its declaration. The sign and range shall not be affected by value overrides.

 Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

— A parameter with no range specification and with either a signed type specification or no type specification shall have an implied range with an *lsb* equal to `0` and an *msb* equal to one less than the size of the final value assigned to the parameter.

— A parameter with no range specification, with either a signed type specification or no type specification, and for which the final value assigned to it is unsized shall have an implied range with an *lsb* equal to `0` and an *msb* equal to an implementation-dependent value of at least 31.

The conversion rules between real and integer values described in 4.8.2 apply to parameters as well.

Bit-selects and part-selects of parameters that are not of type **real** shall be allowed (see 5.2).

For example:

```
parameter  msb = 7;                    // defines msb as a constant value 7
parameter  e = 25, f = 9;              // defines two constant numbers
parameter  r = 5.7;                    // declares r as a real parameter
parameter  byte_size = 8,
           byte_mask = byte_size - 1;
parameter  average_delay = (r + f) / 2;

parameter signed [3:0] mux_selector = 0;
parameter real r1 = 3.5e17;
parameter  p1 = 13'h7e;
parameter  [31:0] dec_const = 1'b1;   // value converted to 32 bits
parameter  newconst = 3'h4;           // implied range of [2:0]
parameter  newconst = 4;              // implied range of at least [31:0]
```

4.10.2 Local parameters (localparam)

Verilog HDL local parameters are identical to parameters except that they cannot directly be modified by **defparam** statements (see 12.2.1) or module instance parameter value assignments (see 12.2.2). Local parameters can be assigned constant expressions containing parameters, which can be modified with **defparam** statements or module instance parameter value assignments.

Bit-selects and part-selects of local parameters that are not of type **real** shall be allowed (see 5.2).

The syntax for local parameter declarations is given in Syntax 4-4.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

4.10.3 Specify parameters

The syntax for declaring specify parameters is shown in Syntax 4-5.

```
specparam_declaration ::= (From A.2.1.1)
      specparam [ range ] list_of_specparam_assignments ;
list_of_specparam_assignments ::= (From A.2.3)
      specparam_assignment { , specparam_assignment }
specparam_assignment ::= (From A.2.4)
        specparam_identifier = constant_mintypmax_expression
      | pulse_control_specparam
pulse_control_specparam ::=
        PATHPULSE$ = ( reject_limit_value [ , error_limit_value ] )
      | PATHPULSE$specify_input_terminal_descriptor$specify_output_terminal_descriptor
        = ( reject_limit_value [ , error_limit_value ] )
error_limit_value ::=
      limit_value
reject_limit_value ::=
      limit_value
limit_value ::=
      constant_mintypmax_expression
range ::= (From A.2.5)
      [ msb_constant_expression : lsb_constant_expression ]
```

Syntax 4-5—Syntax for specparam declaration

The keyword **specparam** declares a special type of parameter that is intended only for providing timing and delay values, but can appear in any expression that is not assigned to a parameter and is not part of the range specification of a declaration. Specify parameters (also called *specparams*) are permitted both within the specify block (see Clause 14) and in the main module body.

A specify parameter declared outside a specify block shall be declared before it is referenced. The value assigned to a specify parameter can be any constant expression. A specify parameter can be used as part of a constant expression for a subsequent specify parameter declaration. Unlike a module parameter, a specify parameter cannot be modified from within the language, but it can be modified through SDF annotation (see Clause 16).

Specify parameters and module parameters are not interchangeable. In addition, module parameters shall not be assigned a constant expression that includes any specify parameters. Table 4-7 summarizes the differences between the two types of parameter declarations.

Table 4-7—Differences between specparams and parameters

Specparams (specify parameter)	Parameters (module parameter)
Use keyword **specparam**	Use keyword **parameter**
Shall be declared *inside* a module or specify block	Shall be declared *outside* specify blocks
May only be used inside a module or specify block	May not be used inside specify blocks
May be assigned specparams and parameters	May not be assigned specparams
Use SDF annotation to override values	Use **defparam** or instance declaration parameter value passing to override values

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

A specify parameter can have a range specification. The range of specify parameters shall be in accordance with the following rules:

— A specparam declaration with no range specification shall default to the range of the final value assigned to the parameter, after any value overrides have been applied.
— A specparam with a range specification shall be the range of the parameter declaration. The range shall not be affected by value overrides.

Bit-selects and part-selects of specify parameters that are not of type **real** shall be allowed (see 5.2).

For example:

```
specify
    specparam tRise_clk_q = 150, tFall_clk_q = 200;
    specparam tRise_control = 40, tFall_control = 50;
endspecify
```

The lines between the keywords **specify** and **endspecify** declare four specify parameters. The first line declares specify parameters called `tRise_clk_q` and `tFall_clk_q` with values `150` and `200`, respectively; the second line declares `tRise_control` and `tFall_control` specify parameters with values `40` and `50`, respectively.

For example:

```
module RAM16GEN (output [7:0] DOUT, input [7:0] DIN, input [5:0] ADR,
    input WE, CE);
specparam dhold = 1.0;
specparam ddly = 1.0;
parameter width = 1;
parameter regsize = dhold + 1.0;  // Illegal - cannot assign
                                  // specparams to parameters
endmodule
```

4.11 Name spaces

In Verilog HDL, there are several name spaces; two are global and the rest are local. The global name spaces are definitions and text macros. The *definitions name space* unifies all the **module** (see 12.1) and **primitive** (see 8.1) definitions. Once a name is used to define a module or primitive, the name shall not be used again to declare another module or primitive.

The *text macro name space* is global. Because text macro names are introduced and used with a leading ` character, they remain unambiguous with any other name space (see 19.3). The text macro names are defined in the linear order of appearance in the set of input files that make up the description of the design unit. Subsequent definitions of the same name override the previous definitions for the balance of the input files.

The local name spaces are block, module, generate block, port, specify block, and attribute. Once a name is defined within the block, module, port, generate block, or specify block name space, it shall not be defined again in that space (with the same or a different type). As described in 3.8, it is legal to redefine names within the attribute name space.

The *block name space* is introduced by the named block (see 9.8), function (see 10.4), and task (see 10.2) constructs. It unifies the definitions of the named blocks, functions, tasks, parameters, named events, and

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

variable type of declaration (see 4.2.2). The variable type of declaration includes the **reg**, **integer**, **time**, **real**, and **realtime** declarations.

The *module name space* is introduced by the **module** and **primitive** constructs. It unifies the definition of functions, tasks, named blocks, module instances, generate blocks, parameters, named events, genvars, net type of declaration, and variable type of declaration. The net type of declaration includes **wire**, **wor**, **wand**, **tri**, **trior**, **triand**, **tri0**, **tri1**, **trireg**, **uwire**, **supply0**, and **supply1** (see 4.6).

The *generate block name space* is introduced by generate constructs (see 12.4). It unifies the definition of functions, tasks, named blocks, module instances, generate blocks, local parameters, named events, genvars, net type of declaration, and variable type of declaration.

The *port name space* is introduced by the **module**, **primitive**, **function**, and **task** constructs. It provides a means of structurally defining connections between two objects that are in two different name spaces. The connection can be unidirectional (either **input** or **output**) or bidirectional (**inout**). The port name space overlaps the module and the block name spaces. Essentially, the port name space specifies the type of connection between names in different name spaces. The port type of declarations include **input**, **output**, and **inout** (see 12.3). A port name introduced in the port name space may be reintroduced in the module name space by declaring a variable or a wire with the same name as the port name.

The *specify block name space* is introduced by the **specify** construct (see 14.2).

The *attribute name space* is enclosed by the **(*** and ***)** constructs attached to a language element (see 3.8). An attribute name can be defined and used only in the attribute name space. Any other type of name cannot be defined in this name space.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

5. Expressions

This clause describes the operators and operands available in the Verilog HDL and how to use them to form expressions.

An *expression* is a construct that combines operands with operators to produce a result that is a function of the values of the operands and the semantic meaning of the operator. Any legal operand, such as a net bit-select, without any operator is considered an expression. Wherever a value is needed in a Verilog HDL statement, an expression can be used.

Some statement constructs require an expression to be a *constant expression*. The operands of a constant expression consist of constant numbers, strings, parameters, constant bit-selects and part-selects of parameters, constant function calls (see 10.4.5), and constant system function calls only; but they can use any of the operators defined in Table 5-1.

Constant system function calls are calls to certain built-in system functions where the arguments are constant expressions. When used in constant expressions, these function calls shall be evaluated at elaboration time. The system functions that may be used in constant system function calls are *pure functions*, i.e., those whose value depends only on their input arguments and which have no side effects. Specifically, the system functions allowed in constant expressions are the conversion system functions listed in 17.8 and the mathematical system functions listed in 17.11.

The data types **reg**, **integer**, **time**, **real**, and **realtime** are all variable data types. Descriptions pertaining to variable usage apply to all of these data types.

An *operand* can be one of the following:

- Constant number (including **real**) or string
- Parameter (including local and specify parameters)
- Parameter (not **real**) bit-select or part-select (including local and specify parameters)
- Net
- Net bit-select or part-select
- **reg**, **integer**, or **time** variable
- **reg**, **integer**, or **time** variable bit-select or part-select
- **real** or **realtime** variable
- Array element
- Array element (not **real**) bit-select or part-select
- A call to a user-defined function or system-defined function that returns any of the above

5.1 Operators

The symbols for the Verilog HDL operators are similar to those in the C programming language. Table 5-1 lists these operators.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 5-1—Operators in Verilog HDL

{} {{}}	Concatenation, replication
unary + unary -	Unary operators
+ - * / **	Arithmetic
%	Modulus
> >= < <=	Relational
!	Logical negation
&&	Logical and
||	Logical or
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality
~	Bitwise negation
&	Bitwise and
|	Bitwise inclusive or
^	Bitwise exclusive or
^~ or ~^	Bitwise equivalence
&	Reduction and
~&	Reduction nand
|	Reduction or
~|	Reduction nor
^	Reduction xor
~^ or ^~	Reduction xnor
<<	Logical left shift
>>	Logical right shift
<<<	Arithmetic left shift
>>>	Arithmetic right shift
? :	Conditional

5.1.1 Operators with real operands

The operators shown in Table 5-2 shall be legal when applied to real operands. All other operators shall be considered illegal when used with real operands.

The result of using logical or relational operators on real numbers is a single-bit scalar value.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 5-2—Legal operators for use in real expressions

unary + unary -	Unary operators
+ - * / **	Arithmetic
> >= < <=	Relational
! && ||	Logical
== !=	Logical equality
?:	Conditional

Table 5-3 lists operators that shall not be used to operate on real numbers.

Table 5-3—Operators not allowed for real expressions

{} {{}}	Concatenate, replicate
%	Modulus
=== !==	Case equality
~ & | ^ ^~ ~^	Bitwise
^ ^~ ~^ & ~& | ~|	Reduction
<< >> <<< >>>	Shift

See 4.8.1 for more information on use of real numbers.

5.1.2 Operator precedence

The precedence order of the Verilog operators is shown in Table 5-4.

Operators shown on the same row in Table 5-4 shall have the same precedence. Rows are arranged in order of decreasing precedence for the operators. For example, *****, /, and **%** all have the same precedence, which is higher than that of the binary + and – operators.

All operators shall associate left to right with the exception of the conditional operator, which shall associate right to left. Associativity refers to the order in which the operators having the same precedence are evaluated. Thus, in the following example, `B` is added to `A`, and then `C` is subtracted from the result of `A+B`.

```
A + B - C
```

When operators differ in precedence, the operators with higher precedence shall associate first. In the following example, `B` is divided by `C` (division has higher precedence than addition), and then the result is added to `A`.

```
A + B / C
```

Parentheses can be used to change the operator precedence.

```
(A + B) / C      // not the same as A + B / C
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 5-4—Precedence rules for operators

+ - ! ~ & ~& | ~| ^ ~^ ^~ (unary)	Highest precedence
**	
* / %	
+ - (binary)	
<< >> <<< >>>	
< <= > >=	
== != === !==	
& (binary)	
^ ^~ ~^ (binary)	
| (binary)	
&&	
||	
?: (conditional operator)	
{} {{}}	Lowest precedence

5.1.3 Using integer numbers in expressions

Integer numbers can be used as operands in expressions. An integer number can be expressed as

— An unsized, unbased integer (e.g., `12`)
— An unsized, based integer (e.g., `'d12`, `'sd12`)
— A sized, based integer (e.g., `16'd12`, `16'sd12`)

A negative value for an integer with no base specifier shall be interpreted differently from an integer with a base specifier. An integer with no base specifier shall be interpreted as a signed value in twos-complement form. An integer with an unsigned base specifier shall be interpreted as an unsigned value.

For example:

This example shows four ways to write the expression “minus 12 divided by 3.” Note that `-12` and `-'d12` both evaluate to the same twos-complement bit pattern, but, in an expression, the `-'d12` loses its identity as a signed negative number.

```
integer IntA;
IntA = -12 / 3;          // The result is -4.

IntA = -'d 12 / 3;       // The result is 1431655761.

IntA = -'sd 12 / 3;      // The result is -4.

IntA = -4'sd 12 / 3;     // -4'sd12 is the negative of the 4-bit
                         // quantity 1100, which is -4. -(-4) = 4.
                         // The result is 1.
```


--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

5.1.4 Expression evaluation order

The operators shall follow the associativity rules while evaluating an expression as described in 5.1.2. However, if the final result of an expression can be determined early, the entire expression need not be evaluated. This is called *short-circuiting* an expression evaluation.

For example:

```
reg regA, regB, regC, result ;
result = regA & (regB | regC) ;
```

If `regA` is known to be zero, the result of the expression can be determined as zero without evaluating the subexpression `regB | regC`.

5.1.5 Arithmetic operators

The binary arithmetic operators are given in Table 5-5.

Table 5-5—Arithmetic operators defined

a + b	a plus b
a - b	a minus b
a * b	a multiplied by b (or a times b)
a / b	a divided by b
a % b	a modulo b
a ** b	a to the power of b

The integer division shall truncate any fractional part toward zero. For the division or modulus operators, if the second operand is a zero, then the entire result value shall be `x`. The modulus operator (for example, `y % z`) gives the remainder when the first operand is divided by the second and thus is zero when z divides y exactly. The result of a modulus operation shall take the sign of the first operand.

If either operand of the power operator is real, then the result type shall be real. The result of the power operator is unspecified if the first operand is zero and the second operand is nonpositive or if the first operand is negative and the second operand is not an integral value.

If neither operand of the power operator is real, then the result type shall be determined as outlined in 5.4.1 and 5.5.1. The result value is `'bx` if the first operand is zero and the second operand is negative. The result value is `1` if the second operand is zero.

In all cases, the second operand of the power operator shall be treated as self-determined.

These statements are illustrated in Table 5-6.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 5-6—Power operator rules

op1 is op2 is	negative < –1	–1	zero	1	positive > 1
positive	op1 ** op2	op2 is odd -> –1 op2 is even -> 1	0	1	op1 ** op2
zero	1	1	1	1	1
negative	0	op2 is odd -> –1 op2 is even -> 1	'bx	1	0

The unary arithmetic operators shall take precedence over the binary operators. The unary operators are given in Table 5-7.

Table 5-7—Unary operators defined

+m	Unary plus m (same as m)
-m	Unary minus m

For the arithmetic operators, if any operand bit value is the unknown value `x` or the high-impedance value `z`, then the entire result value shall be `x`.

For example:

Table 5-8 gives examples of some modulus and power operations.

Table 5-8—Examples of modulus and power operators

Expression	Result	Comments
10 % 3	1	10/3 yields a remainder of 1.
11 % 3	2	11/3 yields a remainder of 2.
12 % 3	0	12/3 yields no remainder.
–10 % 3	–1	The result takes the sign of the first operand.
11 % –3	2	The result takes the sign of the first operand
–4'd12 % 3	1	–4'd12 is seen as a large positive number that leaves a remainder of 1 when divided by 3.
3 ** 2	9	3 * 3
2 ** 3	8	2 * 2 * 2
2 ** 0	1	Anything to the zero exponent is 1.
0 ** 0	1	Zero to the zero exponent is also 1.
2.0 ** –3'sb1	0.5	2.0 is real, giving real reciprocal.
2 ** –3 'sb1	0	2 ** –1 = 1/2. Integer division truncates to zero.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 5-8—Examples of modulus and power operators *(continued)*

Expression	Result	Comments
0 ** –1	'bx	0 ** –1 = 1/0. Integer division by zero is 'bx.
9 ** 0.5	3.0	Real square root.
9.0 ** (1/2)	1.0	Integer division truncates exponent to zero.
–3.0 ** 2.0	9.0	Defined because real 2.0 is still integral value.

5.1.6 Arithmetic expressions with regs and integers

A value assigned to a **reg** variable or a net shall be treated as an unsigned value unless the **reg** variable or net has been explicitly declared to be signed. A value assigned to an **integer**, **real** or **realtime** variable shall be treated as signed. A value assigned to a **time** variable shall be treated as unsigned. Signed values, except for those assigned to **real** and **realtime** variables, shall use a twos-complement representation. Values assigned to **real** and **realtime** variables shall use a floating-point representation. Conversions between signed and unsigned values shall keep the same bit representation; only the interpretation changes.

Table 5-9 lists how arithmetic operators interpret each data type.

Table 5-9—Data type interpretation by arithmetic operators

Data type	Interpretation
unsigned net	Unsigned
signed net	Signed, twos complement
unsigned reg	Unsigned
signed reg	Signed, twos complement
integer	Signed, twos complement
time	Unsigned
real, realtime	Signed, floating point

For example:

The following example shows various ways to divide “minus twelve by three”—using **integer** and **reg** data types in expressions.

```
integer intA;
reg [15:0] regA;
reg signed [15:0] regS;

intA = -4'd12;
regA = intA / 3;     // expression result is -4,
                     // intA is an integer data type, regA is 65532

regA = -4'd12;       // regA is 65524
intA = regA / 3;     // expression result is 21841,
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
                        // regA is a reg data type

intA = -4'd12 / 3;      // expression result is 1431655761.
                        // -4'd12 is effectively a 32-bit reg data type

regA = -12 / 3;         // expression result is -4, -12 is effectively
                        // an integer data type. regA is 65532

regS = -12 / 3;         // expression result is -4. regS is a signed reg

regS = -4'sd12 / 3;     // expression result is 1. -4'sd12 is actually 4.
                        // The rules for integer division yield 4/3==1.
```

5.1.7 Relational operators

Table 5-10 lists and defines the relational operators.

Table 5-10—Definitions of relational operators

a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b

An expression using these relational operators shall yield the scalar value `0` if the specified relation is false or the value `1` if it is true. If either operand of a relational operator contains an unknown (`x`) or high-impedance (`z`) value, then the result shall be a 1-bit unknown value `(x)`.

When one or both operands of a relational expression are unsigned, the expression shall be interpreted as a comparison between unsigned values. If the operands are of unequal bit lengths, the smaller operand shall be zero-extended to the size of the larger operand.

When both operands are signed, the expression shall be interpreted as a comparison between signed values. If the operands are of unequal bit lengths, the smaller operand shall be sign-extended to the size of the larger operand.

If either operand is a real operand, then the other operand shall be converted to an equivalent real value and the expression shall be interpreted as a comparison between real values.

All the relational operators shall have the same precedence. Relational operators shall have lower precedence than arithmetic operators.

For example:

The following examples illustrate the implications of this precedence rule:

```
a < foo - 1         // this expression is the same as
a < (foo - 1)       // this expression, but . . .
foo - (1 < a)       // this one is not the same as
foo - 1 < a         // this expression
```


--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

When `foo - (1 < a)` evaluates, the relational expression evaluates first, and then either zero or one is subtracted from `foo`. When `foo - 1 < a` evaluates, the value of `foo` operand is reduced by one and then compared with `a`.

5.1.8 Equality operators

The equality operators shall rank lower in precedence than the relational operators. Table 5-11 lists and defines the equality operators.

Table 5-11—Definitions of equality operators

a === b	a equal to b, including x and z
a !== b	a not equal to b, including x and z
a == b	a equal to b, result can be unknown
a != b	a not equal to b, result can be unknown

All four equality operators shall have the same precedence. These four operators compare operands bit for bit. As with the relational operators, the result shall be `0` if comparison fails and `1` if it succeeds.

If the operands are of unequal bit lengths and if one or both operands are unsigned, the smaller operand shall be zero-extended to the size of the larger operand. If both operands are signed, the smaller operand shall be sign-extended to the size of the larger operand.

If either operand is a real operand, then the other operand shall be converted to an equivalent real value, and the expression shall be interpreted as a comparison between real values.

For the *logical equality* and *logical inequality* operators (`==` and `!=`), if, due to unknown or high-impedance bits in the operands, the relation is ambiguous, then the result shall be a 1-bit unknown value (`x`).

For the *case equality* and *case inequality* operators (`===` and `!==`), the comparison shall be done just as it is in the procedural case statement (see 9.5). Bits that are `x` or `z` shall be included in the comparison and shall match for the result to be considered equal. The result of these operators shall always be a known value, either `1` or `0`.

5.1.9 Logical operators

The operators *logical and* (`&&`) and *logical or* (||) are logical connectives. The result of the evaluation of a logical comparison shall be `1` (defined as true), `0` (defined as false), or, if the result is ambiguous, the unknown value (`x`). The precedence of `&&` is greater than that of `||`, and both are lower than relational and equality operators.

A third logical operator is the unary *logical negation* operator (`!`). The negation operator converts a nonzero or true operand into `0` and a zero or false operand into `1`. An ambiguous truth value remains as `x`.

For example:

Example 1—If reg `alpha` holds the integer value 237 and `beta` holds the value zero, then the following examples perform as described:

```
regA = alpha && beta;      // regA is set to 0
regB = alpha || beta;      // regB is set to 1
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Example 2—The following expression performs a logical and of three subexpressions without needing any parentheses:

```
a < size-1 && b != c && index != lastone
```

However, it is recommended for readability purposes that parentheses be used to show very clearly the precedence intended, as in the following rewrite of this example:

```
(a < size-1) && (b != c) && (index != lastone)
```

Example 3—A common use of `!` is in constructions like the following:

```
if (!inword)
```

In some cases, the preceding construct makes more sense to someone reading the code than this equivalent construct:

```
if (inword == 0)
```

5.1.10 Bitwise operators

The *bitwise operators* shall perform bitwise manipulations on the operands; that is, the operator shall combine a bit in one operand with its corresponding bit in the other operand to calculate 1 bit for the result. Logic Table 5-12 through Table 5-16 show the results for each possible calculation.

Table 5-12—Bitwise binary and operator

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Table 5-13—Bitwise binary or operator

|	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 5-14—Bitwise binary exclusive or operator

^	**0**	**1**	**x**	**z**
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

Table 5-15—Bitwise binary exclusive nor operator

^~ ~^	**0**	**1**	**x**	**z**
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Table 5-16—Bitwise unary negation operator

~	
0	1
1	0
x	x
z	x

When the operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions.

5.1.11 Reduction operators

The *unary reduction operators* shall perform a bitwise operation on a single operand to produce a single-bit result. For *reduction and*, *reduction or*, and *reduction xor* operators, the first step of the operation shall apply the operator between the first bit of the operand and the second using logic Table 5-17 through Table 5-19. The second and subsequent steps shall apply the operator between the 1-bit result of the prior step and the next bit of the operand using the same logic table. For *reduction nand*, *reduction nor*, and *reduction xnor* operators, the result shall be computed by inverting the result of the reduction and, reduction or, and reduction xor operation, respectively.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 5-17—Reduction unary and operator

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Table 5-18—Reduction unary or operator

|	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Table 5-19—Reduction unary exclusive or operator

^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

For example:

Table 5-20 shows the results of applying reduction operators on different operands.

Table 5-20—Results of unary reduction operations

Operand	&	~&	|	~|	^	~^	Comments
4'b0000	0	1	0	1	0	1	No bits set
4'b1111	1	0	1	0	0	1	All bits set
4'b0110	0	1	1	0	0	1	Even number of bits set
4'b1000	0	1	1	0	1	0	Odd number of bits set

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

5.1.12 Shift operators

There are two types of *shift operators*: the logical shift operators, << and >>, and the arithmetic shift operators, <<< and >>>. The left shift operators, << and <<<, shall shift their left operand to the left by the number by the number of bit positions given by the right operand. In both cases, the vacated bit positions shall be filled with zeroes. The right shift operators, >> and >>>, shall shift their left operand to the right by the number of bit positions given by the right operand. The logical right shift shall fill the vacated bit positions with zeroes. The arithmetic right shift shall fill the vacated bit positions with zeroes if the result type is unsigned. It shall fill the vacated bit positions with the value of the most significant (i.e., sign) bit of the left operand if the result type is signed. If the right operand has an `x` or `z` value, then the result shall be unknown. The right operand is always treated as an unsigned number and has no effect on the signedness of the result. The result signedness is determined by the left-hand operand and the remainder of the expression, as outlined in 5.5.1.

For example:

Example 1—In this example, the reg `result` is assigned the binary value `0100`, which is `0001` shifted to the left two positions and zero-filled.

```
module shift;
reg [3:0] start, result;
initial begin
    start = 1;
    result = (start << 2);
end
endmodule
```

Example 2—In this example, the reg `result` is assigned the binary value `1110`, which is `1000` shifted to the right two positions and sign-filled.

```
module ashift;
reg signed [3:0] start, result;
initial begin
    start = 4'b1000;
    result = (start >>> 2);
end
endmodule
```

5.1.13 Conditional operator

The conditional operator, also known as *ternary operator*, shall be right associative and shall be constructed using three operands separated by two operators in the format given in Syntax 5-1.

conditional_expression ::= *(From A.8.3)*
    expression1 **?** { attribute_instance } expression2 **:** expression3
expression1 ::=
    expression
expression2 ::=
    expression
expression3 ::=
    expression

Syntax 5-1—Syntax for conditional operator

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The evaluation of a conditional operator shall begin with a logical equality comparison (see 5.1.8) of expression1 with zero, termed the *condition*. If the condition evaluates to false (0), then expression3 shall be evaluated and used as the result of the conditional expression. If the condition evaluates to true (1), then expression2 is evaluated and used as the result. If the condition evaluates to an ambiguous value (`x` or `z`), then both expression2 and expression3 shall be evaluated; and their results shall be combined, bit by bit, using Table 5-21 to calculate the final result unless expression2 or expression3 is real, in which case the result shall be `0`. If the lengths of expression2 and expression3 are different, the shorter operand shall be lengthened to match the longer and zero-filled from the left (the high-order end).

Table 5-21—Ambiguous condition results for conditional operator

?:	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

For example:

The following example of a three-state output bus illustrates a common use of the conditional operator:

```
wire [15:0] busa = drive_busa ? data : 16'bz;
```

The bus called `data` is driven onto `busa` when `drive_busa` is `1`. If `drive_busa` is unknown, then an unknown value is driven onto `busa`. Otherwise, `busa` is not driven.

5.1.14 Concatenations

A concatenation is the result of the joining together of bits resulting from one or more expressions. The concatenation shall be expressed using the brace characters **{** and **}**, with commas separating the expressions within.

Unsized constant numbers shall not be allowed in concatenations. This is because the size of each operand in the concatenation is needed to calculate the complete size of the concatenation.

For example:

This example concatenates four expressions:

```
{a, b[3:0], w, 3'b101}
```

It is equivalent to the following example:

```
{a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1}
```

An operator that can be applied only to concatenations is *replication*, which is expressed by a concatenation preceded by a non-negative, non-`x` and non-`z` constant expression, called a *replication constant*, enclosed together within brace characters, and which indicates a joining together of that many copies of the

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

concatenation. Unlike regular concatenations, expressions containing replications shall not appear on the left-hand side of an assignment and shall not be connected to **output** or **inout** ports.

This example replicates `w` four times.

```
{4{w}} // This yields the same value as {w, w, w, w}
```

The following examples show illegal replications:

```
{1'bz{1'b0}}    // illegal
{1'bx{1'b0}}    // illegal
```

The next example illustrates a replication nested within a concatenation:

```
{b, {3{a, b}}}   // This yields the same value as
                 // {b, a, b, a, b, a, b}
```

A replication operation may have a replication constant with a value of zero. This is useful in parameterized code. A replication with a zero replication constant is considered to have a size of zero and is ignored. Such a replication shall appear only within a concatenation in which at least one of the operands of the concatenation has a positive size.

For example:

```
parameter P = 32;

// The following is legal for all P from 1 to 32

assign b[31:0] = { {32-P{1'b1}}, a[P-1:0] } ;

// The following is illegal for P=32 because the zero
// replication appears alone within a concatenation

assign c[31:0] = { {{32-P{1'b1}}}, a[P-1:0] }

// The following is illegal for P=32

initial
  $displayb({32-P{1'b1}}, a[P-1:0]);
```

When a replication expression is evaluated, the operands shall be evaluated exactly once, even if the replication constant is zero. For example:

```
result = {4{func(w)}} ;
```

would be computed as

```
y = func(w) ;
result = {y, y, y, y} ;
```

5.2 Operands

There are several types of operands that can be specified in expressions. The simplest type is a reference to a net, variable, or parameter in its complete form; that is, just the name of the net, variable, or parameter is

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

given. In this case, all of the bits making up the net, variable, or parameter value shall be used as the operand.

If a single bit of a vector net, vector **reg**, **integer**, or **time** variable, or parameter is required, then a bit-select operand shall be used. A part-select operand shall be used to reference a group of adjacent bits in a vector net, vector **reg**, **integer**, or **time** variable, or parameter.

An array element or a bit-select or part-select of an array element can be referenced as an operand. A concatenation of other operands (including nested concatenations) can be specified as an operand. A function call is an operand.

5.2.1 Vector bit-select and part-select addressing

Bit-selects extract a particular bit from a vector net, vector **reg**, **integer**, or **time** variable, or parameter. The bit can be addressed using an expression. If the bit-select is out of the address bounds or the bit-select is `x` or `z`, then the value returned by the reference shall be `x`. A bit-select or part-select of a scalar, or of a variable or parameter of type **real** or **realtime**, shall be illegal.

Several contiguous bits in a vector net, vector **reg**, **integer**, or **time** variable, or parameter can be addressed and are known as *part-selects*. There are two types of part-selects, a constant part-select and an indexed part-select. A constant part-select of a vector reg or net is given with the following syntax:

```
vect[msb_expr:lsb_expr]
```

Both `msb_expr` and `lsb_expr` shall be constant integer expressions. The first expression has to address a more significant bit than the second expression.

An indexed part-select of a vector net, vector **reg**, **integer**, or **time** variable, or parameter is given with the following syntax:

```
reg [15:0] big_vect;
reg [0:15] little_vect;

   big_vect[lsb_base_expr +: width_expr]
little_vect[msb_base_expr +: width_expr]

   big_vect[msb_base_expr -: width_expr]
little_vect[lsb_base_expr -: width_expr]
```

The `msb_base_expr` and `lsb_base_expr` shall be integer expressions, and the `width_expr` shall be a positive constant integer expression. The `lsb_base_expr` and `msb_base_expr` can vary at run time. The first two examples select bits starting at the base and ascending the bit range. The number of bits selected is equal to the width expression. The second two examples select bits starting at the base and descending the bit range.

A part-select of any type that addresses a range of bits that are completely out of the address bounds of the net, **reg**, **integer**, **time** variable, or parameter or a part-select that is `x` or `z` shall yield the value `x` when read and shall have no effect on the data stored when written. Part-selects that are partially out of range shall, when read, return `x` for the bits that are out of range and shall, when written, only affect the bits that are in range.

For example:

```
reg [31: 0] big_vect;
reg [0 :31] little_vect;
```

 Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
reg [63: 0] dword;
integer sel;

big_vect[ 0 +: 8]     // == big_vect[ 7 : 0]
big_vect[15 -: 8]     // == big_vect[15 : 8]

little_vect[ 0 +: 8] // == little_vect[0 : 7]
little_vect[15 -: 8] // == little_vect[8 :15]

dword[8*sel +: 8]     // variable part-select with fixed width
```

For example:

Example 1—The following example specifies the single bit of `acc` vector that is addressed by the operand `index`:

```
acc[index]
```

The actual bit that is accessed by an address is, in part, determined by the declaration of `acc`. For instance, each of the declarations of `acc` shown in the next example causes a particular value of `index` to access a *different* bit:

```
reg [15:0] acc;
reg [2:17] acc
```

Example 2—The next example and the bullet items that follow it illustrate the principles of bit addressing. The code declares an 8-bit reg called `vect` and initializes it to a value of 4. The list describes how the separate bits of that vector can be addressed.

```
reg [7:0] vect;
vect = 4; // fills vect with the pattern 00000100
          // msb is bit 7, lsb is bit 0
```

- If the value of `addr` is 2, then `vect[addr]` returns `1`.
- If the value of `addr` is out of bounds, then `vect[addr]` returns `x`.
- If `addr` is 0, 1, or 3 through 7, `vect[addr]` returns `0`.
- `vect[3:0]` returns the bits `0100`.
- `vect[5:1]` returns the bits `00010`.
- `vect[`*expression that returns x*`]` returns `x`.
- `vect[`*expression that returns z*`]` returns `x`.
- If any bit of `addr` is `x` or `z`, then the value of `addr` is `x`.

NOTE 1—Part-select indices that evaluate to x or z may be flagged as a compile time error.

NOTE 2—Bit-select or part-select indices that are outside of the declared range may be flagged as a compile time error.

5.2.2 Array and memory addressing

Declaration of arrays and memories (one-dimensional arrays of reg) are discussed in 4.9. This subclause discusses array addressing.

For example:

The next example declares a memory of 1024 eight-bit words:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
reg [7:0] mem_name[0:1023];
```

The syntax for a memory address shall consist of the name of the memory and an expression for the address, specified with the following format:

```
mem_name[addr_expr]
```

The `addr_expr` can be any integer expression; therefore, memory indirections can be specified in a single expression. The next example illustrates memory indirection:

```
mem_name[mem_name[3]]
```

In this example, `mem_name[3]` addresses word three of the memory called `mem_name`. The value at word three is the index into `mem_name` that is used by the memory address `mem_name[mem_name[3]]`. As with bit-selects, the address bounds given in the declaration of the memory determine the effect of the address expression. If the index is out of the address bounds or if any bit in the address is `x` or `z`, then the value of the reference shall be `x`.

For example:

The next example declares an array of 256-by-256 eight-bit elements and an array 256-by-256-by-8 one-bit elements:

```
reg [7:0] twod_array[0:255][0:255];
wire threed_array[0:255][0:255][0:7];
```

The syntax for access to the array shall consist of the name of the memory or array and an integer expression for each addressed dimension:

```
twod_array[addr_expr][addr_expr]
threed_array[addr_expr][addr_expr][addr_expr]
```

As before, the `addr_expr` can be any integer expression. The array `twod_array` accesses a whole 8-bit vector, while the array `threed_array` accesses a single bit of the three-dimensional array.

To express bit-selects or part-selects of array elements, the desired word shall first be selected by supplying an address for each dimension. Once selected, bit-selects and part-selects shall be addressed in the same manner as net and reg bit-selects and part-selects (see 5.2.1).

For example:

```
twod_array[14][1][3:0]      // access lower 4 bits of word
twod_array[1][3][6]         // access bit 6 of word
twod_array[1][3][sel]       // use variable bit-select
threed_array[14][1][3:0]    // Illegal
```

5.2.3 Strings

String operands shall be treated as constant numbers consisting of a sequence of 8-bit ASCII codes, one per character. Any Verilog HDL operator can manipulate string operands. The operator shall behave as though the entire string were a single numeric value.

When a variable is larger than required to hold the value being assigned, the contents after the assignment shall be padded on the left with zeros. This is consistent with the padding that occurs during assignment of nonstring values.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

For example:

The following example declares a string variable large enough to hold 14 characters and assigns a value to it. The example then manipulates the string using the concatenation operator.

```
module string_test;
reg [8*14:1] stringvar;

initial begin
    stringvar = "Hello world";
    $display("%s is stored as %h", stringvar, stringvar);
    stringvar = {stringvar,"!!!"};
    $display("%s is stored as %h", stringvar, stringvar);
end
endmodule
```

The result of simulating the above description is

```
   Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
```

5.2.3.1 String operations

The common string operations *copy*, *concatenate*, and *compare* are supported by Verilog HDL operators. Copy is provided by simple assignment. Concatenation is provided by the concatenation operator. Comparison is provided by the equality operators.

When manipulating string values in vector regs, the regs should be at least `8*n` bits (where `n` is the number of ASCII characters) in order to preserve the 8-bit ASCII code.

5.2.3.2 String value padding and potential problems

When strings are assigned to variables, the values stored shall be padded on the left with zeros. Padding can affect the results of comparison and concatenation operations. The comparison and concatenation operators shall not distinguish between zeros resulting from padding and the original string characters (`\0`, `ASCII NUL`).

For example:

The following example illustrates the potential problem:

```
reg [8*10:1] s1, s2;
initial begin
    s1 = "Hello";
    s2 = " world!";
    if ({s1,s2} == "Hello world!")
        $display("strings are equal");
end
```

The comparison in this example fails because during the assignment the string variables are padded as illustrated in the next example:

```
s1 = 000000000048656c6c6f
s2 = 00000020776f726c6421
```

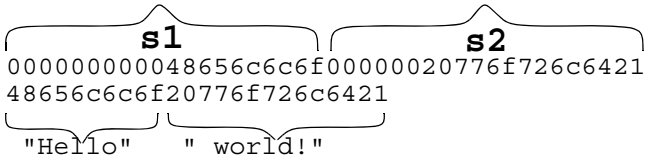
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The concatenation of `s1` and `s2` includes the zero padding, resulting in the following value:

```
000000000048656c6c6f00000020776f726c6421
```

Because the string “Hello world!” contains no zero padding, the comparison fails, as shown in the following example:



This comparison yields a result of zero, which is equivalent to false.

5.2.3.3 Null string handling

The null string (`""`) shall be considered equivalent to the ASCII NUL (`"\0"`), which has a value zero (0), which is different from a string `"0"`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

5.3 Minimum, typical, and maximum delay expressions

Verilog HDL delay expressions can be specified as three expressions separated by colons and enclosed by parentheses. This is intended to represent minimum, typical, and maximum values—in that order. The syntax is given in Syntax 5-2.

```
constant_expression ::= (From A.8.3)
        constant_primary
      | unary_operator { attribute_instance } constant_primary
      | constant_expression binary_operator { attribute_instance } constant_expression
      | constant_expression ? { attribute_instance } constant_expression
          constant_expression
constant_mintypmax_expression ::=
        constant_expression
      | constant_expression : constant_expression : constant_expression
expression ::=
        primary
      | unary_operator { attribute_instance } primary
      | expression binary_operator { attribute_instance } expression
      | conditional_expression
mintypmax_expression ::=
        expression
      | expression : expression : expression
constant_primary ::= (From A.8.4)
        number
      | parameter_identifier [ [ constant_range_expression ] ]
      | specparam_identifier [ [ constant_range_expression ] ]
      | constant_concatenation
      | constant_multiple_concatenation
      | constant_function_call
      | constant_system_function_call
      | ( constant_mintypmax_expression )
      | string
primary ::=
        number
      | hierarchical_identifier [ { [ expression ] } [ range_expression ] ]
      | concatenation
      | multiple_concatenation
      | function_call
      | system_function_call
      | ( mintypmax_expression )
      | string
```

Syntax 5-2—Syntax for mintypmax expression

Verilog HDL models typically specify three values for delay expressions. The three values allow a design to be tested with minimum, typical, or maximum delay values.

Values expressed in min:typ:max format can be used in expressions. The min:typ:max format can be used wherever expressions can appear.

For example:

Example 1—This example shows an expression that defines a single triplet of delay values. The minimum value is the sum of `a+d`; the typical value is `b+e`; the maximum value is `c+f`, as follows:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
(a:b:c) + (d:e:f)
```

Example 2—The next example shows a typical expression that is used to specify `min:typ:max` format values:

```
val - (32'd 50: 32'd 75: 32'd 100)
```

5.4 Expression bit lengths

Controlling the number of bits that are used in expression evaluations is important if consistent results are to be achieved. Some situations have a simple solution; for example, if a bitwise and operation is specified on two 16-bit regs, then the result is a 16-bit value. However, in some situations, it is not obvious how many bits are used to evaluate an expression or what size the result should be.

For example, should an arithmetic add of two 16-bit values perform the evaluation using 16 bits, or should the evaluation use 17 bits in order to allow for a possible carry overflow? The answer depends on the type of device being modeled and whether that device handles carry overflow. The Verilog HDL uses the bit length of the operands to determine how many bits to use while evaluating an expression. The bit length rules are given in 5.4.1. In the case of the addition operator, the bit length of the largest operand, including the left-hand side of an assignment, shall be used.

For example:

```
reg [15:0] a, b; // 16-bit regs
reg [15:0] sumA; // 16-bit reg
reg [16:0] sumB; // 17-bit reg

sumA = a + b; // expression evaluates using 16 bits
sumB = a + b; // expression evaluates using 17 bits
```

5.4.1 Rules for expression bit lengths

The rules governing the expression bit lengths have been formulated so that most practical situations have a natural solution.

The number of bits of an expression (known as the *size* of the expression) shall be determined by the operands involved in the expression and the context in which the expression is given.

A *self-determined expression* is one where the bit length of the expression is solely determined by the expression itself—for example, an expression representing a delay value.

A *context-determined expression* is one where the bit length of the expression is determined by the bit length of the expression and by the fact that it is part of another expression. For example, the bit size of the right-hand expression of an assignment depends on itself and the size of the left-hand side.

Table 5-22 shows how the form of an expression shall determine the bit lengths of the results of the expression. In Table 5-22, `i`, `j`, and `k` represent expressions of an operand, and `L(i)` represents the bit length of the operand represented by `i`.

Multiplication may be performed without losing any overflow bits by assigning the result to something wide enough to hold it.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 5-22—Bit lengths resulting from self-determined expressions

Expression	Bit length	Comments
Unsized constant number[a]	Same as integer	
Sized constant number	As given	
i op j, where op is: + - * / % & | ^ ^~ ~^	max(L(i),L(j))	
op i, where op is: + - ~	L(i)	
i op j, where op is: === !== == != > >= < <=	1 bit	Operands are sized to max(L(i),L(j))
i op j, where op is: && ||	1 bit	All operands are self-determined
op i, where op is: & ~& | ~| ^ ~^ ^~ !	1 bit	All operands are self-determined
i op j, where op is: >> << ** >>> <<<	L(i)	j is self-determined
i ? j : k	max(L(j),L(k))	i is self-determined
{i,...,j}	L(i)+..+L(j)	All operands are self-determined
{i{j,..,k}}	i * (L(j)+..+L(k))	All operands are self-determined

[a]If an unsized constant is part of an expression that is longer than 32 bits and if the most significant bit is unknown (`X` or `x`) or three-state (`Z` or `z`), the most significant bit is extended up to the size of the expression. Otherwise, signed constants are sign-extended and unsigned constants are zero-extended.

5.4.2 Example of expression bit-length problem

During the evaluation of an expression, interim results shall take the size of the largest operand (in case of an assignment, this also includes the left-hand side). Care has to be taken to prevent loss of a significant bit during expression evaluation. The example below describes how the bit lengths of the operands could result in the loss of a significant bit.

Given the following declarations:

```
reg [15:0] a, b, answer; // 16-bit regs
```

the intent is to evaluate the expression

```
answer = (a + b) >> 1; //will not work properly
```

where `a` and `b` are to be added, which can result in an overflow, and then shifted right by 1 bit to preserve the carry bit in the 16-bit `answer`.

A problem arises, however, because all operands in the expression are of a 16-bit width. Therefore, the expression (`a + b`) produces an interim result that is only 16 bits wide, thus losing the carry bit before the evaluation performs the 1-bit right shift operation.

Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The solution is to force the expression (`a + b`) to evaluate using at least 17 bits. For example, adding an integer value of 0 to the expression will cause the evaluation to be performed using the bit size of integers. The following example will produce the intended result:

```
answer = (a + b + 0) >> 1; //will work correctly
```

In the following example:

```
module bitlength();
   reg [3:0] a,b,c;
   reg [4:0] d;

   initial begin
     a = 9;
     b = 8;
     c = 1;
     $display("answer = %b", c ? (a&b) : d);
     end
   endmodule
```

the `$display` statement will display

```
answer = 01000
```

By itself, the expression `a&b` would have the bit length 4, but because it is in the context of the conditional expression, which uses the maximum bit length, the expression `a&b` actually has length 5, the length of `d`.

5.4.3 Example of self-determined expressions

```
reg [3:0] a;
reg [5:0] b;
reg [15:0] c;

initial begin
   a = 4'hF;
   b = 6'hA;
   $display("a*b=%h", a*b);// expression size is self-determined
   c = {a**b};           // expression a**b is self-determined
                         // due to concatenation operator {}
   $display("a**b=%h", c);
   c = a**b;             // expression size is determined by c
   $display("c=%h", c);
end
```

Simulator output for this example:

```
a*b=16  // 'h96 was truncated to 'h16 since expression size is 6
a**b=1  // expression size is  4 bits (size of a)
c=ac61  // expression size is 16 bits (size of c)
```

5.5 Signed expressions

Controlling the sign of an expression is important if consistent results are to be achieved. In addition to the rules outlined in 5.5.1 through 5.5.4, two system functions shall be used to handle type casting on expressions: **$signed()** and **$unsigned()**. These functions shall evaluate the input expression and return a value with the same size and value of the input expression and the type defined by the function:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
$signed - returned value is signed
$unsigned - returned value is unsigned
```

For example:

```
reg [7:0] regA, regB;
reg signed [7:0] regS;

regA = $unsigned(-4);      // regA = 8'b11111100
regB = $unsigned(-4'sd4); // regB = 8'b00001100
regS = $signed (4'b1100); // regS = -4
```

5.5.1 Rules for expression types

The following are the rules for determining the resulting type of an expression:

- Expression type depends only on the operands. It does not depend on the left-hand side (if any).
- Decimal numbers are signed.
- Based_numbers are unsigned, except where the `s` notation is used in the base specifier (as in "`4'sd12`").
- Bit-select results are unsigned, regardless of the operands.
- Part-select results are unsigned, regardless of the operands even if the part-select specifies the entire vector.

```
reg [15:0] a;
reg signed [7:0] b;

initial
    a = b[7:0]; // b[7:0] is unsigned and therefore zero-extended
```

- Concatenate results are unsigned, regardless of the operands.
- Comparison results (1, 0) are unsigned, regardless of the operands.
- Reals converted to integers by type coercion are signed
- The sign and size of any self-determined operand are determined by the operand itself and independent of the remainder of the expression.
- For nonself-determined operands, the following rules apply:
  - If any operand is real, the result is real.
  - If any operand is unsigned, the result is unsigned, regardless of the operator.
  - If all operands are signed, the result will be signed, regardless of operator, except when specified otherwise.

5.5.2 Steps for evaluating an expression

The following are the steps for evaluating an expression:

- Determine the expression size based upon the standard rules of expression size determination.
- Determine the sign of the expression using the rules outlined in 5.5.1.
- Propagate the type and size of the expression (or self-determined subexpression) back down to the context-determined operands of the expression. In general, any context-determined operand of an operator shall be the same type and size as the result of the operator. However, there are two exceptions:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

— If the result type of the operator is real and if it has a context-determined operand that is not real, that operand shall be treated as if it were self-determined and then converted to real just before the operator is applied.

— The relational and equality operators have operands that are neither fully self-determined nor fully context-determined. The operands shall affect each other as if they were context-determined operands with a result type and size (maximum of the two operand sizes) determined from them. However, the actual result type shall always be 1 bit unsigned. The type and size of the operand shall be independent of the rest of the expression and vice versa.

— When propagation reaches a simple operand as defined in 5.2 (a primary as defined in A.8.4), then that operand shall be converted to the propagated type and size. If the operand must be extended, then it shall be sign-extended only if the propagated type is signed.

5.5.3 Steps for evaluating an assignment

The following are the steps for evaluating an assignment:

— Determine the size of the right-hand side by the standard assignment size determination rules (see 5.4).

— If needed, extend the size of the right-hand side, performing sign extension if, and only if, the type of the right-hand side is signed.

5.5.4 Handling X and Z in signed expressions

If a signed operand is to be resized to a larger signed width and the value of the sign bit is x, the resulting value shall be bit-filled with xs. If the sign bit of the value is z, then the resulting value shall be bit-filled with zs. If any bit of a signed value is x or z, then any nonlogical operation involving the value shall result in the entire resultant value being an x and the type consistent with the expression's type.

5.6 Assignments and truncation

If the width of the right-hand expression is larger than the width of the left-hand side in an assignment, the MSBs of the right-hand expression will always be discarded to match the size of the left-hand side. Implementations are not required to warn or report any errors related to assignment size mismatch or truncation. Truncating the sign bit of a signed expression may change the sign of the result.

For example:

```
reg        [5:0] a;
reg signed [4:0] b;

initial  begin
  a = 8'hff; // After the assignment, a = 6'h3f
  b = 8'hff; // After the assignment, b = 5'h1f
end
```

For example:

```
reg        [0:5] a;
reg signed [0:4] b, c;

initial  begin
  a = 8'sh8f; // After the assignment, a = 6'h0f
  b = 8'sh8f; // After the assignment, b = 5'h0f
  c = -113;   // After the assignment, c = 15
```


Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
    // 1000_1111 = (-'h71 = -113) truncates to ('h0F = 15)
end
```

For example:

```
reg       [7:0] a;
reg signed [7:0] b;
reg signed [5:0] c, d;

initial  begin
  a = 8'hff;
  c = a;      // After the assignment, c = 6'h3f
  b = -113;
  d = b;      // After the assignment, d = 6'h0f
end
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

6. Assignments

The assignment is the basic mechanism for placing values into nets and variables. There are two basic forms of assignments:

— The *continuous assignment*, which assigns values to nets
— The *procedural assignment*, which assigns values to variables

There are two additional forms of assignments, **assign**/**deassign** and **force**/**release**, which are called *procedural continuous assignments*, described in 9.3.

An assignment consists of two parts, a left-hand side and a right-hand side, separated by the equals (=) character; or, in the case of nonblocking procedural assignment, the less-than-equals (<=) character pair. The right-hand side can be any expression that evaluates to a value. The left-hand side indicates the variable to which the right-hand side value is to be assigned. The left-hand side can take one of the forms given in Table 6-1, depending on whether the assignment is a continuous assignment or a procedural assignment.

Table 6-1—Legal left-hand forms in assignment statements

Statement type	Left-hand side
Continuous assignment	Net (vector or scalar) Constant bit-select of a vector net Constant part-select of a vector net Constant indexed part-select of a vector net Concatenation or nested concatenation of any of the above left-hand side
Procedural assignment	Variables (vector or scalar) Bit-select of a vector reg, integer, or time variable Constant part-select of a vector reg, integer, or time variable Indexed part-select of a vector reg, integer, or time variable Memory word Concatenation or nested concatenation of any of the above left-hand side

6.1 Continuous assignments

Continuous assignments shall drive values onto nets, both vector and scalar. This assignment shall occur whenever the value of the right-hand side changes. Continuous assignments provide a way to model combinational logic without specifying an interconnection of gates. Instead, the model specifies the logical expression that drives the net.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The syntax for continuous assignments is given in Syntax 6-1.

```
net_declaration ::= (From A.2.1.3)
        net_type [ signed ]
          [ delay3 ] list_of_net_identifiers ;
      | net_type [ drive_strength ] [ signed ]
          [ delay3 ] list_of_net_decl_assignments ;
      | net_type [ vectored | scalared ] [ signed ]
          range [ delay3 ] list_of_net_identifiers ;
      | net_type [ drive_strength ] [ vectored | scalared ] [ signed ]
          range [ delay3 ] list_of_net_decl_assignments ;
      | trireg [ charge_strength ] [ signed ]
          [ delay3 ] list_of_net_identifiers ;
      | trireg [ drive_strength ] [ signed ]
          [ delay3 ] list_of_net_decl_assignments ;
      | trireg [ charge_strength ] [ vectored | scalared ] [ signed ]
          range [ delay3 ] list_of_net_identifiers ;
      | trireg [ drive_strength ] [ vectored | scalared ] [ signed ]
          range [ delay3 ] list_of_net_decl_assignments ;
list_of_net_decl_assignments ::= (From A.2.3)
      net_decl_assignment { , net_decl_assignment }
net_decl_assignment ::= (From A.2.4)
      net_identifier = expression
continuous_assign ::= (From A.6.1)
      assign [ drive_strength ] [ delay3 ] list_of_net_assignments ;
list_of_net_assignments ::=
      net_assignment { , net_assignment }
net_assignment ::=
      net_lvalue = expression
```

Syntax 6-1—Syntax for continuous assignment

6.1.1 The net declaration assignment

The first two alternatives in the net declaration are discussed in 4.2. The third alternative, the net declaration assignment, allows a continuous assignment to be placed on a net in the same statement that declares the net.

For example:

The following is an example of the net declaration form of a continuous assignment:

```
wire (strong1, pull0) mynet = enable ;
```

NOTE—Because a net can be declared only once, only one net declaration assignment can be made for a particular net. This contrasts with the continuous assignment statement; one net can receive multiple assignments of the continuous assignment form.

6.1.2 The continuous assignment statement

The continuous assignment statement shall place a continuous assignment on a net data type. The net may be explicitly declared or may inherit an implicit declaration in accordance with the implicit declaration rules defined in 4.5.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Assignments on nets shall be continuous and automatic. In other words, whenever an operand in the right-hand expression changes value, the whole right-hand side shall be evaluated. If the new value is different from the previous value, then the new value shall be assigned to the left-hand side.

For example:

Example 1—The following is an example of a continuous assignment to a net that has been previously declared:

```
wire mynet ;
assign (strong1, pull0) mynet = enable ;
```

Example 2—The following is an example of the use of a continuous assignment to model a 4-bit adder with carry. The assignment could not be specified directly in the declaration of the nets because it requires a concatenation on the left-hand side.

```
module adder (sum_out, carry_out, carry_in, ina, inb);
output [3:0] sum_out;
output carry_out;
input [3:0] ina, inb;
input carry_in;
wire carry_out, carry_in;
wire [3:0] sum_out, ina, inb;
assign {carry_out, sum_out} = ina + inb + carry_in;
endmodule
```

Example 3—The following example describes a module with one 16-bit output bus. It selects between one of four input busses and connects the selected bus to the output bus.

```
module select_bus(busout, bus0, bus1, bus2, bus3, enable, s);
parameter n = 16;
parameter Zee = 16'bz;
output [1:n] busout;
input [1:n] bus0, bus1, bus2, bus3;
input enable;
input [1:2] s;
tri [1:n] data;          // net declaration
// net declaration with continuous assignment
tri [1:n] busout = enable ? data : Zee;
// assignment statement with four continuous assignments
assign
   data = (s == 0) ? bus0 : Zee,
   data = (s == 1) ? bus1 : Zee,
   data = (s == 2) ? bus2 : Zee,
   data = (s == 3) ? bus3 : Zee;
endmodule
```

The following sequence of events is experienced during simulation of this example:

a) The value of `s`, a bus selector input variable, is checked in the assign statement. Based on the value of `s`, the net `data` receives the data from one of the four input buses.
b) The setting of `data` net triggers the continuous assignment in the net declaration for `busout`. If `enable` is set, the contents of `data` are assigned to `busout`; if `enable` is `0`, the contents of `Zee` are assigned to `busout`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

6.1.3 Delays

A delay given to a continuous assignment shall specify the time duration between a right-hand operand value change and the assignment made to the left-hand side. If the left-hand references a scalar net, then the delay shall be treated in the same way as for gate delays; that is, different delays can be given for the output rising, falling, and changing to high impedance (see Clause 7).

If the left-hand references a vector net, then up to three delays can be applied. The following rules determine which delay controls the assignment:

— If the right-hand side makes a transition from nonzero to zero, then the falling delay shall be used.
— If the right-hand side makes a transition to `z`, then the turn-off delay shall be used.
— For all other cases, the rising delay shall be used.

Specifying the delay in a continuous assignment that is part of the net declaration shall be treated differently from specifying a net delay and then making a continuous assignment to the net. A delay value can be applied to a net in a net declaration, as in the following example:

```
wire #10 wireA;
```

This syntax, called a *net delay*, means that any value change that is to be applied to `wireA` by some other statement shall be delayed for ten time units before it takes effect. When there is a continuous assignment in a declaration, the delay is part of the continuous assignment and is not a net delay. Thus, it shall not be added to the delay of other drivers on the net. Furthermore, if the assignment is to a vector net, then the rising and falling delays shall not be applied to the individual bits if the assignment is included in the declaration.

In situations where a right-hand operand changes before a previous change has had time to propagate to the left-hand side, then the following steps are taken:

a) The value of the right-hand expression is evaluated.
b) If this right-hand side value differs from the value currently scheduled to propagate to the left-hand side, then the currently scheduled propagation event is descheduled.
c) If the new right-hand side value equals the current left-hand side value, no event is scheduled.
d) If the new right-hand side value differs from the current left-hand side value, a delay is calculated in the standard way using the current value of the left-hand side, the newly calculated value of the right-hand side, and the delays indicated on the statement; a new propagation event is then scheduled to occur delay time units in the future.

6.1.4 Strength

The driving strength of a continuous assignment can be specified by the user. This applies only to assignments to scalar nets of the following types:

wire	**tri**	**trireg**
wand	**triand**	**tri0**
wor	**trior**	**tri1**

Continuous assignments driving strengths can be specified either in a net declaration or in a stand-alone assignment, using the **assign** keyword. The strength specification, if provided, shall immediately follow the keyword (either the keyword for the net type or **assign**) and precede any delay specified. Whenever the continuous assignment drives the net, the strength of the value shall be simulated as specified.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

A drive strength specification shall contain one strength value that applies when the value being assigned to the net is `1` and a second strength value that applies when the assigned value is `0`. The following keywords shall specify the strength value for an assignment of `1`:

supply1 **strong1** **pull1** **weak1** **highz1**

The following keywords shall specify the strength value for an assignment of `0`:

supply0 **strong0** **pull0** **weak0** **highz0**

The order of the two strength specifications shall be arbitrary. The following two rules shall constrain the use of drive strength specifications:

— The strength specifications (**highz1**, **highz0**) and (**highz0**, **highz1**) shall be treated as illegal constructs.
— If drive strength is not specified, it shall default to (**strong1**, **strong0**).

6.2 Procedural assignments

The primary discussion of procedural assignments is in 9.2. However, a description of the basic ideas in this clause highlights the differences between continuous assignments and procedural assignments.

As stated in 6.1, continuous assignments drive nets in a manner similar to the way gates drive nets. The expression on the right-hand side can be thought of as a combinatorial circuit that drives the net continuously. In contrast, procedural assignments put values in variables. The assignment does not have duration; instead, the variable holds the value of the assignment until the next procedural assignment to that variable.

Procedural assignments occur within procedures such as **always**, **initial** (see 9.9), **task**, and **function** (see Clause 10) and can be thought of as "triggered" assignments. The trigger occurs when the flow of execution in the simulation reaches an assignment within a procedure. Reaching the assignment can be controlled by conditional statements. Event controls, delay controls, **if** statements, **case** statements, and looping statements can all be used to control whether assignments are evaluated. Clause 9 gives details and examples.

6.2.1 Variable declaration assignment

The variable declaration assignment is a special case of procedural assignment as it assigns a value to a variable. It allows an initial value to be placed in a variable in the same statement that declares the variable. The assignment shall be to a constant expression. The assignment does not have duration; instead, the variable holds the value until the next assignment to that variable. Variable declaration assignments to an array are not allowed. Variable declaration assignments are only allowed at the module level. If the same variable is assigned different values both in an initial block and in a variable declaration assignment, the order of the evaluation is undefined.

For example:

Example 1—Declare a 4-bit reg and assign it the value 4.

```
reg[3:0] a = 4'h4;
```

This is equivalent to writing

```
reg[3:0] a;
initial a = 4'h4;
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Example 2—The following example is not legal:

```
reg [3:0] array [3:0] = 0;
```

Example 3—Declare two integers; the first is assigned the value of 0.

```
integer i = 0, j;
```

Example 4—Declare two real variables, assigned to the values 2.5 and 300,000.

```
real r1 = 2.5, n300k = 3E6;
```

Example 5—Declare a time variable and realtime variable with initial values.

```
time t1 = 25;
realtime rt1 = 2.5;
```

6.2.2 Variable declaration syntax

The syntax for variable declaration assignments is given in Syntax 6-2.

```
integer_declaration ::= (From A.2.1.3)
      integer list_of_variable_identifiers ;
real_declaration ::=
      real list_of_real_identifiers ;
realtime_declaration ::=
      realtime list_of_real_identifiers ;
reg_declaration ::=
      reg [ signed ] [ range ] list_of_variable_identifiers ;
time_declaration ::=
      time list_of_variable_identifiers ;
real_type ::= (From A.2.2.1)
        real_identifier { dimension }
      | real_identifier = constant_expression
variable_type ::=
        variable_identifier { dimension }
      | variable_identifier = constant_expression
list_of_real_identifiers ::= (From A.2.3)
      real_type { , real_type }
list_of_variable_identifiers ::=
      variable_type { , variable_type }
```

Syntax 6-2—Syntax for variable declaration assignment

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7. Gate- and switch-level modeling

This clause describes the syntax and semantics of the built-in primitives of gate- and switch-level modeling and how a hardware design can be described using these primitives.

There are 14 logic gates and 12 switches predefined in the Verilog HDL to provide the gate- and switch-level modeling facility. Modeling with logic gates and switches has the following advantages:

— Gates provide a much closer one-to-one mapping between the actual circuit and the model.
— There is no continuous assignment equivalent to the bidirectional transfer gate.

7.1 Gate and switch declaration syntax

Syntax 7-1 shows the gate and switch declaration syntax.

A gate or a switch instance declaration shall have the following specifications:

— The keyword that names the type of gate or switch primitive
— An optional drive strength
— An optional propagation delay
— An optional identifier that names each gate or switch instance
— An optional range for array of instances
— The terminal connection list

Multiple instances of the one type of gate or switch primitive can be declared as a comma-separated list. All such instances shall have the same drive strength and delay specification.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
gate_instantiation ::= (From A.3.1)
        cmos_switchtype [delay3] cmos_switch_instance { , cmos_switch_instance } ;
      | enable_gatetype [drive_strength] [delay3] enable_gate_instance { , enable_gate_instance } ;
      | mos_switchtype [delay3] mos_switch_instance { , mos_switch_instance } ;
      | n_input_gatetype [drive_strength] [delay2] n_input_gate_instance {, n_input_gate_instance };
      | n_output_gatetype [drive_strength] [delay2] n_output_gate_instance
          { , n_output_gate_instance } ;
      | pass_en_switchtype [delay2] pass_enable_switch_instance {, pass_enable_switch_instance } ;
      | pass_switchtype pass_switch_instance { , pass_switch_instance } ;
      | pulldown [pulldown_strength] pull_gate_instance { , pull_gate_instance } ;
      | pullup [pullup_strength] pull_gate_instance { , pull_gate_instance } ;
cmos_switch_instance ::= [ name_of_gate_instance ]
      ( output_terminal , input_terminal , ncontrol_terminal , pcontrol_terminal )
enable_gate_instance ::= [ name_of_gate_instance ]
      ( output_terminal , input_terminal , enable_terminal )
mos_switch_instance ::= [ name_of_gate_instance ]
      ( output_terminal , input_terminal , enable_terminal )
n_input_gate_instance ::= [ name_of_gate_instance ]
      ( output_terminal , input_terminal { , input_terminal } )
n_output_gate_instance ::= [ name_of_gate_instance ]
      ( output_terminal { , output_terminal } , input_terminal )
pass_switch_instance ::= [ name_of_gate_instance ] ( inout_terminal , inout_terminal )
pass_enable_switch_instance ::= [ name_of_gate_instance ]
      ( inout_terminal , inout_terminal , enable_terminal )
pull_gate_instance ::= [ name_of_gate_instance ] ( output_terminal )
name_of_gate_instance ::= gate_instance_identifier [ range ]
pulldown_strength ::= (From A.3.2)
        ( strength0 , strength1 )
      | ( strength1 , strength0 )
      | ( strength0 )
pullup_strength ::= ( strength0 , strength1 )
      | ( strength1 , strength0 )
      | ( strength1 )
enable_terminal ::= (From A.3.3)
      expression
inout_terminal ::= net_lvalue
input_terminal ::= expression
ncontrol_terminal ::= expression
output_terminal ::= net_lvalue
pcontrol_terminal ::= expression
cmos_switchtype ::= (From A.3.4)
      cmos | rcmos
enable_gatetype ::= bufif0 | bufif1 | notif0 | notif1
mos_switchtype ::= nmos | pmos | rnmos | rpmos
n_input_gatetype ::= and | nand | or | nor | xor | xnor
n_output_gatetype ::= buf | not
pass_en_switchtype ::= tranif0 | tranif1 | rtranif1 | rtranif0
pass_switchtype ::= tran | rtran
```

Syntax 7-1—Syntax for gate instantiation

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.1.1 The gate type specification

A gate or switch instance declaration shall begin with the keyword that specifies the gate or switch primitive being used by the instances that follow in the declaration. Table 7-1 lists the keywords that shall begin a gate or a switch instance declaration.

Table 7-1—Built-in gates and switches

n_input gates	n_output gates	Three-state gates	Pull gates	MOS switches	Bidirectional switches
and	**buf**	**bufif0**	**pulldown**	**cmos**	**rtran**
nand	**not**	**bufif1**	**pullup**	**nmos**	**rtranif0**
nor		**notif0**		**pmos**	**rtranif1**
or		**notif1**		**rcmos**	**tran**
xnor				**rnmos**	**tranif0**
xor				**rpmos**	**tranif1**

Explanations of the built-in gates and switches shown in Table 7-1 begin in 7.2.

7.1.2 The drive strength specification

An optional drive strength specification shall specify the *strength* of the logic values on the output terminals of the gate instance. Only the instances of the gate primitives shown in Table 7-2 can have the drive strength specification.

Table 7-2—Valid gate types for strength specifications

and	**nand**	**buf**	**not**	**pulldown**
or	**nor**	**bufif0**	**notif0**	**pullup**
xor	**xnor**	**bufif1**	**notif1**	

The drive strength specification for a gate instance, with the exception of **pullup** and **pulldown**, shall have a *strength1* specification and a *strength0* specification. The *strength1* specification shall specify the strength of signals with a logic value 1, and the *strength0* specification shall specify the strength of signals with a logic value 0. The strength specification shall follow the gate type keyword and precede any delay specification. The *strength0* specification can precede or follow the *strength1* specification. The *strength1* and *strength0* specifications shall be separated by a comma and enclosed within a pair of parentheses.

The **pullup** gate can have only a *strength1* specification; a *strength0* specification shall be optional. The **pulldown** gate can have only a *strength0* specification; a *strength1* specification shall be optional. See 7.8 for more details.

The *strength1* specification shall be one of the following keywords:

supply1 **strong1** **pull1** **weak1**

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The *strength0* specification shall be one of the following keywords:

supply0 **strong0** **pull0** **weak0**

Specifying **highz1** as *strength1* shall cause the gate or switch to output a logic value `z` in place of a `1`. Specifying **highz0** shall cause the gate to output a logic value `z` in place of a `0`. The strength specifications **(highz0, highz1)** and **(highz1, highz0)** shall be considered invalid.

In the absence of a strength specification, the instances shall have the default strengths **strong1** and **strong0**.

For example:

The following example shows a drive strength specification in a declaration of an open collector **nor** gate:

nor (highz1,strong0) `n1(out1,in1,in2);`

In this example, the **nor** gate outputs a `z` in place of a `1`.

Logic strength modeling is discussed in more detail in 7.9 through 7.13.

7.1.3 The delay specification

An optional delay specification shall specify the propagation delay through the gates and switches in a declaration. Gates and switches in declarations with no delay specification shall have no propagation delay. A delay specification can contain up to three delay values, depending on the gate type. The **pullup** and **pulldown** instance declarations shall not include delay specifications. Delays are discussed in more detail in 7.14.

7.1.4 The primitive instance identifier

An optional name can be given to a gate or switch instance. If multiple instances are declared as an array of instances, an identifier shall be used to name the instances.

7.1.5 The range specification

There are many situations when repetitive instances are required. These instances shall differ from each other only by the index of the vector to which they are connected.

In order to specify an array of instances, the instance name shall be followed by the range specification. The range shall be specified by two constant expressions, left-hand index (`lhi`) and right-hand index (`rhi`), separated by a colon and enclosed within a pair of square brackets. A `[lhi:rhi]` range specification shall represent an array of `abs(lhi-rhi)+1` instances. Neither of the two constant expressions are required to be zero, and `lhi` is not required to be larger than `rhi`. If both constant expressions are equal, only one instance shall be generated.

An array of instances shall have a continuous range. One instance identifier shall be associated with only one range to declare an array of instances.

The range specification shall be optional. If no range specification is given, a single instance shall be created.

For example:

The declaration shown below is illegal:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
nand #2 t_nand[0:3] ( ... ), t_nand[4:7] ( ... );
```

It could be declared correctly as one array of eight instances or as two arrays with unique names of four elements each:

```
nand #2 t_nand[0:7]( ... );
nand #2 x_nand[0:3] ( ... ), y_nand[4:7] ( ... );
```

7.1.6 Primitive instance connection list

The terminal list describes how the gate or switch connects to the rest of the model. The gate or switch type can limit these expressions. The connection list shall be enclosed in a pair of parentheses, and the terminals shall be separated by commas. The output or bidirectional terminals shall always come first in the terminal list, followed by the input terminals.

The terminal connections for an array of instances shall follow these rules:

— The bit length of each port expression in the declared instance-array shall be compared with the bit length of each single-instance port or terminal in the instantiated module or primitive.
— For each port or terminal where the bit length of the instance-array port expression is the same as the bit length of the single-instance port, the instance-array port expression shall be connected to each single-instance port.
— If bit lengths are different, each instance shall get a part-select of the port expression as specified in the range, starting with the right-hand index.
— Too many or too few bits to connect to all the instances shall be considered an error.

An individual instance from an array of instances shall be referenced in the same manner as referencing an element of an array of regs.

For example:

Example 1—The following declaration of `nand_array` declares four instances that can be referenced by `nand_array[1]`, `nand_array[2]`, `nand_array[3]`, and `nand_array[4]`, respectively.

```
nand #2 nand_array[1:4]( ... ) ;
```

Example 2—The two module descriptions that follow are equivalent except for indexed instance names, and they demonstrate the range specification and connection rules for declaring an array of instances:

```
module driver (in, out, en);
input [3:0] in;
output [3:0] out;
input en;

bufif0 ar[3:0] (out, in, en); // array of three-state buffers

endmodule

module driver_equiv (in, out, en);
input [3:0] in;
output [3:0] out;
input en;

bufif0 ar3 (out[3], in[3], en); // each buffer declared separately
bufif0 ar2 (out[2], in[2], en);
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
bufif0 ar1 (out[1], in[1], en);
bufif0 ar0 (out[0], in[0], en);

endmodule
```

Example 3—The two module descriptions that follow are equivalent except for indexed instance names, and they demonstrate how different instances within an array of instances are connected when the port sizes do not match:

```
module busdriver (busin, bushigh, buslow, enh, enl);
input [15:0] busin;
output [7:0] bushigh, buslow;
input enh, enl;

driver busar3 (busin[15:12], bushigh[7:4], enh);
driver busar2 (busin[11:8], bushigh[3:0], enh);
driver busar1 (busin[7:4], buslow[7:4], enl);
driver busar0 (busin[3:0], buslow[3:0], enl);

endmodule

module busdriver_equiv (busin, bushigh, buslow, enh, enl);
input [15:0] busin;
output [7:0] bushigh, buslow;
input enh, enl;

driver busar[3:0]    (.out({bushigh, buslow}), .in(busin),
                      .en({enh, enh, enl, enl}));
endmodule
```

Example 4—This example demonstrates how a series of modules can be chained together. Figure 7-1 shows an equivalent schematic interconnection of DFF instances.

```
module dffn (q, d, clk);
parameter bits = 1;
input [bits-1:0] d;
output [bits-1:0] q;
input clk ;

DFF dff[bits-1:0] (q, d, clk); // create a row of D flip-flops

endmodule

module MxN_pipeline (in, out, clk);
parameter M = 3, N = 4;        // M=width,N=depth
input [M-1:0] in;
output [M-1:0] out;
input clk;
wire [M*(N-1):1] t;

// #(M) redefines the bits parameter for dffn
// create p[1:N] columns of dffn rows (pipeline)

dffn #(M) p[1:N] ({out, t}, {t, in}, clk);

endmodule
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

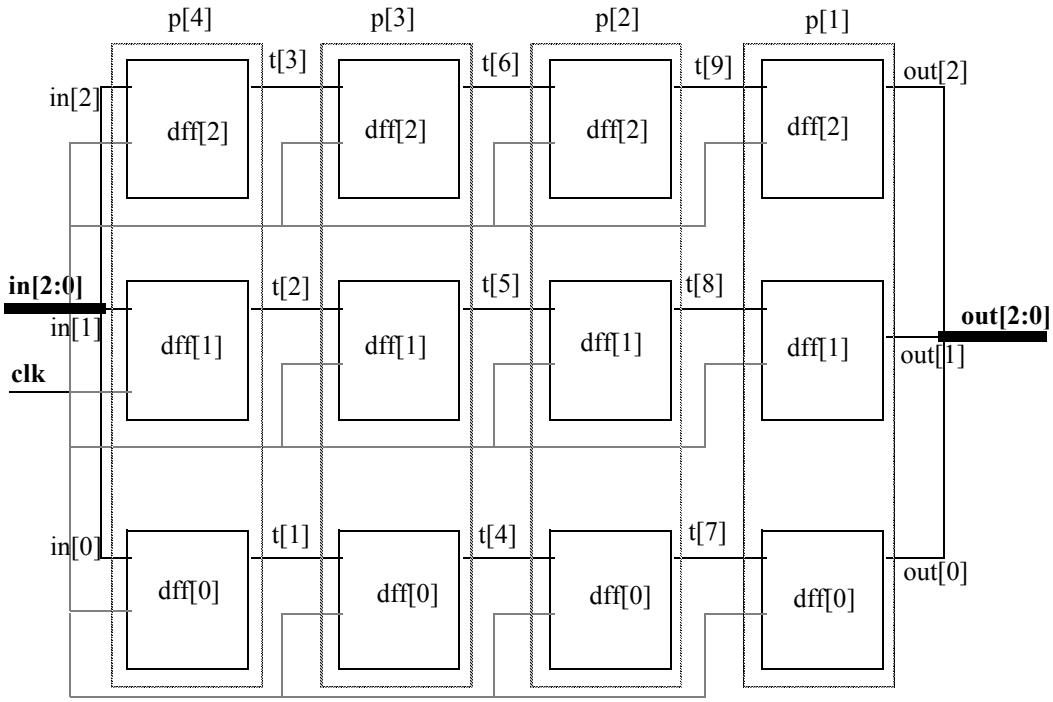


Figure 7-1—Schematic diagram of interconnections in array of instances

7.2 and, nand, nor, or, xor, and xnor gates

The instance declaration of a multiple input logic gate shall begin with one of the following keywords:

and nand nor or xor xnor

The delay specification shall be zero, one, or two delays. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to `x`. If only one delay is specified, it shall specify both the rise delay and the fall delay. If there is no delay specification, there shall be no propagation delay through the gate.

These six logic gates shall have one output and one or more inputs. The first terminal in the terminal list shall connect to the output of the gate and all other terminals connect to its inputs.

The truth tables for these gates, showing the result of two input values, appear in Table 7-3.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 7-3—Truth tables for multiple input logic gates

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Versions of these six logic gates having more than two inputs shall have a natural extension, but the number of inputs shall not alter propagation delays.

For example:

The following example declares a two-input **and** gate:

```
and a1 (out, in1, in2);
```

The inputs are `in1` and `in2`. The output is `out`. The instance name is `a1`.

7.3 buf and not gates

The instance declaration of a multiple output logic gate shall begin with one of the following keywords:

buf **not**

The delay specification shall be zero, one, or two delays. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to `x`. If only one delay is specified, it shall specify both the rise delay and the fall delay. If there is no delay specification, there shall be no propagation delay through the gate.

These two logic gates shall have one input and one or more outputs. The last terminal in the terminal list shall connect to the input of the logic gate, and the other terminals shall connect to the outputs of the logic gate.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Truth tables for these logic gates with one input and one output are shown in Table 7-4.

Table 7-4—Truth tables for multiple output logic gates

buf	
input	output
0	0
1	1
x	x
z	x

not	
input	output
0	1
1	0
x	x
z	x

For example:

The following example declares a two-output **buf**:

```
buf b1 (out1, out2, in);
```

The input is `in`. The outputs are `out1` and `out2`. The instance name is `b1`.

7.4 bufif1, bufif0, notif1, and notif0 gates

The instance declaration of these three-state logic gates shall begin with one of the following keywords:

bufif0 **bufif1** **notif1** **notif0**

These four logic gates model three-state drivers. In addition to logic values `1` and `0`, these gates can output `z`.

The delay specification shall be zero, one, two, or three delays. If the delay specification contains three delays, the first delay shall determine the rise delay, the second delay shall determine the fall delay, the third delay shall determine the delay of transitions to `z`, and the smallest of the three delays shall determine the delay of transitions to `x`. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to `x` and `z`. If only one delay is specified, it shall specify the delay for all output transitions. If there is no delay specification, there shall be no propagation delay through the gate.

Some combinations of data input values and control input values can cause these gates to output either of two values, without a preference for either value (see 7.10.2). These logic tables for these gates include two symbols representing such unknown results. The symbol `L` shall represent a result that has a value `0` or `z`. The symbol `H` shall represent a result that has a value `1` or `z`. Delays on transitions to `H` or `L` shall be treated the same as delays on transitions to `x`.

These four logic gates shall have one output, one data input, and one control input. The first terminal in the terminal list shall connect to the output, the second terminal shall connect to the data input, and the third terminal shall connect to the control input.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 7-5 presents the logic tables for these gates.

Table 7-5—Truth tables for three-state logic gates

bufif0	CONTROL				
		0	**1**	**x**	**z**
D	0	0	z	L	L
A	1	1	z	H	H
T	x	x	z	x	x
A	z	x	z	x	x

bufif1	CONTROL				
		0	**1**	**x**	**z**
D	0	z	0	L	L
A	1	z	1	H	H
T	x	z	x	x	x
A	z	z	x	x	x

notif0	CONTROL				
		0	**1**	**x**	**z**
D	0	1	z	H	H
A	1	0	z	L	L
T	x	x	z	x	x
A	z	x	z	x	x

notif1	CONTROL				
		0	**1**	**x**	**z**
D	0	z	1	H	H
A	1	z	0	L	L
T	x	z	x	x	x
A	z	z	x	x	x

For example:

The following example declares an instance of **bufif1**:

```
bufif1 bf1 (outw, inw, controlw);
```

The output is `outw`, the input is `inw`, and the control is `controlw`. The instance name is `bf1`.

7.5 MOS switches

The instance declaration of a MOS switch shall begin with one of the following keywords:

cmos **nmos** **pmos** **rcmos** **rnmos** **rpmos**

The **cmos** and **rcmos** switches are described in 7.7.

The **pmos** keyword stands for the P-type metal-oxide semiconductor (PMOS) transistor and the **nmos** keyword stands for the N-type metal-oxide semiconductor (NMOS) transistor. PMOS and NMOS transistors have relatively low impedance between their sources and drains when they conduct. The **rpmos** keyword stands for resistive PMOS transistor and the **rnmos** keyword stands for resistive NMOS transistor. Resistive PMOS and resistive NMOS transistors have significantly higher impedance between their sources and drains when they conduct than PMOS and NMOS transistors have. The load devices in static MOS networks are examples of **rpmos** and **rnmos** transistors. These four switches are *unidirectional channels* for data similar to the **bufif** gates.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The delay specification shall be zero, one, two, or three delays. If the delay specification contains three delays, the first delay shall determine the rise delay, the second delay shall determine the fall delay, the third delay shall determine the delay of transitions to `z`, and the smallest of the three delays shall determine the delay of transitions to `x`. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to `x` and `z`. If only one delay is specified, it shall specify the delay for all output transitions. If there is no delay specification, there shall be no propagation delay through the switch.

Some combinations of data input values and control input values can cause these switches to output either of two values, without a preference for either value. The logic tables for these switches include two symbols representing such unknown results. The symbol `L` represents a result that has a value `0` or `z`. The symbol `H` represents a result that has a value `1` or `z`. Delays on transitions to `H` and `L` shall be the same as delays on transitions to `x`.

These four switches shall have one output, one data input, and one control input. The first terminal in the terminal list shall connect to the output, the second terminal shall connect to the data input, and the third terminal shall connect to the control input.

The **nmos** and **pmos** switches shall pass signals from their inputs and through their outputs with a change in the strength of the signal in only one case, as discussed in 7.11. The **rnmos** and **rpmos** switches shall reduce the strength of signals that propagate through them, as discussed in 7.12.

Table 7-6 presents the logic tables for these switches.

Table 7-6—Truth tables for MOS switches

pmos rpmos	CONTROL				
		0	**1**	**x**	**z**
D	0	0	z	L	L
A	1	1	z	H	H
T	x	x	z	x	x
A	z	z	z	z	z

nmos rnmos	CONTROL				
		0	**1**	**x**	**z**
D	0	z	0	L	L
A	1	z	1	H	H
T	x	z	x	x	x
A	z	z	z	z	z

For example:

The following example declares a **pmos** switch:

```
pmos p1 (out, data, control);
```

The output is `out`, the data input is `data`, and the control input is `control`. The instance name is `p1`.

7.6 Bidirectional pass switches

The instance declaration of a bidirectional pass switch shall begin with one of the following keywords:

tran	**tranif1**	**tranif0**
rtran	**rtranif1**	**rtranif0**

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The bidirectional pass switches shall not delay signals propagating through them. When **tranif0**, **tranif1**, **rtranif0**, or **rtranif1** devices are turned off, they shall block signals; and when they are turned on, they shall pass signals. The **tran** and **rtran** devices cannot be turned off, and they shall always pass signals.

The delay specifications for **tranif1**, **tranif0**, **rtranif1**, and **rtranif0** devices shall be zero, one, or two delays. If the specification contains two delays, the first delay shall determine the turn-on delay, the second delay shall determine the turn-off delay, and the smaller of the two delays shall apply to output transitions to `x` and `z`. If only one delay is specified, it shall specify both the turn-on and the turn-off delays. If there is no delay specification, there shall be no turn-on or turn-off delay for the bidirectional pass switch.

The bidirectional pass switches **tran** and **rtran** shall not accept delay specification.

The **tranif1**, **tranif0**, **rtranif1**, and **rtranif0** devices shall have three items in their terminal lists. The first two shall be bidirectional terminals that conduct signals to and from the devices, and the third terminal shall connect to a control input. The **tran** and **rtran** devices shall have terminal lists containing two bidirectional terminals. Both bidirectional terminals shall unconditionally conduct signals to and from the devices, allowing signals to pass in either direction through the devices. The bidirectional terminals of all six devices shall be connected only to scalar nets or bit-selects of vector nets.

The **tran**, **tranif0**, and **tranif1** devices shall pass signals with an alteration in their strength in only one case, as discussed in 7.11. The **rtran**, **rtranif0**, and **rtranif1** devices shall reduce the strength of the signals passing through them according to rules discussed in 7.12.

For example:

The following example declares an instance of **tranif1**:

```
tranif1 t1 (inout1,inout2,control);
```

The bidirectional terminals are `inout1` and `inout2`. The control input is `control`. The instance name is `t1`.

7.7 CMOS switches

The instance declaration of a CMOS switch shall begin with one of the following keywords:

cmos **rcmos**

The delay specification shall be zero, one, two, or three delays. If the delay specification contains three delays, the first delay shall determine the rise delay, the second delay shall determine the fall delay, the third delay shall determine the delay of transitions to `z`, and the smallest of the three delays shall determine the delay of transitions to `x`. Delays in transitions to `H` or `L` are the same as delays in transitions to `x`. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to `x` and `z`. If only one delay is specified, it shall specify the delay for all output transitions. If there is no delay specification, there shall be no propagation delay through the switch.

The **cmos** and **rcmos** switches shall have a data input, a data output, and two control inputs. In the terminal list, the first terminal shall connect to the data output, the second terminal shall connect to the data input, the third terminal shall connect to the n-channel control input, and the last terminal shall connect to the p-channel control input.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The **cmos** gate shall pass signals with an alteration in their strength in only one case, as discussed in 7.11. The **rcmos** gate shall reduce the strength of signals passing through it according to rules described in 7.12.

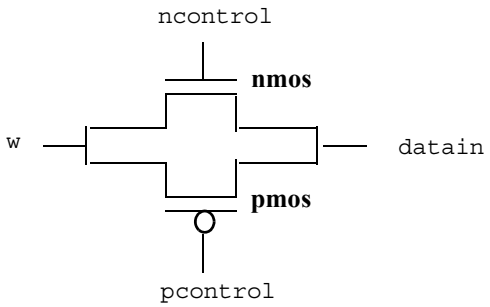
The **cmos** switch shall be treated as the combination of a **pmos** switch and an **nmos** switch. The **rcmos** switch shall be treated as the combination of an **rpmos** switch and an **rnmos** switch. The combined switches in these configurations shall share data input and data output terminals, but they shall have separate control inputs.

For example:

The equivalence of the **cmos** gate to the pairing of an **nmos** gate and a **pmos** gate is shown in the following example:

```
cmos (w, datain, ncontrol, pcontrol);
```

is equivalent to:




```
nmos (w, datain, ncontrol);
pmos (w, datain, pcontrol);
```

7.8 pullup and pulldown sources

The instance declaration of a pullup or a pulldown source shall begin with one of the following keywords:

pullup **pulldown**

A **pullup** source shall place a logic value `1` on the nets connected in its terminal list. A **pulldown** source shall place a logic value `0` on the nets connected in its terminal list.

The signals that these sources place on nets shall have **pull** strength in the absence of a strength specification. If there is a *strength1* specification on a **pullup** source or a *strength0* specification on a **pulldown** source, the signals shall have the strength specified. A *strength0* specification on a **pullup** source and a *strength1* specification on a **pulldown** source shall be ignored.

There shall be no delay specifications for these sources.

For example:

The following example declares two **pullup** instances:

```
pullup (strong1) p1 (neta), p2 (netb);
```

In this example, the `p1` instance drives `neta` and the `p2` instance drives `netb` with **strong** strength.

7.9 Logic strength modeling

The Verilog HDL provides for accurate modeling of signal contention, bidirectional pass gates, resistive MOS devices, dynamic MOS, charge sharing, and other technology-dependent network configurations by

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

allowing scalar net signal values to have a full range of unknown values and different levels of strength or combinations of levels of strength. This multiple-level logic strength modeling resolves combinations of signals into known or unknown values to represent the behavior of hardware with improved accuracy.

A strength specification shall have two components:

a) The strength of the 0 portion of the net value, called *strength0*, designated as one of the following:

supply0 **strong0** **pull0** **weak0** **highz0**

b) The strength of the 1 portion of the net value, called *strength1*, designated as one of the following:

supply1 **strong1** **pull1** **weak1** **highz1**

The combinations **(highz0, highz1)** and **(highz1, highz0)** shall be considered illegal.

Despite this division of the strength specification, it is helpful to consider strength as a property occupying regions of a continuum in order to predict the results of combinations of signals.

Table 7-7 demonstrates the continuum of strengths. The left column lists the keywords used in specifying strengths. The right column gives correlated strength levels.

Table 7-7—Strength levels for scalar net signal values

Strength name	Strength level
supply0	7
strong0	6
pull0	5
large0	4
weak0	3
medium0	2
small0	1
highz0	0
highz1	0
small1	1
medium1	2
weak1	3
large1	4
pull1	5
strong1	6
supply1	7

Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

In Table 7-7, there are four *driving strengths*:

supply **strong** **pull** **weak**

Signals with driving strengths shall propagate from gate outputs and continuous assignment outputs.

In Table 7-7, there are three *charge storage strengths*:

large **medium** **small**

Signals with the charge storage strengths shall originate in the **trireg** net type.

It is possible to think of the strengths of signals in Table 7-7 as locations on the scale in Figure 7-2.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Figure 7-2—Scale of strengths

Discussions of signal combinations later in this clause employ graphics similar to those used in Figure 7-2.

If the signal value of a net is known, all of its strength levels shall be in either the *strength0* part of the scale represented by Figure 7-2, or all strength levels shall be in its *strength1* part. If the signal value of a net is unknown, it shall have strength levels in both the *strength0* and the *strength1* parts. A net with a signal value `z` shall have a strength level only in one of the 0 subdivisions of the parts of the scale.

7.10 Strengths and values of combined signals

In addition to a signal value, a net shall have either a single unambiguous strength level or an ambiguous strength consisting of more than one level. When signals combine, their strengths and values shall determine the strength and value of the resulting signal in accordance with the principles in 7.10.1 through 7.10.4.

7.10.1 Combined signals of unambiguous strength

This subclause deals with combinations of signals in which each signal has a known value and a single strength level.

If two or more signals of unequal strength combine in a wired net configuration, the stronger signal shall dominate all the weaker drivers and determine the result. The combination of two or more signals of like value shall result in the same value with the greater of all the strengths. The combination of signals identical in strength and value shall result in the same signal.

The combination of signals with unlike values and the same strength can have three possible results. Two of the results occur in the presence of wired logic, and the third occurs in its absence. Wired logic is discussed in 7.10.4. The result in the absence of wired logic is the subject of Figure 7-4 (in 7.10.2).

For example:

In Figure 7-3, the numbers in parentheses indicate the relative strengths of the signals. The combination of a **pull1** and a **strong0** results in a **strong0**, which is the stronger of the two signals.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

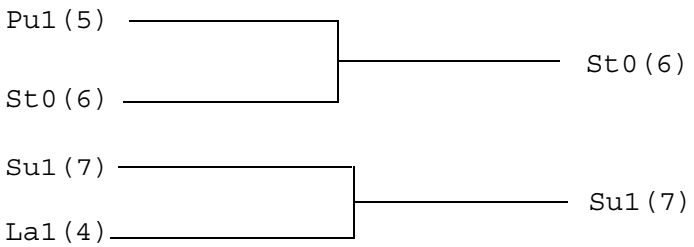


Figure 7-3—Combining unequal strengths

7.10.2 Ambiguous strengths: sources and combinations

There are several classifications of signals possessing ambiguous strengths:

— Signals with known values and multiple strength levels
— Signals with a value `x`, which have strength levels consisting of subdivisions of both the *strength1* and the *strength0* parts of the scale of strengths in Figure 7-2
— Signals with a value `L`, which have strength levels that consist of high impedance joined with strength levels in the *strength0* part of the scale of strengths in Figure 7-2
— Signals with a value `H`, which have strength levels that consist of high impedance joined with strength levels in the *strength1* part of the scale of strengths in Figure 7-2

Many configurations can produce signals of ambiguous strength. When two signals of equal strength and opposite value combine, the result shall be a value `x`, along with the strength levels of both signals and all the smaller strength levels.

For example:

Figure 7-4 shows the combination of a **weak** signal with a value `1` and a **weak** signal with a value `0` yielding a signal with **weak** strength and a value `x`.

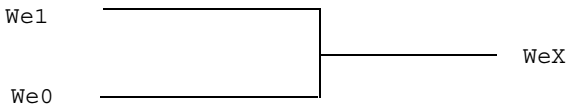


Figure 7-4—Combination of signals of equal strength and opposite values

This output signal is described in Figure 7-5.

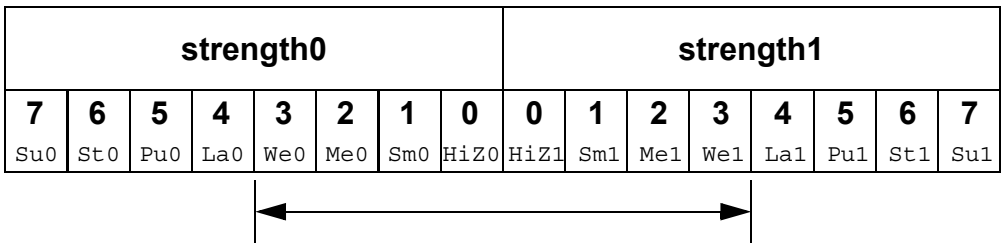


Figure 7-5—Weak x signal strength

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

An ambiguous signal strength can be a range of possible values. An example is the strength of the output from the three-state drivers with unknown control inputs as shown in Figure 7-6.

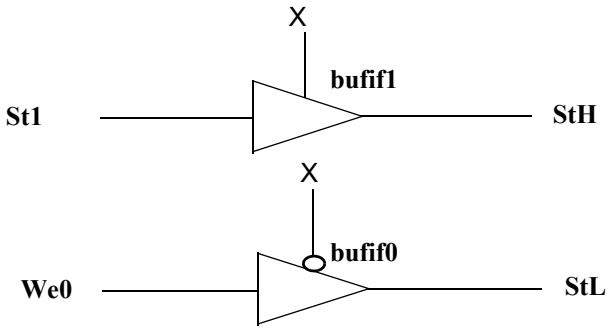


Figure 7-6—Bufifs with control inputs of x

The output of the **bufif1** in Figure 7-6 is a **strong** `H`, composed of the range of values described in Figure 7-7.

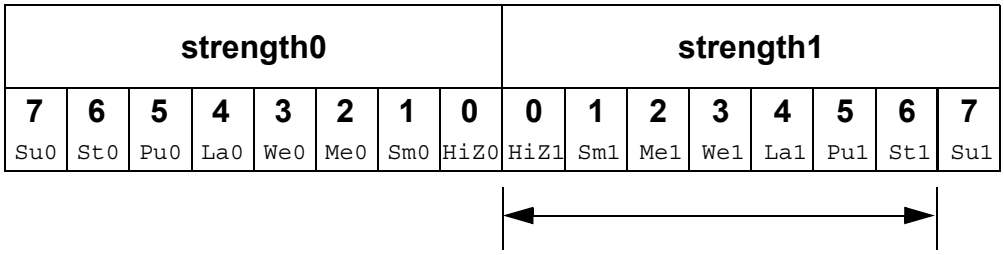


Figure 7-7—Strong H range of values

The output of the **bufif0** in Figure 7-6 is a **strong** `L`, composed of the range of values described in Figure 7-8.

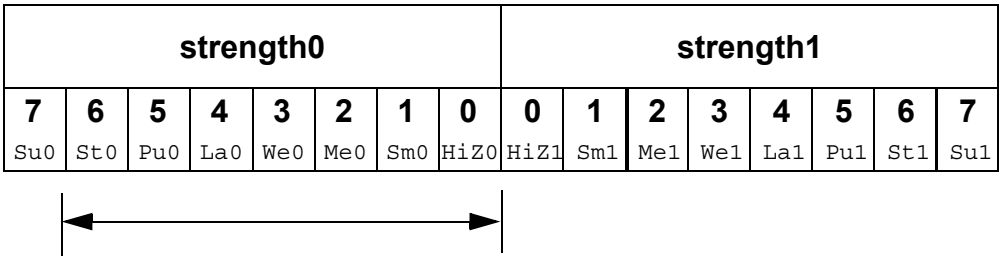


Figure 7-8—Strong L range of values

The combination of two signals of ambiguous strength shall result in a signal of ambiguous strength. The resulting signal shall have a range of strength levels that includes the strength levels in its component signals. The combination of outputs from two three-state drivers with unknown control inputs, shown in Figure 7-9, is an example.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

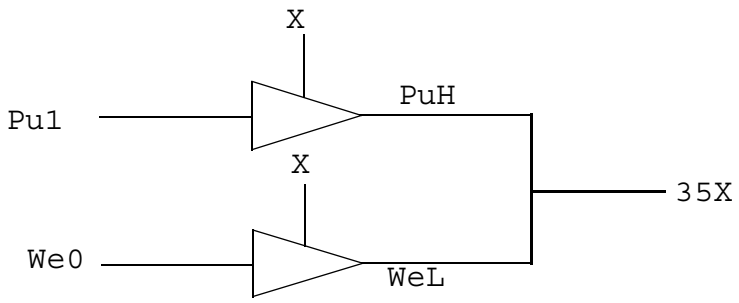


Figure 7-9—Combined signals of ambiguous strength

In Figure 7-9, the combination of signals of ambiguous strengths produces a range that includes the extremes of the signals and all the strengths between them, as described in Figure 7-10.

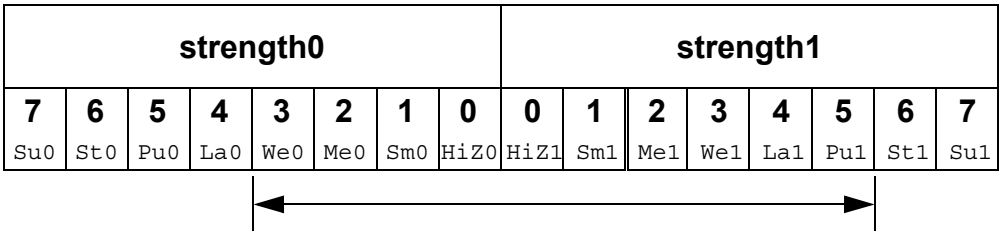


Figure 7-10—Range of strengths for an unknown signal

The result is a value `x` because its range includes the values `1` and `0`. The number `35`, which precedes the `x`, is a concatenation of two digits. The first is the digit 3, which corresponds to the highest *strength0* level for the result. The second digit, 5, corresponds to the highest *strength1* level for the result.

Switch networks can produce a ranges of strengths of the same value, such as the signals from the upper and lower configurations in Figure 7-11.

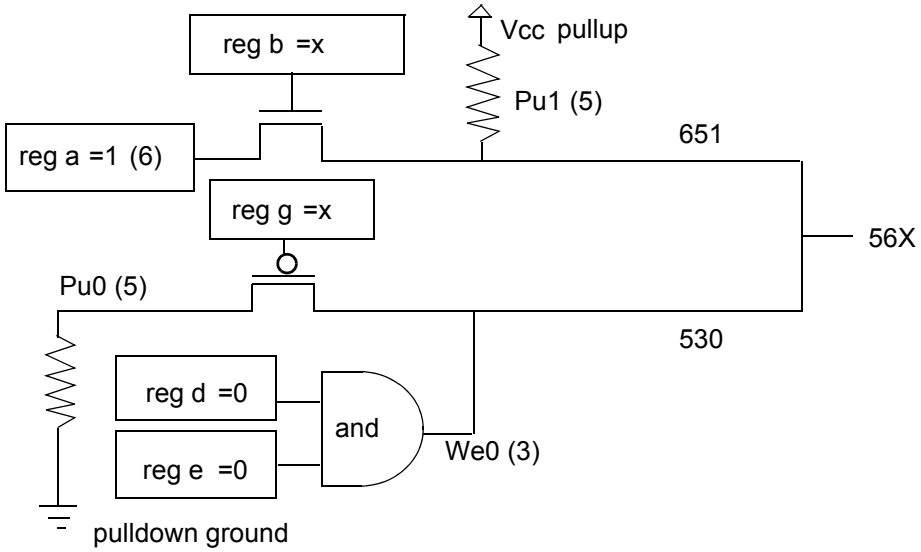


Figure 7-11—Ambiguous strengths from switch networks

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

In Figure 7-11, the upper combination of a reg, a gate controlled by a reg of unspecified value, and a pullup produces a signal with a value of 1 and a range of strengths (651) described in Figure 7-12.

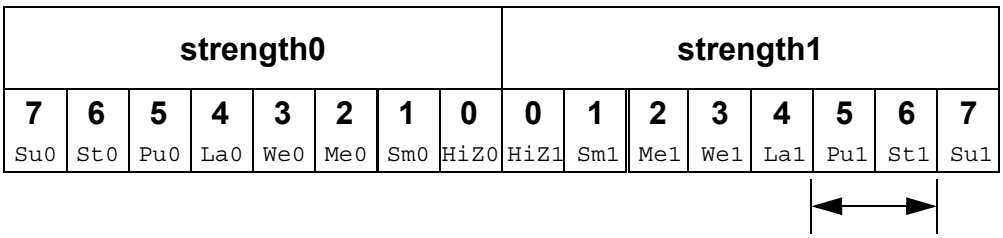


Figure 7-12—Range of two strengths of a defined value

In Figure 7-11, the lower combination of a **pulldown**, a gate controlled by a reg of unspecified value, and an **and** gate produces a signal with a value 0 and a range of strengths (530) described in Figure 7-13.

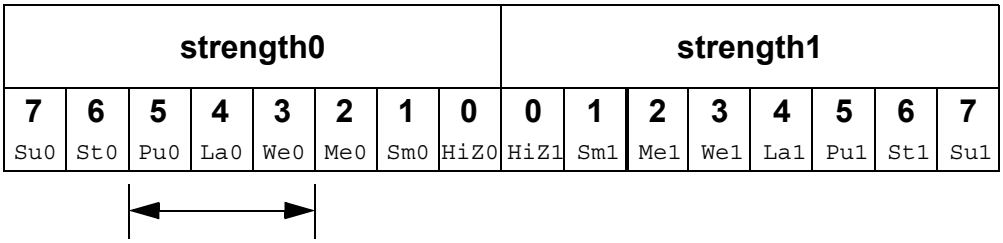


Figure 7-13—Range of three strengths of a defined value

When the signals from the upper and lower configurations in Figure 7-11 combine, the result is an unknown with a range (56x) determined by the extremes of the two signals shown in Figure 7-14.

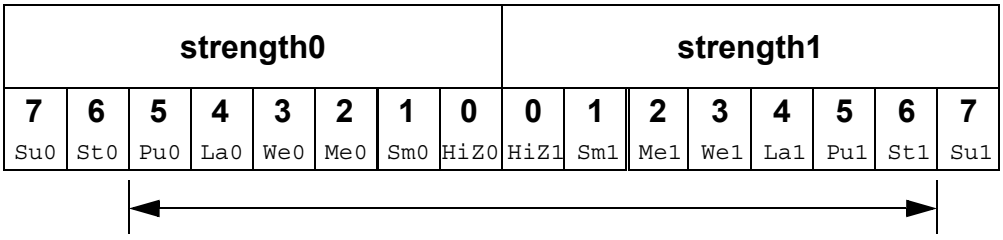


Figure 7-14—Unknown value with a range of strengths

In Figure 7-11, replacing the **pulldown** in the lower configuration with a **supply0** would change the range of the result to the range (StX) described in Figure 7-15.

The range in Figure 7-15 is **strong** x because it is unknown and the extremes of both its components are **strong**. The extreme of the output of the lower configuration is **strong** because the lower **pmos** reduces the strength of the **supply0** signal. This modeling feature is discussed in 7.11.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

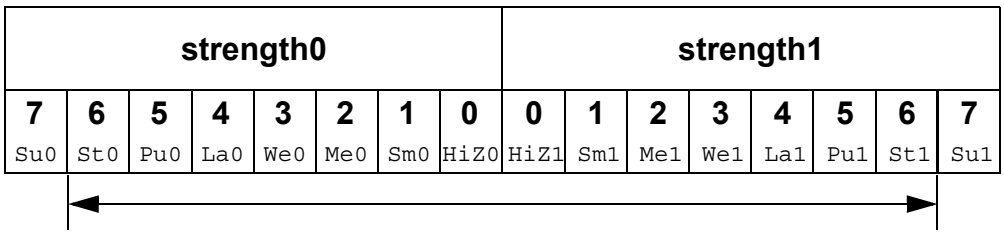


Figure 7-15—Strong X range

Logic gates produce results with ambiguous strengths as well as three-state drivers. Such a case appears in Figure 7-16. The **and** gate `N1` is declared with **highz0** strength, and `N2` is declared with **weak0** strength.

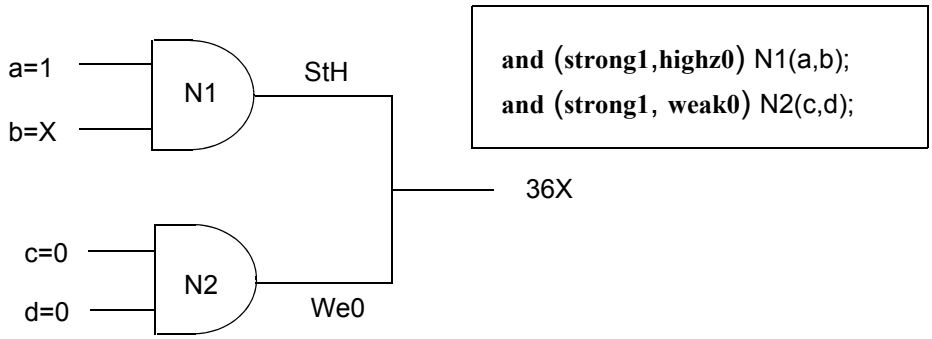


Figure 7-16—Ambiguous strength from gates

In Figure 7-16, reg `b` has an unspecified value; therefore, input to the upper **and** gate is **strong** `x`. The upper **and** gate has a strength specification including **highz0**. The signal from the upper **and** gate is a **strong** `H` composed of the values as described in Figure 7-17.

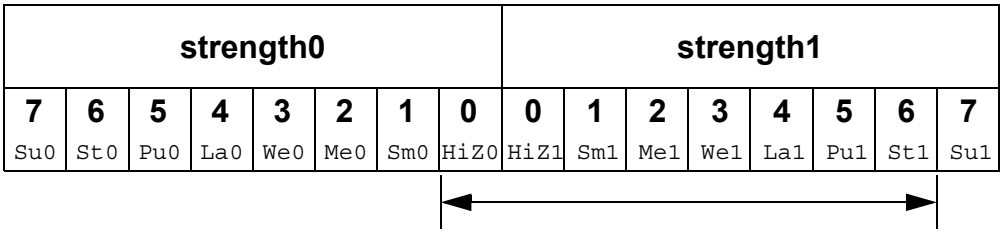


Figure 7-17—Ambiguous strength signal from a gate

`HiZ0` is part of the result because the strength specification for the gate in question specified that strength for an output with a value `0`. A strength specification other than high impedance for the `0` value output results in a gate output value `x`. The output of the lower **and** gate is a **weak** `0` as described in Figure 7-18.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

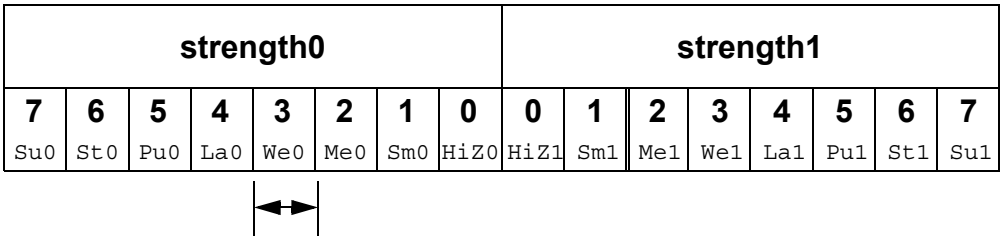


Figure 7-18—Weak 0

When the signals combine, the result is the range (`36x`) as described in Figure 7-19.

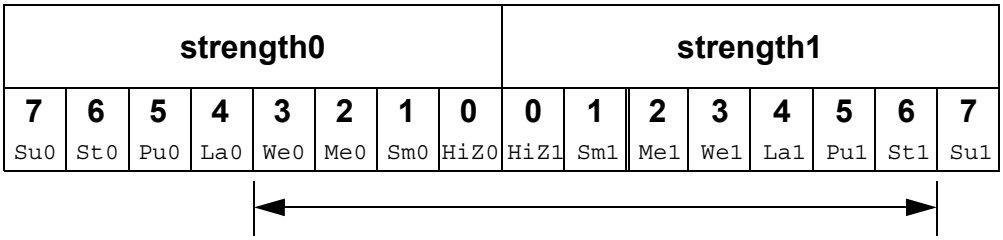


Figure 7-19—Ambiguous strength in combined gate signals

Figure 7-19 presents the combination of an ambiguous signal and an unambiguous signal. Such combinations are the topic of 7.10.3.

7.10.3 Ambiguous strength signals and unambiguous signals

The combination of a signal with unambiguous strength and known value with another signal of ambiguous strength presents several possible cases. To understand a set of rules governing this type of combination, it is necessary to consider the strength levels of the ambiguous strength signal separately from each other and relative to the unambiguous strength signal. When a signal of known value and unambiguous strength combines with a component of a signal of ambiguous strength, these shall be the rules:

a) The strength levels of the ambiguous strength signal that are greater than the strength level of the unambiguous signal shall remain in the result.
b) The strength levels of the ambiguous strength signal that are smaller than or equal to the strength level of the unambiguous signal shall disappear from the result, subject to rule c.
c) If the operation of rule a and rule b results in a gap in strength levels because the signals are of opposite value, the signals in the gap shall be part of the result.

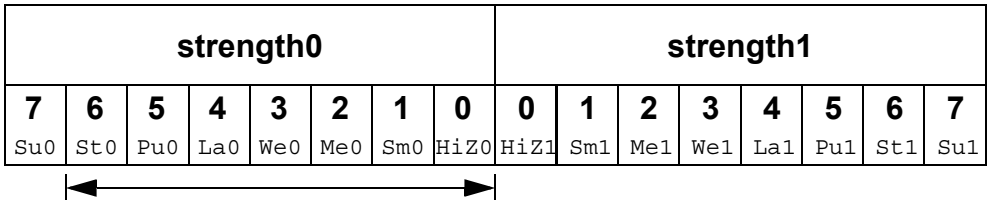
The following figures show some applications of the rules.

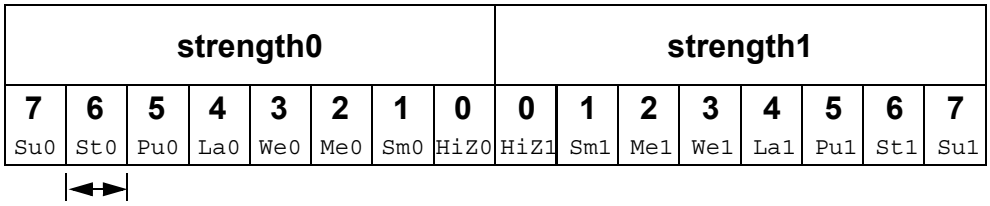
In Figure 7-20, the strength levels in the ambiguous strength signal that are smaller than or equal to the strength level of the unambiguous strength signal disappear from the result, demonstrating rule b.

In Figure 7-21, rule a, rule b, and rule c apply. The strength levels of the ambiguous strength signal that are of opposite value and lesser strength than the unambiguous strength signal disappear from the result. The strength levels in the ambiguous strength signal that are less than the strength level of the unambiguous strength signal, and of the same value, disappear from the result. The strength level of the unambiguous strength signal and the greater extreme of the ambiguous strength signal define a range in the result.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale





Combining the two signals above results in the following signal:

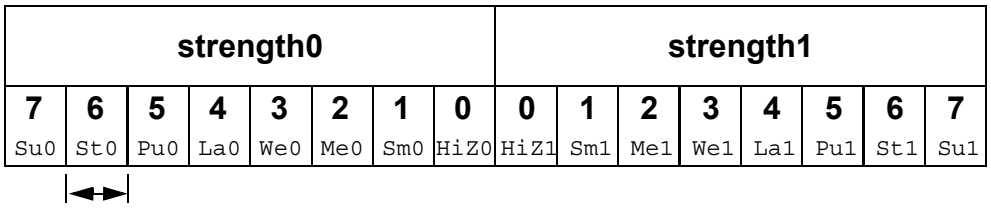
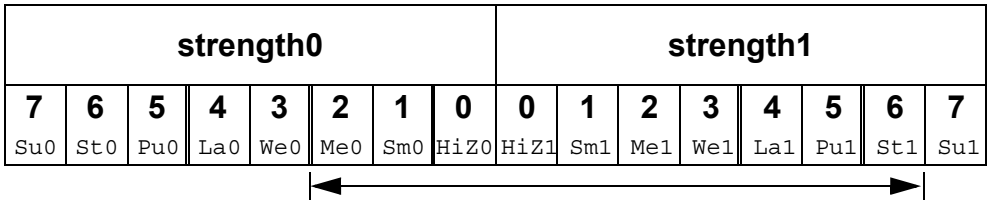
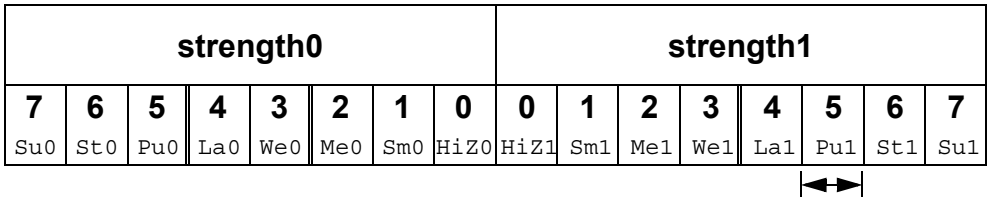


Figure 7-20—Elimination of strength levels





Combining the two signals above results in the following signal:

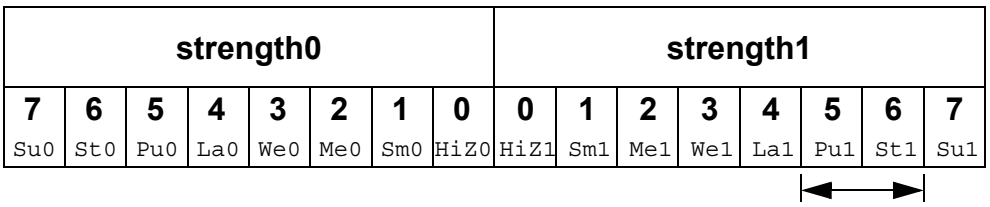


Figure 7-21—Result showing a range and the elimination of strength levels of two values

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

In Figure 7-22, rule a and rule b apply. The strength levels in the ambiguous strength signal that are less than the strength level of the unambiguous strength signal disappear from the result. The strength level of the unambiguous strength signal and the strength level at the greater extreme of the ambiguous strength signal define a range in the result.

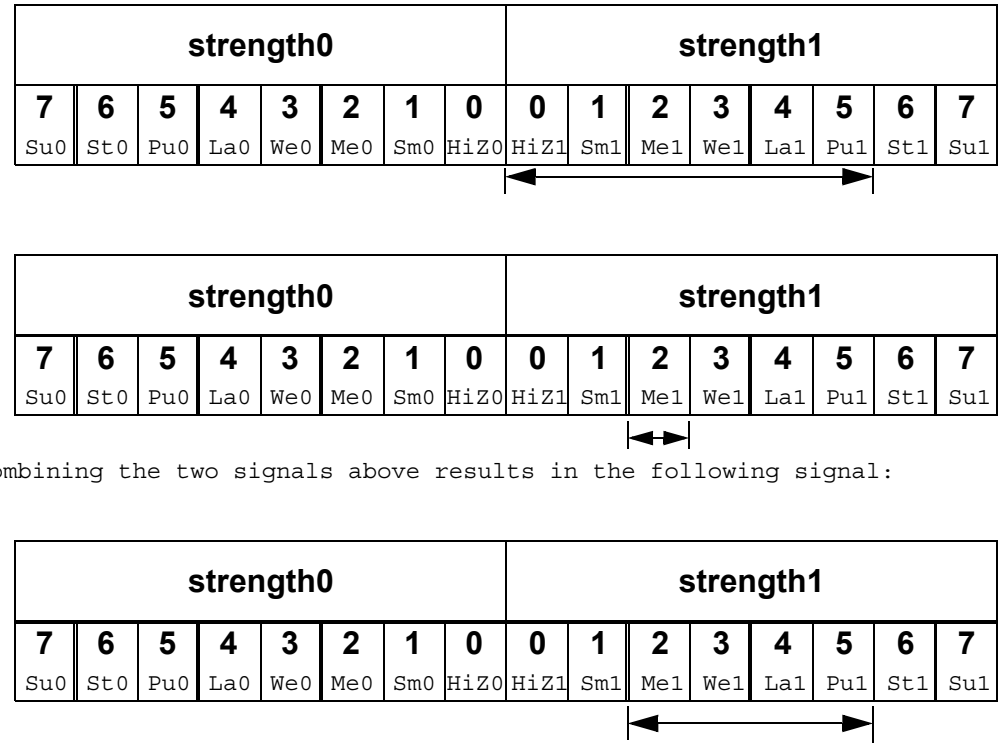


Figure 7-22—Result showing a range and the elimination of strength levels of one value

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

In Figure 7-23, rule a, rule b, and rule c apply. The greater extreme of the range of strengths for the ambiguous strength signal is larger than the strength level of the unambiguous strength signal. The result is a range defined by the greatest strength in the range of the ambiguous strength signal and by the strength level of the unambiguous strength signal.

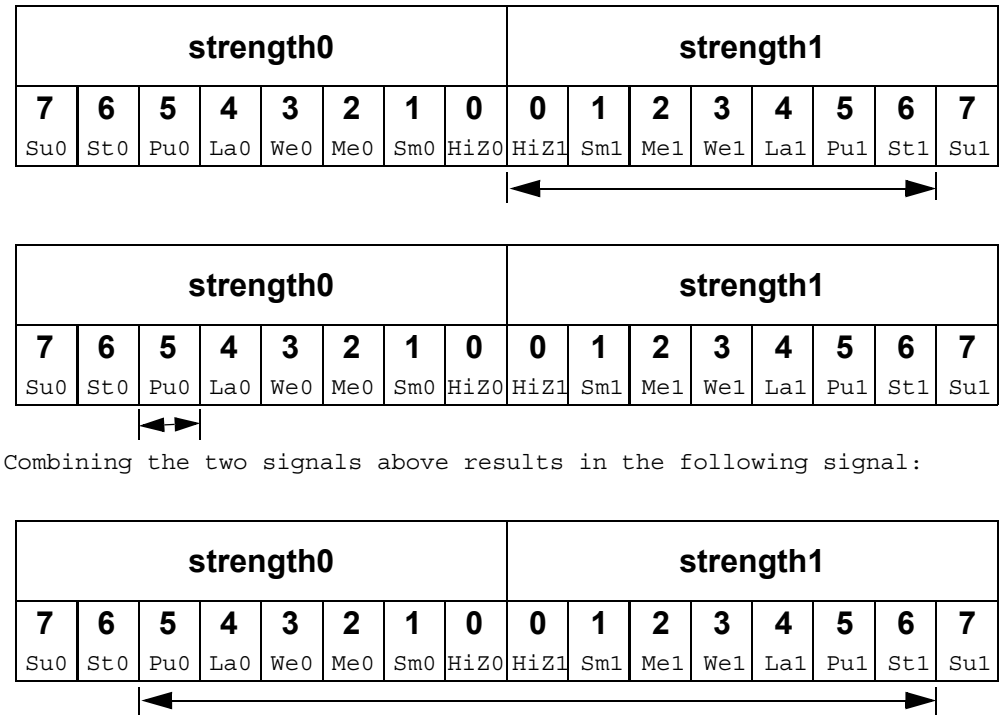


Figure 7-23—A range of both values

Copyright © 2006 IEEE. All rights reserved.

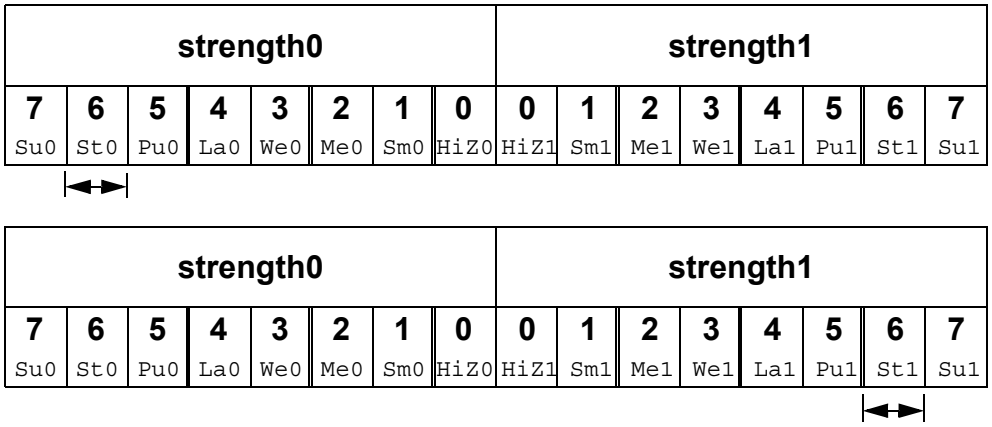
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.10.4 Wired logic net types

The net types **triand**, **wand**, **trior**, and **wor** shall resolve conflicts when multiple drivers have the same strength. These net types shall resolve signal values by treating signals as inputs of logic functions.

For example:

Consider the combination of two signals of unambiguous strength in Figure 7-24.




```
wired AND logic value result:    0
wired OR  logic value result:    1
```

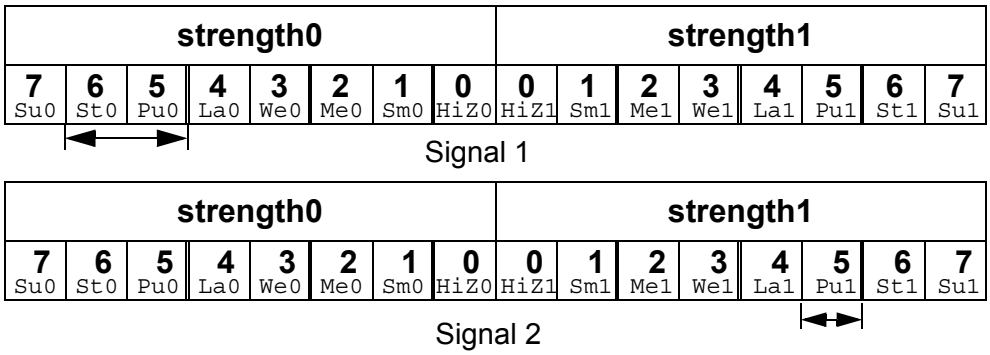
Figure 7-24—Wired logic with unambiguous strength signals

The combination of the signals in Figure 7-24, using *wired and* logic, produces a result with the same value as the result produced by an **and** gate with the value of the two signals as its inputs. The combination of signals using *wired or* logic produces a result with the same value as the result produced by an **or** gate with the values of the two signals as its inputs. The strength of the result is the same as the strength of the combined signals in both cases. If the value of the upper signal changes so that both signals in Figure 7-24 possess a value `1`, then the results of both types of logic have a value `1`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

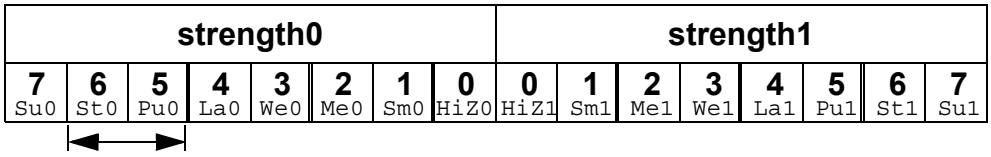
When ambiguous strength signals combine in wired logic, it is necessary to consider the results of all combinations of each of the strength levels in the first signal with each of the strength levels in the second signal, as shown in Figure 7-25.



The combinations of strength levels for *and* logic appear in the following chart:

signal1		signal2		result	
strength	**value**	**strength**	**value**	**strength**	**value**
5	0	5	1	5	0
6	0	5	1	6	0

The result is the following signal:



The combinations of strength levels for *or* logic appear in the following chart:

signal1		signal2		result	
strength	**value**	**strength**	**value**	**strength**	**value**
5	0	5	1	5	1
6	0	5	1	6	0

The result is the following signal:

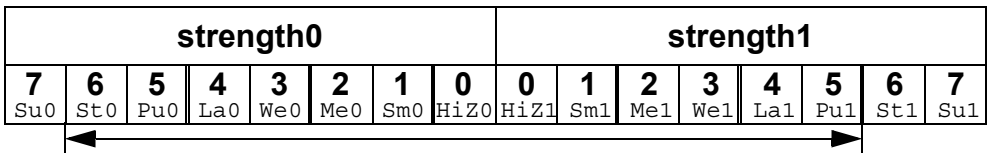


Figure 7-25—Wired logic and ambiguous strengths

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.11 Strength reduction by nonresistive devices

The **nmos**, **pmos**, and **cmos** switches shall pass the strength from the data input to the output, except that a **supply** strength shall be reduced to a **strong** strength.

The **tran**, **tranif0**, and **tranif1** switches shall not affect signal strength across the bidirectional terminals, except that a **supply** strength shall be reduced to a **strong** strength.

7.12 Strength reduction by resistive devices

The **rnmos**, **rpmos**, **rcmos**, **rtran**, **rtranif1**, and **rtranif0** devices shall reduce the strength of signals that pass through them according to Table 7-8.

Table 7-8—Strength reduction rules

Input strength	Reduced strength
Supply drive	Pull drive
Strong drive	Pull drive
Pull drive	Weak drive
Large capacitor	Medium capacitor
Weak drive	Medium capacitor
Medium capacitor	Small capacitor
Small capacitor	Small capacitor
High impedance	High impedance

7.13 Strengths of net types

The **tri0**, **tri1**, **supply0**, and **supply1** net types shall generate signals with specific strength levels. The **trireg** declaration can specify either of two signal strength levels other than a default strength level.

7.13.1 tri0 and tri1 net strengths

The **tri0** net type models a net connected to a resistive **pulldown** device. In the absence of an overriding source, such a signal shall have a value `0` and a **pull** strength. The **tri1** net type models a net connected to a resistive **pullup** device. In the absence of an overriding source, such a signal shall have a value `1` and a **pull** strength.

7.13.2 trireg strength

The **trireg** net type models charge storage nodes. The strength of the drive resulting from a **trireg** net that is in the charge storage state (that is, a driver charged the net and then went to high impedance) shall be one of these three strengths: **large**, **medium**, or **small**. The specific strength associated with a particular **trireg** net shall be specified by the user in the net declaration. The default shall be **medium**. The syntax of this specification is described in 4.4.1.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

7.13.3 supply0 and supply1 net strengths

The **supply0** net type models ground connections. The **supply1** net type models connections to power supplies. The **supply0** and **supply1** net types shall have **supply** driving strengths.

7.14 Gate and net delays

Gate and net delays provide a means of more accurately describing delays through a circuit. The *gate delays* specify the signal propagation delay from any gate input to the gate output. Up to three values per output representing rise, fall, and turn-off delays can be specified (see 7.2 through 7.8).

Net delays refer to the time it takes from any driver on the net changing value to the time when the net value is updated and propagated further. Up to three delay values per net can be specified.

For both gates and nets, the *default delay* shall be zero when no delay specification is given. When one delay value is given, then this value shall be used for all propagation delays associated with the gate or the net. When two delays are given, the first delay shall specify the rise delay, and the second delay shall specify the fall delay. The delay when the signal changes to high impedance or to unknown shall be the lesser of the two delay values.

For a three-delay specification,

— The first delay refers to the transition to the 1 value (rise delay).
— The second delay refers to the transition to the 0 value (fall delay).
— The third delay refers to the transition to the high-impedance value.

When a value changes to the unknown (x) value, the delay is the smallest of the three delays. The strength of the input signal shall not affect the propagation delay from an input to an output.

Table 7-9 summarizes the from-to propagation delay choice for the two- and three-delay specifications.

Table 7-9—Rules for propagation delays

From value:	**To value**:	**Delay used if there are**	
		2 delays	**3 delays**
0	1	d1	d1
0	x	min(d1, d2)	min(d1, d2, d3)
0	z	min(d1, d2)	d3
1	0	d2	d2
1	x	min(d1, d2)	min(d1, d2, d3)
1	z	min(d1, d2)	d3
x	0	d2	d2
x	1	d1	d1
x	z	min(d1, d2)	d3
z	0	d2	d2

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 7-9—Rules for propagation delays *(continued)*

From value:	To value:	Delay used if there are	
		2 delays	3 delays
z	1	d1	d1
z	x	min(d1, d2)	min(d1, d2, d3)

For example:

Example 1—The following is an example of a delay specification with one, two, and three delays:

```
and #(10) a1 (out, in1, in2);        // only one delay
and #(10,12) a2 (out, in1, in2);     // rise and fall delays
bufif0 #(10,12,11) b3 (out, in, ctrl);// rise, fall, and turn-off delays
```

Example 2—The following example specifies a simple latch module with three-state outputs, where individual delays are given to the gates. The propagation delay from the primary inputs to the outputs of the module will be cumulative, and it depends on the signal path through the network.

```
module tri_latch (qout, nqout, clock, data, enable);
output qout, nqout;
input clock, data, enable;
tri qout, nqout;

not     #5          n1 (ndata, data);
nand    #(3,5)      n2 (wa, data, clock),
                    n3 (wb, ndata, clock);
nand    #(12,15)    n4 (q, nq, wa),
                    n5 (nq, q, wb);
bufif1  #(3,7,13)   q_drive (qout, q, enable),
                    nq_drive (nqout, nq, enable);

endmodule
```

7.14.1 min:typ:max delays

The syntax for delays on gate primitives (including UDPs; see Clause 8), nets, and continuous assignments shall allow three values each for the rising, falling, and turn-off delays. The minimum, typical, and maximum values for each delay shall be specified as expressions separated by colons. There shall be no required relation (e.g., min ≤ typ ≤ max) between the expressions for minimum, typical, and maximum delays. These can be any three expressions.

For example:

The following example shows `min:typ:max` values for rising, falling, and turn-off delays:

```
module iobuf (io1, io2, dir);
       ...
bufif0 #(5:7:9, 8:10:12, 15:18:21) b1 (io1, io2, dir);
bufif1 #(6:8:10, 5:7:9, 13:17:19)  b2 (io2, io1, dir);
       ...
endmodule
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The syntax for delay controls in procedural statements (see 9.7) also allows minimum, typical, and maximum values. These are specified by expressions separated by colons. The following example illustrates this concept.

```
parameter min_hi = 97, typ_hi = 100, max_hi = 107;
reg clk;

always begin
    #(95:100:105) clk = 1;
    #(min_hi:typ_hi:max_hi) clk = 0;
end
```

7.14.2 trireg net charge decay

Like all nets, the delay specification in a **trireg** net declaration can contain up to three delays. The first two delays shall specify the delay for transition to the `1` and `0` logic states when the **trireg** net is driven to these states by a driver. The third delay shall specify the *charge decay time* instead of the delay in a transition to the `z` logic state. The charge decay time specifies the delay between when the drivers of a **trireg** net turn off and when its stored charge can no longer be determined.

A **trireg** net does not need a turn-off delay specification because a **trireg** net never makes a transition to the `z` logic state. When the drivers of a **trireg** net make transitions from the `1`, `0`, or `x` logic states to off, the **trireg** net shall retain the previous `1`, `0`, or `x` logic state that was on its drivers. The `z` value shall not propagate from the drivers of a **trireg** net to a **trireg** net. A **trireg** net can only hold a `z` logic state when `z` is the initial logic state of the **trireg** net or when the **trireg** net is forced to the `z` state with a **force** statement (see 9.3.2).

A delay specification for charge decay models a charge storage node that is not ideal, i.e., a charge storage node whose charge leaks out through its surrounding devices and connections.

The charge decay process and the delay specification for charge decay are described in 7.14.2.1 and 7.14.2.2, respectively.

7.14.2.1 Charge decay process

Charge decay is the cause of transition of a 1 or 0 that is stored in a **trireg** net to an unknown value (`x`) after a specified delay. The charge decay process shall begin when the drivers of the **trireg** net turn off and the **trireg** net starts to hold charge. The charge decay process shall end under the following two conditions:

a) The delay specified by charge decay time elapses, and the **trireg** net makes a transition from `1` or `0` to `x`.
b) The drivers of **trireg** net turn on and propagate a `1`, `0`, or `x` into the **trireg** net.

7.14.2.2 Delay specification for charge decay time

The third delay in a **trireg** net declaration shall specify the charge decay time. A three-valued delay specification in a **trireg** net declaration shall have the following form:

```
#(d1, d2, d3)        // (rise_delay, fall_delay, charge_decay_time)
```

The charge decay time specification in a **trireg** net declaration shall be preceded by a rise and a fall delay specification.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

For example:

Example 1—The following example shows a specification of the charge decay time in a **trireg** net declaration:

```
trireg (large) #(0,0,50) cap1;
```

This example declares a **trireg** net named `cap1`. This **trireg** net stores a **large** charge. The delay specifications for the rise delay is 0, the fall delay is 0, and the charge decay time specification is 50 time units.

Example 2—The next example presents a source description file that contains a **trireg** net declaration with a charge decay time specification. Figure 7-26 shows an equivalent schematic for the source description.

```
module capacitor;
reg data, gate;

// trireg declaration with a charge decay time of 50 time units
trireg (large) #(0,0,50) cap1;

nmos nmos1 (cap1, data, gate); // nmos that drives the trireg

initial begin
    $monitor("%0d data=%v gate=%v cap1=%v", $time, data, gate, cap1);
    data = 1;
    // Toggle the driver of the control input to the nmos switch
     gate = 1;
    #10 gate = 0;
    #30 gate = 1;
    #10 gate = 0;
    #100 $finish;
end
endmodule
```

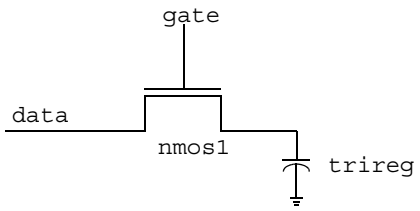


Figure 7-26—Trireg net with capacitance

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

8. User-defined primitives (UDPs)

This clause describes a modeling technique to augment the set of predefined gate primitives by designing and specifying new primitive elements called UDPs. Instances of these new UDPs can be used in exactly the same manner as the gate primitives to represent the circuit being modeled.

The following two types of behavior can be represented in a UDP:

a) Combinational—modeled by a *combinational UDP*
b) Sequential—modeled by a *sequential UDP*

A combinational UDP uses the value of its inputs to determine the next value of its output. A sequential UDP uses the value of its inputs and the current value of its output to determine the value of its output. Sequential UDPs provide a way to model sequential circuits such as flip-flops and latches. A sequential UDP can model both level-sensitive and edge-sensitive behavior.

Each UDP has exactly one output, which can be in one of three states: `0`, `1`, or `x`. The three-state value `z` is not supported. In sequential UDPs, the output always has the same value as the internal state.

The `z` values passed to UDP inputs shall be treated the same as `x` values.

8.1 UDP definition

UDP definitions are independent of modules; they are at the same level as module definitions in the syntax hierarchy. They can appear anywhere in the source text, either before or after they are instantiated inside a module. They shall not appear between the keywords **module** and **endmodule**.

Implementations may limit the maximum number of UDP definitions in a model, but they shall allow at least 256.

The formal syntax of the UDP definition is given in Syntax 8-1.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
udp_declaration ::= (From A.5.1)
        { attribute_instance } primitive udp_identifier ( udp_port_list ) ;
        udp_port_declaration { udp_port_declaration }
        udp_body
        endprimitive
      | { attribute_instance } primitive udp_identifier ( udp_declaration_port_list ) ;
        udp_body
        endprimitive
udp_port_list ::= (From A.5.2)
      output_port_identifier , input_port_identifier { , input_port_identifier }
udp_declaration_port_list ::=
      udp_output_declaration , udp_input_declaration { , udp_input_declaration }
udp_port_declaration ::=
        udp_output_declaration ;
      | udp_input_declaration ;
      | udp_reg_declaration ;
udp_output_declaration ::=
        { attribute_instance } output port_identifier
      | { attribute_instance } output reg port_identifier [ = constant_expression ]
udp_input_declaration ::=
      { attribute_instance } input list_of_port_identifiers
udp_reg_declaration ::=
      { attribute_instance } reg variable_identifier
udp_body ::= (From A.5.3)
      combinational_body | sequential_body
combinational_body ::=
      table combinational_entry { combinational_entry } endtable
combinational_entry ::=
      level_input_list : output_symbol ;
sequential_body ::=
      [ udp_initial_statement ] table sequential_entry { sequential_entry } endtable
udp_initial_statement ::=
      initial output_port_identifier = init_val ;
init_val ::= 1'b0 | 1'b1 | 1'bx | 1'bX | 1'B0 | 1'B1 | 1'Bx | 1'BX | 1 | 0
sequential_entry ::=
      seq_input_list : current_state : next_state ;
seq_input_list ::=
      level_input_list | edge_input_list
level_input_list ::=
      level_symbol { level_symbol }
edge_input_list ::=
      { level_symbol } edge_indicator { level_symbol }
edge_indicator ::=
      ( level_symbol level_symbol ) | edge_symbol
current_state ::= level_symbol
next_state ::=output_symbol | -
output_symbol ::= 0 | 1 | x | X
level_symbol ::= 0 | 1 | x | X | ? | b | B
edge_symbol ::= r | R | f | F | p | P | n | N | *
```

Syntax 8-1—Syntax for UDPs

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

8.1.1 UDP header

A UDP definition shall have one of two alternate forms. The first form shall begin with the keyword **primitive**, followed by an identifier, which shall be the name of the UDP. This in turn shall be followed by a comma-separated list of port names enclosed in parentheses, which shall be followed by a semicolon. The UDP definition header shall be followed by port declarations and a state table. The UDP definition shall be terminated by the keyword **endprimitive**.

The second form shall begin with the keyword **primitive**, followed by an identifier, which shall be the name of the UDP. This in turn shall be followed by a comma-separated list of port declarations enclosed in parentheses, followed by a semicolon. The UDP definition header shall be followed by a state table. The UDP definition shall be terminated by the keyword **endprimitive**.

UDPs have multiple input ports and exactly one output port; bidirectional inout ports are not permitted on UDPs. All ports of a UDP shall be scalar; vector ports are not permitted.

The output port shall be the first port in the port list.

8.1.2 UDP port declarations

UDPs shall contain input and output port declarations. The output port declaration begins with the keyword **output**, followed by one output port name. The input port declaration begins with the keyword **input**, followed by one or more input port names.

Sequential UDPs shall contain a **reg** declaration for the output port, either in addition to the output declaration, when the UDP is declared using the first form of a UDP Header, or as part of the output_declaration. Combinational UDPs cannot contain a **reg** declaration. The initial value of the output port can be specified in an **initial** statement in a sequential UDP (see 8.1.3).

Implementations may limit the maximum number of inputs to a UDP, but they shall allow at least 9 inputs for sequential UDPs and 10 inputs for combinational UDPs.

8.1.3 Sequential UDP initial statement

The sequential UDP initial statement specifies the value of the output port when simulation begins. This statement begins with the keyword **initial**. The statement that follows shall be an assignment statement that assigns a single-bit literal value to the output port.

8.1.4 UDP state table

The state table defines the behavior of a UDP. It begins with the keyword **table** and is terminated with the keyword **endtable**. Each row of the table is terminated by a semicolon.

Each row of the table is created using a variety of characters (see Table 8-1), which indicate input values and output state. Three states—`0`, `1`, and `x`—are supported. The `z` state is explicitly excluded from consideration in UDPs. A number of special characters are defined to represent certain combinations of state possibilities. These are described in Table 8-1.

The order of the input state fields of each row of the state table is taken directly from the port list in the UDP definition header. It is not related to the order of the input port declarations.

Combinational UDPs have one field per input and one field for the output. The input fields are separated from the output field by a colon (:). Each row defines the output for a particular combination of the input values (see 8.2).

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Sequential UDPs have an additional field inserted between the input fields and the output field. This additional field represents the current state of the UDP and is considered equivalent to the current output value. It is delimited by colons. Each row defines the output based on the current state, particular combinations of input values, and at most one input transition (see 8.4). A row such as the one shown below is illegal:

```
(01) (10) 0 : 0 : 1 ;
```

If all input values are specified as `x`, then the output state shall be specified as `x`.

It is not necessary to explicitly specify every possible input combination. All combinations of input values that are not explicitly specified result in a default output state of `x`.

It is illegal to have the same combination of inputs, including edges, specified for different outputs.

8.1.5 Z values in UDP

The `z` value in a table entry is not supported, and it is considered illegal. The `z` values passed to UDP inputs shall be treated the same as `x` values.

8.1.6 Summary of symbols

To improve the readability and to ease writing of the state table, several special symbols are provided. Table 8-1 summarizes the meaning of all the value symbols that are valid in the table part of a UDP definition.

Table 8-1—UDP table symbols

Symbol	Interpretation	Comments
0	Logic 0	
1	Logic 1	
x	Unknown	Permitted in the input and output fields of all UDPs and in the current state field of sequential UDPs.
?	Iteration of 0, 1, and x	Not permitted in output field.
b	Iteration of 0 and 1	Permitted in the input fields of all UDPs and in the current state field of sequential UDPs. Not permitted in the output field.
-	No change	Permitted only in the output field of a sequential UDP.
(vw)	Value change from v to w	v and w can be any one of 0, 1, x, ?, or b, and are only permitted in the input field.
*	Same as (??)	Any value change on input.
r	Same as (01)	Rising edge on input.
f	Same as (10)	Falling edge on input.
p	Iteration of (01), (0 x) and (x1)	Potential positive edge on the input.
n	Iteration of (10), (1x)and (x0	Potential negative edge on the input.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

8.2 Combinational UDPs

In combinational UDPs, the output state is determined solely as a function of the current input states. Whenever an input state changes, the UDP is evaluated and the output state is set to the value indicated by the row in the state table that matches all the input states. All combinations of the inputs that are not explicitly specified will drive the output state to the unknown value `x`.

For example:

The following example defines a multiplexer with two data inputs and a control input:

```
primitive multiplexer (mux, control, dataA, dataB);
output mux;
input control, dataA, dataB;
table
// control  dataA   dataB  mux
       0      1       0   :  1 ;
       0      1       1   :  1 ;
       0      1       x   :  1 ;
       0      0       0   :  0 ;
       0      0       1   :  0 ;
       0      0       x   :  0 ;
       1      0       1   :  1 ;
       1      1       1   :  1 ;
       1      x       1   :  1 ;
       1      0       0   :  0 ;
       1      1       0   :  0 ;
       1      x       0   :  0 ;
       x      0       0   :  0 ;
       x      1       1   :  1 ;
endtable
endprimitive
```

The first entry in this example can be explained as follows: when `control` equals `0`, `dataA` equals `1`, and `dataB` equals `0`, then output `mux` equals `1`.

The input combination `0xx (control=0, dataA=x, dataB=x)` is not specified. If this combination occurs during simulation, the value of output port `mux` will become `x`.

Using `?`, the description of a multiplexer can be abbreviated as follows:

```
primitive multiplexer (mux, control, dataA, dataB);
output mux;
input control, dataA, dataB;
table
// control  dataA  dataB  mux
       0      1      ?   :  1 ;    // ? = 0 1 x
       0      0      ?   :  0 ;
       1      ?      1   :  1 ;
       1      ?      0   :  0 ;
       x      0      0   :  0 ;
       x      1      1   :  1 ;
endtable
endprimitive
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

8.3 Level-sensitive sequential UDPs

Level-sensitive sequential behavior is represented the same way as combinational behavior, except that the output is declared to be of type **reg** and there is an additional field in each table entry. This new field represents the current state of the UDP. The output field in a sequential UDP represents the next state.

For example:

Consider the example of a latch:

```
primitive latch (q, clock, data);
output q; reg q;
input clock, data;
table
// clock data  q   q+
     0     1 : ? : 1 ;
     0     0 : ? : 0 ;
     1     ? : ? : - ; // - = no change
endtable
endprimitive
```

This description differs from a combinational UDP model in two ways. First, the output identifier `q` has an additional **reg** declaration to indicate that there is an internal state `q`. The output value of the UDP is always the same as the internal state. Second, a field for the current state, which is separated by colons from the inputs and the output, has been added.

8.4 Edge-sensitive sequential UDPs

In level-sensitive behavior, the values of the inputs and the current state are sufficient to determine the output value. Edge-sensitive behavior differs in that changes in the output are triggered by specific transitions of the inputs. This makes the state table a transition table.

Each table entry can have a transition specification on at most one input. A transition is specified by a pair of values in parentheses such as `(01)` or a transition symbol such as `r`. Entries such as the following are illegal:

```
(01)(01)0 : 0 : 1 ;
```

All transitions that do not affect the output shall be explicitly specified. Otherwise, such transitions cause the value of the output to change to `x`. All unspecified transitions default to the output value `x`**.**

If the behavior of the UDP is sensitive to edges of any input, the desired output state shall be specified for all edges of all inputs.

For example:

The following example describes a rising edge D flip-flop:

```
primitive d_edge_ff (q, clock, data);
output q; reg q;
input clock, data;
table
// clock  data      q      q+
   // obtain output on rising edge of clock
   (01)    0     :   ?   :   0   ;
   (01)    1     :   ?   :   1   ;
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
    (0?)    1    :   1   :   1   ;
    (0?)    0    :   0   :   0   ;
    // ignore negative edge of clock
    (?0)    ?    :   ?   :   -   ;
    // ignore data changes on steady clock
     ?     (??)  :   ?   :   -   ;
endtable
endprimitive
```

The terms such as `(01)` represent transitions of the input values. Specifically, `(01)` represents a transition from `0` to `1`. The first line in the table of the preceding UDP definition is interpreted as follows: when clock changes value from `0` to `1` and data equals `0`, the output goes to `0` no matter what the current state.

The transition of clock from `0` to `x` with data equal to `0` and current state equal to `1` will result in the output `q` going to `x`.

8.5 Sequential UDP initialization

The initial value on the output port of a sequential UDP can be specified with an initial statement that provides a procedural assignment. The initial statement is optional.

Like initial statements in modules, the initial statement in UDPs begins with the keyword **initial**. The valid contents of initial statements in UDPs and the valid left-hand and right-hand sides of their procedural assignment statements differ from initial statements in modules. A partial list of differences between these two types of initial statements is described in Table 8-2.

Table 8-2—Initial statements in UDPs and modules

Initial statements in UDPs	Initial statements in modules
Contents limited to one procedural assignment statement	Contents can be one procedural statement of any type or a block statement that contains more than one procedural statement
The procedural assignment statement shall assign a value to a reg whose identifier matches the identifier of an output terminal	Procedural assignment statements in initial statements can assign values to a reg whose identifier does not match the identifier of an output terminal
The procedural assignment statement shall assign one of the following values: 1'b1, 1'b0, 1'bx, 1, 0	Procedural assignment statements can assign values of any size, radix, and value

For example:

Example 1—The following example shows a sequential UDP that contains an initial statement.

```
primitive srff (q, s, r);
output q; reg q;
input s, r;
initial q = 1'b1;
table
//  s  r   q   q+
    1  0 : ? : 1 ;
    f  0 : 1 : - ;
    0  r : ? : 0 ;
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
    0   f : 0 : - ;
    1   1 : ? : 0 ;
endtable
endprimitive
```

The output `q` has an initial value of 1 at the start of the simulation; a delay specification on an instantiated UDP does not delay the simulation time of the assignment of this initial value to the output. When simulation starts, this value is the current state in the state table. Delays are not permitted in a UDP initial statement.

Example 2—The following example and Figure 8-1 show how values are applied in a module that instantiates a sequential UDP with an initial statement:

```
primitive dff1 (q, clk, d);
input clk, d;
output q; reg q;
initial q = 1'b1;
table
// clk     d       q       q+
    r      0   :   ?   :   0   ;
    r      1   :   ?   :   1   ;
    f      ?   :   ?   :   -   ;
    ?      *   :   ?   :   -   ;
endtable
endprimitive

module dff (q, qb, clk, d);
input clk, d;
output q, qb;
   dff1   g1 (qi, clk, d);
   buf #3 g2 (q, qi);
   not #5 g3 (qb, qi);
endmodule
```

The UDP `dff1` contains an initial statement that sets the initial value of its output to 1. The module `dff` contains an instance of UDP `dff1`.

Figure 8-1 shows the schematic of the preceding module and the simulation propagation times of the initial value of the UDP output.

In Figure 8-1, the fanout from the UDP output `qi` includes nets `q` and `qb`. At simulation time 0, `qi` changes value to `1`. That initial value of `qi` does not propagate to net `q` until simulation time 3, and it does not propagate to net `qb` until simulation time 5.

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

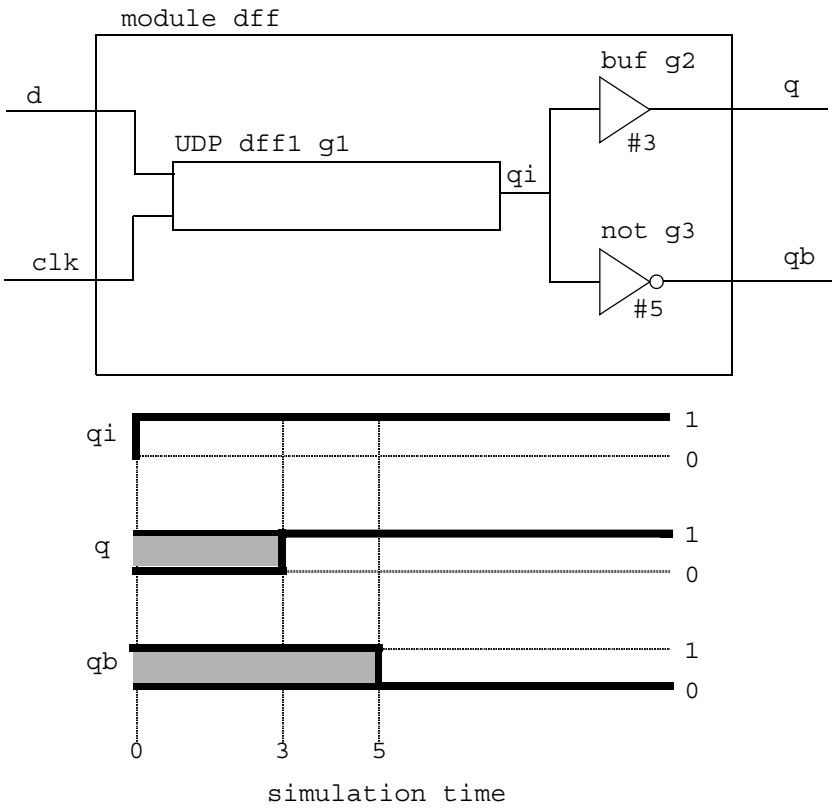


Figure 8-1—Module schematic and simulation times of initial value propagation

8.6 UDP instances

The syntax for creating a UDP instance is shown in Syntax 8-2.

```
udp_instantiation ::= (From A.5.4)
        udp_identifier [ drive_strength ] [ delay2 ]
            udp_instance { , udp_instance } ;
udp_instance ::=
        [ name_of_udp_instance ] ( output_terminal , input_terminal
            { , input_terminal } )
name_of_udp_instance ::=
        udp_instance_identifier [ range ]
```

Syntax 8-2—Syntax for UDP instances

Instances of UDPs are specified inside modules in the same manner as gates (see 7.1). The instance name is optional, just as for gates. The port connection order is as specified in the UDP definition. Only two delays may be specified because `z` is not supported for UDPs. An optional range may be specified for an array of UDP instances. The port connection rules remain the same as outlined in 7.1.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

For example:

The following example creates an instance of the D-type flip-flop `d_edge_ff` (defined in 8.4).

```
module flip;
reg clock, data;
parameter p1 = 10;
parameter p2 = 33;
parameter p3 = 12;

d_edge_ff #p3 d_inst (q, clock, data);

initial begin
    data = 1;
    clock = 1;
    #(20 * p1) $finish;
end
always #p1 clock = ~clock;
always #p2 data = ~data;
endmodule
```

8.7 Mixing level-sensitive and edge-sensitive descriptions

UDP definitions allow a mixing of the level-sensitive and the edge-sensitive constructs in the same table. When the input changes, the edge-sensitive cases are processed first, followed by level-sensitive cases. Thus, when level-sensitive and edge-sensitive cases specify different output values, the result is specified by the level-sensitive case.

For example:

```
primitive jk_edge_ff (q, clock, j, k, preset, clear);
output q; reg q;
input clock, j, k, preset, clear;
table
// clock  jk pc state output/next state
     ?     ?? 01 : ? : 1 ; // preset logic
     ?     ?? *1 : 1 : 1 ;
     ?     ?? 10 : ? : 0 ; // clear logic
     ?     ?? 1* : 0 : 0 ;
     r     00 00 : 0 : 1 ; // normal clocking cases
     r     00 11 : ? : - ;
     r     01 11 : ? : 0 ;
     r     10 11 : ? : 1 ;
     r     11 11 : 0 : 1 ;
     r     11 11 : 1 : 0 ;
     f     ?? ?? : ? : - ;
     b     *? ?? : ? : - ; // j and k transition cases
     b     ?* ?? : ? : - ;
endtable
endprimitive
```

In this example, the preset and clear logic is level-sensitive. Whenever the preset and clear combination is `01`, the output has value `1`. Similarly, whenever the preset and clear combination has value `10`, the output has value `0`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

The remaining logic is sensitive to edges of the clock. In the normal clocking cases, the flip-flop is sensitive to the rising clock edge, as indicated by an `r` in the clock field in those entries. The insensitivity to the falling edge of clock is indicated by a hyphen (-) in the output field (see Table 8-1) for the entry with an `f` as the value of clock. Remember that the desired output for this input transition shall be specified to avoid unwanted `x` values at the output. The last two entries show that the transitions in `j` and `k` inputs do not change the output on a steady low or high `clock`.

8.8 Level-sensitive dominance

Table 8-3 shows level-sensitive and edge-sensitive entries in the example from 8.7, their level-sensitive or edge-sensitive behavior, and a case of input values that each includes.

Table 8-3—Mixing of level-sensitive and edge-sensitive entries

Entry	Included case	Behavior
? ?? 01: ?: 1;	0 00 01: 0: 1;	Level-sensitive
f ?? ??: ?: -;	f 00 01: 0: 0;	Edge-sensitive

The included cases specify opposite next state values for the same input and current state combination. The level-sensitive included case specifies that when the inputs `clock`, `jk`, and `pc` values are `0`, `00`, and `01` and the current state is `0`, the output changes to `1`. The edge-sensitive included case specifies that when `clock` falls from `1` to `0`, the other inputs `jk` and `pc` are `00` and `01`, and the current state is `0`, then the output changes to `0`.

When the edge-sensitive case is processed first, followed by the level-sensitive case, the output changes to `1`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

9. Behavioral modeling

The language constructs introduced so far allow hardware to be described at a relatively detailed level. Modeling a circuit with logic gates and continuous assignments reflects quite closely the logic structure of the circuit being modeled; however, these constructs do not provide the power of abstraction necessary for describing complex high-level aspects of a system. The procedural constructs described in this clause are well suited to tackling problems such as describing a microprocessor or implementing complex timing checks.

This clause starts with a brief overview of a behavioral model to provide a context for many types of behavioral statements in the Verilog HDL.

9.1 Behavioral model overview

Verilog *behavioral models* contain *procedural statements* that control the simulation and manipulate variables of the data types previously described. These statements are contained within procedures. Each procedure has an activity flow associated with it.

The activity starts at the control constructs **initial** and **always**. Each initial construct and each always construct starts a separate activity flow. All of the activity flows are concurrent to model the inherent concurrence of hardware. These constructs are formally described in 9.9.

The following example shows a complete Verilog behavioral model.

```
module behave;
reg [1:0] a, b;

initial begin
    a = 'b1;
    b = 'b0;
end
always begin
    #50 a = ~a;
end
always begin
    #100 b = ~b;
end

endmodule
```

During simulation of this model, all of the flows defined by the initial and always constructs start together at simulation time zero. The initial constructs execute once, and the always constructs execute repetitively.

In this model, the reg variables `a` and `b` initialize to 1 and 0, respectively, at simulation time zero. The initial construct is then complete and does not execute again during this simulation run. This initial construct contains a *begin-end block* (also called a *sequential block*) of statements. In this begin-end block, `a` is initialized first, followed by `b`.

The always constructs also start at time zero, but the values of the variables do not change until the times specified by the delay controls (introduced by #) have elapsed. Thus, reg `a` inverts after 50 time units and reg `b` inverts after 100 time units. Because the always constructs repeat, this model will produce two square waves. The reg `a` toggles with a period of 100 time units, and reg `b` toggles with a period of 200 time units. The two always constructs proceed concurrently throughout the entire simulation run.

 Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

9.2 Procedural assignments

As described in Clause 6, procedural assignments are used for updating **reg**, **integer**, **time**, **real**, **realtime**, and memory data types. There is a significant difference between procedural assignments and continuous assignments:

— Continuous assignments drive nets and are evaluated and updated whenever an input operand changes value.
— Procedural assignments update the value of variables under the control of the procedural flow constructs that surround them.

The right-hand side of a procedural assignment can be any expression that evaluates to a value. The left-hand side shall be a variable that receives the assignment from the right-hand side. The left-hand side of a procedural assignment can take one of the following forms:

— **reg, integer, real, realtime,** or **time** data type: an assignment to the name reference of one of these data types.
— Bit-select of a **reg, integer,** or **time** data type: an assignment to a single bit that leaves the other bits untouched.
— Part-select of a **reg, integer,** or **time** data type: a part-select of one or more contiguous bits that leaves the rest of the bits untouched.
— Memory word: a single word of a memory.
— Concatenation or nested concatenation of any of the above: a concatenation or nested concatenation of any of the previous four forms. Such specification effectively partitions the result of the right-hand expression and assigns the partition parts, in order, to the various parts of the concatenation or nested concatenation.

As described in 5.4, when the right-hand side evaluates to fewer bits than the left-hand side, the right-hand side value is padded to the size of the left-hand side. If the right-hand side is unsigned, it is padded according to the rules specified in 5.4.1. If the right-hand side is signed, it is sign-extended.

The Verilog HDL contains two types of procedural assignment statements:

— Blocking procedural assignment statements
— Nonblocking procedural assignment statements

Blocking and nonblocking procedural assignment statements specify different procedural flows in sequential blocks.

9.2.1 Blocking procedural assignments

A *blocking procedural assignment* statement shall be executed before the execution of the statements that follow it in a sequential block (see 9.8.1). A blocking procedural assignment statement shall not prevent the execution of statements that follow it in a parallel block (see 9.8.2).

The syntax for a blocking procedural assignment is given in Syntax 9-1.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
blocking_assignment ::= (From A.6.2)
      variable_lvalue = [ delay_or_event_control ] expression
delay_control ::= (From A.6.5)
       # delay_value
      | # ( mintypmax_expression )
delay_or_event_control ::=
       delay_control
      | event_control
      | repeat ( expression ) event_control
event_control ::=
       @ hierarchical_event_identifier
      | @ ( event_expression )
      | @*
      | @ (*)
event_expression ::=
       expression
      | posedge expression
      | negedge expression
      | event_expression or event_expression
      | event_expression , event_expression
variable_lvalue ::= (From A.8.5)
       hierarchical_variable_identifier [ { [ expression ] } [ range_expression ] ]
      | { variable_lvalue { , variable_lvalue } }
```

Syntax 9-1—Syntax for blocking assignments

In this syntax, `variable_lvalue` is a data type that is valid for a procedural assignment statement, `=` is the assignment operator, and `delay_or_event_control` is the optional intra-assignment timing control. The control can be either a `delay_control` (e.g., `#6`) or an `event_control` (e.g., `@(posedge clk)`). The expression is the right-hand side value that shall be assigned to the left-hand side. If `variable_lvalue` requires an evaluation, it shall be evaluated at the time specified by the intra-assignment timing control.

The `=` assignment operator used by blocking procedural assignments is also used by procedural continuous assignments and continuous assignments.

For example:

The following examples show blocking procedural assignments:

```
rega = 0;
rega[3] = 1;                  // a bit-select
rega[3:5] = 7;                // a part-select
mema[address] = 8'hff;        // assignment to a mem element
{carry, acc} = rega + regb;   // a concatenation
```

9.2.2 The nonblocking procedural assignment

The *nonblocking procedural assignment* allows assignment scheduling without blocking the procedural flow. The nonblocking procedural assignment statement can be used whenever several variable assignments within the same time step can be made without regard to order or dependence upon each other.

The syntax for a nonblocking procedural assignment is given in Syntax 9-2.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
nonblocking_assignment ::= (From A.6.2)
        variable_lvalue <= [ delay_or_event_control ] expression
delay_control ::= (From A.6.5)
         # delay_value
        | # ( mintypmax_expression )
delay_or_event_control ::=
         delay_control
        | event_control
        | repeat ( expression ) event_control
event_control ::=
         @ hierarchical_event_identifier
        | @ ( event_expression )
        | @*
        | @ (*)
event_expression ::=
         expression
        | posedge expression
        | negedge expression
        | event_expression or event_expression
        | event_expression , event_expression
variable_lvalue ::= (From A.8.5)
         hierarchical_variable_identifier [ { [ expression ] } [ range_expression ] ]
        | { variable_lvalue { , variable_lvalue } }
```

Syntax 9-2—Syntax for nonblocking assignments

In this syntax, `variable_lvalue` is a data type that is valid for a procedural assignment statement, `<=` is the nonblocking assignment operator, and `delay_or_event_control` is the optional intra-assignment timing control. If `variable_lvalue` requires an evaluation, it shall be evaluated at the same time as the expression on the right-hand side. The order of evaluation of the `variable_lvalue` and the expression on the right-hand side is undefined if timing control is not specified.

The nonblocking assignment operator is the same operator as the less-than-or-equal-to relational operator. The interpretation shall be decided from the context in which `<=` appears. When `<=` is used in an expression, it shall be interpreted as a relational operator; and when it is used in a nonblocking procedural assignment, it shall be interpreted as an assignment operator.

The nonblocking procedural assignments shall be evaluated in two steps as discussed in Clause 11. These two steps are shown in the following example:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Example 1

```
module evaluates2 (out);
output out;
reg a, b, c;

initial begin
    a = 0;
    b = 1;
    c = 0;
end

always c = #5 ~c;

always @(posedge c) begin
    a <= b; // evaluates, schedules,
    b <= a; // and executes in two steps
end
endmodule
```

Step 1:

At posedge c, the simulator evaluates the right-hand sides of the nonblocking assignments and schedules the assignments of the new values at the end of the *nonblocking assign update* events (see 11.4).

Step 2:

When the simulator activates the *nonblocking assign update* events, the simulator updates the left-hand side of each nonblocking assignment statement.

Nonblocking assignment schedules changes at time 5

a = 0
b = 1

Assignment values:

a = 1
b = 0

At the end of the time step means that the nonblocking assignments are the last assignments executed in a time step—with one exception. Nonblocking assignment events can create blocking assignment events. These blocking assignment events shall be processed after the scheduled nonblocking events.

Unlike an event or delay control for blocking assignments, the nonblocking assignment does not block the procedural flow. The nonblocking assignment evaluates and schedules the assignment, but it does not block the execution of subsequent statements in a **begin-end** block.

Example 2

```
//non_block1.v
module non_block1;
reg a, b, c, d, e, f;

//blocking assignments
initial begin
    a = #10 1; // a will be assigned 1 at time 10
    b = #2 0;  // b will be assigned 0 at time 12
    c = #4 1;  // c will be assigned 1 at time 16
end
//non-blocking assignments
initial begin
    d <= #10 1; // d will be assigned 1 at time 10
    e <= #2 0;  // e will be assigned 0 at time 2
    f <= #4 1;  // f will be assigned 1 at time 4
end
endmodule
```

scheduled changes at time 2

e = 0

scheduled changes at time 4

f = 1

scheduled changes at time 10

d = 1

As shown in the previous example, the simulator evaluates and schedules assignments for the end of the current time step and can perform swapping operations with the nonblocking procedural assignments.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Example 3

```
//non_block1.v
module non_block1;
reg a, b;
initial begin
    a  = 0;
    b  = 1;
    a <= b; // evaluates, schedules, and
    b <= a; // executes in two steps
end
initial begin
    $monitor ($time, ,"a = %b b = %b", a, b);
    #100 $finish;
end
endmodule
```

Step 1:

The simulator evaluates the right-hand side of the nonblocking assignments and schedules the assignments for the end of the current time step.

Step 2:

At the end of the current time step, the simulator updates the left-hand side of each nonblocking assignment statement.

assignment values:

a = 1
b = 0

The order of the execution of distinct nonblocking assignments to a given variable shall be preserved. In other words, if there is clear ordering of the execution of a set of nonblocking assignments, then the order of the resulting updates of the destination of the nonblocking assignments shall be the same as the ordering of the execution (see 11.4.1).

Example 4

```
module multiple;
reg a;

initial a = 1;
// The assigned value of the reg is determinate
initial begin
    a <= #4 0;    // schedules a = 0 at time 4
    a <= #4 1;    // schedules a = 1 at time 4
end               // At time 4, a = 1
endmodule
```

If the simulator executes two procedural blocks concurrently and if these procedural blocks contain nonblocking assignment operators to the same variable, the final value of that variable is indeterminate. For example, the value of reg `a` is indeterminate in the following example:

Example 5

```
module multiple2;
reg a;

initial a  = 1;
initial a <= #4 0;     // schedules 0 at time 4
initial a <= #4 1;     // schedules 1 at time 4

// At time 4, a = ??
// The assigned value of the reg is indeterminate
endmodule
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The fact that two nonblocking assignments targeting the same variable are in different blocks is not by itself sufficient to make the order of assignments to a variable indeterminate. For example, the value of reg `a` at the end of time cycle 16 is determinate in the following example:

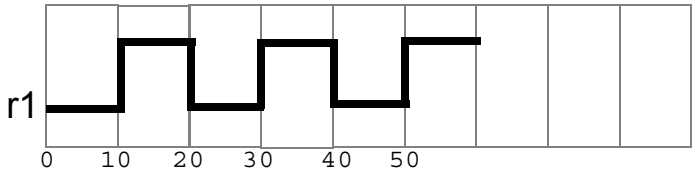
Example 6

```
module multiple3;
reg a;

initial #8 a  <= #8 1;  // executed at time 8;
                        // schedules an update of 1 at time 16
initial #12 a <= #4 0;  // executed at time 12;
                        // schedules an update of 0 at time 16
// Because it is determinate that the update of a to the value 1
// is scheduled before the update of a to the value 0,
// then it is determinate that a will have the value 0
// at the end of time slot 16.
endmodule
```

The following example shows how the value of `i[0]` is assigned to `r1` and how the assignments are scheduled to occur after each time delay:

Example 7

```
module multiple4;
reg r1;
reg [2:0] i;

initial begin
// makes assignments to r1 without cancelling previous assignments
   for (i = 0; i <= 5; i = i+1)
   r1 <= # (i*10) i[0];
end
endmodule
```



9.3 Procedural continuous assignments

The *procedural continuous assignments* (using keywords **assign** and **force**) are procedural statements that allow expressions to be driven continuously onto variables or nets. The syntax for these statements is given in Syntax 9-3.

The left-hand side of the assignment in the *assign statement* shall be a variable reference or a concatenation of variables. It shall not be a memory word (array reference) or a bit-select or a part-select of a variable.

In contrast, the left-hand side of the assignment in the *force statement* can be a variable reference or a net reference. It can be a concatenation of any of the above. Bit-selects and part-selects of vector variables are not allowed.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
net_assignment ::= (From A.6.1)
        net_lvalue = expression
procedural_continuous_assignments ::= (From A.6.2)
         assign variable_assignment
       | deassign variable_lvalue
       | force variable_assignment
       | force net_assignment
       | release variable_lvalue
       | release net_lvalue
variable_assignment ::=
        variable_lvalue = expression
net_lvalue ::= (From A.8.5)
         hierarchical_net_identifier [ { [ constant_expression ] } [ constant_range_expression ] ]
       | { net_lvalue { , net_lvalue } }
variable_lvalue ::=
         hierarchical_variable_identifier [ { [ expression ] } [ range_expression ] ]
       | { variable_lvalue { , variable_lvalue } }
```

Syntax 9-3—Syntax for procedural continuous assignments

9.3.1 The assign and deassign procedural statements

The *assign* procedural continuous assignment statement shall override all procedural assignments to a variable. The *deassign* procedural statement shall end a procedural continuous assignment to a variable. The value of the variable shall remain the same until the variable is assigned a new value through a procedural assignment or a procedural continuous assignment. The assign and deassign procedural statements allow, for example, modeling of asynchronous clear/preset on a D-type edge-triggered flip-flop, where the clock is inhibited when the clear or preset is active.

If the keyword **assign** is applied to a variable for which there is already a procedural continuous assignment, then this new procedural continuous assignment shall deassign the variable before making the new procedural continuous assignment.

For example:

The following example shows a use of the assign and deassign procedural statements in a behavioral description of a D-type flip-flop with preset and clear inputs:

```
module dff (q, d, clear, preset, clock);
output q;
input d, clear, preset, clock;
reg q;

always @(clear or preset)
    if (!clear)
        assign q = 0;
    else if (!preset)
        assign q = 1;
    else
        deassign q;

always @(posedge clock)
    q = d;
endmodule
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

If either `clear` or `preset` is low, then the output `q` will be held continuously to the appropriate constant value, and a positive edge on the `clock` will not affect `q`. When both the `clear` and `preset` are high, then `q` is deassigned.

9.3.2 The force and release procedural statements

Another form of procedural continuous assignment is provided by the *force* and *release* procedural statements. These statements have a similar effect to the **assign**-**deassign** pair, but a force can be applied to nets as well as to variables. The left-hand side of the assignment can be a variable, a net, a constant bit-select of a vector net, a part-select of a vector net, or a concatenation. It cannot be a memory word (array reference) or a bit-select or a part-select of a vector variable.

A force statement to a variable shall override a procedural assignment or an assign procedural continuous assignment to the variable until a release procedural statement is executed on the variable. When released, then if the variable does not currently have an active assign procedural continuous assignment, the variable shall not immediately change value. The variable shall maintain its current value until the next procedural assignment or procedural continuous assignment to the variable. Releasing a variable that currently has an active assign procedural continuous assignment shall immediately reestablish that assignment.

A force procedural statement on a net shall override all drivers of the net—gate outputs, module outputs, and continuous assignments—until a release procedural statement is executed on the net. When released, the net shall immediately be assigned the value determined by the drivers of the net.

For example:

```
module test;
reg a, b, c, d;
wire e;

and and1 (e, a, b, c);

initial begin
    $monitor("%d d=%b,e=%b", $stime, d, e);
    assign d = a & b & c;
    a = 1;
    b = 0;
    c = 1;
    #10;
    force d = (a | b | c);
    force e = (a | b | c);
    #10;
    release d;
    release e;
    #10 $finish;
end
endmodule

Results:

     0 d=0,e=0
    10 d=1,e=1
    20 d=0,e=0
```

In this example, an **and** gate instance `and1` is “patched” to act like an **or** gate by a **force** procedural statement that forces its output to the value of its ORed inputs, and an **assign** procedural statement of ANDed values is “patched” to act like an **assign** statement of ORed values.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

The right-hand side of a procedural continuous assignment or a **force** statement can be an expression. This shall be treated just as a continuous assignment; that is, if any variable on the right-hand side of the assignment changes, the assignment shall be reevaluated while the **assign** or **force** is in effect. For example:

```
force a = b + f(c) ;
```

Here, if `b` changes or `c` changes, `a` will be forced to the new value of the expression `b+f(c)`.

9.4 Conditional statement

The *conditional statement* (or *if-else* statement) is used to make a decision about whether a statement is executed. Formally, the syntax is given in Syntax 9-4.

```
conditional_statement ::= (From A.6.6)
        if ( expression )
          statement_or_null [ else statement_or_null ]
      | if_else_if_statement
```

Syntax 9-4—Syntax for if statement

If the expression evaluates to true (that is, has a nonzero known value), the first statement shall be executed. If it evaluates to false (that is, has a zero value or the value is `x` or `z`), the first statement shall not execute. If there is an else statement and expression is false, the else statement shall be executed.

Because the numeric value of the `if` expression is tested for being zero, certain shortcuts are possible. For example, the following two statements express the same logic:

```
if (expression)
if (expression != 0)
```

Because the else part of an if-else is optional, there can be confusion when an else is omitted from a nested if sequence. This is resolved by always associating the else with the closest previous if that lacks an else. In the example below, the else goes with the inner if, as shown by indentation.

```
if (index > 0)
    if (rega > regb)
        result = rega;
    else        // else applies to preceding if
        result = regb;
```

If that association is not desired, a begin-end block statement shall be used to force the proper association, as shown below.

```
if (index > 0) begin
    if (rega > regb)
        result = rega;
end
else result = regb;
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

9.4.1 If-else-if construct

The construction in Syntax 9-5 occurs so often that it is worth a brief separate discussion:

```
if_else_if_statement ::= (From A.6.6)
      if ( expression ) statement_or_null
      { else if ( expression ) statement_or_null }
      [ else statement_or_null ]
```

Syntax 9-5—Syntax for if-else-if construct

This sequence of if statements (known as an *if-else-if* construct) is the most general way of writing a multiway decision. The expressions shall be evaluated in order. If any expression is true, the statement associated with it shall be executed, and this shall terminate the whole chain. Each statement is either a single statement or a block of statements.

The last else part of the if-else-if construct handles the none-of-the-above or default case where none of the other conditions were satisfied. Sometimes there is no explicit action for the default. In that case, the trailing else statement can be omitted, or it can be used for error checking to catch an impossible condition.

For example:

The following module fragment uses the if-else statement to test the variable `index` to decide whether one of three `modify_segn` regs has to be added to the memory address and which increment is to be added to the `index` reg. The first ten lines declare the regs and parameters.

```
// declare regs and parameters
reg [31:0] instruction, segment_area[255:0];
reg [7:0] index;
reg [5:0] modify_seg1,
    modify_seg2,
    modify_seg3;
parameter
    segment1 = 0,  inc_seg1 = 1,
    segment2 = 20, inc_seg2 = 2,
    segment3 = 64, inc_seg3 = 4,
    data = 128;

// test the index variable
if (index < segment2) begin
    instruction = segment_area [index + modify_seg1];
    index = index + inc_seg1;
end
else if (index < segment3) begin
    instruction = segment_area [index + modify_seg2];
    index = index + inc_seg2;
end
else if (index < data) begin
    instruction = segment_area [index + modify_seg3];
    index = index + inc_seg3;
end
else
    instruction = segment_area [index];
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

9.5 Case statement

The *case* statement is a multiway decision statement that tests whether an expression matches one of a number of other expressions and branches accordingly. The case statement has the syntax shown in Syntax 9-6.

```
case_statement ::= (From A.6.7)
        case ( expression )
          case_item { case_item } endcase
      | casez ( expression )
          case_item { case_item } endcase
      | casex ( expression )
          case_item { case_item } endcase
case_item ::=
        expression { , expression } : statement_or_null
      | default [ : ] statement_or_null
```

Syntax 9-6—Syntax for case statement

The default statement shall be optional. Use of multiple default statements in one case statement shall be illegal.

The case expression and the case item expression can be computed at run time; neither expression is required to be a constant expression.

For example:

A simple example of the use of the case statement is the decoding of reg `rega` to produce a value for `result` as follows:

```
reg [15:0] rega;
reg [9:0] result;

case (rega)
    16'd0: result = 10'b0111111111;
    16'd1: result = 10'b1011111111;
    16'd2: result = 10'b1101111111;
    16'd3: result = 10'b1110111111;
    16'd4: result = 10'b1111011111;
    16'd5: result = 10'b1111101111;
    16'd6: result = 10'b1111110111;
    16'd7: result = 10'b1111111011;
    16'd8: result = 10'b1111111101;
    16'd9: result = 10'b1111111110;
    default result = 'bx;
endcase
```

The case expression given in parentheses shall be evaluated exactly once and before any of the case item expressions. The case item expressions shall be evaluated and compared in the exact order in which they are given. If there is a default case item, it is ignored during this linear search. During the linear search, if one of the `case` item expressions matches the case expression given in parentheses, then the statement associated with that case item shall be executed, and the linear search shall terminate. If all comparisons fail and the default item is given, then the default item statement shall be executed. If the default statement is not given and all of the comparisons fail, then none of the case item statements shall be executed.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Apart from syntax, the `case` statement differs from the multiway if-else-if construct in two important ways:

a) The conditional expressions in the if-else-if construct are more general than comparing one expression with several others, as in the case statement.
b) The case statement provides a definitive result when there are `x` and `z` values in an expression.

In a case expression comparison, the comparison only succeeds when each bit matches exactly with respect to the values `0`, `1`, `x`, and `z`. As a consequence, care is needed in specifying the expressions in the `case` statement. The bit length of all the expressions shall be equal so that exact bitwise matching can be performed. The length of all the `case` item expressions, as well as the case expression in the parentheses, shall be made equal to the length of the longest case expression and case item expression. If any of these expressions is unsigned, then all of them shall be treated as unsigned. If all of these expressions are signed, then they shall be treated as signed.

The reason for providing a case expression comparison that handles the `x` and `z` values is that it provides a mechanism for detecting such values and reducing the pessimism that can be generated by their presence.

For example:

Example 1—The following example illustrates the use of a case statement to handle `x` and `z` values properly:

```
case (select[1:2])
    2'b00: result = 0;
    2'b01: result = flaga;
    2'b0x,
    2'b0z: result = flaga ? 'bx : 0;
    2'b10: result = flagb;
    2'bx0,
    2'bz0: result = flagb ? 'bx : 0;
    default result = 'bx;
endcase
```

In this example, if `select[1]` is `0` and `flaga` is `0`, then even if the value of `select[2]` is `x` or `z`, `result` should be `0`—which is resolved by the third case.

Example 2—The following example shows another way to use a case statement to detect `x` and `z` values:

```
case (sig)
    1'bz: $display("signal is floating");
    1'bx: $display("signal is unknown");
    default: $display("signal is %b", sig);
endcase
```

9.5.1 Case statement with do-not-cares

Two other types of case statements are provided to allow handling of do-not-care conditions in the case comparisons. One of these treats high-impedance values (`z`) as do-not-cares, and the other treats both high-impedance and unknown (`x`) values as do-not-cares.

These case statements can be used in the same way as the traditional case statement, but they begin with keywords **casez** and **casex**, respectively.

Do-not-care values (`z` values for casez, `z` and `x` values for casex) in any bit of either the case expression or the case items shall be treated as do-not-care conditions during the comparison, and that bit position shall

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

not be considered. The do-not-care conditions in case expression can be used to control dynamically which bits should be compared at any time.

The syntax of literal numbers allows the use of the question mark `(?)` in place of `z` in these case statements. This provides a convenient format for specification of do-not-care bits in case statements.

For example:

Example 1—The following is an example of the casez statement. It demonstrates an instruction decode, where values of the most significant bits select which task should be called. If the most significant bit of `ir` is a `1`, then the task `instruction1` is called, regardless of the values of the other bits of `ir`.

```
reg [7:0] ir;

casez (ir)
    8'b1???????: instruction1(ir);
    8'b01??????: instruction2(ir);
    8'b00010???: instruction3(ir);
    8'b000001??: instruction4(ir);
endcase
```

Example 2—The following is an example of the casex statement. It demonstrates an extreme case of how do-not-care conditions can be dynamically controlled during simulation. In this case, if `r = 8'b01100110`, then the task `stat2` is called.

```
reg [7:0] r, mask;

mask = 8'bx0x0x0x0;
casex (r ^ mask)
    8'b001100xx: stat1;
    8'b1100xx00: stat2;
    8'b00xx0011: stat3;
    8'bxx010100: stat4;
endcase
```

9.5.2 Constant expression in case statement

A constant expression can be used for case expression. The value of the constant expression shall be compared against case item expressions.

For example:

The following example demonstrates the usage by modeling a 3-bit priority encoder:

```
reg [2:0] encode ;

case (1)
    encode[2] : $display("Select Line 2") ;
    encode[1] : $display("Select Line 1") ;
    encode[0] : $display("Select Line 0") ;
    default $display("Error: One of the bits expected ON");
endcase
```

In this example, the case expression is a constant expression (1). The case items are expressions (bit-selects) and are compared against the constant expression for a match.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

9.6 Looping statements

There are four types of looping statements. These statements provide a means of controlling the execution of a statement zero, one, or more times.

forever Continuously executes a statement.

repeat Executes a statement a fixed number of times. If the expression evaluates to unknown or high impedance, it shall be treated as zero, and no statement shall be executed.

while Executes a statement until an expression becomes false. If the expression starts out false, the statement shall not be executed at all.

for Controls execution of its associated statement(s) by a three-step process, as follows:

a) Executes an assignment normally used to initialize a variable that controls the number of loops executed.
b) Evaluates an expression. If the result is zero, the for loop shall exit. If it is not zero, the for loop shall execute its associated statement(s) and then perform step c). If the expression evaluates to an unknown or high-impedance value, it shall be treated as zero.
c) Executes an assignment normally used to modify the value of the loop-control variable, then repeats step b).

Syntax 9-7 shows the syntax for various looping statements.

```
loop_statement ::= (From A.6.8)
        forever statement
      | repeat ( expression ) statement
      | while ( expression ) statement
      | for ( variable_assignment ; expression ; variable_assignment )
          statement
```

Syntax 9-7—Syntax for looping statements

The rest of this subclause presents examples for three of the looping statements. The forever loop should only be used in conjunction with the timing controls or the disable statement; therefore, this example is presented in 9.7.2.

For example:

Example 1—Repeat statement: In the following example of a repeat loop, add and shift operators implement a multiplier:

```
parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;

begin : mult
    reg [longsize:1] shift_opa, shift_opb;
    shift_opa = opa;
    shift_opb = opb;
    result = 0;
    repeat (size) begin
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
        if (shift_opb[1])
            result = result + shift_opa;
        shift_opa = shift_opa << 1;
        shift_opb = shift_opb >> 1;
    end
end
```

Example 2—While statement: The following example counts the number of logic `1` values in `rega`:

```
begin : count1s
    reg [7:0] tempreg;
    count = 0;
    tempreg = rega;
    while (tempreg) begin
        if (tempreg[0])
            count = count + 1;
        tempreg = tempreg >> 1;
    end
end
```

Example 3—For statement: The for statement accomplishes the same results as the following pseudo-code that is based on the while loop:

```
begin
    initial_assignment;
    while (condition) begin
            statement
            step_assignment;
        end
end
```

The for loop implements this logic while using only two lines, as shown in the pseudo-code below:

```
for (initial_assignment; condition; step_assignment)
    statement
```

9.7 Procedural timing controls

The Verilog HDL has two types of explicit timing control over when procedural statements can occur. The first type is a *delay control*, in which an expression specifies the time duration between initially encountering the statement and when the statement actually executes. The delay expression can be a dynamic function of the state of the circuit, but it can be a simple number that separates statement executions in time. The delay control is an important feature when specifying stimulus waveform descriptions. It is described in 9.7.1 and 9.7.7.

The second type of timing control is the *event expression*, which allows statement execution to be delayed until the occurrence of some simulation event occurring in a procedure executing concurrently with this procedure. A simulation event can be a change of value on a net or variable (an *implicit event*) or the occurrence of an explicitly named event that is triggered from other procedures (an *explicit event*). Most often, an event control is a positive or negative edge on a clock signal. Event control is discussed in 9.7.2 through 9.7.7.

The procedural statements encountered so far all execute without advancing simulation time. Simulation time can advance by one of the following three methods:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

— A **`delay`** control, which is introduced by the symbol #
— An **`event`** control, which is introduced by the symbol @
— The **`wait`** statement, which operates like a combination of the event control and the while loop

Syntax 9-8 describes timing control in procedural statements.

```
delay_control ::= (From A.6.5)
        # delay_value
      | # ( mintypmax_expression )
event_control ::=
        @ hierarchical_event_identifier
      | @ ( event_expression )
      | @*
      | @ (*)
procedural_timing_control ::=
        delay_control
      | event_control
procedural_timing_control_statement ::=
      | procedural_timing_control  statement_or_null
```

Syntax 9-8—Syntax for procedural timing control

The gate and net delays also advance simulation time, as discussed in Clause 6. The three procedural timing control methods are discussed in 9.7.1 through 9.7.7.

9.7.1 Delay control

A procedural statement following the delay control shall be delayed in its execution with respect to the procedural statement preceding the delay control by the specified delay. If the delay expression evaluates to an unknown or high-impedance value, it shall be interpreted as zero delay. If the delay expression evaluates to a negative value, it shall be interpreted as a twos-complement unsigned integer of the same size as a time variable. Specify parameters are permitted in the delay expression. They can be overridden by SDF annotation, in which case the expression is reevaluated.

For example:

Example 1—The following example delays the execution of the assignment by 10 time units:

```
#10 rega = regb;
```

Example 2—The next three examples provide an expression following the number sign (#). Execution of the assignment is delayed by the amount of simulation time specified by the value of the expression.

```
#d rega = regb;            // d is defined as a parameter
#((d+e)/2) rega = regb;    // delay is average of d and e
#regr regr = regr + 1;     // delay is the value in regr
```

9.7.2 Event control

The execution of a procedural statement can be synchronized with a value change on a net or variable or the occurrence of a declared event. The value changes on nets and variable can be used as events to trigger the execution of a statement. This is known as *detecting an implicit event*. The event can also be based on

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

the direction of the change, that is, toward the value 1 (**posedge**) or toward the value 0 (**negedge**). The behavior of posedge and negedge events is shown in Table 9-1 and can be described as follows:

— A *negedge* shall be detected on the transition from `1` to `x`, `z`, or `0`, and from `x` or `z` to `0`
— A *posedge* shall be detected on the transition from `0` to `x`, `z`, or `1`, and from `x` or `z` to `1`

Table 9-1—Detecting posedge and negedge

From	To			
	0	**1**	**x**	**z**
0	No edge	posedge	posedge	posedge
1	negedge	No edge	negedge	negedge
x	negedge	posedge	No edge	No edge
z	negedge	posedge	No edge	No edge

An implicit event shall be detected on any change in the value of the expression. An edge event shall be detected only on the least significant bit of the expression. A change of value in any operand of the expression without a change in the result of the expression shall not be detected as an event.

For example:

The following example shows illustrations of edge-controlled statements:

```
@r rega = regb; // controlled by any value change in the reg r
@(posedge clock) rega = regb; // controlled by posedge on clock
forever @(negedge clock) rega = regb; // controlled by negative edge
```

9.7.3 Named events

A new data type, in addition to nets and variables, called *event* can be declared. An identifier declared as an event data type is called a *named event*. A named event can be triggered explicitly. It can be used in an event expression to control the execution of procedural statements in the same manner as event controls described in 9.7.2. Named events can be made to occur from a procedure. This allows control over the enabling of multiple actions in other procedures.

An event name shall be declared explicitly before it is used. Syntax 9-9 gives the syntax for declaring events.

```
event_declaration ::= (From A.2.1.3)
        event list_of_event_identifiers ;
list_of_event_identifiers ::= (From A.2.3)
        event_identifier { dimension }
            { , event_identifier { dimension } }
dimension ::= (From A.2.5)
        [ dimension_constant_expression : dimension_constant_expression ]
```

Syntax 9-9—Syntax for event declaration

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

An event shall not hold any data. The following are the characteristics of a named event:

— It can be made to occur at any particular time.
— It has no time duration.
— Its occurrence can be recognized by using the event control syntax described in 9.7.2.

A declared event is made to occur by the activation of an event triggering statement with the syntax given in Syntax 9-10. An event is not made to occur by changing the index of an event array in an event control expression.

```
event_trigger ::= (From A.6.5)
    -> hierarchical_event_identifier { [ expression ] } ;
```

Syntax 9-10—Syntax for event trigger

An event-controlled statement (for example, `@trig rega = regb;`) shall cause simulation of its containing procedure to wait until some other procedure executes the appropriate event-triggering statement (for example, `-> trig`).

Named events and event control give a powerful and efficient means of describing the communication between, and synchronization of, two or more concurrently active processes. A basic example of this is a small waveform clock generator that synchronizes control of a synchronous circuit by signalling the occurrence of an explicit event periodically while the circuit waits for the event to occur.

9.7.4 Event or operator

The logical or of any number of events can be expressed so that the occurrence of any one of the events triggers the execution of the procedural statement that follows it. The keyword **or** or a comma character (**,**) is used as an event logical or operator. A combination of these can be used in the same event expression. Comma-separated sensitivity lists shall be synonymous to **or**-separated sensitivity lists.

For example:

The next two examples show the logical or of two and three events, respectively:

```
@(trig or enable) rega = regb; // controlled by trig or enable
@(posedge clk_a or posedge clk_b or trig) rega = regb;
```

The following examples show the use of the comma (`,`) as an event logical or operator:

```
always @(a, b, c, d, e)
always @(posedge clk, negedge rstn)
always @(a or b, c, d or e)
```

9.7.5 Implicit event_expression list

The `event_expression` list of an event control is a common source of bugs in register transfer level (RTL) simulations. Users tend to forget to add some of the nets or variables read in the timing control statement. This is often found when comparing RTL and gate-level versions of a design. The implicit `event_expression`, **@***, is a convenient shorthand that eliminates these problems by adding all nets and variables that are read by the statement (which can be a statement group) of a `procedural_timing_control_statement` to the `event_expression`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

All net and variable identifiers that appear in the statement will be automatically added to the event expression with these exceptions:

— Identifiers that only appear in wait or event expressions.
— Identifiers that only appear as a *hierarchical_variable_identifier* in the `variable_lvalue` of the left-hand side of assignments.

Nets and variables that appear on the right-hand side of assignments, in function and task calls, in case and conditional expressions, as an index variable on the left-hand side of assignments, or as variables in case item expressions shall all be included by these rules.

For example:

Example 1

```
always @(*)  // equivalent to @(a or b or c or d or f)
   y = (a & b) | (c & d) | myfunction(f);
```

Example 2

```
always @* begin // equivalent to @(a or b or c or d or tmp1 or tmp2)
   tmp1 = a & b;
   tmp2 = c & d;
   y = tmp1 | tmp2;
end
```

Example 3

```
always @* begin  // equivalent to @(b)
   @(i) kid = b;  // i is not added to @*
end
```

Example 4

```
always @* begin  // equivalent to @(a or b or c or d)
   x = a ^ b;
   @*             // equivalent to @(c or d)
      x = c ^ d;
end
```

Example 5

```
always @* begin  // same as @(a or en)
   y = 8'hff;
   y[a] = !en;
end
```

Example 6

```
always @* begin  // same as @(state or go or ws)
   next = 4'b0;
   case (1'b1)
      state[IDLE]: if (go)  next[READ] = 1'b1;
                   else     next[IDLE] = 1'b1;
      state[READ]:          next[DLY ] = 1'b1;
      state[DLY ]: if (!ws) next[DONE] = 1'b1;
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
                    else        next[READ] = 1'b1;
            state[DONE]:        next[IDLE] = 1'b1;
        endcase
    end
```

9.7.6 Level-sensitive event control

The execution of a procedural statement can also be delayed until a condition becomes true. This is accomplished using the *wait* statement, which is a special form of event control. The nature of the wait statement is level-sensitive, as opposed to basic event control (specified by the `@` character), which is edge-sensitive.

The wait statement shall evaluate a condition; and, if it is false, the procedural statements following the wait statement shall remain blocked until that condition becomes true before continuing. The wait statement has the form given in Syntax 9-11.

```
wait_statement ::= (From A.6.5)
    wait ( expression ) statement_or_null
```

Syntax 9-11—Syntax for wait statement

For example:

The following example shows the use of the wait statement to accomplish level-sensitive event control:

```
begin
    wait (!enable) #10 a = b;
    #10 c = d;
end
```

If the value of `enable` is `1` when the block is entered, the wait statement will delay the evaluation of the next statement (`#10 a = b;`) until the value of `enable` changes to `0`. If `enable` is already `0` when the `begin-end` block is entered, then the assignment "`a = b;`" is evaluated after a delay of 10 and no additional delay occurs.

9.7.7 Intra-assignment timing controls

The delay and event control constructs previously described precede a statement and delay its execution. In contrast, the *intra-assignment delay and event controls* are contained within an assignment statement and modify the flow of activity in a different way. This subclause describes the purpose of intra-assignment timing controls and the repeat timing control that can be used in intra-assignment delays.

An intra-assignment delay or event control shall delay the assignment of the new value to the left-hand side, but the right-hand expression shall be evaluated before the delay, instead of after the delay. The syntax for intra-assignment delay and event control is given in Syntax 9-12.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
blocking_assignment ::= (From A.6.2)
      variable_lvalue = [ delay_or_event_control ] expression
nonblocking_assignment ::=
      variable_lvalue <= [ delay_or_event_control ] expression
delay_control ::= (From A.6.5)
       # delay_value
      | # ( mintypmax_expression )
delay_or_event_control ::=
       delay_control
      | event_control
      | repeat ( expression ) event_control
event_control ::=
       @ hierarchical_event_identifier
      | @ ( event_expression )
      | @*
      | @ (*)
event_expression ::=
       expression
      | posedge expression
      | negedge expression
      | event_expression or event_expression
      | event_expression , event_expression
```

Syntax 9-12—Syntax for intra-assignment delay and event control

The intra-assignment delay and event control can be applied to both blocking assignments and nonblocking assignments. The *repeat* event control shall specify an intra-assignment delay of a specified number of occurrences of an event. If the repeat count literal, or signed reg holding the repeat count, is less than or equal to `0` at the time of evaluation, the assignment occurs as if there is no repeat construct.

For example:

```
repeat (-3) @ (event_expression)
       // will not execute event_expression.

repeat (a) @ (event_expression)
    // if a is assigned -3, it will execute the event_expression
    // if a is declared as an unsigned reg, but not if a is signed
```

This construct is convenient when events have to be synchronized with counts of clock signals.

For example:

Table 9-2 illustrates the philosophy of intra-assignment timing controls by showing the code that could accomplish the same timing effect without using intra-assignment.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 9-2—Intra-assignment timing control equivalence

Intra-assignment timing control	
With intra-assignment construct	**Without intra-assignment construct**

`a = #5 b;`	**begin**
	`  temp = b;`
	`  #5 a = temp;`
	end

`a = @(`**posedge** `clk) b;`	**begin**
	`  temp = b;`
	`  @(`**posedge** `clk) a = temp;`
	end

`a =` **repeat**`(3)`
`  @(`**posedge** `clk) b;`
	begin
	`  temp = b;`
	`  @(`**posedge** `clk);`
	`  @(`**posedge** `clk);`
	`  @(`**posedge** `clk) a = temp;`
	end

The next three examples use the fork-join behavioral construct. All statements between the keywords **fork** and **join** execute concurrently. This construct is described in more detail in 9.8.2.

The following example shows a race condition that could be prevented by using intra-assignment timing control:

```
fork
        #5 a = b;
        #5 b = a;
join
```

The code in this example samples and sets the values of both `a` and `b` at the same simulation time, thereby creating a race condition. The intra-assignment form of timing control used in the next example prevents this race condition.

```
fork                    // data swap
        a = #5 b;
        b = #5 a;
join
```

Intra-assignment timing control works because the intra-assignment delay causes the values of `a` and `b` to be evaluated before the delay and causes the assignments to be made after the delay. Some existing tools that implement intra-assignment timing control use temporary storage in evaluating each expression on the right-hand side.

Intra-assignment waiting for events is also effective. In the following example, the right-hand expressions are evaluated when the assignment statements are encountered, but the assignments are delayed until the rising edge of the clock signal:

```
fork                                // data shift
        a = @(posedge clk) b;
        b = @(posedge clk) c;
join
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The following is an example of a repeat event control as the intra-assignment delay of a nonblocking assignment:

```
a <= repeat(5) @(posedge clk) data;
```

Figure 9-1 illustrates the activities that result from this `repeat` event control.

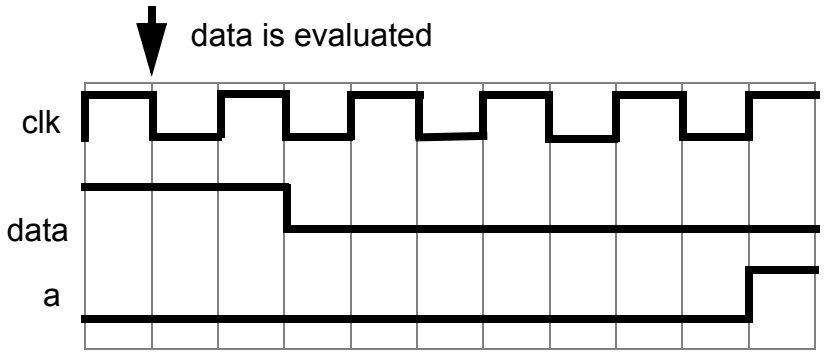


Figure 9-1—Repeat event control utilizing a clock edge

In this example, the value of `data` is evaluated when the assignment is encountered. After five occurrences of **posedge** `clk`, `a` is assigned the value of `data`.

The following is an example of a repeat event control as the intra-assignment delay of a procedural assignment:

```
a = repeat(num) @(clk) data;
```

In this example, the value of `data` is evaluated when the assignment is encountered. After the number of transitions of `clk` equals the value of `num`, `a` is assigned the value of `data`.

The following is an example of a repeat event control with expressions containing operations to specify both the number of event occurrences and the event that is counted:

```
a <= repeat(a+b) @(posedge phi1 or negedge phi2) data;
```

In this example, the value of `data` is evaluated when the assignment is encountered. After the sum of the positive edges of `phi1` and the negative edges of `phi2` equals the sum of `a` and `b`, `a` is assigned the value of `data`. Even if **posedge** `phi1` and **negedge** `phi2` occurred at the same simulation time, each will be detected separately.

9.8 Block statements

The *block statements* are a means of grouping statements together so that they act syntactically like a single statement. There are two types of blocks in the Verilog HDL:

— *Sequential block*, also called *begin-end block*
— *Parallel block*, also called *fork-join block*

The sequential block shall be delimited by the keywords **begin** and **end**. The procedural statements in sequential block shall be executed sequentially in the given order.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The parallel block shall be delimited by the keywords **fork** and **join**. The procedural statements in parallel block shall be executed concurrently.

9.8.1 Sequential blocks

A sequential block shall have the following characteristics:

— Statements shall be executed in sequence, one after another.
— Delay values for each statement shall be treated relative to the simulation time of the execution of the previous statement.
— Control shall pass out of the block after the last statement executes.

Syntax 9-13 gives the formal syntax for a sequential block.

```
seq_block ::= (From A.6.3)
        begin [ : block_identifier
            { block_item_declaration } ] { statement } end
block_item_declaration ::= (From A.2.8)
          { attribute_instance } reg [ signed ] [ range ] list_of_block_variable_identifiers ;
        | { attribute_instance } integer list_of_block_variable_identifiers ;
        | { attribute_instance } time list_of_block_variable_identifiers ;
        | { attribute_instance } real list_of_block_real_identifiers ;
        | { attribute_instance } realtime list_of_block_real_identifiers ;
        | { attribute_instance } event_declaration
        | { attribute_instance } local_parameter_declaration ;
        | { attribute_instance } parameter_declaration ;
```

Syntax 9-13—Syntax for sequential block

For example:

Example 1—A sequential block enables the following two assignments to have a deterministic result:

```
begin
    areg = breg;
    creg = areg;     // creg stores the value of breg
end
```

The first assignment is performed, and `areg` is updated before control passes to the second assignment.

Example 2—Delay control can be used in a sequential block to separate the two assignments in time.

```
begin
    areg = breg;
    @(posedge clock) creg = areg;  // assignment delayed until
end                                // posedge on clock
```

Example 3—The following example shows how the combination of the sequential block and delay control can be used to specify a time-sequenced waveform:

```
parameter d = 50;     // d declared as a parameter and
reg [7:0] r;          // r declared as an 8-bit reg

begin    // a waveform controlled by sequential delay
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
    #d r = 'h35;
    #d r = 'hE2;
    #d r = 'h00;
    #d r = 'hF7;
    #d -> end_wave; //trigger an event called end_wave
end
```

9.8.2 Parallel blocks

A *parallel block* shall have the following characteristics:

— Statements shall execute concurrently.
— Delay values for each statement shall be considered relative to the simulation time of entering the block.
— Delay control can be used to provide time-ordering for assignments.
— Control shall pass out of the block when the last time-ordered statement executes.

Syntax 9-14 gives the formal syntax for a parallel block.

```
par_block ::= (From A.6.3)
      fork [ : block_identifier
         { block_item_declaration } ] { statement } join
block_item_declaration ::= (From A.2.8)
        { attribute_instance } reg [ signed ] [ range ] list_of_block_variable_identifiers ;
      | { attribute_instance } integer list_of_block_variable_identifiers ;
      | { attribute_instance } time list_of_block_variable_identifiers ;
      | { attribute_instance } real list_of_block_real_identifiers ;
      | { attribute_instance } realtime list_of_block_real_identifiers ;
      | { attribute_instance } event_declaration
      | { attribute_instance } local_parameter_declaration ;
      | { attribute_instance } parameter_declaration ;
```

Syntax 9-14—Syntax for parallel block

The timing controls in a fork-join block do not have to be ordered sequentially in time.

For example:

The following example codes the waveform description shown in Example 3 of 9.8.1 by using a parallel block instead of a sequential block. The waveform produced on the reg is exactly the same for both implementations.

```
fork
    #50 r = 'h35;
    #100 r = 'hE2;
    #150 r = 'h00;
    #200 r = 'hF7;
    #250 -> end_wave;
join
```

9.8.3 Block names

Both sequential and parallel blocks can be named by adding `: name_of_block` after the keywords **begin** or **fork**. The naming of blocks serves several purposes:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

— It allows local variables, parameters, and named events to be declared for the block.
— It allows the block to be referenced in statements such as the disable statement (see 10.3).

All variables shall be static; that is, a unique location exists for all variables, and leaving or entering blocks shall not affect the values stored in them.

The block names give a means of uniquely identifying all variables at any simulation time.

9.8.4 Start and finish times

Both sequential and parallel blocks have the notion of a start and finish time. For sequential blocks, the start time is when the first statement is executed, and the finish time is when the last statement has been executed. For parallel blocks, the start time is the same for all the statements, and the finish time is when the last time-ordered statement has been executed.

Sequential and parallel blocks can be embedded within each other, allowing complex control structures to be expressed easily and with a high degree of structure. When blocks are embedded within each other, the timing of when a block starts and finishes is important. Execution shall not continue to the statement following a block until the finish time for the block has been reached, that is, until the block has completely finished executing.

For example:

Example 1—The following example shows the statements from the example in 9.8.2 written in the reverse order and still producing the same waveform.

```
fork
    #250 -> end_wave;
    #200 r = 'hF7;
    #150 r = 'h00;
    #100 r = 'hE2;
    #50 r = 'h35;
join
```

Example 2—When an assignment is to be made after two separate events have occurred, known as the *joining of events*, a `fork-join` block can be useful.

```
begin
    fork
        @Aevent;
        @Bevent;
    join
    areg = breg;
end
```

The two events can occur in any order (or even at the same simulation time), the `fork-join` block will complete, and the assignment will be made. In contrast, if the `fork-join` block was a `begin-end` block and the `Bevent` occurred before the `Aevent`, then the block would be waiting for the next `Bevent`.

Example 3—This example shows two sequential blocks, each of which will execute when its controlling event occurs. Because the event controls are within a `fork-join` block, they execute in parallel, and the sequential blocks can, therefore, also execute in parallel.

```
fork
    @enable_a
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
        begin
            #ta wa = 0;
            #ta wa = 1;
            #ta wa = 0;
        end
    @enable_b
        begin
            #tb wb = 1;
            #tb wb = 0;
            #tb wb = 1;
        end
join
```

9.9 Structured procedures

All procedures in the Verilog HDL are specified within one of the following four statements:

— *initial construct*
— *always construct*
— Task
— Function

The initial and always constructs are enabled at the beginning of a simulation. The initial construct shall execute only once, and its activity shall cease when the statement has finished. In contrast, the always construct shall execute repeatedly. Its activity shall cease only when the simulation is terminated. There shall be no implied order of execution between initial and always constructs. The initial constructs need not be scheduled and executed before the always constructs. There shall be no limit to the number of initial and always constructs that can be defined in a module.

Tasks and functions are procedures that are enabled from one or more places in other procedures. Tasks and functions are described in Clause 10.

9.9.1 Initial construct

The syntax for the initial construct is given in Syntax 9-15.

```
initial_construct ::= (From A.6.2)
    initial statement
```

Syntax 9-15—Syntax for initial construct

For example:

The following example illustrates use of the initial construct for initialization of variables at the start of simulation.

```
initial begin
    areg = 0; // initialize a reg
    for (index = 0; index < size; index = index + 1)
        memory[index] = 0; //initialize memory word
end
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Another typical usage of the initial construct is specification of waveform descriptions that execute once to provide stimulus to the main part of the circuit being simulated.

```
initial begin
    inputs = 'b000000;     // initialize at time zero
    #10 inputs = 'b011001; // first pattern
    #10 inputs = 'b011011; // second pattern
    #10 inputs = 'b011000; // third pattern
    #10 inputs = 'b001000; // last pattern
end
```

9.9.2 Always construct

The always construct repeats continuously throughout the duration of the simulation. Syntax 9-16 shows the syntax for the always construct.

always_construct ::= *(From A.6.2)*
always statement

Syntax 9-16—Syntax for always construct

The always construct, because of its looping nature, is only useful when used in conjunction with some form of timing control. If an always construct has no control for simulation time to advance, it will create a simulation deadlock condition.

The following code, for example, creates a zero-delay infinite loop:

```
always areg = ~areg;
```

Providing a timing control to the above code creates a potentially useful description as shown in the following:

```
always #half_period areg = ~areg;
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10. Tasks and functions

Tasks and functions provide the ability to execute common procedures from several different places in a description. They also provide a means of breaking up large procedures into smaller ones to make it easier to read and debug the source descriptions. This clause discusses the differences between tasks and functions, describes how to define and invoke tasks and functions, and presents examples of each.

10.1 Distinctions between tasks and functions

The following rules distinguish tasks from functions:

— A function shall execute in one simulation time unit; a task can contain time-controlling statements.
— A function cannot enable a task; a task can enable other tasks and functions.
— A function shall have at least one **input** type argument and shall not have an **output** or **inout** type argument; a task can have zero or more arguments of any type.
— A function shall return a single value; a task shall not return a value.

The purpose of a function is to respond to an input value by returning a single value. A task can support multiple goals and can calculate multiple result values. However, only the **output** or **inout** type arguments pass result values back from the invocation of a task. A function is used as an operand in an expression; the value of that operand is the value returned by the function.

For example:

Either a task or a function can be defined to switch bytes in a 16-bit word. The task would return the switched word in an output argument; therefore, the source code to enable a task called `switch_bytes` could look like the following example:

```
switch_bytes (old_word, new_word);
```

The task `switch_bytes` would take the bytes in `old_word`, reverse their order, and place the reversed bytes in `new_word`.

A word-switching function would return the switched word as the return value of the function. Thus, the function call for the function `switch_bytes` could look like the following example:

```
new_word = switch_bytes (old_word);
```

10.2 Tasks and task enabling

A task shall be enabled from a statement that defines the argument values to be passed to the task and the variables that receive the results. Control shall be passed back to the enabling process after the task has completed. Thus, if a task has timing controls inside it, then the time of enabling a task can be different from the time at which the control is returned. A task can enable other tasks, which in turn can enable still other tasks—with no limit on the number of tasks enabled. Regardless of how many tasks have been enabled, control shall not return until all enabled tasks have completed.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

10.2.1 Task declarations

The syntax for defining tasks is given in Syntax 10-1.

```
task_declaration ::= (From A.2.7)
        task [ automatic ] task_identifier ;
          { task_item_declaration }
          statement_or_null
        endtask
      | task [ automatic ] task_identifier ( [ task_port_list ] ) ;
          { block_item_declaration }
          statement_or_null
        endtask
task_item_declaration ::=
        block_item_declaration
      | { attribute_instance } tf_ input_declaration ;
      | { attribute_instance } tf_output_declaration ;
      | { attribute_instance } tf_inout_declaration ;
task_port_list ::=
      task_port_item { , task_port_item }
task_port_item ::=
        { attribute_instance } tf_input_declaration
      | { attribute_instance } tf_output_declaration
      | { attribute_instance } tf_inout_declaration
tf_input_declaration ::=
        input [ reg ] [ signed ] [ range ] list_of_port_identifiers
      | input task_port_type list_of_port_identifiers
tf_output_declaration ::=
        output [ reg ] [ signed ] [ range ] list_of_port_identifiers
      | output task_port_type list_of_port_identifiers
tf_inout_declaration ::=
        inout [ reg ] [ signed ] [ range ] list_of_port_identifiers
      | inout task_port_type list_of_port_identifiers
task_port_type ::=
        integer | real | realtime | time
block_item_declaration ::= (From A.2.8)
        { attribute_instance } reg [ signed ] [ range ] list_of_block_variable_identifiers ;
      | { attribute_instance } integer list_of_block_variable_identifiers ;
      | { attribute_instance } time list_of_block_variable_identifiers ;
      | { attribute_instance } real list_of_block_real_identifiers ;
      | { attribute_instance } realtime list_of_block_real_identifiers ;
      | { attribute_instance } event_declaration
      | { attribute_instance } local_parameter_declaration ;
      | { attribute_instance } parameter_declaration ;
list_of_block_variable_identifiers ::=
      block_variable_type { , block_variable_type }
list_of_block_real_identifiers ::=
      block_real_type { , block_real_type }
block_variable_type ::=
      variable_identifier { dimension }
block_real_type ::=
      real_identifier { dimension }
```

Syntax 10-1—Syntax for task declaration

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

There are two alternate task declaration syntaxes.

The first syntax shall begin with the keyword **task**, followed by the optional keyword **automatic**, followed by a name for the task and a semicolon, and ending with the keyword **endtask**. The keyword **automatic** declares an automatic task that is reentrant, with all the task declarations allocated dynamically for each concurrent task entry. Task item declarations can specify the following:

— Input arguments
— Output arguments
— Inout arguments
— All data types that can be declared in a procedural block

The second syntax shall begin with the keyword **task**, followed by a name for the task and a parenthesis-enclosed *task_port_list*. The *task_port_list* shall consist of zero or more comma separated *task_port_items*. There shall be a semicolon after the close parenthesis. The task body shall follow and then the keyword **endtask**.

In both syntaxes, the port declarations shall have the same syntax as defined by the *tf_input_declaration*, *tf_output_declaration*, and *tf_inout_declaration*, as detailed in Syntax 10-1 above.

Tasks without the optional keyword **automatic** are static tasks, with all declared items being statically allocated. These items shall be shared across all uses of the task executing concurrently. Task with the optional keyword **automatic** are automatic tasks. All items declared inside automatic tasks are allocated dynamically for each invocation. Automatic task items cannot be accessed by hierarchical references. Automatic tasks can be invoked through use of their hierarchical name.

10.2.2 Task enabling and argument passing

The task-enabling statement shall pass arguments as a comma-separated list of expressions enclosed in parentheses. The formal syntax of the task-enabling statement is given in Syntax 10-2.

```
task_enable ::= (From A.6.9)
    hierarchical_task_identifier [ ( expression { , expression } ) ] ;
```

Syntax 10-2—Syntax for task-enabling statement

If the task definition has no arguments, a list of arguments shall not be provided in the task-enabling statement. Otherwise, there shall be an ordered list of expressions that matches the length and order of the list of arguments in the task definition. A null expression shall not be used as an argument in a task-enabling statement.

If an argument in the task is declared as an **input**, then the corresponding expression can be any expression. The order of evaluation of the expressions in the argument list is undefined. If the argument is declared as an **output** or an **inout**, then the expression shall be restricted to an expression that is valid on the left-hand side of a procedural assignment (see 9.2). The following items satisfy this requirement:

— **reg**, **integer**, **real**, **realtime**, and **time** variables
— Memory references
— Concatenations of **reg**, **integer**, and **time** variables
— Concatenations of memory references
— Bit-selects and part-selects of **reg**, **integer**, and **time** variables

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The execution of the task-enabling statement shall pass input values from the expressions listed in the enabling statement to the arguments specified within the task. Execution of the return from the task shall pass values from the task **output** and **inout** type arguments to the corresponding variables in the task-enabling statement. All arguments to the task shall be passed by value rather than by reference (that is, a *pointer* to the value).

For example:

Example 1—The following example illustrates the basic structure of a task definition with five arguments:

```
task my_task;
input a, b;
inout c;
output d, e;
begin
    . . .     // statements that perform the work of the task
    . . .
    c = foo1;  // the assignments that initialize result regs
    d = foo2;
    e = foo3;
end
endtask
```

Or using the second form of a task declaration, the task could be defined as follows:

```
task my_task (input a, b, inout c, output d, e);
begin
    . . .     // statements that perform the work of the task
    . . .
    c = foo1;  // the assignments that initialize result regs
    d = foo2;
    e = foo3;
end
endtask
```

The following statement enables the task:

```
my_task (v, w, x, y, z);
```

The task-enabling arguments (`v`, `w`, `x`, `y`, and `z`) correspond to the arguments (`a`, `b`, `c`, `d`, and `e`) defined by the task. At task-enabling time, the **input** and **inout** type arguments (`a`, `b`, and `c`) receive the values passed in `v`, `w`, and `x`. Thus, execution of the task-enabling call effectively causes the following assignments:

```
a = v;
b = w;
c = x;
```

As part of the processing of the task, the task definition for `my_task` shall place the computed result values into `c`, `d`, and `e`. When the task completes, the following assignments to return the computed values to the calling process are performed:

```
x = c;
y = d;
z = e;
```

Example 2—The following example illustrates the use of tasks by describing a traffic light sequencer:

```
module traffic_lights;
reg clock, red, amber, green;
```


Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
parameter  on = 1, off = 0, red_tics = 350,
           amber_tics = 30, green_tics = 200;

// initialize colors.
initial red = off;
initial amber = off;
initial green = off;

always begin                          // sequence to control the lights.
    red = on;                         // turn red light on
    light(red, red_tics);             // and wait.
    green = on;                       // turn green light on
    light(green, green_tics);         // and wait.
    amber = on;                       // turn amber light on
    light(amber, amber_tics);         // and wait.
end

// task to wait for 'tics' positive edge clocks
// before turning 'color' light off.
task light;
output color;
input [31:0] tics;
begin
    repeat (tics) @ (posedge clock);
    color = off;                      // turn light off.
end
endtask

always begin                          // waveform for the clock.
    #100 clock = 0;
    #100 clock = 1;
end
endmodule // traffic_lights.
```

10.2.3 Task memory usage and concurrent activation

A task may be enabled more than once concurrently. All variables of an automatic task shall be replicated on each concurrent task invocation to store state specific to that invocation. All variables of a static task shall be static in that there shall be a single variable corresponding to each declared local variable in a module instance, regardless of the number of concurrent activations of the task. However, static tasks in different instances of a module shall have separate storage from each other.

Variables declared in static tasks, including **input**, **output**, and **inout** type arguments, shall retain their values between invocations. They shall be initialized to the default initialization value as described in 4.2.2.

Variables declared in automatic tasks, including **output** type arguments, shall be initialized to the default initialization value whenever execution enters their scope. **input** and **inout** type arguments shall be initialized to the values passed from the expressions corresponding to these arguments listed in the task-enabling statements.

Because variables declared in automatic tasks are deallocated at the end of the task invocation, they shall not be used in certain constructs that might refer to them after that point:

— They shall not be assigned values using nonblocking assignments or procedural continuous assignments.
— They shall not be referenced by procedural continuous assignments or procedural force statements.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

— They shall not be referenced in intra-assignment event controls of nonblocking assignments.
— They shall not be traced with system tasks such as **$monitor** and **$dumpvars**.

10.3 Disabling of named blocks and tasks

The *disable* statement provides the ability to terminate the activity associated with concurrently active procedures, while maintaining the structured nature of Verilog HDL procedural descriptions. The disable statement gives a mechanism for terminating a task before it executes all its statements, breaking from a looping statement, or skipping statements in order to continue with another iteration of a looping statement. It is useful for handling exception conditions such as hardware interrupts and global resets.

The disable statement has the syntax form shown in Syntax 10-3.

```
disable_statement ::= (From A.6.5)
        disable hierarchical_task_identifier ;
      | disable hierarchical_block_identifier ;
```

Syntax 10-3—Syntax for disable statement

Either form of disable statement shall terminate the activity of a task or a named block. Execution shall resume at the statement following the block or following the task-enabling statement. All activities enabled within the named block or task shall be terminated as well. If task enable statements are nested (that is, one task enables another, and that one enables yet another), then disabling a task within the chain shall disable all tasks downward on the chain. If a task is enabled more than once, then disabling such a task shall disable all activations of the task.

The results of the following activities that can be initiated by a task are not specified if the task is disabled:

— Results of output and inout arguments
— Scheduled, but not executed, nonblocking assignments
— Procedural continuous assignments (**assign** and **force** statements)

The disable statement can be used within blocks and tasks to disable the particular block or task containing the disable statement. The disable statement can be used to disable named blocks within a function, but cannot be used to disable functions. In cases where a disable statement within a function disables a block or a task that called the function, the behavior is undefined. Disabling an automatic task or a block inside an automatic task proceeds as for regular tasks for all concurrent executions of the task.

For example:

Example 1—This example illustrates how a block disables itself.

```
begin : block_name
    rega = regb;
    disable block_name;
    regc = rega; // this assignment will never execute
end
```

Example 2—This example shows the disable statement being used within a named block in a manner similar to a forward *goto*. The next statement executed after the disable statement is the one following the named block.

```
begin : block_name
    ...
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
    ...
    if (a == 0)
        disable block_name;
    ...
end     // end of named block
// continue with code following named block
    ...
```

Example 3—This example shows the disable statement being used as an early return from a task. However, a task disabling itself using a disable statement is not a shorthand for the return statement found in programming languages.

```
task proc_a;
begin
    ...
    ...
    if (a == 0)
        disable proc_a; // return if true
    ...
    ...
end
endtask
```

Example 4—This example shows the disable statement being used in an equivalent way to the two statements *continue* and *break* in the C programming language. The example illustrates control code that would allow a named block to execute until a loop counter reaches `n` iterations or until the variable `a` is set to the value of `b`. The named block `break` contains the code that executes until `a == b`, at which point the `disable break;` statement terminates execution of that block. The named block `continue` contains the code that executes for each iteration of the `for` loop. Each time this code executes the `disable continue;` statement, the `continue` block terminates, and execution passes to the next iteration of the `for` loop. For each iteration of the `continue` block, a set of statements executes if `(a != 0)`. Another set of statements executes if `(a! = b)`.

```
begin : break
    for (i = 0; i < n; i = i+1) begin : continue
        @clk
            if (a == 0) // "continue" loop
                disable continue;
            statements
            statements
        @clk
            if (a == b) // "break" from loop
                disable break;
            statements
            statements
    end
end
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Example 5—This example shows the disable statement being used to disable concurrently a sequence of timing controls and the task `action` when the `reset` event occurs. The example shows a `fork-join` block within which are a named sequential block (`event_expr`) and a disable statement that waits for occurrence of the event `reset`. The sequential block and the wait for `reset` execute in parallel. The `event_expr` block waits for one occurrence of event `ev1` and three occurrences of event `trig`. When these four events have happened, plus a delay of `d` time units, the task `action` executes. When the event `reset` occurs, regardless of events within the sequential block, the `fork-join` block terminates—including the task `action`.

```
fork
    begin : event_expr
        @ev1;
        repeat (3) @trig;
        #d action (areg, breg);
    end
    @reset disable event_expr;
join
```

Example 6—The next example is a behavioral description of a retriggerable monostable. The named event `retrig` restarts the monostable time period. If `retrig` continues to occur within 250 time units, then `q` will remain at `1`.

```
always begin : monostable
    #250 q = 0;

end

always @retrig begin
    disable monostable;
    q = 1;
end
```

10.4 Functions and function calling

The purpose of a function is to return a value that is to be used in an expression. The rest of this clause explains how to define and use functions.

10.4.1 Function declarations

The syntax for defining a function is given in Syntax 10-4.

Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
function_declaration ::= (From A.2.6)
        function [ automatic ] [ function_range_or_type ]
          function_identifier ;
          function_item_declaration { function_item_declaration }
          function_statement
        endfunction
      | function [ automatic ] [ function_range_or_type ]
          function_identifier ( function_port_list ) ;
          { block_item_declaration }
          function_statement
        endfunction
function_item_declaration ::=
        block_item_declaration
      | { attribute_instance } tf_input_declaration ;
function_port_list ::=
      { attribute_instance } tf_input_declaration
        { , { attribute_instance }tf_input_declaration }
tf_input_declaration  ::=
        input [ reg ] [ signed ] [ range ] list_of_port_identifiers
      | input task_port_type list_of_port_identifiers
function_range_or_type ::=
        [ signed ] [ range ]
      | integer
      | real
      | realtime
      | time
block_item_declaration ::= (From A.2.8)
        { attribute_instance } reg [ signed ] [ range ] list_of_block_variable_identifiers ;
      | { attribute_instance } integer list_of_block_variable_identifiers ;
      | { attribute_instance } time list_of_block_variable_identifiers ;
      | { attribute_instance } real list_of_block_real_identifiers ;
      | { attribute_instance } realtime list_of_block_real_identifiers ;
      | { attribute_instance } event_declaration
      | { attribute_instance } local_parameter_declaration ;
      | { attribute_instance } parameter_declaration ;
list_of_block_variable_identifiers ::=
      block_variable_type { , block_variable_type }
list_of_block_real_identifiers ::=
      block_real_type { , block_real_type }
block_variable_type ::=
      variable_identifier { dimension }
block_real_type ::=
      real_identifier { dimension }
```

Syntax 10-4—Syntax for function declaration

A function definition shall begin with the keyword **function**, followed by the optional keyword **automatic**, followed by an optional *function_range_or_type* of the return value from the function, followed by the name of the function, followed either by a semicolon or by a function port list enclosed in parentheses and then a semicolon, and then shall end with the keyword **endfunction**.

The use of a *function_range_or_type* shall be optional. A function specified without a *function_range_or_type* defaults to a scalar for the return value. If used, *function_range_or_type* shall specify that the return value of the function is a **real**, an **integer**, a **time**, a **realtime**, or a vector (optionally signed) with a range of `[n:m]` bits.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

A function shall have at least one input declared.

The keyword **automatic** declares an automatic function that is reentrant, with all the function declarations allocated dynamically for each concurrent function call. Automatic function items cannot be accessed by hierarchical references. Automatic functions can be invoked through the use of their hierarchical name.

Function inputs shall be declared one of two ways. The first method shall have the name of the function followed by a semicolon. After the semicolon, one or more input declarations optionally mixed with block item declarations shall follow. After the function item declarations, there shall be a behavioral statement and then the **endfunction** keyword.

The second method shall have the name of the function, followed by an open parenthesis and one or more input declarations, separated by commas. After all the input declarations, there shall be a close parenthesis and a semicolon. After the semicolon, there shall be zero or more block item declarations, followed by a behavioral statement, and then the **endfunction** keyword.

For example:

The following example defines a function called `getbyte`, using a range specification:

```
function [7:0] getbyte;
input [15:0] address;
begin
   // code to extract low-order byte from addressed word
   . . .
   getbyte = result_expression;
end
endfunction
```

Or using the second form of a function declaration, the function could be defined as follows:

```
function [7:0] getbyte (input [15:0] address);
begin
   // code to extract low-order byte from addressed word
   . . .
   getbyte = result_expression;
end
endfunction
```

10.4.2 Returning a value from a function

The function definition shall implicitly declare a variable, internal to the function, with the same name as the function. This variable either defaults to a 1-bit reg or is the same type as the type specified in the function declaration. The function definition initializes the return value from the function by assigning the function result to the internal variable with the same name as the function.

It is illegal to declare another object with the same name as the function in the scope where the function is declared. Inside a function, there is an implied variable with the name of the function, which may be used in expressions within the function. It is, therefore, also illegal to declare another object with the same name as the function inside the function scope.

The following line from the example in 10.4.1 illustrates this concept:

```
getbyte = result_expression;
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

10.4.3 Calling a function

A *function call* is an operand within an expression. The function call has the syntax given in Syntax 10-5.

```
function_call ::= (From A.8.2)
    hierarchical_function_identifier{ attribute_instance } ( expression { , expression } )
```

Syntax 10-5—Syntax for function call

The order of evaluation of the arguments to a function call is undefined.

For example:

The following example creates a word by concatenating the results of two calls to the function `getbyte` (defined in 10.4.1):

```
word = control ? {getbyte(msbyte), getbyte(lsbyte)}:0;
```

10.4.4 Function rules

Functions are more limited than tasks. The following rules govern their usage:

a) A function definition shall not contain any time-controlled statements, that is, any statements containing #, @, or **wait**.
b) Functions shall not enable tasks.
c) A function definition shall contain at least one input argument.
d) A function definition shall not have any argument declared as output or inout.
e) A function shall not have any nonblocking assignments or procedural continuous assignments.
f) A function shall not have any event triggers.

For example:

This example defines a function called `factorial` that returns an integer value. The `factorial` function is called iteratively and the results are printed.

```
module tryfact;

// define the function
function automatic integer factorial;
input [31:0] operand;
integer i;
if (operand >= 2)
    factorial = factorial (operand - 1) * operand;
else
    factorial = 1;
endfunction

// test the function
integer result;
integer n;
initial begin
    for (n = 0; n <= 7; n = n+1) begin
        result = factorial(n);
        $display("%0d factorial=%0d", n, result);
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
    end
end
endmodule // tryfact
```

The simulation results are as follows:

```
0 factorial=1
1 factorial=1
2 factorial=2
3 factorial=6
4 factorial=24
5 factorial=120
6 factorial=720
7 factorial=5040
```

10.4.5 Use of constant functions

Constant function calls are used to support the building of complex calculations of values at elaboration time (see 12.8). A constant function call shall be a function invocation of a *constant function* local to the calling module where the arguments to the function are *constant expressions*. Constant functions are a subset of normal Verilog functions that shall meet the following constraints:

- They shall contain no hierarchical references.
- Any function invoked within a constant function shall be a constant function local to the current module.
- It shall be legal to call any system function that is allowed in a *constant_expression* (see Clause 5). Calls to other system functions shall be illegal.
- All system tasks within a constant function shall be ignored.
- All parameter values used within the function shall be defined before the use of the invoking constant function call (i.e., any parameter use in the evaluation of a constant function call constitutes a use of that parameter at the site of the original constant function call).
- All identifiers that are not parameters or functions shall be declared locally to the current function.
- If they use any parameter value that is affected directly or indirectly by a **defparam** statement (see 12.2.1), the result is undefined. This can produce an error or the constant function can return an indeterminate value.
- They shall not be declared inside a generate block (see 12.4).
- They shall not themselves use constant functions in any context requiring a constant expression.

Constant function calls are evaluated at elaboration time. Their execution has no effect on the initial values of the variables used either at simulation time or among multiple invocations of a function at elaboration time. In each of these cases, the variables are initialized as they would be for normal simulation.

For example:

This example defines a function called `clogb2` that returns an integer with the value of the ceiling of the log base 2.

```
module ram_model (address, write, chip_select, data);
  parameter data_width = 8;
  parameter ram_depth = 256;
  localparam addr_width = clogb2(ram_depth);
  input [addr_width - 1:0] address;
  input write, chip_select;
```


Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
  inout [data_width - 1:0] data;

  //define the clogb2 function
  function integer clogb2;
    input [31:0] value;
    begin
      value = value - 1;
      for (clogb2 = 0; value > 0; clogb2 = clogb2 + 1)
        value = value >> 1;
    end
    endfunction

    reg [data_width - 1:0] data_store[0:ram_depth - 1];
    //the rest of the ram model
```

An instance of this `ram_model` with parameters assigned is as follows:

```
ram_model #(32,421) ram_a0(a_addr,a_wr,a_cs,a_data);
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

11. Scheduling semantics

11.1 Execution of a model

The balance of the clauses of this standard describe the behavior of each of the elements of the language. This clause gives an overview of the interactions between these elements, especially with respect to the scheduling and execution of events.

The elements that make up the Verilog HDL can be used to describe the behavior, at varying levels of abstraction, of electronic hardware. An HDL has to be a parallel programming language. The execution of certain language constructs is defined by parallel execution of blocks or processes. It is important to understand what execution order is guaranteed to the user and what execution order is indeterminate.

Although the Verilog HDL is used for more than simulation, the semantics of the language are defined for simulation, and everything else is abstracted from this base definition.

11.2 Event simulation

The Verilog HDL is defined in terms of a discrete event execution model. The discrete event simulation is described in more detail in this subclause to provide a context to describe the meaning and valid interpretation of Verilog HDL constructs. These resulting definitions provide the standard Verilog reference model for simulation, which all compliant simulators shall implement. However, there is a great deal of choice in the definitions that follow, and differences in some details of execution are to be expected between different simulators. In addition, Verilog HDL simulators are free to use different algorithms from those described in this clause, provided the user-visible effect is consistent with the reference model.

A design consists of connected threads of execution or processes. Processes are objects that can be evaluated, that may have state, and that can respond to changes on their inputs to produce outputs. Processes include primitives, modules, initial and always procedural blocks, continuous assignments, asynchronous tasks, and procedural assignment statements.

Every change in value of a net or variable in the circuit being simulated, as well as the named event, is considered an *update event*.

Processes are sensitive to update events. When an update event is executed, all the processes that are sensitive to that event are evaluated in an arbitrary order. The evaluation of a process is also an event, known as an *evaluation event*.

In addition to events, another key aspect of a simulator is time. The term *simulation time* is used to refer to the time value maintained by the simulator to model the actual time it would take for the circuit being simulated. The term *time* is used interchangeably with simulation time in this clause.

Events can occur at different times. In order to keep track of the events and to make sure they are processed in the correct order, the events are kept on an *event queue*, ordered by simulation time. Putting an event on the queue is called *scheduling an event*.

11.3 The stratified event queue

The Verilog event queue is logically segmented into five different regions. Events are added to any of the five regions, but are only removed from the active region.

 Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

a) *Active events* occur at the current simulation time and can be processed in any order.
b) *Inactive* events occur at the current simulation time, but shall be processed after all the active events are processed.
c) *Nonblocking assign update* events have been evaluated during some previous simulation time, but shall be assigned at this simulation time after all the active and inactive events are processed.
d) *Monitor* events shall be processed after all the active, inactive, and nonblocking assign update events are processed.
e) *Future* events occur at some future simulation time. Future events are divided into *future inactive events* and *future nonblocking assignment update events*.

The processing of all the active events is called a *simulation cycle*.

The freedom to choose any active event for immediate processing is an essential source of nondeterminism in the Verilog HDL.

An *explicit zero delay* (#0) requires that the process be suspended and added as an inactive event for the current time so that the process is resumed in the next simulation cycle in the current time.

A nonblocking assignment (see 9.2.2) creates a nonblocking assign update event, scheduled for a current or later simulation time.

The **$monitor** and **$strobe** system tasks (see 17.1) create monitor events for their arguments. These events are continuously reenabled in every successive time step. The monitor events are unique in that they cannot create any other events.

The callback procedures scheduled with PLI routines such as **vpi_register_cb(cbReadWriteSynch)** (see 27.33) shall be treated as inactive events.

11.4 Verilog simulation reference model

In all the examples that follow, T refers to the current simulation time, and all events are held in the event queue, ordered by simulation time.

```
while (there are events) {
        if (no active events) {
                if (there are inactive events) {
                        activate all inactive events;
                } else if (there are nonblocking assign update events) {
                        activate all nonblocking assign update events;
                } else if (there are monitor events) {
                        activate all monitor events;
                } else {
                        advance T to the next event time;
                        activate all inactive events for time T;
                }
        }
        E = any active event;
        if (E is an update event) {
                update the modified object;
                add evaluation events for sensitive processes to event queue;
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
        } else { /* shall be an evaluation event */
                evaluate the process;
                add update events to the event queue;
        }
}
```

11.4.1 Determinism

This standard guarantees a certain scheduling order:

a) Statements within a `begin-end` block shall be executed in the order in which they appear in that `begin-end` block. Execution of statements in a particular `begin-end` block can be suspended in favor of other processes in the model; however, in no case shall the statements in a `begin-end` block be executed in any order other than that in which they appear in the source.
b) Nonblocking assignments shall be performed in the order the statements were executed (see 9.2.2). Consider the following example:

```
initial begin
    a <= 0;
    a <= 1;
end
```

When this block is executed, there will be two events added to the nonblocking assign update queue. The previous rule requires that they be entered on the queue in source order; this rule requires that they be taken from the queue and performed in source order as well. Hence, at the end of simulation time `0`, the variable `a` will be assigned `0` and then `1`.

11.4.2 Nondeterminism

One source of nondeterminism is the fact that active events can be taken off the queue and processed in any order. Another source of nondeterminism is that statements without time-control constructs in behavioral blocks do not have to be executed as one event. Time control statements are the # expression and @ expression constructs (see 9.7). At any time while evaluating a behavioral statement, the simulator may suspend execution and place the partially completed event as a pending active event on the event queue. The effect of this is to allow the interleaving of process execution, although the order of interleaved execution is nondeterministic and not under control of the user.

11.5 Race conditions

Because the execution of expression evaluation and net update events may be intermingled, race conditions are possible:

```
assign p = q;
initial begin
    q = 1;
    #1 q = 0;
    $display(p);
end
```

The simulator is correct in displaying either a `1` or a `0`. The assignment of `0` to `q` enables an update event for `p`. The simulator may either continue and execute the $display task or execute the update for `p`, followed by the $display task.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

11.6 Scheduling implication of assignments

Assignments are translated into processes and events as detailed in 11.6.1 through 11.6.7.

11.6.1 Continuous assignment

A continuous assignment statement (Clause 6) corresponds to a process, sensitive to the source elements in the expression. When the value of the expression changes, it causes an active update event to be added to the event queue, using current values to determine the target. A continuous assignment process is also evaluated at time 0 to ensure that constant values are propagated. This includes implicit continuous assignments (see 11.6.6).

11.6.2 Procedural continuous assignment

A procedural continuous assignment (which is the **assign** or **force** statement; see 9.3) corresponds to a process that is sensitive to the source elements in the expression. When the value of the expression changes, it causes an active update event to be added to the event queue, using current values to determine the target.

A **deassign** or a **release** statement deactivates any corresponding **assign** or **force** statement(s).

11.6.3 Blocking assignment

A blocking assignment statement (see 9.2.1) with a delay computes the right-hand side value using the current values, then causes the executing process to be suspended and scheduled as a future event. If the delay is 0, the process is scheduled as an inactive event for the current time.

When the process is returned (or if it returns immediately if no delay is specified), the process performs the assignment to the left-hand side and enables any events based upon the update of the left-hand side. The values at the time the process resumes are used to determine the target(s). Execution may then continue with the next sequential statement or with other active events.

11.6.4 Nonblocking assignment

A nonblocking assignment statement (see 9.2.2) always computes the updated value and schedules the update as a nonblocking assign update event, either in this time step if the delay is zero or as a future event if the delay is nonzero. The values in effect when the update is placed on the event queue are used to compute both the right-hand value and the left-hand target.

11.6.5 Switch (transistor) processing

The event-driven simulation algorithm described in 11.4 depends on unidirectional signal flow and can process each event independently. The inputs are read, the result is computed, and the update is scheduled.

The Verilog HDL provides switch-level modeling in addition to behavioral and gate-level modeling. Switches provide bidirectional signal flow and require coordinated processing of nodes connected by switches.

The Verilog HDL source elements that model switches are various forms of transistors, called **tran**, **tranif0**, **tranif1**, **rtran**, **rtranif0**, and **rtranif1**.

Switch processing shall consider all the devices in a bidirectional switch-connected net before it can determine the appropriate value for any node on the net because the inputs and outputs interact. A simulator can do this using a relaxation technique. The simulator can process tran at any time. It can process a subset of tran-connected events at a particular time, intermingled with the execution of other active events.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Further refinement is required when some transistors have gate value x. A conceptually simple technique is to solve the network repeatedly with these transistors set to all possible combinations of fully conducting and nonconducting transistors. Any node that has a unique logic level in all cases has steady-state response equal to this level. All other nodes have steady-state response x.

11.6.6 Port connections

Ports connect processes through implicit continuous assignment statements or implicit bidirectional connections. Bidirectional connections are analogous to an always-enabled tran connection between the two nets, but without any strength reduction. Port connection rules require that a value receiver be a net or a structural net expression.

Ports can always be represented as declared objects connected as follows:

— If an input port, then a continuous assignment from an outside expression to a local (input) net
— If an output port, then a continuous assignment from a local output expression to an outside net
— If an inout, then a nonstrength-reducing transistor connecting the local net to an outside net

Primitive terminals are different from module ports. Primitive output and inout terminals shall be connected directly to 1-bit nets or 1-bit structural net expressions (see 12.3.9.2), with no intervening process that could alter the strength. Changes from primitive evaluations are scheduled as active update events on the connected nets. Input terminals connected to 1-bit nets or 1-bit structural net expressions are also connected directly, with no intervening process that could affect the strength. Input terminals connected to other kinds of expressions are represented as implicit continuous assignments from the expression to an implicit net that is connected to the input terminal.

11.6.7 Functions and tasks

Task/function argument passing is by value, and it copies in on invocation and copies out on return. The copy-out-on-the-return function behaves in the same manner as does any blocking assignment.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

12. Hierarchical structures

The Verilog HDL supports a hierarchical hardware description structure by allowing modules to be embedded within other modules. Higher level modules create instances of lower level modules and communicate with them through input, output, and bidirectional ports. These module input/output (I/O) ports can be scalar or vector.

As an example of a module hierarchy, consider a system consisting of printed circuit boards (PCBs). The system would be represented as the top-level module and would create instances of modules that represent the boards. The board modules would, in turn, create instances of modules that represent integrated circuits (ICs), and the ICs could, in turn, create instances of modules such as flip-flops, muxes, and alus.

To describe a hierarchy of modules, the user provides textual definitions of the various modules. Each module definition stands alone; the definitions are not nested. Statements within the module definitions create instances of other modules, thus describing the hierarchy.

12.1 Modules

This subclause gives the formal syntax for a module definition and then gives the syntax for module instantiation, along with an example of a module definition and a module instantiation.

A module definition shall be enclosed between the keywords **module** and **endmodule**. The identifier following the keyword **module** shall be the name of the module being defined. The optional list of parameter definitions shall specify an ordered list of the parameters for the module. The optional list of ports or port declarations shall specify an ordered list of the ports for the module. The order used in defining the list of parameters in the `module_parameter_port_list` and in the list of ports can be significant when instantiating the module (see 12.2.2.1 and 12.3.5). The identifiers in this list shall be declared in input, output, and inout statements within the module definition. Ports declared in the list of port declarations shall not be redeclared within the body of the module. The module items define what constitutes a module, and they include many different types of declarations and definitions, many of which have already been introduced.

The keyword **macromodule** can be used interchangeably with the keyword **module** to define a module. An implementation may choose to treat module definitions beginning with the **macromodule** keyword differently.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
module_declaration ::= (From A.1.2)
        { attribute_instance } module_keyword module_identifier [ module_parameter_port_list ]
          list_of_ports ; { module_item }
          endmodule
      | { attribute_instance } module_keyword module_identifier [ module_parameter_port_list ]
          [ list_of_port_declarations ] ; { non_port_module_item }
          endmodule
module_keyword ::= module | macromodule
module_parameter_port_list ::= (From A.1.3
      # ( parameter_declaration { , parameter_declaration } )
list_of_ports ::= ( port { , port } )
list_of_port_declarations ::= ( port_declaration { , port_declaration } ) | ( )
port ::= [ port_expression ] | . port_identifier ( [ port_expression ] )
port_expression ::= port_reference | { port_reference { , port_reference } }
port_reference ::= port_identifier [ [ constant_range_expression ] ]
port_declaration ::= {attribute_instance} inout_declaration
      | {attribute_instance} input_declaration
      | {attribute_instance} output_declaration
module_item ::= (From A.1.4)
        port_declaration ;
      | non_port_module_item
module_or_generate_item ::=
        { attribute_instance } module_or_generate_item_declaration
      | { attribute_instance } local_parameter_declaration ;
      | { attribute_instance } parameter_override
      | { attribute_instance } continuous_assign
      | { attribute_instance } gate_instantiation
      | { attribute_instance } udp_instantiation
      | { attribute_instance } module_instantiation
      | { attribute_instance } initial_construct
      | { attribute_instance } always_construct
      | { attribute_instance } loop_generate_construct
      | { attribute_instance } conditional_generate_construct
module_or_generate_item_declaration ::=
        net_declaration
      | reg_declaration
      | integer_declaration
      | real_declaration
      | time_declaration
      | realtime_declaration
      | event_declaration
      | genvar_declaration
      | task_declaration
      | function_declaration
non_port_module_item ::=
        module_or_generate_item
      | generate_region
      | specify_block
      | { attribute_instance } parameter_declaration ;
      | { attribute_instance } specparam_declaration
parameter_override ::= defparam list_of_defparam_assignments ;
```

Syntax 12-1—Syntax for module

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

See 12.3 for the definitions of ports.

12.1.1 Top-level modules

Top-level modules are modules that are included in the source text, but do not appear in any module instantiation statement, as described in 12.1.2. This applies even if the module instantiation appears in a generate block that is not itself instantiated (see 12.4). A model shall contain at least one top-level module.

12.1.2 Module instantiation

Instantiation allows one module to incorporate a copy of another module into itself. Module definitions do not nest. In other words, one module definition shall not contain the text of another module definition within its **module**-**endmodule** keyword pair. A module definition nests another module by *instantiating* it. The *module instantiation statement* creates one or more named *instances* of a defined module.

For example, a counter module might instantiate a D flip-flop module to create multiple instances of the flip-flop.

Syntax 12-2 gives the syntax for specifying instantiations of modules.

```
module_instantiation ::= (From A.4.1)
        module_identifier [ parameter_value_assignment ]
            module_instance { , module_instance } ;
parameter_value_assignment ::=
        # ( list_of_parameter_assignments )
list_of_parameter_assignments ::=
          ordered_parameter_assignment { , ordered_parameter_assignment }
        | named_parameter_assignment   { , named_parameter_assignment }
ordered_parameter_assignment ::=
        expression
named_parameter_assignment ::=
        . parameter_identifier ( [ mintypmax_expression ] )
module_instance ::=
        name_of_module_instance ( [ list_of_port_connections ] )
name_of_module_instance ::=
        module_instance_identifier [ range ]
list_of_port_connections ::=
          ordered_port_connection { , ordered_port_connection }
        | named_port_connection { , named_port_connection }
ordered_port_connection ::=
        { attribute_instance } [ expression ]
named_port_connection ::=
        { attribute_instance } . port_identifier ( [ expression ] )
```

Syntax 12-2—Syntax for module instantiation

The instantiations of modules can contain a range specification. This allows an array of instances to be created. The array of instances is described in 7.1. The syntax and semantics of arrays of instances defined for gates and primitives apply for modules as well.

One or more module instances (identical copies of a module) can be specified in a single module instantiation statement.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The list of port connections shall be provided only for modules defined with ports. The parentheses, however, are always required. When a list of port connections is given using the ordered port connection method, the first element in the list shall connect to the first port declared in the module, the second to the second port, and so on. See 12.3 for a more detailed discussion of ports and port connection rules.

A connection can be a simple reference to a variable or a net identifier, an expression, or a blank. An expression can be used for supplying a value to a module input port. A blank port connection shall represent the situation where the port is not to be connected.

When connecting ports by name, an unconnected port can be indicated either by omitting it in the port list or by providing no expression in the parentheses [i.e., `.port_name ()`].

For example:

Example 1—The following example illustrates a circuit (the lower level module) being driven by a simple waveform description (the higher level module) where the circuit module is instantiated inside the waveform module:

```
// Lower level module:
// module description of a nand flip-flop circuit
module ffnand (q, qbar, preset, clear);
output q, qbar;          //declares 2 circuit output nets
input preset, clear;     //declares 2 circuit input nets

// declaration of two nand gates and their interconnections
nand g1 (q, qbar, preset),
     g2 (qbar, q, clear);
endmodule

// Higher level module:
// a waveform description for the nand flip-flop
module ffnand_wave;
wire out1, out2;     //outputs from the circuit
reg in1, in2;        //variables to drive the circuit
parameter d = 10;

// instantiate the circuit ffnand, name it "ff",
// and specify the IO port interconnections
ffnand ff(out1, out2, in1, in2);

// define the waveform to stimulate the circuit
initial begin
   #d in1 = 0; in2 = 1;
   #d in1 = 1;
   #d in2 = 0;
   #d in2 = 1;
end
endmodule
```

Example 2—The following example creates two instances of the flip-flop module `ffnand` defined in Example 1. It connects only to the `q` output in one instance and only to the `qbar` output in the other instance.

```
// a waveform description for testing
// the nand flip-flop, without the output ports
module ffnand_wave;
reg in1, in2; //variables to drive the circuit
parameter d = 10;
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
// make two copies of the circuit ffnand
// ff1 has qbar unconnected, ff2 has q unconnected
ffnand ff1(out1, , in1, in2),
       ff2(.qbar(out2), .clear(in2), .preset(in1), .q());
   //  ff3(.q(out3),.clear(in1),,,); is illegal

// define the waveform to stimulate the circuit
initial begin
   #d in1 = 0; in2 = 1;
   #d in1 = 1;
   #d in2 = 0;
   #d in2 = 1;
end
endmodule
```

12.2 Overriding module parameter values

There are two different ways that parameters can be defined. The first is the *module_parameter_port_list* (see 12.1), and the second is as a *module_item* (see 4.10). A module declaration can contain parameter definitions of either or both types or can contain no parameter definitions.

A module parameter can have a type specification and a range specification. The effect of parameter overrides on a parameter's type and range shall be in accordance with the following rules:

- A parameter declaration with no type or range specification shall default to the type and range of the final override value assigned to the parameter.
- A parameter with a range specification, but with no type specification, shall be the range of the parameter declaration and shall be unsigned. An override value shall be converted to the type and range of the parameter.
- A parameter with a type specification, but with no range specification, shall be of the type specified. An override value shall be converted to the type of the parameter. A signed parameter shall default to the range of the final override value assigned to the parameter.
- A parameter with a signed type specification and with a range specification shall be signed and shall be the range of its declaration. An override value shall be converted to the type and range of the parameter.

For example:

```
module generic_fifo
  #(parameter MSB=3, LSB=0, DEPTH=4)
                              //These parameters can be overridden
   (input [MSB:LSB] in,
    input clk, read, write, reset,
    output [MSB:LSB] out,
    output full, empty );

   localparam FIFO_MSB = DEPTH*MSB;
   localparam FIFO_LSB = LSB;
         // These parameters are local, and cannot be overridden.
         // They can be affected by altering the public parameters
         // above, and the module will work correctly.

   reg [FIFO_MSB:FIFO_LSB] fifo;
   reg [LOG2(DEPTH):0] depth;
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
    always @(posedge clk or reset) begin
      casex ({read,write,reset})
        // implementation of fifo
      endcase
    end
  endmodule
```

There are two ways to alter nonlocal parameter values: the *defparam statement*, which allows assignment to parameters using their hierarchical names, and the *module instance parameter value assignment*, which allows values to be assigned in line during module instantiation. If a **defparam** assignment conflicts with a module instance parameter, the parameter in the module will take the value specified by the **defparam**. The module instance parameter value assignment comes in two forms, by ordered list or by name. The next two subclauses describe these two methods.

There are two kinds of parameter declarations. The first kind of parameter declaration has a type and/or range qualification, and the second does not. When an untyped and unranged parameter's value is overridden, the parameter takes on the size and type of the override.

When a typed and/or ranged parameter is overridden, the new value is converted to the type and size of the destination and assigned to that parameter.

For example:

```
module foo(a,b);
   real r1,r2;
   parameter [2:0] A = 3'h2;
   parameter B = 3'h2;
   initial begin
      r1 = A;
      r2 = B;
      $display("r1 is %f r2 is %f",r1,r2);
   end
endmodule // foo
module bar;
   wire a,b;
   defparam f1.A = 3.1415;
   defparam f1.B = 3.1415;
   foo f1(a,b);
endmodule // bar
```

Parameter `A` is a typed and/or ranged parameter; therefore, when its value is redefined, the parameter retains its original type and sign. Therefore, the **defparam** of `f1.A` with the value 3.1415 is performed by converting the floating point number 3.1415 into a fixed-point number 3, and then the low 3 bits of 3 are assigned to `A`.

Parameter `B` is not a typed and/or ranged parameter; therefore, when its value is redefined, the parameter type and range take on the type and range of the new value. Therefore, the **defparam** of `f1.B` with the value 3.1415 replaces `B`'s current value of 3'h2 with the floating point number 3.1415.

12.2.1 defparam statement

Using the *defparam statement*, parameter values can be changed in any module instance throughout the design using the hierarchical name of the parameter. See 12.5 for hierarchical names.

However, a **defparam** statement in a hierarchy in or under a generate block instance (see 12.4) or an array of instances (see 7.1 and 12.1.2) shall not change a parameter value outside that hierarchy.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

Each instantiation of a generate block is considered to be a separate hierarchy scope. Therefore, this rule implies that a **defparam** statement in a generate block may not target a parameter in another instantiation of the same generate block, even when the other instantiation is created by the same loop generate construct. For example, the following code is not allowed:

```
genvar i;

generate
  for (i = 0; i < 8; i = i + 1) begin : somename
    flop my_flop(in[i], in1[i], out1[i]);
    defparam somename[i+1].my_flop.xyz = i ;
  end
endgenerate
```

Similarly, a **defparam** statement in one instance of an array of instances may not target a parameter in another instance of the array.

The expression on the right-hand side of the **defparam** assignments shall be a constant expression involving only numbers and references to parameters. The referenced parameters (on the right-hand side of the **defparam**) shall be declared in the same module as the **defparam** statement.

The **defparam** statement is particularly useful for grouping all of the parameter value override assignments together in one module.

In the case of multiple **defparam**s for a single parameter, the parameter takes the value of the last **defparam** statement encountered in the source text. When **defparam**s are encountered in multiple source files, e.g., found by library searching, the **defparam** from which the parameter takes its value is undefined.

For example:

```
module top;
reg clk;
reg [0:4] in1;
reg [0:9] in2;
wire [0:4] o1;
wire [0:9] o2;

vdff m1 (o1, in1, clk);
vdff m2 (o2, in2, clk);
endmodule

module vdff (out, in, clk);
parameter size = 1, delay = 1;
input [0:size-1] in;
input clk;
output [0:size-1] out;
reg [0:size-1] out;

always @(posedge clk)
   # delay out = in;
endmodule

module annotate;
defparam
       top.m1.size = 5,
       top.m1.delay = 10,
       top.m2.size = 10,
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
        top.m2.delay = 20;
endmodule
```

The module `annotate` has the **defparam** statement, which overrides `size` and `delay` parameter values for instances `m1` and `m2` in the top-level module `top`. The modules `top` and `annotate` would both be considered top-level modules.

12.2.2 Module instance parameter value assignment

An alternative method for assigning values to parameters within module instances is to use one of the two forms of module instance parameter value assignment. They are *assignment by ordered list* and *assignment by name*. The two types of module instance parameter value assignment shall not be mixed; parameter assignments to a particular module instance shall be entirely by order or entirely by name.

Module instance parameter value assignment by ordered list is similar in appearance to the assignment of delay values to gate instances, and assignment by name is similar to connecting module ports by name. It supplies values for particular instances of a module to any parameters that have been specified in the definition of that module.

A parameter declared in a named block, task, or function can only be directly redefined using a **defparam** statement. However, if the parameter value is dependent on a second parameter, then redefining the second parameter will update the value of the first parameter as well (see 12.2.3).

12.2.2.1 Parameter value assignment by ordered list

The order of the assignments in the module instance parameter value assignment by ordered list shall follow the order of declaration of the parameters within the module. It is not necessary to assign values to all of the parameters within a module when using this method. However, it is not possible to skip over a parameter. Therefore, to assign values to a subset of the parameters declared within a module, the declarations of the parameters that make up this subset shall precede the declarations of the remaining parameters. An alternative is to assign values to all of the parameters, but to use the default value (the same value assigned in the declaration of the parameter within the module definition) for those parameters that do not need new values.

For example:

Consider the following example, where the parameters within module instances `mod_a`, `mod_c`, and `mod_d` are changed during instantiation:

```
module tb1;

  wire [9:0] out_a, out_d;
  wire [4:0] out_b, out_c;
  reg  [9:0] in_a, in_d;
  reg  [4:0] in_b, in_c;
  reg        clk;

  // testbench clock & stimulus generation code ...

  // Four instances of vdff with parameter value assignment
  // by ordered list

  // mod_a has new parameter values size=10 and delay=15
  // mod_b has default parameters (size=5, delay=1)
  // mod_c has one default size=5 and one new delay=12
  //   In order to change the value of delay,
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
  //   it is necessary to specify the (default) value of size as well.
  // mod_d has a new parameter value size=10.
  //   delay retains its default value

  vdff #(10,15) mod_a (.out(out_a), .in(in_a), .clk(clk));
  vdff          mod_b (.out(out_b), .in(in_b), .clk(clk));
  vdff #( 5,12) mod_c (.out(out_c), .in(in_c), .clk(clk));
  vdff #(10)    mod_d (.out(out_d), .in(in_d), .clk(clk));

endmodule

module vdff (out, in, clk);
  parameter size=5, delay=1;
  output [size-1:0] out;
  input  [size-1:0] in;
  input             clk;
  reg    [size-1:0] out;

  always @(posedge clk)
    #delay out = in;

endmodule
```

Local parameters cannot be overridden; therefore, they are not considered part of the ordered list for parameter value assignment. In the following example, `addr_width` will be assigned the value 12, and `data_width` will be assigned the value 16. `mem_size` will not be explicitly assigned a value due to the ordered list, but will have the value 4096 due to its declaration expression.

```
module my_mem (addr, data);

parameter addr_width = 16;
localparam mem_size = 1 << addr_width;
parameter data_width = 8;
...
endmodule

module top;
...
my_mem #(12, 16) m(addr,data);
endmodule
```

12.2.2.2 Parameter value assignment by name

Parameter assignment by name consists of explicitly linking the parameter name and its new value. The name of the parameter shall be the name specified in the instantiated module.

It is not necessary to assign values to all of the parameters within a module when using this method. Only parameters that are assigned new values need to be specified.

The parameter expression is optional so that the instantiating module can document the existence of a parameter without assigning anything to it. The parentheses are required, and in this case the parameter retains its default value. Once a parameter is assigned a value, there shall not be another assignment to this parameter name.

Consider the following example, where both parameters of `mod_a` and only one parameter of `mod_c` and `mod_d` are changed during instantiation:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
module tb2;

  wire [9:0] out_a, out_d;
  wire [4:0] out_b, out_c;
  reg  [9:0] in_a, in_d;
  reg  [4:0] in_b, in_c;
  reg        clk;

  // testbench clock & stimulus generation code ...

  // Four instances of vdff with parameter value assignment by name

  // mod_a has new parameter values size=10 and delay=15
  // mod_b has default parameters (size=5, delay=1)
  // mod_c has one default size=5 and one new delay=12
  // mod_d has a new parameter value size=10.
  //   delay retains its default value

  vdff #(.size(10),.delay(15)) mod_a (.out(out_a),.in(in_a),.clk(clk));
  vdff                         mod_b (.out(out_b),.in(in_b),.clk(clk));
  vdff #(.delay(12))           mod_c (.out(out_c),.in(in_c),.clk(clk));
  vdff #(.delay( ),.size(10) ) mod_d (.out(out_d),.in(in_d),.clk(clk));

endmodule

module vdff (out, in, clk);
  parameter size=5, delay=1;
  output [size-1:0] out;
  input  [size-1:0] in;
  input             clk;
  reg    [size-1:0] out;

  always @(posedge clk)
    #delay out = in;

endmodule
```

It shall be legal to instantiate modules using different types of parameter redefinition in the same top-level module. Consider the following example, where the parameters of `mod_a` are changed using parameter redefinition by ordered list and the second parameter of `mod_c` is changed using parameter redefinition by name during instantiation:

```
module tb3;

  // declarations & code

  // legal mixture of instance with positional parameters and
  // another instance with named parameters

  vdff #(10, 15)     mod_a (.out(out_a), .in(in_a), .clk(clk));
  vdff               mod_b (.out(out_b), .in(in_b), .clk(clk));
  vdff #(.delay(12)) mod_c (.out(out_c), .in(in_c), .clk(clk));

endmodule
```

It shall be illegal to instantiate any module using a mixture of parameter redefinitions by order and by name as shown in the instantiation of `mod_a` below:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
// mod_a instance with ILLEGAL mixture of parameter assignments
vdff #(10, .delay(15)) mod_a (.out(out_a), .in(in_a), .clk(clk));
```

12.2.3 Parameter dependence

A parameter (for example, `memory_size`) can be defined with an expression containing another parameter (for example, `word_size`). However, overriding a parameter, whether by a **defparam** statement or in a module instantiation statement, effectively replaces the parameter definition with the new expression. Because `memory_size` depends on the value of `word_size`, a modification of `word_size` changes the value of `memory_size`. For example, in the following parameter declaration, an update of `word_size`, whether by **defparam** statement or in an instantiation statement for the module that defined these parameters, automatically updates `memory_size`. If `memory_size` is updated due to either a **defparam** or an instantiation statement, then it will take on that value, regardless of the value of `word_size`.

```
parameter
    word_size = 32,
    memory_size = word_size * 4096;
```

12.3 Ports

Ports provide a means of interconnecting a hardware description consisting of modules and primitives. For example, module A can instantiate module B, using port connections appropriate to module A. These port names can differ from the names of the internal nets and variables specified in the definition of module B.

12.3.1 Port definition

The syntax for ports and a list of ports is given in Syntax 12-3.

```
list_of_ports ::= (From A.1.3)
      ( port { , port } )
list_of_port_declarations ::=
       ( port_declaration { , port_declaration } )
      | ( )
port ::=
       [ port_expression ]
      | . port_identifier ( [ port_expression ] )
port_expression ::=
       port_reference
      | { port_reference { , port_reference } }
port_reference ::=
      port_identifier [ [ constant_range_expression ] ]
port_declaration ::=
       {attribute_instance} inout_declaration
      | {attribute_instance} input_declaration
      | {attribute_instance} output_declaration
```

Syntax 12-3—Syntax for port

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

12.3.2 List of ports

The port reference for each port in the list of ports at the top of each module declaration can be one of the following:

— A simple identifier or escaped identifier
— A bit-select of a vector declared within the module
— A part-select of a vector declared within the module
— A concatenation of any of the above

The port expression is optional because ports can be defined that do not connect to anything internal to the module. Once a port has been defined, there shall not be another port definition with this same name.

The first type of module port, with only a `port_expression`, is an implicit port. The second type is the explicit port. This explicitly specifies the `port_identifier` used for connecting module instance ports by name (see 12.3.6) and the `port_expression` that contains identifiers declared inside the module as described in 12.3.3. Named port connections shall not be used for implicit ports unless the `port_expression` is a simple identifier or escaped identifier, which shall be used as the port name.

12.3.3 Port declarations

Each *port_identifier* in a *port_expression* in the list of ports for the module declaration shall also be declared in the body of the module as one of the following port declarations: **input**, **output**, or **inout** (bidirectional). This is in addition to any other data type declaration for a particular port— for example, a **reg** or **wire**. The syntax for port declarations is given in Syntax 12-4.

```
inout_declaration ::= (From A.2.1.2)
        inout [ net_type ] [ signed ] [ range ] list_of_port_identifiers
input_declaration ::=
        input [ net_type ] [ signed ] [ range ] list_of_port_identifiers
output_declaration ::=
         output [ net_type ] [ signed ] [ range ]
            list_of_port_identifiers
        | output reg [ signed ] [ range ]
            list_of_variable_port_identifiers
        | output output_variable_type
            list_of_variable_port_identifiers
list_of_port_identifiers ::= (From A.2.3)
        port_identifier { , port_identifier }
```

Syntax 12-4—Syntax for port declarations

If a port declaration includes a net or variable type, then the port is considered completely declared, and it is an error for the port to be declared again in a variable or net data type declaration. Because of this, all other aspects of the port shall be declared in such a port declaration, including the signed and range definitions if needed.

If a port declaration does not include a net or variable type, then the port can be again declared in a net or variable declaration. If the net or variable is declared as a vector, the range specification between the two declarations of a port shall be identical. Once a name is used in a port declaration, it shall not be declared again in another port declaration or in a data type declaration.

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Implementations may limit the maximum number of ports in a module definition, but the limit shall be at least 256.

For example:

```
input  aport;    // First declaration - okay.
input  aport;    // Error - multiple declaration, port declaration
output aport;    // Error - multiple declaration, port declaration
```

The signed attribute can be attached either to a port declaration or the corresponding net or **reg** declaration or to both. If either the port or the net/**reg** is declared as signed, then the other shall also be considered signed.

Implicit nets shall be considered unsigned. Nets connected to ports without an explicit net declaration shall be considered unsigned, unless the port is declared as signed.

For example:

```
module test(a,b,c,d,e,f,g,h);
input [7:0] a;        // no explicit declaration - net is unsigned
input [7:0] b;
input signed [7:0] c;
input signed [7:0] d;  // no explicit net declaration - net is signed
output [7:0] e;       // no explicit declaration - net is unsigned
output [7:0] f;
output signed [7:0] g;
output signed [7:0] h;  // no explicit net declaration - net is signed

wire signed [7:0] b;  // port b inherits signed attribute from net decl.
wire [7:0] c;         // net  c inherits signed attribute from port
reg signed [7:0] f;   // port f inherits signed attribute from reg decl.
reg [7:0] g;          // reg  g inherits signed attribute from port

endmodule

module complex_ports ({c,d}, .e(f));
                // Nets {c,d} receive the first port bits.
                // Name 'f' is declared inside the module.
                // Name 'e' is defined outside the module.
                // Can't use named port connections of first port.

module split_ports (a[7:4], a[3:0]);
                // First port  is upper 4 bits of 'a'.
                // Second port is lower 4 bits of 'a'.
                // Can't use named port connections because
                // of part-select port 'a'.

module same_port (.a(i), .b(i));
          // Name 'i' is declared inside the module as an inout port.
          // Names 'a' and 'b' are defined for port connections.

module renamed_concat (.a({b,c}), f, .g(h[1]));
         // Names 'b', 'c', 'f', 'h' are defined inside the module.
         // Names 'a', 'f', 'g' are defined for port connections.
         // Can use named port connections.

module same_input (a,a);
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
input a;              // This is legal. The inputs are tied together.

module mixed_direction (.p({a, e}));
input a;              // p contains both input and output directions.
output e;
```

12.3.4 List of ports declarations

An alternate syntax that minimizes the duplication of data can be used to specify the ports of a module. Each module shall be declared either entirely with the list of ports syntax as described in 12.3.2 or entirely using the *list_of_port_declarations* as described in this subclause.

Each declared port provides the complete information about the port. The port's direction, width, net, or variable type and whether the port is signed or unsigned are completely described. The same syntax for input, inout, and output declarations is used in the module header as would be used for the list of port style declaration, except the *list_of_port_declarations* is included in the module header rather than separately (after the `;` that terminates the module header).

For example:

As an example, the module named test given in the previous example could alternatively be declared as follows:

```
module test (
   input [7:0] a,
   input signed [7:0] b, c, d,   // Multiple ports that share all
                                 // attributes can be declared together.
   output [7:0] e,               // Every attribute of the declaration
                                 // must be in the one declaration.
   output reg signed [7:0] f, g,
   output signed    [7:0] h) ;
   // It is illegal to redeclare any ports of
   // the module in the body of the module.
endmodule
```

The *port_reference* type of module port declaration shall not be done using *list_of_port_declarations* style of module declarations. Also ports declared using the *list_of_port_declarations* shall only be simple identifiers or escaped identifiers. They shall not be bit-selects, part-selects, or concatenations (as in the example `complex_ports`); nor can ports be split (as in the example `split_ports`); nor can they be named ports (as in the example `same_port`).

Designs may freely mix modules declared using each syntax; hence implementations desiring the above special cases of port declaration can be done using the first `list_of_ports` syntax.

12.3.5 Connecting module instance ports by ordered list

One method of making the connection between the port expressions listed in a module instantiation and the ports declared within the instantiated module is the ordered list; that is, the port expressions listed for the module instance shall be in the same order as the ports listed in the module declaration.

For example:

The following example illustrates a top-level module (`topmod`) that instantiates a second module (`modB`). Module `modB` has ports that are connected by an ordered list. The connections made are as follows:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

— Port wa in the modB definition connects to the bit-select v[0] in the topmod module.
— Port wb connects to v[3].
— Port c connects to w.
— Port d connects to v[4].

In the modB definition, ports wa and wb are declared as inouts while ports c and d are declared as input.

```
module topmod;
    wire [4:0] v;
    wire a,b,c,w;

    modB b1 (v[0], v[3], w, v[4]);
    endmodule

    module modB (wa, wb, c, d);
    inout wa, wb;
    input c, d;

    tranif1         g1 (wa, wb, cinvert);
    not #(2, 6)     n1 (cinvert, int);
    and #(6, 5)     g2 (int, c, d);
endmodule
```

During simulation of the b1 instance of modb, the **and** gate g2 activates first to produce a value on int. This value triggers the **not** gate n1 to produce output on cinvert, which then activates the **tranif1** gate g1.

12.3.6 Connecting module instance ports by name

The second way to connect module ports consists of explicitly linking the two names for each side of the connection: the port declaration name from the module declaration to the expression, i.e., the name used in the module declaration, followed by the name used in the instantiating module. This compound name is then placed in the list of module connections. The port name shall be the name specified in the module declaration. The port name cannot be a bit-select, a part-select, or a concatenation of ports. If the module port declaration was implicit, the port_expression shall be a simple identifier or escaped identifier, which shall be used as the port name. If the module port declaration was explicit, the explicit name is used as the name of port.

The port expression can be any valid expression.

The port expression is optional so that the instantiating module can document the existence of the port without connecting it to anything. The parentheses are required.

The two types of module port connections shall not be mixed; connections to the ports of a particular module instance shall be all by order or all by name.

For example:

Example 1—In the following example, the instantiating module connects its signals topA and topB to the ports In1 and Out defined by the module ALPHA. At least one port provided by ALPHA is unused; it is named In2. There could be other unused ports not mentioned in the instantiation.

```
ALPHA instance1 (.Out(topB),.In1(topA),.In2());
```

Example 2—This example defines the modules modB and topmod, and then topmod instantiates modB using ports connected by name.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
module topmod;
    wire [4:0] v;
    wire a,b,c,w;

    modB b1 (.wb(v[3]),.wa(v[0]),.d(v[4]),.c(w));
endmodule

module modB(wa, wb, c, d);
    inout wa, wb;
    input c, d;

    tranif1         g1(wa, wb, cinvert);
    not #(6, 2)     n1(cinvert, int);
    and #(5, 6)     g2(int, c, d);
endmodule
```

Because these connections are made by name, the order in which they appear is irrelevant.

Multiple module instance port connections are not allowed, e.g., the following example is illegal:

Example 3—This example shows illegal port connections.

```
module test;
  a ia (.i (a), .i (b), // illegal connection of input port twice.
        .o (c), .o (d), // illegal connection of output port twice.
        .e (e), .e (f)); // illegal connection of inout port twice.
endmodule
```

12.3.7 Real numbers in port connections

The **real** data type shall not be directly connected to a port. It shall be connected indirectly, as shown in the following example. The system functions **$realtobits** and **$bitstoreal** shall be used for passing the bit patterns across module ports. (See 17.8 for a description of these system tasks.)

For example:

```
module driver (net_r);
    output net_r;
    real r;
    wire [64:1] net_r = $realtobits(r);
endmodule

module receiver (net_r);
    input net_r;
    wire [64:1] net_r;
    real r;

    initial assign r = $bitstoreal(net_r);

endmodule
```

12.3.8 Connecting dissimilar ports

A port of a module can be viewed as providing a link or connection between two items (e.g., nets, regs, expressions)—one internal to the module instance and one external to the module instance.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

Examination of the port connection rules described in 12.3.9 will show that the item receiving the value through the port (the internal item for inputs, the external item for outputs) shall be a structural net expression. The item that provides the value can be any expression.

A port that is declared as input (output) but used as an output (input) or inout may be coerced to inout. If not coerced to inout, a warning has to be issued.

12.3.9 Port connection rules

The rules in 12.3.9.1 through 12.3.9.3 shall govern the way module ports are declared and the way they are interconnected.

12.3.9.1 Rule 1

An input or inout port shall be of type net.

12.3.9.2 Rule 2

Each port connection shall be a continuous assignment of source to sink, where one connected item shall be a signal source and the other shall be a signal sink. The assignment shall be a continuous assignment from source to sink for input or output ports. The assignment is a nonstrength reducing transistor connection for inout ports. Only nets or structural net expressions shall be the sinks in an assignment.

A *structural net expression* is a port expression whose operands can be the following:

- A scalar net
- A vector net
- A constant bit-select of a vector net
- A part-select of a vector net
- A concatenation of structural net expressions

The following external items shall not be connected to the output or inout ports of modules:

- Variables
- Expressions other than the following:
  - A scalar net
  - A vector net
  - A constant bit-select of a vector net
  - A part-select of a vector net
  - A concatenation of the expressions listed above

12.3.9.3 Rule 3

If the net on either side of a port has the net type **uwire**, a warning shall be issued if the nets are not merged into a single net, as described in 12.3.10.

12.3.10 Net types resulting from dissimilar port connections

When different net types are connected through a module port, the nets on both sides of the port can take on the same type. The resulting net type can be determined as shown in Table 12-1. In the table, *external net* means the net specified in the module instantiation, and *internal net* means the net specified in the module

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

definition. The net whose type is used is said to be the *dominating net*. The net whose type is changed is said to be the *dominated net*. It is permissible to merge the dominating and dominated nets into a single net, whose type shall be that of the dominating net. The resulting net is called the *simulated net*, and the dominated net is called a *collapsed net*.

The simulated net shall take the delay specified for the dominating net. If the dominating net is of the type **trireg**, any strength value specified for the trireg net shall apply to the simulated net.

Table 12-1—Net types resulting from dissimilar port connections

Internal net	External net								
	wire, tri	**wand, triand**	**wor, trior**	**trireg**	**tri0**	**tri1**	**uwire**	**supply0**	**supply1**
wire, tri	ext	ext	ext	ext	ext	ext	ext	ext	ext
wand, triand	int	ext	ext warn	ext warn	ext warn	ext warn	ext warn	ext	ext
wor, trior	int	ext warn	ext	ext warn	ext warn	ext warn	ext warn	ext	ext
trireg	int	ext warn	ext warn	ext	ext	ext	ext warn	ext	ext
tri0	int	ext warn	ext warn	int	ext	ext warn	ext warn	ext	ext
tri1	int	ext warn	ext warn	int	ext warn	ext	ext warn	ext	ext
uwire	int	int warn	int warn	int warn	int warn	int warn	ext	ext	ext
supply0	int	int	int	int	int	int	int	ext	ext warn
supply1	int	int	int	int	int	int	int	ext warn	ext

KEY:
ext = The external net type shall be used.
int = The internal net type shall be used.
warn = A warning shall be issued.

12.3.10.1 Net type resolution rule

When the two nets connected by a port are of different net type, the resulting single net can be assigned one of the following:

— The dominating net type if one of the two nets is dominating, or
— The net type external to the module

When a dominating net type does not exist, the external net type shall be used.

12.3.10.2 Net type table

Table 12-1 shows the net type dictated by net type resolution rule.

Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The simulated net shall take the net type specified in the table and the delay specified for that net. If the simulated net selected is a **trireg**, any strength value specified for the trireg net applies to the simulated net.

12.3.11 Connecting signed values via ports

The sign attribute shall not cross hierarchy. In order to have the signed type cross hierarchy, the **signed** keyword must be used in the object's declaration at the different levels of hierarchy. Any expressions on a port shall be treated as any other expression in an assignment. It shall be typed, sized, and evaluated, and the resulting value assigned to the object on the other side of the port using the same rules as an assignment.

12.4 Generate constructs

Generate constructs are used to either conditionally or multiply instantiate generate blocks into a model. A generate block is a collection of one or more module items. A generate block may not contain port declarations, parameter declarations, specify blocks, or specparam declarations. All other module items, including other generate constructs, are allowed in a generate block. Generate constructs provide the ability for parameter values to affect the structure of the model. They also allow for modules with repetitive structure to be described more concisely, and they make recursive module instantiation possible.

There are two kinds of generate constructs: loops and conditionals. *Loop generate constructs* allow a single generate block to be instantiated into a model multiple times. *Conditional generate constructs*, which include if-generate and case-generate constructs, instantiate at most one generate block from a set of alternative generate blocks. The term *generate scheme* refers to the method for determining which or how many generate blocks are instantiated. It includes the conditional expressions, case alternatives, and loop control statements that appear in a generate construct.

Generate schemes are evaluated during elaboration of the model. Elaboration occurs after parsing the HDL and before simulation; and it involves expanding module instantiations, computing parameter values, resolving hierarchical names (see 12.5), establishing net connectivity and in general preparing the model for simulation. Although generate schemes use syntax that is similar to behavioral statements, it is important to recognize that they do not execute at simulation time. They are evaluated at elaboration time, and the result is determined before simulation begins. Therefore, all expressions in generate schemes shall be constant expressions, deterministic at elaboration time. For more details on elaboration, see 12.8.

The elaboration of a generate construct results in zero or more instances of a generate block. An instance of a generate block is similar in some ways to an instance of a module. It creates a new level of hierarchy. It brings the objects, behavioral constructs, and module instances within the block into existence. These constructs act the same as they would if they were in a module brought into existence with a module instantiation, except that object declarations from the enclosing scope can be referenced directly (see 12.7). Names in instantiated named generate blocks can be referenced hierarchically as described in 12.5.

The keywords **generate** and **endgenerate** may be used in a module to define a *generate region*. A generate region is a textual span in the module description where generate constructs may appear. Use of generate regions is optional. There is no semantic difference in the module when a generate region is used. A parser may choose to recognize the generate region to produce different error messages for misused generate construct keywords. Generate regions do not nest, and they may only occur directly within a module. If the **generate** keyword is used, it shall be matched by an **endgenerate** keyword.

The syntax for generate constructs is given in Syntax 12-5.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
module_or_generate_item ::= (From A.1.4)
        { attribute_instance } module_or_generate_item_declaration
      | { attribute_instance } local_parameter_declaration ;
      | { attribute_instance } parameter_override
      | { attribute_instance } continuous_assign
      | { attribute_instance } gate_instantiation
      | { attribute_instance } udp_instantiation
      | { attribute_instance } module_instantiation
      | { attribute_instance } initial_construct
      | { attribute_instance } always_construct
      | { attribute_instance } loop_generate_construct
      | { attribute_instance } conditional_generate_construct
generate_region ::= (From A.4.2)
      generate { module_or_generate_item } endgenerate
genvar_declaration ::=
      genvar list_of_genvar_identifiers ;
list_of_genvar_identifiers ::=
      genvar_identifier { , genvar_identifier }
loop_generate_construct ::=
      for ( genvar_initialization ; genvar_expression ; genvar_iteration )
        generate_block
genvar_initialization ::=
      genvar_identifier = constant_expression
genvar_expression ::=
        genvar_primary
      | unary_operator { attribute_instance } genvar_primary
      | genvar_expression binary_operator { attribute_instance } genvar_expression
      | genvar_expression ? { attribute_instance } genvar_expression : genvar_expression
genvar_iteration ::=
      genvar_identifier = genvar_expression
genvar_primary ::=
        constant_primary
      | genvar_identifier
conditional_generate_construct ::=
        if_generate_construct
      | case_generate_construct
if_generate_construct ::=
      if ( constant_expression ) generate_block_or_null
        [ else generate_block_or_null ]
case_generate_construct ::=
      case ( constant_expression )
        case_generate_item { case_generate_item } endcase
case_generate_item ::=
        constant_expression { , constant_expression } : generate_block_or_null
      | default [ : ] generate_block_or_null
generate_block ::=
        module_or_generate_item
      | begin [ : generate_block_identifier ] { module_or_generate_item } end
generate_block_or_null ::=
        generate_block | ;
```

Syntax 12-5—Syntax for generate constructs

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

12.4.1 Loop generate constructs

A loop generate construct permits a generate block to be instantiated multiple times using syntax that is similar to a for loop statement. The loop index variable shall be declared in a **genvar** declaration prior to its use in a loop generate scheme.

The **genvar** is used as an integer during elaboration to evaluate the generate loop and create instances of the generate block, but it does not exist at simulation time. A **genvar** shall not be referenced anywhere other than in a loop generate scheme.

Both the initialization and iteration assignments in the loop generate scheme shall assign to the same **genvar**. The initialization assignment shall not reference the loop index variable on the right-hand side.

Within the generate block of a loop generate construct, there is an implicit **localparam** declaration. This is an integer parameter that has the same name and type as the loop index variable, and its value within each instance of the generate block is the value of the index variable at the time the instance was elaborated. This parameter can be used anywhere within the generate block that a normal parameter with an integer value can be used. It can be referenced with a hierarchical name.

Because this implicit **localparam** has the same name as the **genvar**, any reference to this name inside the loop generate block will be a reference to the **localparam**, not to the **genvar**. As a consequence, it is not possible to have two nested loop generate constructs that use the same **genvar**.

Generate blocks in loop generate constructs can be named or unnamed, and they can consist of only one item, which need not be surrounded by **begin**/**end** keywords. Even if the **begin**/**end** keywords are absent, it is still a generate block, which, like all generate blocks, comprises a separate scope and a new level of hierarchy when it is instantiated.

If the generate block is named, it is a declaration of an array of generate block instances. The index values in this array are the values assumed by the **genvar** during elaboration. This can be a sparse array because the **genvar** values do not have to form a contiguous range of integers. The array is considered to be declared even if the loop generate scheme resulted in no instances of the generate block. If the generate block is not named, the declarations within it cannot be referenced using hierarchical names other than from within the hierarchy instantiated by the generate block itself.

It shall be an error if the name of a generate block instance array conflicts with any other declaration, including any other generate block instance array. It shall be an error if the loop generate scheme does not terminate. It shall be an error if a **genvar** value is repeated during the evaluation of the loop generate scheme. It shall be an error if any bit of the **genvar** is set to `x` or `z` during the evaluation of the loop generate scheme.

For example:

Example 1—Examples of legal and illegal generate loops

```
module mod_a;
genvar i;

// "generate",  "endgenerate" keywords are not required

for (i=0; i<5; i=i+1) begin:a
  for (i=0; i<5; i=i+1) begin:b
  ...               // error -- using "i" as loop index for
  ...               // two nested generate loops
  end
end
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
endmodule

------

module mod_b;
genvar i;
reg a;

for (i=1; i<0; i=i+1) begin: a
  ...               // error -- "a" conflicts with name of reg "a"
end

endmodule

------

module mod_c;
genvar i;

for (i=1; i<5; i=i+1) begin: a
 ...
end

for (i=10; i<15; i=i+1) begin: a
  ...              // error -- "a" conflicts with name of previous
  ...              // loop even though indices are unique
end

endmodule
```

Example 2—A parameterized gray-code–to–binary-code converter module using a loop to generate continuous assignments

```
module gray2bin1 (bin, gray);
  parameter SIZE = 8;      // this module is parameterizable
  output [SIZE-1:0] bin;
  input  [SIZE-1:0] gray;

  genvar i;
  generate
    for (i=0; i<SIZE; i=i+1) begin:bit
      assign bin[i] = ^gray[SIZE-1:i];
            // i refers to the implicitly defined localparam whose
            // value in each instance of the generate block is
            // the value of the genvar when it was elaborated.
    end
  endgenerate
endmodule
```

The models in Example 3 and Example 4 are parameterized modules of ripple adders using a loop to generate Verilog gate primitives. Example 3 uses a two-dimensional net declaration outside of the generate loop to make the connections between the gate primitives while Example 4 makes the net declaration inside of the generate loop to generate the wires needed to connect the gate primitives for each iteration of the loop.

Example 3—Generated ripple adder with two-dimensional net declaration outside of the generate loop

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
module addergen1 (co, sum, a, b, ci);
  parameter SIZE = 4;
  output [SIZE-1:0] sum;
  output            co;
  input  [SIZE-1:0] a, b;
  input             ci;
  wire   [SIZE  :0] c;
  wire   [SIZE-1:0] t [1:3];
  genvar            i;

  assign c[0] = ci;

  // Hierarchical gate instance names are:
  // xor gates: bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
  //            bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
  // and gates: bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
  //            bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
  // or  gates: bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
  // Generated instances are connected with
  // multidimensional nets t[1][3:0] t[2][3:0] t[3][3:0]
  // (12 nets total)

    for(i=0; i<SIZE; i=i+1) begin:bit
      xor g1 ( t[1][i],    a[i],    b[i]);
      xor g2 (  sum[i], t[1][i],    c[i]);
      and g3 ( t[2][i],    a[i],    b[i]);
      and g4 ( t[3][i], t[1][i],    c[i]);
      or  g5 (  c[i+1], t[2][i], t[3][i]);
    end

  assign co = c[SIZE];
endmodule
```

Example 4—Generated ripple adder with net declaration inside of the generate loop

```
module addergen1 (co, sum, a, b, ci);
  parameter SIZE = 4;
  output [SIZE-1:0] sum;
  output            co;
  input  [SIZE-1:0] a, b;
  input             ci;
  wire   [SIZE  :0] c;

  genvar            i;

  assign c[0] = ci;

  // Hierarchical gate instance names are:
  // xor gates: bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
  //            bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
  // and gates: bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
  //            bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
  // or  gates: bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
  // Gate instances are connected with nets named:
  //            bit[0].t1 bit[1].t1 bit[2].t1 bit[3].t1
  //            bit[0].t2 bit[1].t2 bit[2].t2 bit[3].t2
  //            bit[0].t3 bit[1].t3 bit[2].t3 bit[3].t3
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
    for(i=0; i<SIZE; i=i+1) begin:bit
      wire   t1, t2, t3;

      xor g1 (     t1, a[i], b[i]);
      xor g2 ( sum[i],   t1, c[i]);
      and g3 (     t2, a[i], b[i]);
      and g4 (     t3,   t1, c[i]);
      or  g5 ( c[i+1],   t2,   t3);
    end

  assign co = c[SIZE];
endmodule
```

The hierarchical generate block instance names in a multilevel generate loop are shown in Example 5. For each block instance created by the generate loop, the generate block identifier for the loop is indexed by adding the "[genvar value]" to the end of the generate block identifier. These names can be used in hierarchical path names (see 12.5).

Example 5—A multilevel generate loop

```
parameter SIZE = 2;
genvar i, j, k, m;
generate
  for (i=0; i<SIZE; i=i+1) begin:B1     // scope B1[i]
    M1 N1();                            // instantiates B1[i].N1
    for (j=0; j<SIZE; j=j+1) begin:B2   // scope B1[i].B2[j]
      M2 N2();                          // instantiates B1[i].B2[j].N2
      for (k=0; k<SIZE; k=k+1) begin:B3 // scope B1[i].B2[j].B3[k]
        M3 N3();                        // instantiates B1[i].B2[j].B3[k].N3
      end
    end
    if (i>0) begin:B4                   // scope B1[i].B4
      for (m=0; m<SIZE; m=m+1) begin:B5 // scope B1[i].B4.B5[m]
        M4 N4();                        // instantiates B1[i].B4.B5[m].N4
      end
    end
  end
endgenerate

  // Some examples of hierarchical names for the module instances:
  // B1[0].N1              B1[1].N1
  // B1[0].B2[0].N2        B1[0].B2[1].N2
  // B1[0].B2[0].B3[0].N3  B1[0].B2[0].B3[1].N3
  // B1[0].B2[1].B3[0].N3
  // B1[1].B4.B5[0].N4     B1[1].B4.B5[1].N4
```

12.4.2 Conditional generate constructs

The conditional generate constructs, if-generate and case-generate, select at most one generate block from a set of alternative generate blocks based on constant expressions evaluated during elaboration. The selected generate block, if any, is instantiated into the model.

Generate blocks in conditional generate constructs can be named or unnamed, and they may consist of only one item, which need not be surrounded by **begin**/**end** keywords. Even if the **begin**/**end** keywords are absent, it is still a generate block, which, like all generate blocks, comprises a separate scope and a new level of hierarchy when it is instantiated.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

Because at most one of the alternative generate blocks is instantiated, it is permissible for there to be more than one block with the same name within a single conditional generate construct. It is not permissible for any of the named generate blocks to have the same name as generate blocks in any other conditional or loop generate construct in the same scope, even if the blocks with the same name are not selected for instantiation. It is not permissible for any of the named generate blocks to have the same name as any other declaration in the same scope, even if that block is not selected for instantiation.

If the generate block selected for instantiation is named, then this name declares a generate block instance and is the name for the scope it creates. Normal rules for hierarchical naming apply. If the generate block selected for instantiation is not named, it still creates a scope; but the declarations within it cannot be referenced using hierarchical names other than from within the hierarchy instantiated by the generate block itself.

If a generate block in a conditional generate construct consists of only one item that is itself a conditional generate construct and if that item is not surrounded by **begin**/**end** keywords, then this generate block is not treated as a separate scope. The generate construct within this block is said to be directly nested. The generate blocks of the directly nested construct are treated as if they belong to the outer construct. Therefore, they can have the same name as the generate blocks of the outer construct, and they cannot have the same name as any declaration in the scope enclosing the outer construct (including other generate blocks in other generate constructs in that scope). This allows complex conditional generate schemes to be expressed without creating unnecessary levels of generate block hierarchy.

The most common use of this would be to create an **if-else-if** generate scheme with any number of **else-if** clauses, all of which can have generate blocks with the same name because only one will be selected for instantiation. It is permissible to combine if-generate and case-generate constructs in the same complex generate scheme. Direct nesting applies only to conditional generate constructs nested in conditional generate constructs. It does not apply in any way to loop generate constructs.

Example 1

```
module test;
parameter p = 0, q = 0;
wire a, b, c;

//-----------------------------------------------------------
// Code to either generate a u1.g1 instance or no instance.
// The u1.g1 instance of one of the following gates:
// (and, or, xor, xnor) is generated if
// {p,q} == {1,0}, {1,2}, {2,0}, {2,1}, {2,2}, {2, default}
//-----------------------------------------------------------

if (p == 1)
  if (q == 0)
    begin : u1           // If p==1 and q==0, then instantiate
      and g1(a, b, c); // AND with hierarchical name test.u1.g1
    end
  else if (q == 2)
    begin : u1           // If p==1 and q==2, then instantiate
      or  g1(a, b, c); // OR with hierarchical name test.u1.g1
    end
                  // "else" added to end "if (q == 2)" statement
  else ;             // If p==1 and q!=0 or 2, then no instantiation
else if (p == 2)
  case (q)
  0, 1, 2:
    begin : u1          // If p==2 and q==0,1, or 2, then instantiate
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
      xor g1(a, b, c);// XOR with hierarchical name test.u1.g1
    end
  default:
    begin : u1          // If p==2 and q!=0,1, or 2, then instantiate
      xnor g1(a, b, c);// XNOR with hierarchical name test.u1.g1
    end
  endcase

endmodule
```

This generate construct will select at most one of the generate blocks named `u1`. The hierarchical name of the gate instantiation in that block would be `test.u1.g1`. When nesting if-generate constructs, the **else** always belongs to the nearest **if** construct.

NOTE—As in the example above, an **else** with a null generate block can be inserted to make a subsequent **else** belong to an outer **if** construct. **begin**/**end** keywords can also be used to disambiguate. However, this would violate the criteria for direct nesting, and an extra level of generate block hierarchy would be created.

Conditional generate constructs make it possible for a module to contain an instantiation of itself. The same can be said of loop generate constructs, but it is more easily done with conditional generates. With proper use of parameters, the resulting recursion can be made to terminate, resulting in a legitimate model hierarchy. Because of the rules for determining top-level modules, a module containing an instantiation of itself will not be a top-level module.

Example 2—An implementation of a parameterized multiplier module

```
module multiplier(a,b,product);
parameter a_width = 8, b_width = 8;
localparam product_width = a_width+b_width;
                  // cannot be modified directly with the defparam
                  // statement or the module instance statement #
input  [a_width-1:0] a;
input  [b_width-1:0] b;
output [product_width-1:0] product;

generate
  if((a_width < 8) || (b_width < 8)) begin: mult
    CLA_multiplier #(a_width,b_width) u1(a, b, product);
    // instantiate a CLA  multiplier
  end
  else begin: mult
    WALLACE_multiplier #(a_width,b_width) u1(a, b, product);
    // instantiate a Wallace-tree  multiplier
  end
endgenerate
// The hierarchical instance name is mult.u1

endmodule
```

Example 3—Generate with a case to handle widths less than 3

```
generate
  case (WIDTH)
    1: begin: adder              // 1-bit adder implementation
         adder_1bit x1(co, sum, a, b, ci);
       end
    2: begin: adder              // 2-bit adder implementation
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
        adder_2bit x1(co, sum, a, b, ci);
      end
    default:
      begin: adder              // others - carry look-ahead adder
        adder_cla #(WIDTH) x1(co, sum, a, b, ci);
      end
  endcase
// The hierarchical instance name is adder.x1

endgenerate
```

Example 4—A module of memory dimm

```
module dimm(addr, ba, rasx, casx, csx, wex, cke, clk, dqm, data, dev_id);

  parameter [31:0] MEM_WIDTH = 16, MEM_SIZE  = 8; // in mbytes

  input [10:0] addr;
  input        ba, rasx, casx, csx, wex, cke, clk;
  input [ 7:0] dqm;
  inout [63:0] data;
  input [ 4:0] dev_id;

  genvar       i;

    case ({MEM_SIZE, MEM_WIDTH})
      {32'd8, 32'd16}: // 8Meg x 16 bits wide
      begin: memory
        for (i=0; i<4; i=i+1) begin:word
          sms_08b216t0 p(.clk(clk), .csb(csx), .cke(cke),.ba(ba),
                         .addr(addr), .rasb(rasx), .casb(casx),
                         .web(wex), .udqm(dqm[2*i+1]), .ldqm(dqm[2*i]),
                         .dqi(data[15+16*i:16*i]), .dev_id(dev_id));
        // The hierarchical instance names are memory.word[3].p,
        // memory.word[2].p, memory.word[1].p, memory.word[0].p,
        // and the task memory.read_mem
        end
        task read_mem;
          input  [31:0] address;
          output [63:0] data;
          begin                  // call read_mem in sms module
            word[3].p.read_mem(address, data[63:48]);
            word[2].p.read_mem(address, data[47:32]);
            word[1].p.read_mem(address, data[31:16]);
            word[0].p.read_mem(address, data[15: 0]);
          end
        endtask
      end

      {32'd16, 32'd8}: // 16Meg x 8 bits wide
      begin: memory
        for (i=0; i<8; i=i+1) begin:byte
          sms_16b208t0 p(.clk(clk), .csb(csx), .cke(cke),.ba(ba),
                         .addr(addr), .rasb(rasx), .casb(casx),
                         .web(wex), .dqm(dqm[i]),
                         .dqi(data[7+8*i:8*i]), .dev_id(dev_id));
        // The hierarchical instance names are memory.byte[7].p,
        // memory.byte[6].p, ... , memory.byte[1].p, memory.byte[0].p,
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
        // and the task memory.read_mem
        end
        task read_mem;
          input  [31:0] address;
          output [63:0] data;
          begin                  // call read_mem in sms module
            byte[7].p.read_mem(address, data[63:56]);
            byte[6].p.read_mem(address, data[55:48]);
            byte[5].p.read_mem(address, data[47:40]);
            byte[4].p.read_mem(address, data[39:32]);
            byte[3].p.read_mem(address, data[31:24]);
            byte[2].p.read_mem(address, data[23:16]);
            byte[1].p.read_mem(address, data[15: 8]);
            byte[0].p.read_mem(address, data[ 7: 0]);
          end
        endtask
      end
      // Other memory cases ...
    endcase
endmodule
```

12.4.3 External names for unnamed generate blocks

Although an unnamed generate block has no name that can be used in a hierarchical name, it needs to have a name by which external interfaces can refer to it. A name will be assigned for this purpose to each unnamed generate block as described in the next paragraph.

Each generate construct in a given scope is assigned a number. The number will be 1 for the construct that appears textually first in that scope and will increase by 1 for each subsequent generate construct in that scope. All unnamed generate blocks will be given the name “genblk<n>” where <n> is the number assigned to its enclosing generate construct. If such a name would conflict with an explicitly declared name, then leading zeroes are added in front of the number until the name does not conflict.

NOTE—Each generate construct is assigned its number as described in the previous paragraph even if it does not contain any unnamed generate bocks.

For example:

```
module top;

  parameter genblk2 = 0;
  genvar i;

  // The following generate block is implicitly named genblk1

  if (genblk2) reg a;  // top.genblk1.a
  else         reg b;  // top.genblk1.b

  // The following generate block is implicitly named genblk02
  // as genblk2 is already a declared identifier

  if (genblk2) reg a;  // top.genblk02.a
  else         reg b;  // top.genblk02.b

  // The following generate block would have been named genblk3
  // but is explicitly named g1
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
  for (i = 0; i < 1; i = i + 1) begin : g1    // block name
    // The following generate block is implicitly named genblk1
    // as the first nested scope inside of g1
    if (1)      reg a;  // top.g1[0].genblk1.a
  end

  // The following generate block is implicitly named genblk4 since
  // it belongs to the fourth generate construct in scope "top".
  // The previous generate block would have been
  // named genblk3 if it had not been explicitly named g1

  for (i = 0; i < 1; i = i + 1)
    // The following generate block is implicitly named genblk1
    // as the first nested generate block in genblk4
    if (1)      reg a;  // top.genblk4[0].genblk1.a


  // The following generate block is implicitly named genblk5
  if (1)        reg a;  // top.genblk5.a

endmodule
```

12.5 Hierarchical names

Every identifier in a Verilog HDL description shall have a unique *hierarchical path name*. The hierarchy of modules and the definition of items such as tasks and named blocks within the modules shall define these names. The hierarchy of names can be viewed as a tree structure, where each module instance, generate block instance, task, function, or named **begin**-**end** or **fork**-**join** block defines a new hierarchical level, or *scope*, in a particular branch of the tree.

A design description contains one or more top-level modules (see 12.1.1). Each such module forms the top of a name hierarchy. This root or these parallel root modules make up one or more hierarchies in a *design description* or *description*. Inside any module, each module instance (including an arrayed instance), generate block instance, task definition, function definition, and named **begin**-**end** or **fork**-**join** block shall define a new branch of the hierarchy. Named blocks within named blocks and within tasks and functions shall create new branches. Unnamed generate blocks are exceptions. They create branches that are visible only from within the block and within any hierarchy instantiated by the block. See 12.4.3 for a discussion of unnamed generate blocks.

Each node in the hierarchical name tree shall be a separate scope with respect to identifiers. A particular identifier can be declared at most once in any scope. See 12.7 for a discussion of scope rules and 4.11 for a discussion of name spaces.

Any named Verilog object or *hierarchical name reference* can be referenced uniquely in its full form by concatenating the names of the modules, module instance names, generate blocks, tasks, functions, or named blocks that contain it. The period character shall be used to separate each of the names in the hierarchy, except for escaped identifiers embedded in the hierarchical name reference, which are followed by separators composed of white space and a period-character. The complete path name to any object shall start at a top-level (root) module. This path name can be used from any level in the hierarchy or from a parallel hierarchy. The first node name in a path name can also be the top of a hierarchy that starts at the level where the path is being used (which allows and enables downward referencing of items). Objects declared in automatic tasks and functions are exceptions and cannot be accessed by hierarchical name references. Objects declared in unnamed generate blocks are also exceptions. They can be referenced by hierarchical names only from within the block and within any hierarchy instantiated by the block.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Names in a hierarchical path name that refer to instance arrays or loop generate blocks may be followed immediately by a constant expression in square brackets. This expression selects a particular instance of the array and is, therefore, called an *instance select*. The expression shall evaluate to one of the legal index values of the array. If the array name is not the last path element in the hierarchical name, the instance select expression is required.

The syntax for hierarchical path names is given in Syntax 12-6.

```
escaped_identifier ::= (From A.9.3)
      \ {Any_ASCII_character_except_white_space} white_space
hierarchical_identifier ::=
       { identifier [ [ constant_expression ] ] . } identifier
identifier ::=
       simple_identifier
     | escaped_identifier
simple_identifier[a] ::= [ a-zA-Z_ ] { [ a-zA-Z0-9_$ ] }
white_space ::= (From A.9.4)
      space | tab | newline | eof[b]
```

[a]A `simple_identifier` shall start with an alpha or underscore (_) character, shall have at least one character, and shall not have any spaces.
[b]End of file.

Syntax 12-6—Syntax for hierarchical path names

For example:

Example 1—The code in this example defines a hierarchy of module instances and named blocks.

```
module mod (in);
input in;

always @(posedge in) begin : keep
reg hold;
   hold = in;
end
endmodule

module wave;
reg stim1, stim2;

cct a(stim1, stim2); // instantiate cct

initial begin :wave1
   #100   fork :innerwave
             reg hold;
          join
   #150   begin
             stim1 = 0;
          end
end
endmodule
```

```
module cct (stim1, stim2);
input stim1, stim2;

// instantiate mod
mod amod(stim1), bmod(stim2);
endmodule
```

Figure 12-1 illustrates the hierarchy implicit in this Verilog code.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

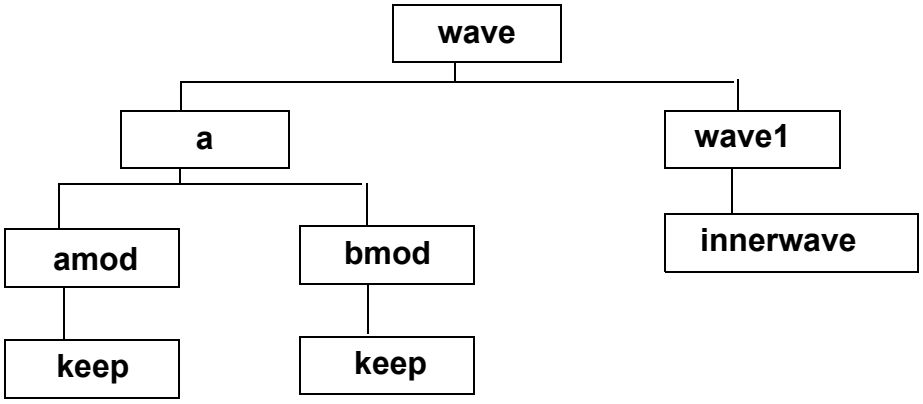


Figure 12-1—Hierarchy in a model

Figure 12-2 is a list of the hierarchical forms of the names of all the objects defined in the code.

```
wave                        wave.a.bmod
wave.stim1                  wave.a.bmod.in
wave.stim2                  wave.a.bmod.keep
wave.a                      wave.a.bmod.keep.hold
wave.a.stim1                wave.wave1
wave.a.stim2                wave.wave1.innerwave
wave.a.amod                 wave.wave1.innerwave.hold
wave.a.amod.in
wave.a.amod.keep
wave.a.amod.keep.hold
```

Figure 12-2—Hierarchical path names in a model

Hierarchical name referencing allows free data access to any object from any level in the hierarchy. If the unique hierarchical path name of an item is known, its value can be sampled or changed from anywhere within the description.

Example 2—The next example shows how a pair of named blocks can refer to items declared within each other.

```
begin
    fork :mod_1
        reg x;
        mod_2.x = 1;
    join
    fork :mod_2
        reg x;
        mod_1.x = 0;
    join
end
```

12.6 Upwards name referencing

The name of a module or module instance is sufficient to identify the module and its location in the hierarchy. A lower level module can reference items in a module above it in the hierarchy. Variables can be

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

referenced if the name of the higher level module or its instance name is known. For tasks, functions, named blocks, and generate blocks, Verilog shall look in the enclosing module for the name until it is found or until the root of the hierarchy is reached. It shall only search in higher enclosing modules for the name, not instances.

The syntax for an upward reference is given in Syntax 12-7.

```
upward_name_reference ::=
        module_identifier.item_name
item_name ::=
          function_identifier
        | block_identifier
        | net_identifier
        | parameter_identifier
        | port_identifier
        | task_identifier
        | variable_identifier
```

Syntax 12-7—Syntax for upward name referencing

Upward name references can also be done with names of the form

```
scope_name.item_name
```

where `scope_name` is either a module instance name or a generate block name. A name of this form shall be resolved as follows:

a) Look in the current scope for a scope named `scope_name`. If not found and the current scope is not the module scope, look for the name in the enclosing scope, repeating as necessary until the name is found or the module scope is reached. If still not found, proceed to step b). Otherwise, this name reference shall be treated as a downward reference from the scope in which the name is found.
b) Look in the parent module's outermost scope for a scope named scope_name. If found, the item name shall be resolved from that scope.
c) Repeat step b), going up the hierarchy.

There is an exception to these rules for hierarchical names on the left-hand side of **defparam** statements. See 12.8 for details.

For example:

In this example, there are four modules, `a`, `b`, `c`, and `d`. Each module contains an integer `i`. The highest level modules in this segment of a model hierarchy are `a` and `d`. There are two copies of module `b` because module `a` and `d` instantiate `b`. There are four copies of `c.i` because each of the two copies of `b` instantiates `c` twice.

```
module a;
integer i;

b   a_b1();

endmodule
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
module b;
integer i;

c  b_c1(), b_c2();

initial                 // downward path references two copies of i:
   #10 b_c1.i = 2;      // a.a_b1.b_c1.i, d.d_b1.b_c1.i

endmodule


module c;
integer i;

initial begin           // local name references four copies of i:
   i = 1;               // a.a_b1.b_c1.i, a.a_b1.b_c2.i,
                        // d.d_b1.b_c1.i, d.d_b1.b_c2.i
   b.i = 1;             // upward path references two copies of i:
                        // a.a_b1.i, d.d_b1.i
   end

endmodule


module d;
integer i;

b d_b1();

initial begin           // full path name references each copy of i
   a.i = 1;                     d.i = 5;
   a.a_b1.i = 2;                d.d_b1.i = 6;
   a.a_b1.b_c1.i = 3;           d.d_b1.b_c1.i = 7;
   a.a_b1.b_c2.i = 4;           d.d_b1.b_c2.i = 8;
end

endmodule
```

12.7 Scope rules

The following elements define a new scope in Verilog:

- Modules
- Tasks
- Functions
- Named blocks
- Generate blocks

An identifier shall be used to declare only one item within a scope. This rule means it is illegal to declare two or more variables that have the same name, or to name a task the same as a variable within the same module, or to give a gate instance the same name as the name of the net connected to its output. For generate blocks, this rule applies regardless of whether the generate block is instantiated. An exception to this is made for generate blocks in a conditional generate construct. See 12.4.3 for a discussion of naming conditional generate blocks.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

If an identifier is referenced directly (without a hierarchical path) within a task, function, named block, or generate block, it shall be declared either within the task, function, named block, or generate block locally or within a module, task, function, named block, or generate block that is higher in the same branch of the name tree that contains the task, function, named block, or generate block. If it is declared locally, then the local item shall be used; if not, the search shall continue upward until an item by that name is found or until a module boundary is encountered. If the item is a variable, it shall stop at a module boundary; if the item is a task, function, named block, or generate block, it continues to search higher level modules until found. This fact means that tasks and functions can use and modify the variables within the containing module by name, without going through their ports.

If an identifier is referenced with a hierarchical name, the path can start with a module name, instance name, task, function, named block, or named generate block. The names shall be searched first at the current level and then in higher level modules until found. Because both module names and instance names can be used, precedence is given to instance names if there is a module named the same as an instance name.

Because of the upward searching, path names that are not strictly on a downward path can be used.

For example:

Example 1—In Figure 12-3, each rectangle represents a local scope. The scope available to upward searching extends outward to all containing rectangles—with the boundary of the module A as the outer limit. Thus block G can directly reference identifiers in F, E, and A; it cannot directly reference identifiers in H, B, C, and D.

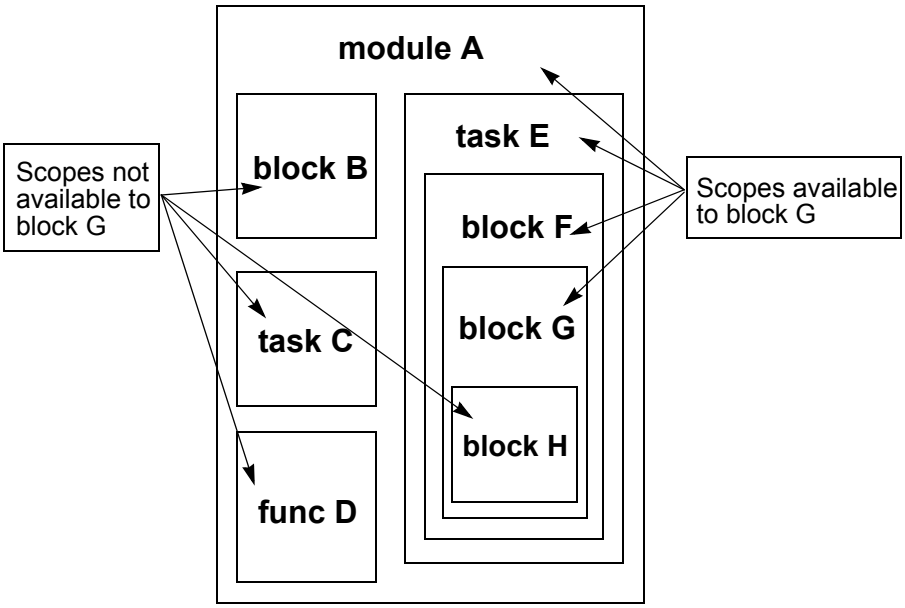


Figure 12-3—Scopes available to upward name referencing

Example 2—The following example shows how variables can be accessed directly or with hierarchical names:

```
task t;
reg s;
begin : b
  reg r;
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
    t.b.r = 0;// These three lines access the same variable r
    b.r = 0;
    r = 0;

    t.s = 0;// These two lines access the same variable s
    s = 0;
  end
  endtask
```

12.8 Elaboration

Elaboration is the process that occurs between parsing and simulation. It binds modules to module instances, builds the model hierarchy, computes parameter values, resolves hierarchical names, establishes net connectivit, and prepares all of this for simulation. With the addition of generate constructs, the order in which these tasks occur becomes significant.

12.8.1 Order of elaboration

Because of generate constructs, the model hierarchy can depend on parameter values. Because **defparam** statements can alter parameter values from almost anywhere in the hierarchy, the result of elaboration can be ambiguous when generate constructs are involved. The final model hierarchy can depend on the order in which **defparams** and generate constructs are evaluated.

The following algorithm defines an order that produces the correct hierarchy:

a) A list of starting points is initialized with the list of top-level modules.
b) The hierarchy below each starting point is expanded as much as possible without elaborating generate constructs. All parameters encountered during this expansion are given their final values by applying initial values, parameter overrides, and **defparam** statements.

   In other words, any **defparam** statement whose target can be resolved within the hierarchy elaborated so far must have its target resolved and its value applied. **defparam** statements whose target cannot be resolved are deferred until the next iteration of this step. Because no **defparam** inside the hierarchy below a generate construct is allowed to refer to a parameter outside the generate construct, it is possible for parameters to get their final values before going to step c).
c) Each generate construct encountered in step b) is revisited, and the generate scheme is evaluated. The resulting generate block instantiations make up the new list of starting points. If the new list of starting points is not empty, go to step b).

12.8.2 Early resolution of hierarchical names

In order to comply with this algorithm, hierarchical names in some **defparam** statements will need to be resolved prior to the full elaboration of the hierarchy. It is possible that when elaboration is complete, rules for name resolution would dictate that a hierarchical name in a **defparam** statement would have resolved differently had early resolution not been required. This could result in a situation where an identical hierarchical name in some other statement in the same scope would resolve differently from the one in the **defparam** statement. Below is an example of a design that has this problem:

```
module m;
  m1 n();
endmodule

module m1;
  parameter p = 2;
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
  defparam m.n.p = 1;
  initial $display(m.n.p);

  generate
    if (p == 1) begin : m
      m2 n();
    end
  endgenerate
endmodule

module m2;
  parameter p = 3;
endmodule
```

In this example, the **defparam** must be evaluated before the conditional generate is elaborated. At this point in elaboration, the name resolves to **parameter** `p` in module `mid1`, and this parameter is used in the generate scheme. The result of the **defparam** is to set that parameter to 1; therefore, the generate condition is true. After the hierarchy below the generate construct is elaborated, the rules for hierarchical name resolution would dictate that the name should have resolved to **parameter** `p` in module `mid2`. In fact, the identical name in the **$display** statement will resove to that other parameter.

It shall be an error if a hierarchical name in a **defparam** is resolved before the hierarchy is completely elaborated and that name would resolve differently once the model is completely elaborated.

This situation will occur very rarely. In order to cause the error, there has to be a named generate block that has the same name as one of the scopes in its full hierarchical name. Furthermore, there have to be two instances with the same name, one in the generate block and one in the other scope with the same name as the generate block. Then, inside these instances there have to be parameters with the same name. If this problem occurs, it can be easily fixed by changing the name of the generate block.

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

13. Configuring the contents of a design

13.1 Introduction

To facilitate both the sharing of Verilog designs between designers and/or design groups and the repeatability of the exact contents of a given simulation (or other tool) session, the concept of *configurations* is used in the Verilog language. A configuration is simply an explicit set of rules to specify the exact source description to be used to represent each instance in a design. The operation of selecting a source representation for an instance is referred to as *binding* the instance.

The example below shows a simple configuration problem.

For example:

```
file top.v                file adder.v              file adder.vg

module top();             module adder(...);        module adder(...);
adder a1(...);            // rtl adder              // gate-level adder
adder a2(...);            // description            // description
endmodule                 ...                       ...
                          endmodule                 endmodule
```

Consider using the `rtl adder` description in `adder.v` for instance `a1` in module top and the gate-level adder description in `adder.vg` for instance `a2`. In order to specify this particular set of instance bindings and to avoid having to change the source description to specify a new set, a configuration can be used.

```
config cfg1; // specify rtl adder for top.a1, gate-level adder for top.a2
    design rtlLib.top;
    default liblist rtlLib;
    instance top.a2 liblist gateLib;
endconfig
```

The elements of a config are explained in subsequent subclauses, but this simple example illustrates some important points about configs. As evidenced by the **config-endconfig** syntax, the config is a design element, similar to a module, which exists in the Verilog name space. The config contains a set of rules that are applied when searching for a source description to bind to a particular instance of the design.

A Verilog design description starts with a top-level module (or modules) (see 12.1.1). From this module's source description, the instantiated modules (or children) are found, then the source descriptions for the module definitions of these subinstances shall be located, and so on until every instance in the design is mapped to a source description.

13.1.1 Library notation

In order to map a Verilog instance to a source description, the concept of a symbolic *library*, which is simply a logical collection of design elements (such as modules, primitives, or configs), can be used. These design elements can be referred to as *cell*s. The cell name shall be the same as the name of the module/primitive/config being processed. Syntax 13-1 specifies a cell from a given library.

```
library_cell ::=
    [library_identifier.]cell_identifier[:config]
```

Syntax 13-1—Syntax for cell

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

This notation gives a symbolic method of referring to source descriptions; the method of mapping source descriptions into libraries is shown in greater detail in 13.2.1. The optional `:config` extension shall be used explicitly to refer to a config in the case where a config has the same name as a module/primitive.

For the purposes of this example, suppose the files `top.v` and `adder.v` (i.e., the RTL descriptions) have been mapped into the library `rtlLib` and the file `adder.vg` (i.e., the gate-level description of the `adder`) has been mapped into the library `gateLib`. The actual mechanism for mapping source descriptions to libraries is detailed in 13.2.

13.1.2 Basic configuration elements

The **design** statement in `config cfg1` of the first example of 13.1 specifies the top-level module in the design and what source description is to be used. In this example, the `rtlLib.top` notation indicates the top-level module description shall be taken from `rtlLib`. Because `top.v` and `adder.v` were mapped to this library, the actual description for the module is known to come from `top.v`.

The **default** statement coupled with the **liblist** clause specifies, by default, all subinstances of top (i.e., `top.a1` and `top.a2`) shall be taken from `rtlLib`, which means the descriptions in `top.v` and `adder.v`, which were mapped to this library, shall be used. For a basic design, which can be completely `rtl`, this can be sufficient to specify completely the binding for the entire design. However, here the `top.a2` instance of adder to the gate-level description shall be bound.

The **instance** statement specifies, for the particular instance `top.a2`, the source description shall be taken from `gateLib`. The instance statement overrides the default rule for this particular instance. Because `adder.vg` was mapped to `gateLib`, this statement dictates the gate-level description in `adder.vg` be used for instance `top.a2`.

13.2 Libraries

As mentioned in the previous subclause, a library is a logical collection of cells that are mapped to particular source description files. The symbolic *lib*`.`*cell*[`:config`] notation supports the separate compilation of source files by providing a file-system-independent name to refer to source descriptions when instances in a design are bound. It also allows multiple tools, which can have different invocation use models, to share the same configuration.

13.2.1 Specifying libraries—the library map file

When parsing a source description file (or files), the parser shall first read the library mapping information from a predefined file prior to reading any source files. The name of this file and the mechanism for reading it shall be tool-specific, but all compliant tools shall provide a mechanism to specify one or more library map files to be used for a particular invocation of the tool. If multiple map files are specified, then they shall be read in the order in which they are specified.

For the purposes of this discussion, assume the existence of a file named `lib.map` in the current working directory, which is automatically read by the parser prior to parsing any source files specified on the command line. The syntax for declaring a library in the library map file is shown in Syntax 13-2.

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
library_text ::= (From A.1.1)
      { library_description }
library_description ::=
        library_declaration
      | include_statement
      | config_declaration
library_declaration ::=
      library library_identifier file_path_spec [ { , file_path_spec } ]
        [ -incdir file_path_spec { , file_path_spec } ] ;
include_statement ::=
      include file_path_spec ;
```

Syntax 13-2—Syntax for declaring library in library map file

Library map file details

1—file_path_spec uses file-system-specific notation to specify an absolute or relative path to a particular file or set of files. The following shortcuts/wildcards can be used:

`?`	single character wildcard (matches any single character)
`*`	multiple character wildcard (matches any number of characters in a directory/file name)
`...`	hierarchical wildcard (matches any number of hierarchical directories)
`..`	specifies the parent directory
`.`	specifies the directory containing the `lib.map`

Paths that end in `/` shall include all files in the specified directory. Identical to `/*`.
Paths that do not begin with `/` are relative to the directory in which the current lib.map file is located.

2—The paths `./*.v` and `*.v` are identical, and both specify all files with a `.v` suffix in the current directory.

Any file encountered by the compiler that does not match any library's file_path_spec shall by default be compiled into a library named `work`.

To perform the library mapping discussed in the example in 13.1, use the following library definitions in the `lib.map` file:

```
library rtlLib *.v;          // matches all files in the current directory with a .v suffix
library gateLib ./*.vg;      // matches all files in the current directory with a .vg suffix
```

13.2.1.1 File path resolution

If a file name potentially matches multiple file path specifications, the path specifications shall be resolved in the following order:

a) File path specifications that end with an explicit filename
b) File path specifications that end with a wildcarded filename
c) File path specifications that end with a directory

If a file name matches path specifications in multiple library definitions (after the above resolution rules have been applied), it shall be an error.

Using these rules with the library definitions in the `lib.map` file, all source files encountered by the parser/compiler can be mapped to a unique library. Once the source descriptions have been mapped to libraries, the cells defined in those libraries are available for binding.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

NOTE—Tool implementers may find it convenient to provide a command-line argument to explicitly specify the library into which the file being parsed is to be mapped, which shall override any library definitions in the `lib.map` file. If these libraries do not exist in the `lib.map` file, they can only be accessed via an explicit config.

If multiple cells with the same name map to the same library, then the `LAST` cell encountered shall be written to the library. This is to support a "separate-compile" use model (see 13.4.3), where it is assumed that encountering a cell after it has previously been compiled is intended to be a recompiling of the cell. In the case where multiple modules with the same name are mapped to the same library in a single invocation of the compiler, then a warning message shall be issued.

13.2.2 Using multiple library map files

In addition to specifying library mapping information, a `lib.map` file can also include references to other `lib.map` files. The **include** command is used to insert the entire contents of a library map file in another file during parsing. The result is as though the contents of the included map file appear in place of the **include** command.

The syntax of a `lib.map` file is limited to library specifications, include statements, and standard Verilog comment syntax. Syntax 13-3 shows the syntax for the **include** command.

```
include_statement ::= (From A.1.1)
    include file_path_spec ;
```

Syntax 13-3—Syntax for include command

If the file path specification, whether in an include or library statement, describes a relative path, it shall be relative to the location of the file that contains the file path. Library providers shall include a local library map file in addition to the source contents of the library. Individual users can then simply include the provider's library map file in their own map file to gain access to the contents of the provided library.

13.2.3 Mapping source files to libraries

For each cell definition encountered during parsing/compiling, the name of the source file being parsed is compared to the file path specifications of the library declarations in all of the library map files being used. The cell is mapped into the library whose file path specification matches the source file name.

13.3 Configurations

As mentioned in the introduction of this clause, a configuration is simply a set of rules to apply when searching for library cells to which to bind instances. The syntax for configurations is shown in 13.3.1.

13.3.1 Basic configuration syntax

The configuration syntax is shown in Syntax 13-4.

13.3.1.1 Design statement

The **design** statement names the library and cell of the top-level module or modules in the design hierarchy configured by the config. There shall be one and only one design statement, but multiple top-level modules can be listed in the design statement. The cell or cells identified cannot be configurations themselves. It is possible the design identified can have the same name as configs, however.

The **design** statement shall appear before any config rule statements in the config.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
config_declaration ::= (From A.1.5)
      config config_identifier ;
      design_statement
      {config_rule_statement}
      endconfig
design_statement ::=
      design { [library_identifier.]cell_identifier } ;
config_rule_statement ::=
        default_clause liblist_clause ;
      | inst_clause liblist_clause ;
      | inst_clause use_clause  ;
      | cell_clause liblist_clause ;
      | cell_clause use_clause ;
```

Syntax 13-4—Syntax for configuration

If the library identifier is omitted, then the library that contains the config shall be used to search for the cell.

13.3.1.2 The default clause

The syntax for the **default** clause is specified in Syntax 13-5.

```
default_clause ::= (From A.1.5)
      default
```

Syntax 13-5—Syntax for default clause

The **default** clause selects all instances that do not match a more specific selection clause. The **use** expansion clause (see 13.3.1.6) cannot be used with a **default** selection clause. For other expansion clauses, there cannot be more than one **default** clause that specifies the expansion clause.

For simple design configurations, it might be sufficient to specify a **default** *liblist* (see 13.3.1.5).

13.3.1.3 The instance clause

The **instance** clause is used to specify the specific instance to which the expansion clause shall apply. The syntax for the **instance** clause is specified in Syntax 13-6.

```
inst_clause ::= (From A.1.5)
      instance inst_name
inst_name ::=
      topmodule_identifier{.instance_identifier}
```

Syntax 13-6—Syntax for instance clause

The instance name associated with the **instance** clause is a Verilog hierarchical name, starting at the top-level module of the config (i.e., the name of the cell in the **design** statement).

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

13.3.1.4 The cell clause

The **cell** selection clause names the cell to which it applies. The syntax for the **cell** clause is specified in Syntax 13-7.

```
cell_clause ::= (From A.1.5)
    cell [ library_identifier.]cell_identifier
```

Syntax 13-7—Syntax for cell clause

If the optional library name is specified, then the selection rule applies to any instance that is bound or is under consideration for being bound to the selected library and cell. It is an error if a library name is included in a **cell** selection clause and the corresponding expansion clause is a library list expansion clause.

13.3.1.5 The liblist clause

The **liblist** clause defines an ordered set of libraries to be searched to find the current instance. The syntax for the **liblist** clause is specified in Syntax 13-8.

```
liblist_clause ::= (From A.1.5)
    liblist { library_identifier }
```

Syntax 13-8—Syntax for liblist clause

*liblist*s are inherited hierarchically downward as instances are bound. When searching for a cell to bind to the current unbound instance, and in the absence of an applicable binding expansion clause, the specified library list is searched in the specified order.

The current library list is selected by the selection clauses. If no library list clause is selected or if the selected library list is empty, then the library list contains the single name that is the library in which the cell containing the unbound instance is found (i.e., the parent cell's library).

13.3.1.6 The use clause

The **use** clause specifies a specific binding for the selected cell. The syntax for the **use** clause is specified in Syntax 13-9.

```
use_clause ::= (From A.1.5)
    use [library_identifier.]cell_identifier[:config]
```

Syntax 13-9—Syntax for use clause

A **use** clause can only be used in conjunction with an **instance** or **cell** selection clause. It specifies the exact library and cell to which a selected cell or instance is bound.

The **use** clause has no effect on the current value of the library list. It can be common in practice to specify multiple config rule statements, one of which specifies a binding and the other of which specifies a library list.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

If the lib.cell to which the **use** clause refers is a config that has the same name as a module/primitive in the same library, then the optional `:config` suffix can be added to the `lib.cell` to specify the config explicitly.

If the library name is omitted, the library shall be inherited from the parent cell.

NOTE—The binding statement can create situations where the unbound instance's module name and the cell name to which it is bound are different.

13.3.2 Hierarchical configurations

For situations where it is desirable to specify a special set of configuration rules for a subsection of a design, it is possible to bind a particular instance directly to a configuration using the binding clause:

```
instance top.a1.foo use lib1.foo:config;
// bind to the config foo in library lib1
```

specifies the instance `top.a1.foo` is to be replaced with the design hierarchy specified by the configuration `lib1.foo:config`. The **design** statement in `lib1.foo:config` shall specify the actual binding for the instance `top.a1.foo`, and the rules specified in the config shall determine the configuration of all other subinstances under `top.a1.foo`.

It shall be an error for an instance clause to specify a hierarchical path to an instance that occurs within a hierarchy specified by another config.

```
config bot;
    design lib1.bot;
    default liblist lib1 lib2;
    instance bot.a1 liblist lib3;
endconfig

config top;
    design lib1.top;
    default liblist lib2 lib1;
    instance top.bot use lib1.bot:config;
    instance top.bot.a1 liblist lib4;
    // ERROR - cannot set liblist for top.bot.a1 from this config
endconfig
```

13.4 Using libraries and configs

This subclause describes potential use models for referencing configs on the command line. It is included for clarification purposes.

The traditional Verilog simulation use model takes a file-based approach, where the source descriptions for all cells in the design are specified on the command line for each invocation of the tool. With the advent of compiled-code simulators, the configuration mechanism shall also support a use model that allows for the source files to be precompiled and then for the precompiled design objects to be referenced on the command line. This subclause explains how configurations can be used in both of these scenarios.

13.4.1 Precompiling in a single-pass use model

The single-pass use model is the traditional use model with which most users are familiar. In this use model, all of the source description files shall be provided to the simulator via the command line, and only these source descriptions can be used to bind cell instances in the current design. A precompiling strategy in this

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

scenario actually parses every cell description provided on the command line and maps it into the library without regard to whether the cell actually is used in the design. The tool can optionally check to see whether the cell already exists in the library and, if it is up-to-date (i.e., the source description has not changed since the last time the cell was compiled), can skip recompiling the cell. After all cells on the command line have been compiled, then the tool can locate the top-level cell (discussed in Clause 12) and proceed down the hierarchy, binding each instance as it is encountered in the hierarchy.

NOTE—With this use model, it is not necessary for library objects to persist from one tool invocation to another (although for performance considerations it is recommended they do).

13.4.2 Elaboration-time compiling in a single-pass use model

An alternate strategy that can be used with a single-pass tool is to parse the source files only to find the top-level module(s), without actually compiling anything into the library during this scanning process. Once the top-level module(s) has been found, then it can be compiled into the library, and the tool can proceed down the hierarchy, only compiling the source descriptions necessary to bind the design successfully. Based on the binding rules in place, only the source files that match the current library specification need to be parsed to find the current cell's source description to compile. As with the precompiled single-pass use model, it is not necessary for library cells to persist from one invocation to another using this strategy.

13.4.3 Precompiling using a separate compilation tool

When using a separate compilation tool, it is essential that library cells persist, and the compiled forms shall, therefore, exist somewhere in the file system. The exact format and location for holding these compiled forms shall be vendor- or tool-specific. Using this separate compiler strategy, the source descriptions shall be parsed and compiled into the library using one or more invocations of the compiler tool. The only restriction is that all cells in a design shall be precompiled prior to binding the design (typically via an invocation of a separate tool). Using this strategy, the tool that actually does the binding only needs to be told the top-level module(s) of the design to be bound, and then it shall use the precompiled form of the cell description(s) from the library to determine the subinstances and descend hierarchically down the design, binding each cell as it is located.

13.4.4 Command line considerations

In each of the three preceding strategies, either the binding rules can be specified via a config, or the default rules (from the library map file) can be used. In the single-pass use models, the config can be specified by including its source description file on the command line. In the case where the config includes a design statement, then the specified cell shall be the top-level module, regardless of the presence of any uninstantiated cells in the rest of the source files. When using a separate compilation tool, the tool that actually does the binding only needs to be given the *lib.cell* specification for the top-level cell(s) and/or the config to be used. In this strategy, the config itself shall also be precompiled.

13.5 Configuration examples

Consider the following set of source descriptions:

```
file top.v
module top(...);
...
adder a1(...);
adder a2(...);
endmodule
module foo(...);
... // rtl
endmodule
```

```
file adder.v
module adder(...);
... // rtl
foo f1(...);
foo f2(...);
endmodule
module foo(...);
... // rtl
endmodule
```

```
file adder.vg
module adder(...);
... // gate-level
foo f1(...);
foo f2(...);
endmodule
module foo(...);
... // gate-level
endmodule
```

```
file lib.map
library rtlLib top.v;
library aLib adder.*;
library gateLib
        adder.vg;
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

All of the examples in this subclause shall assume the `top.v`, `adder.v` and `adder.vg` files get compiled with the given `lib.map` file. This yields the following library structure:

```
rtlLib.top // from top.v
rtlLib.foo // from top.v
aLib.adder // from adder.v
aLib.foo // rtl from adder.v
gateLib.adder // from adder.vg
gateLib.foo // from adder.vg
```

13.5.1 Default configuration from library map file

With no configuration, the libraries are searched according to the library declaration order in the library map file. In other words, all instances of module `adder` shall use `aLib.adder` (because `aLib` is the first library specified that contains a cell named `adder`), and all instances of module `foo` shall use `rtlLib.foo` (because `rtlLib` is the first library that contains `foo`).

13.5.2 Using default clause

To always use the foo definition from file `adder.v`, use the following simple configuration:

```
config cfg1;
    design rtlLib.top;
    default liblist aLib rtlLib;
endconfig
```

The **default liblist** statement overrides the library search order in the `lib.map` file; therefore, `aLib` is always searched before `rtlLib`. Because the `gateLib` library is not included in the `liblist`, the gate-level descriptions of `adder` and `foo` shall not be used.

To use the gate-level representations of `adder` and `foo`, add to the config as follows:

```
config cfg2;
    design rtlLib.top;
    default liblist gateLib aLib rtlLib;
endconfig
```

This shall cause the gate representation always to be taken before the `rtl` representation, using the module definitions for `adder` and `foo` from `adder.vg`. The `rtl` view of top shall be taken because there is no gate representation available.

13.5.3 Using cell clause

To modify the config to use the `rtl` view of `adder` and the gate-level representation of `foo` from `gateLib`, use the following:

```
config cfg3;
    design rtlLib.top;
    default liblist aLib rtlLib;
cell foo use gateLib.foo;
endconfig
```

The cell clause selects all cells named `foo` and explicitly binds them to the gate representation in `gateLib`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

13.5.4 Using instance clause

To modify the config so the `top.a1 adder` (and its descendants) use the gate representation and the `top.a2 adder` (and its descendants), use the `rtl` representation from `aLib`:

```
config cfg4
    design rtlLib.top;
    default liblist gateLib rtlLib;
    instance top.a2 liblist aLib;
endconfig
```

Because the **liblist** is inherited, all of the descendants of `top.a2` inherit its **liblist** from the instance selection clause.

13.5.5 Using hierarchical config

Now suppose all this work has only been on the adder module by itself and a config that uses the `rtlLib.foo` cell for `f1`, and the `gateLib.foo` cell for `f2` has already been developed. Then use the following:

```
config cfg5;
    design aLib.adder;
    default liblist gateLib aLib;
    instance adder.f1 liblist rtlLib;
endconfig
```

To use this configuration `cfg5` for the `top.a2` instance of `adder` and take the full default `aLib adder` for the `top.a1` instance, use the following config:

```
config cfg6;
    design rtlLib.top;
    default liblist aLib rtlLib;
    instance top.a2 use work.cfg5:config;
endconfig
```

The binding clause specifies the `work.cfg5:config` configuration is to be used to resolve the bindings of instance `top.a2` and its descendants. It is the design statement in config `cfg5` that defines the exact binding for the `top.a2` instance itself. The rest of `cfg5` defines the rules to bind the descendants of `top.a2`. Notice the instance clause in `cfg5` is relative to its own top-level module, `adder`.

13.6 Displaying library binding information

It shall be possible to display the actual library binding information for module instances during simulation. The format specifier `%l` or `%L` shall print out the `library.cell` binding information for the module instance containing the display (or other textual output) command. This is similar to the `%m` format specifier, which prints out the hierarchical path name of the module containing it.

It shall also be able to use VPI to display the binding information. The following VPI properties shall exist for objects of type `vpiModule`:

— `vpiLibrary`—the library name into which the module was compiled
— `vpiCell`—the name of the cell bound to the module instance
— `vpiConfig`—the `library.cell` name of the config controlling the binding of the module instance

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

These properties shall be of `string` type, similar to the `vpiName` and `vpiFullName` properties.

13.7 Library mapping examples

In the absence of a configuration, it is possible to perform basic control of the library searching order when binding a design.

When a config is used, the config overrides the rules specified in this subclause.

13.7.1 Using the command line to control library searching

In the absence of a configuration, it shall be necessary for all compliant tools to provide a mechanism of specifying a library search order on the command line that overrides the default order from the library map file. This mechanism shall include specification of library names only, with the definitions of these libraries to be taken from the library map file.

NOTE—It is recommended all compliant tools use "-L <library_name>" to specify this search order.

13.7.2 File path specification examples

For example:

Given the following set of files:

```
/proj/lib1/rtl/a.v
/proj/lib2/gates/a.v
/proj/lib1/rtl/b.v
/proj/lib2/gates/b.v
```

From the `/proj library`, the following absolute `file_path_specs` are resolved as shown:

```
/proj/lib*/*/a.v =/proj/lib1/rtl/a.v, /proj/lib2/gates/a.v
.../a.v =/proj/lib1/rtl/a.v, /proj/lib2/gates/a.v
/proj/.../b.v =/proj/lib1/rtl/b.v, /proj/lib2/gates/b.v
.../rtl/*.v =/proj/lib1/rtl/a.v, /proj/lib1/rtl/b.v
```

From the `/proj/lib1` directory, the following relative `file_path_specs` are resolved as shown:

```
../lib2/gates/*.v = /proj/lib2/gates/a.v, /proj/lib2/gates/b.v
./rtl/?.v = /proj/lib1/rtl/a.v, /proj/lib1/rtl/b.v
./rtl/ = /proj/lib1/rtl/a.v, /proj/lib1/rtl/b.v
```

13.7.3 Resolving multiple path specifications

For example:

```
library lib1 "/proj/lib1/foo*.v";
library lib2 "/proj/lib1/foo.v";
library lib3 "../lib1/";
library lib4 "/proj/lib1/*ver.v";
```

When evaluated from the directory `/proj/tb` directory, the following source files shall map into the specified library:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
../lib1/foobar.v - lib1     // potentially matches lib1 and lib3. Because lib1
                            includes  a filename and lib3 only specifies a directory; lib1 takes
                            precedence
/proj/lib1/foo.v - lib2     // takes precedence over lib1 and lib3 path specifications
/proj/lib1/bar.v -          lib3
/proj/lib1/barver.v -       lib4  // takes precedence over lib3 path specification
/proj/lib1/foover.v -       ERROR  // matches lib1 and lib4
/test/tb/tb.v -             work // does not match any library specifications.
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

14. Specify blocks

Two types of HDL constructs are often used to describe delays for structural models such as ASIC cells. They are as follows:

— *Distributed delays*, which specify the time it takes events to propagate through gates and nets inside the module (see 7.14)
— *Module path delays*, which describe the time it takes an event at a source (input port or inout port) to propagate to a destination (output port or inout port)

This clause describes how paths are specified in a module and how delays are assigned to these paths.

14.1 Specify block declaration

A block statement called the *specify block* is the vehicle for describing paths between a source and a destination and for assigning delays to these paths. The syntax for specify blocks is shown in Syntax 14-1.

```
specify_block ::= (From A.7.1)
      specify { specify_item } endspecify
specify_item ::=
        specparam_declaration
      | pulsestyle_declaration
      | showcancelled_declaration
      | path_declaration
      | system_timing_check
```

Syntax 14-1—Syntax for specify block

The specify block shall be bounded by the keywords **specify** and **endspecify**, and it shall appear inside a module declaration. The specify block can be used to perform the following tasks:

— Describe various paths across the module.
— Assign delays to those paths.
— Perform timing checks to ensure that events occurring at the module inputs satisfy the timing constraints of the device described by the module (see Clause 15).

The paths described in the specify block, called *module paths*, pair a signal source with a signal destination. The source may be unidirectional (an input port) or bidirectional (an inout port) and is referred to as the *module path source*. Similarly, the destination may be unidirectional (an output port) or bidirectional (an inout port) and is referred to as the *module path destination*.

For example:

```
specify
    specparam tRise_clk_q = 150, tFall_clk_q = 200;
    specparam tSetup = 70;

    (clk => q) = (tRise_clk_q, tFall_clk_q);

    $setup(d, posedge clk, tSetup);
endspecify
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The first two lines following the keyword **specify** declare specify parameters, which are discussed in 4.10.3. The line following the declarations of specify parameters describes a module path and assigns delays to that module path. The specify parameters determine the delay assigned to the module path. Specifying module paths is presented in 14.2. Assigning delays to module paths is discussed in 14.3. The line preceding the keyword **endspecify** instantiates one of the system timing checks, which are discussed further in Clause 15.

14.2 Module path declarations

There are two steps required to set up module path delays in a specify block:

a) Describe the module paths.
b) Assign delays to those paths (see 14.3).

The syntax of the module path declaration is described in Syntax 14-2.

```
path_declaration ::= (From A.7.2)
        simple_path_declaration ;
      | edge_sensitive_path_declaration ;
      | state_dependent_path_declaration ;
```

Syntax 14-2—Syntax for module path declaration

A module path may be described as a simple path, an edge-sensitive path, or a state-dependent path. A module path shall be defined inside a specify block as a connection between a source signal and a destination signal. Module paths can connect any combination of vectors and scalars.

For example:

Figure 14-1 illustrates a circuit with module path delays. More than one source (`A`, `B`, `C`, and `D`) may have a module path to the same destination (`Q`), and different delays may be specified for each input to output path.

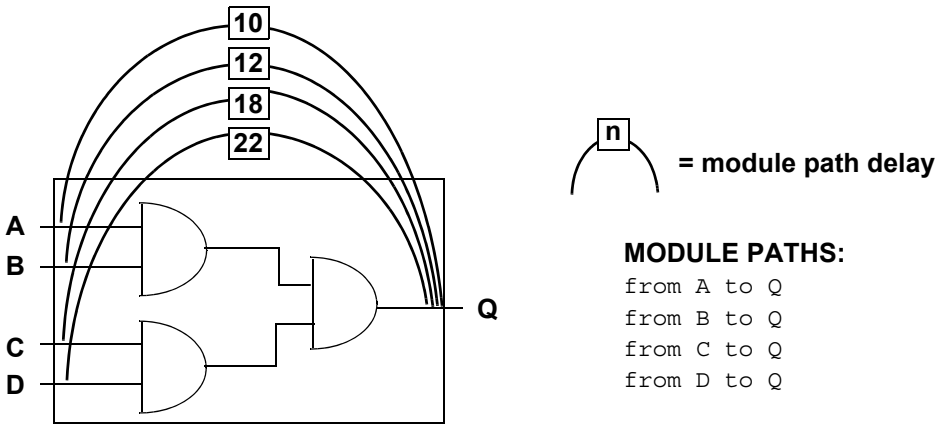


Figure 14-1—Module path delays

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

14.2.1 Module path restrictions

Module paths have the following restrictions:

— The module path source shall be a net that is connected to a module input port or inout port.
— The module path destination shall be a net or variable that is connected to a module output port or inout port.
— The module path destination shall have only one driver inside the module.

14.2.2 Simple module paths

The syntax for specifying a simple module path is given in Syntax 14-3.

```
simple_path_declaration ::= (From A.7.2)
        parallel_path_description = path_delay_value
      | full_path_description = path_delay_value
parallel_path_description ::=
      ( specify_input_terminal_descriptor [ polarity_operator ] =>
        specify_output_terminal_descriptor )
full_path_description ::=
      ( list_of_path_inputs [ polarity_operator ] *> list_of_path_outputs )
list_of_path_inputs ::=
      specify_input_terminal_descriptor { , specify_input_terminal_descriptor }
list_of_path_outputs ::=
      specify_output_terminal_descriptor { , specify_output_terminal_descriptor }
specify_input_terminal_descriptor ::= (From A.7.3)
      input_identifier [ [ constant_range_expression ] ]
specify_output_terminal_descriptor ::=
      output_identifier [ [ constant_range_expression ] ]
input_identifier ::=
      input_port_identifier | inout_port_identifier
output_identifier ::=
      output_port_identifier | inout_port_identifier
polarity_operator ::= (From A.7.4)
      + | -
```

Syntax 14-3—Syntax for simple module path

Simple paths can be declared in one of two forms:

— Source *> destination
— Source => destination

The symbols *> and => each represent a different kind of connection between the module path source and the module path destination. The operator *> establishes a full connection between source and destination. The operator => establishes a parallel connection between source and destination. See 14.2.5 for a description of full connection and parallel connection paths.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

For example:

The following three examples illustrate valid simple module path declarations:

```
(A => Q) = 10;
(B => Q) = (12);
(C, D *> Q) = 18;
```

14.2.3 Edge-sensitive paths

When a module path is described using an edge transition at the source, it is called an *edge-sensitive path*. The edge-sensitive path construct is used to model the timing of input-to-output delays, which only occur when a specified edge occurs at the source signal.

The syntax of the edge-sensitive path declaration is shown in Syntax 14-4.

```
edge_sensitive_path_declaration ::= (From A.7.4)
        parallel_edge_sensitive_path_description = path_delay_value
      | full_edge_sensitive_path_description = path_delay_value
parallel_edge_sensitive_path_description ::=
      ( [ edge_identifier ] specify_input_terminal_descriptor =>
        ( specify_output_terminal_descriptor [ polarity_operator ] : data_source_expression ) )
full_edge_sensitive_path_description ::=
      ( [ edge_identifier ] list_of_path_inputs *>
        ( list_of_path_outputs [ polarity_operator ] : data_source_expression ) )
data_source_expression ::=
      expression
edge_identifier ::=
      posedge | negedge
```

Syntax 14-4—Syntax for edge-sensitive path declaration

The edge identifier may be one of the keywords **posedge** or **negedge**, associated with an input terminal descriptor, which may be any input port or inout port. If a vector port is specified as the input terminal descriptor, the edge transition shall be detected on the least significant bit. If the edge transition is not specified, the path shall be considered active on any transition at the input terminal.

An edge-sensitive path may be specified with full connections (*>) or parallel connections (=>). For parallel connections (=>), the destination shall be any scalar output or inout port or the bit-select of a vector output or inout port. For full connections (*>), the destination shall be a list of one or more of the vector or scalar output and inout ports, and bit-selects or part-selects of vector output and inout ports. See 14.2.5 for a description of parallel paths and full connection paths.

The data source expression is an arbitrary expression, which serves as a description of the flow of data to the path destination. This arbitrary data path description does not affect the actual propagation of data or events through the model; how an event at the data path source propagates to the destination depends on the internal logic of the module. The polarity operator describes whether the data path is inverting or noninverting.

For example:

Example 1—The following example demonstrates an edge-sensitive path declaration with a positive polarity operator:

```
( posedge clock => ( out +: in ) ) = (10, 8);
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

In this example, at the positive edge of `clock`, a module path extends from `clock` to `out` using a rise delay of 10 and a fall delay of 8. The data path is from `in` to `out`, and `in` is not inverted as it propagates to `out`.

Example 2—The following example demonstrates an edge-sensitive path declaration with a negative polarity operator:

```
( negedge clock[0] => ( out -: in ) ) = (10, 8);
```

In this example, at the negative edge of `clock[0]`, a module path extends from `clock[0]` to `out` using a rise delay of 10 and a fall delay of 8. The data path is from `in` to `out`, and `in` is inverted as it propagates to `out`.

Example 3—The following example demonstrates an edge-sensitive path declaration with no edge identifier:

```
( clock => ( out : in ) ) = (10, 8);
```

In this example, at any change in `clock`, a module path extends from `clock` to `out`.

14.2.4 State-dependent paths

A *state-dependent path* makes it possible to assign a delay to a module path that affects signal propagation delay through the path only if specified conditions are true.

A state-dependent path description includes the following items:

— A conditional expression that, when evaluated true, enables the module path
— A module path description
— A delay expression that applies to the module path

The syntax for the state-dependent path declaration is shown in Syntax 14-5.

```
state_dependent_path_declaration ::= (From A.7.4)
        if ( module_path_expression ) simple_path_declaration
      | if ( module_path_expression ) edge_sensitive_path_declaration
      | ifnone simple_path_declaration
```

Syntax 14-5—Syntax for state-dependent paths

14.2.4.1 Conditional expression

The operands in the conditional expression shall be constructed from the following:

— Scalar or vector module input ports or inout ports or their bit-selects or part-selects
— Locally defined variables or nets or their bit-selects or part-selects
— Compile time constants (constant numbers and specify parameters)

Table 14-1 contains a list of valid operators that may be used in conditional expressions.

A conditional expression shall evaluate to true `(1)` for the state-dependent path to be assigned a delay value. If the conditional expression evaluates to `x` or `z`, it shall be treated as true. If the conditional expression evaluates to multiple bits, the least significant bit shall represent the result. The conditional expression can have any number of operands and operators.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 14-1—List of valid operators in state-dependent path delay expression

Operator	Description	Operator	Description
~	bitwise negation	&	reduction and
&	bitwise and	|	reduction or
|	bitwise or	^	reduction xor
^	bitwise xor	~&	reduction nand
^~ ~^	bitwise xnor	~|	reduction nor
==	logical equality	^~ ~^	reduction xnor
!=	logical inequality	{}	concatenation
&&	logical and	{ {} }	replication
||	logical or	?:	conditional
!	logical not		

14.2.4.2 Simple state-dependent paths

If the path description of a state-dependent path is a simple path, then it is called a *simple state-dependent path*. The simple path description is discussed in 14.2.2.

For example:

Example 1—The following example uses state-dependent paths to describe the timing of an XOR gate.

```
module XORgate (a, b, out);
input a, b;
output out;

xor x1 (out, a, b);

specify
    specparam noninvrise = 1, noninvfall = 2;
    specparam invertrise = 3, invertfall = 4;
    if (a) (b => out) = (invertrise, invertfall);
    if (b) (a => out) = (invertrise, invertfall);
    if (~a)(b => out) = (noninvrise, noninvfall);
    if (~b)(a => out) = (noninvrise, noninvfall);
endspecify
endmodule
```

In this example, the first two state-dependent paths describe a pair of output rise and fall delay times when the `XOR` gate `(x1)` inverts a changing input. The last two state-dependent paths describe another pair of output rise and fall delay times when the `XOR` gate buffers a changing input.

Example 2—The following example models a partial ALU. The state-dependent paths specify different delays for different ALU operations.

```
module ALU (o1, i1, i2, opcode);
input [7:0] i1, i2;
input [2:1] opcode;
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
output [7:0] o1;

//functional description omitted
specify
    // add operation
    if (opcode == 2'b00) (i1,i2 *> o1) = (25.0, 25.0);
    // pass-through i1 operation
    if (opcode == 2'b01) (i1 => o1) = (5.6, 8.0);
    // pass-through i2 operation
    if (opcode == 2'b10) (i2 => o1) = (5.6, 8.0);
    // delays on opcode changes
    (opcode *> o1) = (6.1, 6.5);
endspecify
endmodule
```

In the preceding example, the first three path declarations declare paths extending from operand inputs `i1` and `i2` to the `o1` output. The delays on these paths are assigned to operations on the basis of the operation specified by the inputs on `opcode`. The last path declaration declares a path from the `opcode` input to the `o1` output.

14.2.4.3 Edge-sensitive state-dependent paths

If the path description of a state-dependent path describes an edge-sensitive path, then the state-dependent path is called an *edge-sensitive state-dependent path*. The edge-sensitive paths are discussed in 14.2.3.

Different delays can be assigned to the same edge-sensitive path as long as the following criteria are met:

— The edge, condition, or both make each declaration unique.
— The port is referenced in the same way in all path declarations (entire port, bit-select, or part-select).

For example:

Example 1

```
if ( !reset && !clear )
    ( posedge clock => ( out +: in ) ) = (10, 8) ;
```

In this example, if the positive edge of `clock` occurs when `reset` and `clear` are low, a module path extends from `clock` to `out` using a rise delay of 10 and a fall delay of 8.

Example 2—The following example shows two edge-sensitive path declarations, each of which has a unique edge:

```
specify
    ( posedge clk => ( q[0] : data ) ) = (10, 5);
    ( negedge clk => ( q[0] : data ) ) = (20, 12);
endspecify
```

Example 3—The following example shows two edge-sensitive path declarations, each of which has a unique condition:

```
specify
    if (reset)
        ( posedge clk => ( q[0] : data ) ) = (15, 8);
    if (!reset && cntrl)
        ( posedge clk => ( q[0] : data ) ) = (6, 2);
endspecify
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Example 4—The two state-dependent path declarations shown below are not legal because even though they have different conditions, the destinations are not specified in the same way: the first destination is a part-select, the second is a bit-select.

```
specify
    if (reset)
        (posedge clk => (q[3:0]:data)) = (10,5);
    if (!reset)
        (posedge clk => (q[0]:data)) = (15,8);
endspecify
```

14.2.4.4 The ifnone condition

The **ifnone** keyword is used to specify a default state-dependent path delay when all other conditions for the path are false. The **ifnone** condition shall specify the same module path source and destination as the state-dependent module paths. The following rules apply to module paths specified with the **ifnone** condition:

- Only simple module paths may be described with an **ifnone** condition.
- The state-dependent paths that correspond to the **ifnone** path may be either simple module paths or edge-sensitive paths.
- If there are no corresponding state-dependent module paths to the **ifnone** module path, then the **ifnone** module path shall be treated the same as an unconditional simple module path.
- It is illegal to specify both an **ifnone** condition for a module path and an unconditional simple module path for the same module path.

For example:

Example 1—The following are valid state-dependent path combinations:

```
if (C1) (IN => OUT) = (1,1);
ifnone (IN => OUT) = (2,2);

// add operation
if (opcode == 2'b00) (i1,i2 *> o1) = (25.0, 25.0);
// pass-through i1 operation
if (opcode == 2'b01) (i1 => o1) = (5.6, 8.0);
// pass-through i2 operation
if (opcode == 2'b10) (i2 => o1) = (5.6, 8.0);
// all other operations
ifnone (i2 => o1) = (15.0, 15.0);

(posedge CLK => (Q +: D)) = (1,1);
ifnone (CLK => Q) = (2,2);
```

Example 2—The following module path description combination is illegal because it combines a state-dependent path using an **ifnone** condition and an unconditional path for the same module path:

```
if (a) (b => out) = (2,2);
if (b) (a => out) = (2,2);
ifnone (a => out) = (1,1);
(a => out) = (1,1);
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

14.2.5 Full connection and parallel connection paths

The operator *> shall be used to establish a *full connection* between source and destination. In a full connection, every bit in the source shall connect to every bit in the destination. The module path source need not have the same number of bits as the module path destination.

The full connection can handle most types of module paths because it does not restrict the size or number of source signals and destination signals. The following situations require the use of full connections:

— To describe a module path between a vector and a scalar
— To describe a module path between vectors of different sizes
— To describe a module path with multiple sources or multiple destinations in a single statement (see 14.2.6)

The operator => shall be used to establish a *parallel connection* between source and destination. In a parallel connection, each bit in the source shall connect to one corresponding bit in the destination. Parallel module paths can be created only between sources and destinations that contain the same number of bits.

Parallel connections are more restrictive than full connections. They only connect one source to one destination, where each signal contains the same number of bits. Therefore, a parallel connection may only be used to describe a module path between two vectors of the same size. Because scalars are 1 bit wide, either *> or => may be used to set up bit-to-bit connections between two scalars.

For example:

Example 1—Figure 14-2 illustrates how a parallel connection differs from a full connection between two 4-bit vectors.

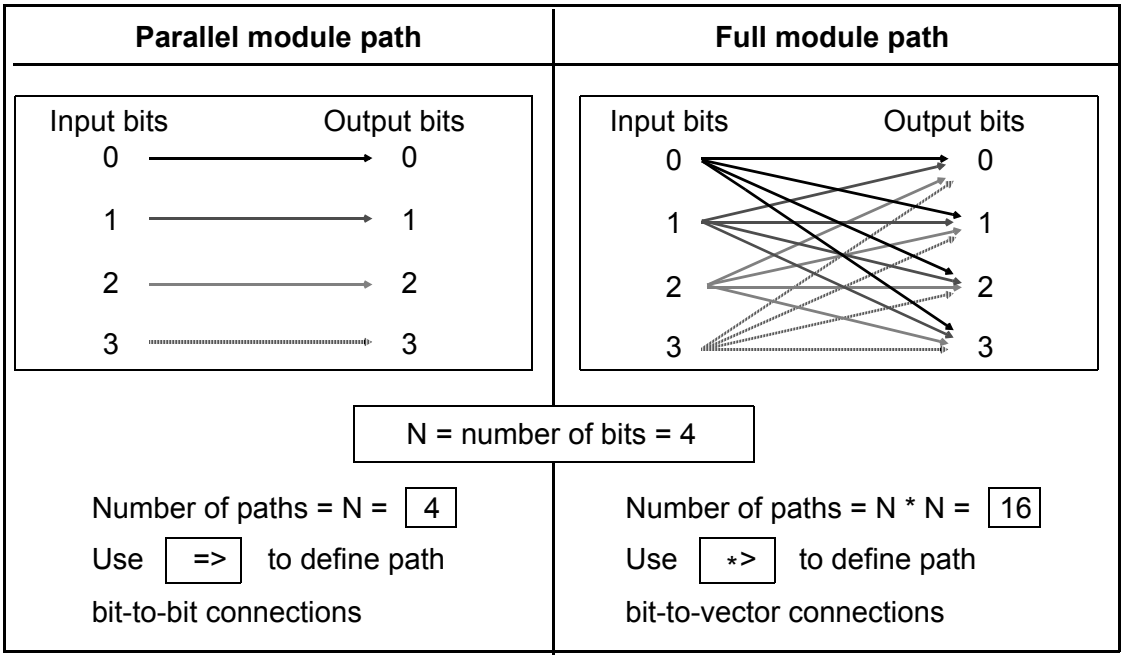


Figure 14-2—Difference between parallel and full connection paths

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Example 2—The following example shows module paths for a 2:1 multiplexor with two 8-bit inputs and one 8-bit output:

```
module mux8 (in1, in2, s, q) ;
output [7:0] q;
input [7:0] in1, in2;
input s;
// Functional description omitted ...
specify
    (in1 => q) = (3, 4) ;
    (in2 => q) = (2, 3) ;
    (s *> q) = 1;
endspecify
endmodule
```

The module path from s to q uses a full connection (*>) because it connects a scalar source—the 1-bit select line—to a vector destination—the 8-bit output bus. The module paths from both input lines in1 and in2 to q use a parallel connection (=>) because they set up parallel connections between two 8-bit buses.

14.2.6 Declaring multiple module paths in a single statement

Multiple module paths may be described in a single statement by using the symbol *> to connect a comma-separated list of sources to a comma-separated list of destinations. When describing multiple module paths in one statement, the lists of sources and destinations may contain a mix of scalars and vectors of any size.

The connection in a multiple module path declaration is always a full connection.

For example:

```
(a, b, c *> q1, q2) = 10;
```

is equivalent to the following six individual module path assignments:

```
(a *> q1) = 10 ;
(b *> q1) = 10 ;
(c *> q1) = 10 ;
(a *> q2) = 10 ;
(b *> q2) = 10 ;
(c *> q2) = 10 ;
```

14.2.7 Module path polarity

The polarity of a module path is an arbitrary specification indicating whether the direction of a signal transition is inverted as it propagates from the input to the output. This arbitrary polarity description does not affect the actual propagation of data or events through the model; how a rise or a fall at the source propagates to the destination depends on the internal logic of the module.

Module paths may specify any of three polarities:

— Unknown polarity
— Positive polarity
— Negative polarity

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

14.2.7.1 Unknown polarity

By default, module paths shall have *unknown polarity*; that is, a transition at the path source may propagate to the destination in an unpredictable way, as follows:

— A rise at the source may cause a rise transition, a fall transition, or no transition at the destination.
— A fall at the source may cause a rise transition, a fall transition, or no transition at the destination.

A module path specified either as a full connection or as a parallel connection, but without a polarity operator `+` or `-`, shall be treated as a module path with unknown polarity.

For example:

```
// Unknown polarity
(In1 => q) = In_to_q ;
(s   *> q) = s_to_q ;
```

14.2.7.2 Positive polarity

For module paths with *positive polarity*, any transition at the source may cause the same transition at the destination, as follows:

— A rise at the source may cause either a rise transition or no transition at the destination.
— A fall at the source may cause either a fall transition or no transition at the destination.

A module path with positive polarity shall be specified by prefixing the `+` polarity operator to `=>` or `*>`.

For example:

```
// Positive polarity
(In1 +=> q) = In_to_q ;
(s   +*> q) = s_to_q ;
```

14.2.7.3 Negative polarity

For module paths with *negative polarity*, any transition at the source may cause the opposite transition at the destination, as follows:

— A rise at the source may cause either a fall transition or no transition at the destination.
— A fall at the source may cause either a rise transition or no transition at the destination.

A module path with negative polarity shall be specified by prefixing the `-` polarity operator to `=>` or `*>`.

For example:

```
// Negative polarity
(In1 -=> q) = In_to_q ;
(s   -*> q) = s_to_q ;
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

14.3 Assigning delays to module paths

The delays that occur at the module outputs where paths terminate shall be specified by assigning delay values to the module path descriptions. The syntax for specifying delay values is shown in Syntax 14-6.

```
path_delay_value ::= (From A.7.4)
        list_of_path_delay_expressions
      | ( list_of_path_delay_expressions )
list_of_path_delay_expressions ::=
        t_path_delay_expression
      | trise_path_delay_expression , tfall_path_delay_expression
      | trise_path_delay_expression , tfall_path_delay_expression , tz_path_delay_expression
      | t01_path_delay_expression , t10_path_delay_expression , t0z_path_delay_expression ,
        tz1_path_delay_expression , t1z_path_delay_expression , tz0_path_delay_expression
      | t01_path_delay_expression , t10_path_delay_expression , t0z_path_delay_expression ,
        tz1_path_delay_expression , t1z_path_delay_expression , tz0_path_delay_expression ,
        t0x_path_delay_expression , tx1_path_delay_expression , t1x_path_delay_expression ,
        tx0_path_delay_expression , txz_path_delay_expression , tzx_path_delay_expression
t_path_delay_expression ::=
      path_delay_expression
```

Syntax 14-6—Syntax for path delay value

In module path delay assignments, a module path description (see 14.2) is specified on the left-hand side, and one or more delay values are specified on the right-hand side. The delay values may be optionally enclosed in a pair of parentheses. There may be one, two, three, six, or twelve delay values assigned to a module path, as described in 14.3.1. The delay values shall be constant expressions containing literals or specparams, and there may be a delay expression of the form `min:typ:max`.

For example:

```
specify
   // Specify Parameters
   specparam tRise_clk_q = 45:150:270, tFall_clk_q=60:200:350;
   specparam tRise_Control = 35:40:45, tFall_control=40:50:65;

   // Module Path Assignments
   (clk => q) = (tRise_clk_q, tFall_clk_q);
   (clr, pre *> q) = (tRise_control, tFall_control);
endspecify
```

In the example above, the specify parameters declared following the **specparam** keyword specify values for the module path delays. The module path assignments assign those module path delays to the module paths.

14.3.1 Specifying transition delays on module paths

Each path delay expression may be a single value—representing the typical delay—or a colon-separated list of three values—representing a *minimum*, *typical*, and *maximum* delay, in that order. If the path delay expression results in a negative value, it shall be treated as zero. Table 14-2 describes how different path delay values shall be associated with various transitions. The path delay expression names refer to the names used in Syntax 14-6.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 14-2—Associating path delay expressions with transitions

Transitions	Number of path delay expressions specified				
	1	2	3	6	12
0 -> 1	t	trise	trise	t01	t01
1 -> 0	t	tfall	tfall	t10	t10
0 -> z	t	trise	tz	t0z	t0z
z -> 1	t	trise	trise	tz1	tz1
1 -> z	t	tfall	tz	t1z	t1z
z -> 0	t	tfall	tfall	tz0	tz0
0 -> x	*	*	*	*	t0x
x -> 1	*	*	*	*	tx1
1 -> x	*	*	*	*	t1x
x -> 0	*	*	*	*	tx0
x -> z	*	*	*	*	txz
z -> x	*	*	*	*	tzx

* See 14.3.2.

For example:

```
// one expression specifies all transitions
(C => Q) = 20;
(C => Q) = 10:14:20;

// two expressions specify rise and fall delays
specparam tPLH1 = 12, tPHL1 = 25;
specparam tPLH2 = 12:16:22, tPHL2 = 16:22:25;
(C => Q) = ( tPLH1, tPHL1 ) ;
(C => Q) = ( tPLH2, tPHL2 ) ;

// three expressions specify rise, fall, and z transition delays
specparam tPLH1 = 12, tPHL1 = 22, tPz1 = 34;
specparam tPLH2 = 12:14:30, tPHL2 = 16:22:40, tPz2 = 22:30:34;
(C => Q) = (tPLH1, tPHL1, tPz1);
(C => Q) = (tPLH2, tPHL2, tPz2);

// six expressions specify transitions to/from 0, 1, and z
specparam t01 = 12, t10 = 16, t0z = 13,
          tz1 = 10, t1z = 14, tz0 = 34 ;
(C => Q) = ( t01, t10, t0z, tz1, t1z, tz0) ;
specparam  T01 = 12:14:24, T10 = 16:18:20, T0z = 13:16:30 ;
specparam  Tz1 = 10:12:16, T1z = 14:23:36, Tz0 = 15:19:34 ;
(C => Q) = ( T01, T10, T0z, Tz1, T1z, Tz0) ;

// twelve expressions specify all transition delays explicitly
specparam t01=10, t10=12, t0z=14, tz1=15, t1z=29, tz0=36,
          t0x=14, tx1=15, t1x=15, tx0=14, txz=20, tzx=30 ;
(C => Q) = (t01, t10, t0z, tz1, t1z, tz0,
             t0x, tx1, t1x, tx0, txz, tzx) ;
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.2 Specifying x transition delays

If the x transition delays are not explicitly specified, the calculation of delay values for x transitions is based on the following two pessimistic rules:

— Transitions from a known state to `x` shall occur as quickly as possible; that is, the shortest possible delay shall be used for any transition to `x`.

— Transitions from `x` to a known state shall take as long as possible; that is, the longest possible delay shall be used for any transition from `x`.

Table 14-3 presents the general algorithm for calculating delay values for x transitions along with specific examples. The following two groups of x transitions are represented in the table:

a) Transition from a known state `s` to `x`: `s -> x`
b) Transition from `x` to a known state `s`: `x -> s`

Table 14-3—Calculating delays for x transitions

X transition	Delay value
General algorithm	
s -> x	minimum (s -> other known signals)
x -> s	maximum (other known signals -> s)
Specific transitions	
0 -> x	minimum (0 -> z delay, 0 -> 1 delay)
1 -> x	minimum (1 -> z delay, 1 -> 0 delay)
z -> x	minimum (z -> 1 delay, z -> 0 delay)
x -> 0	maximum (z -> 0 delay, 1 -> 0 delay)
x -> 1	maximum (z -> 1 delay, 0 -> 1 delay)
x -> z	maximum (1 -> z delay, 0 -> z delay)
Usage: (C => Q) = (5, 12, 17, 10, 6, 22) ;	
0 -> x	minimum (17, 5) = 5
1 -> x	minimum (6, 12) = 6
z -> x	minimum (10, 22) = 10
x -> 0	maximum (22, 12) = 22
x -> 1	maximum (10, 5) = 10
x -> z	maximum (6, 17) = 17

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.3 Delay selection

The simulator shall determine the proper delay to use when a specify path output must be scheduled to transition. There may be specify paths to the output from more than one input, and the simulator must decide which specify path to use.

The simulator shall do this by first determining which specify paths to the output are active. Active specify paths are those whose input has transitioned most recently in time, and either they have no condition or their conditions are true. In the presence of simultaneous input transitions, it is possible for many specify paths to an output to be simultaneously active.

Once the active specify paths are identified, a delay must be selected from among them. This is done by comparing the correct delay for the specific transition being scheduled from each specify path and choosing the smallest.

For example:

Example 1

```
(A => Y) = (6, 9);
(B => Y) = (5, 11);
```

For a `Y` transition from `0` to `1`, if `A` transitioned more recently than `B`, a delay of `6` will be chosen. But if `B` transitioned more recently than `A`, a delay of `5` will be chosen. And if, the last time they transitioned, `A` and `B` did so simultaneously, then the smallest of the two rise delays would be chosen, which is the rise delay from `B` of `5`. The fall delay from `A` of `9` would be chosen if `Y` was instead to transition from `1` to `0`.

Example 2

```
if (MODE < 5) (A => Y) = (5, 9);
if (MODE < 4) (A => Y) = (4, 8);
if (MODE < 3) (A => Y) = (6, 5);
if (MODE < 2) (A => Y) = (3, 2);
if (MODE < 1) (A => Y) = (7, 7);
```

Anywhere from zero to five of these specify paths might be active depending upon the value of `MODE`. For instance, when `MODE` is `2`, the first three specify paths are active. A rise transition would select a delay of `4` because that is the smallest rise delay among the first three. A fall transition would select a delay of `5` because that is the smallest fall delay among the first three.

14.4 Mixing module path delays and distributed delays

If a module contains module path delays and distributed delays (delays on primitive instances within the module), the larger of the two delays for each path shall be used.

For example:

Example 1—Figure 14-3 illustrates a simple circuit modeled with a combination of distributed delays and path delays (only the D input to Q output path is illustrated). Here, the delay on the module path from input `D` to output `Q` is `22`, while the sum of the distributed delays is `0` + `1` = `1`. Therefore, a transition on `Q` caused by a transition on `D` will occur `22` time units after the transition on `D`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

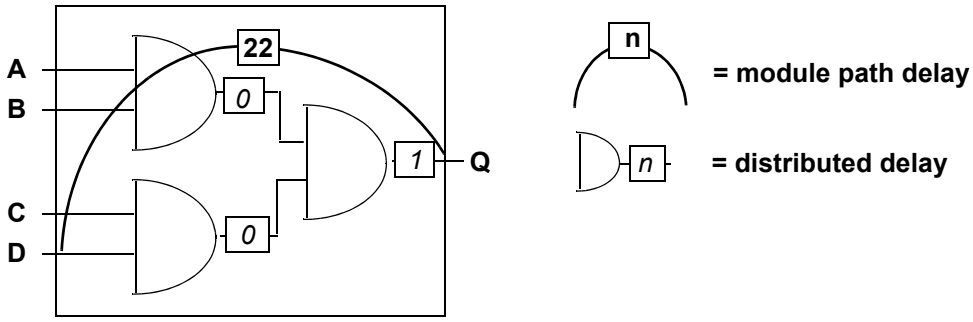


Figure 14-3—Module path delays longer than distributed delays

Example 2—In Figure 14-4, the delay on the module path from `D` to `Q` is `22`, but the distributed delays along that module path now add up to `10 + 20 = 30`. Therefore, an event on `Q` caused by an event on `D` will occur `30` time units after the event on `D`.

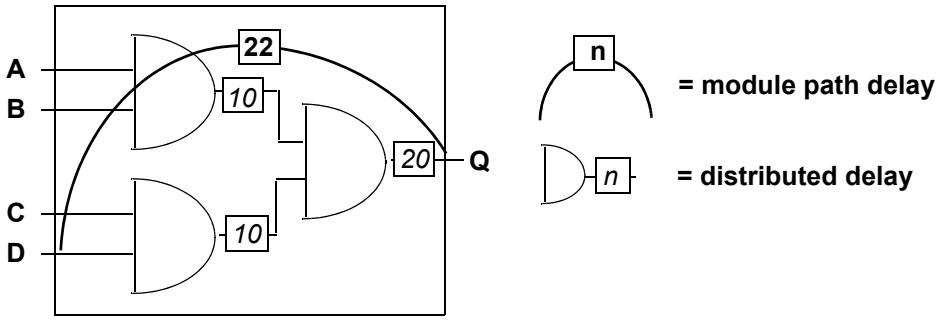


Figure 14-4—Module path delays shorter than distributed delays

14.5 Driving wired logic

Module path output nets shall not have more than one driver within the module. Therefore, wired logic is not allowed at module path outputs.

Figure 14-5 illustrates a violation of this wired-output rule and a method of avoiding the rule violation.

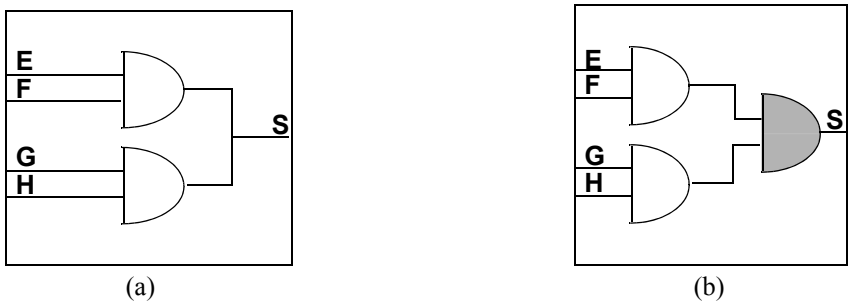


Figure 14-5—Legal and illegal module paths

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

In Figure 14-5 (a), any module path to `S` is illegal because the path destination has two drivers.

Assuming signal `S` in Figure 14-5 (a) is a *wired and*, this limitation can be circumvented by replacing wired logic with gated logic to create a single driver to the output. Figure 14-5 (b) shows how adding a third **and** gate—the shaded gate—solves the problem for the module in Figure 14-5 (a).

The example in Figure 14-6 is also illegal. In this example, when the outputs `Q` and `R` are wired together, it creates a condition where both paths have multiple drivers from within the same module.

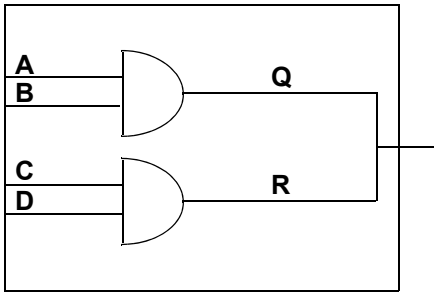


Figure 14-6—Illegal module paths

Although multiple output drivers to a path destination are prohibited inside the same module, they are allowed outside the module. The example in Figure 14-7 is legal because `Q` and `R` each have only one driver within the module in which the module paths are specified.

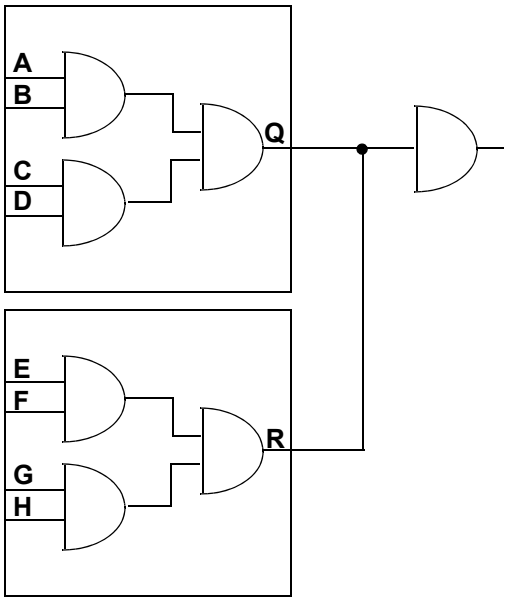


Figure 14-7—Legal module paths

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

14.6 Detailed control of pulse filtering behavior

Two consecutive scheduled transitions closer together in time than the module path delay is deemed a pulse. By default, pulses on a module path output are rejected. Consecutive transitions cannot be closer together than the module path delay, and this is known as the inertial delay model of pulse propagation.

Pulse width ranges control how to handle a pulse presented at a module path output. They are as follows:

— A pulse width range for which a pulse shall be rejected
— A pulse width range for which a pulse shall be allowed to propagate to the path destination
— A pulse width range for which a pulse shall generate a logic `x` on the path destination

Two pulse limit values define the pulse width ranges associated with each module path transition delay. The pulse limit values are called the *error limit* and the *reject limit*. The error limit shall always be at least as large as the reject limit. Pulses greater than or equal to the error limit pass unfiltered. Pulses less than the error limit but greater than or equal to the reject limit are filtered to `X`. Pulses less than the reject limit are rejected, and no pulse emerges. By default, both the error limit and the reject limit are set equal to the delay. These default values yield full inertial pulse behavior, rejecting all pulses smaller than the delay.

In Figure 14-8, the rise delay from input `A` to output `Y` is `7`, and the fall delay is `9`. By default, the error limit and the reject limit for the rise delay are both `7`. The error limit and the reject limit for the fall delay are both `9`. The pulse limits associated with the delay forming the trailing edge of the pulse determine whether and how the pulse should be filtered. Waveform `Y'` shows the waveform resulting from no pulse filtering. The width of the pulse is `2`, which is less than the reject limit for the rise delay of `7`; therefore, the pulse is filtered as shown in waveform `Y`.

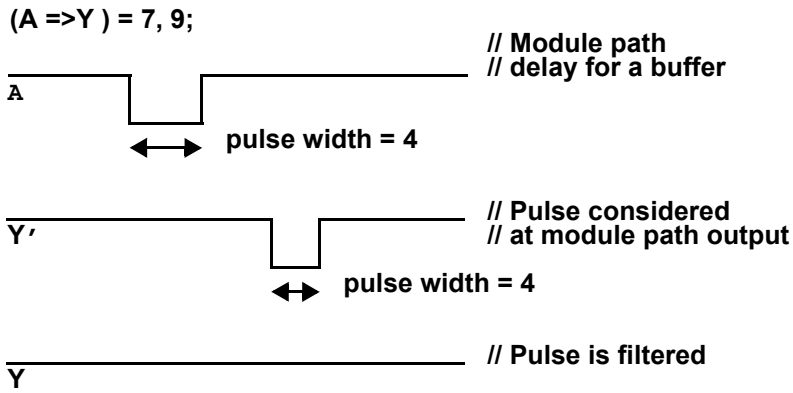


Figure 14-8—Example of pulse filtering

There are three ways to modify the pulse limits from their default values. First, the Verilog language provides the **PATHPULSE$** specparam to modify the pulse limits from their default values. Second, invocation options can specify percentages applying to all module path delays to form the corresponding error limits and reject limits. Third, SDF annotation can individually annotate the error limit and reject limit of each module path transition delay.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

14.6.1 Specify block control of pulse limit values

Pulse limit values may be set from within the specify block with the **PATHPULSE$** specparam. The syntax for using **PATHPULSE$** to specify the reject limit and error limit values is given in Syntax 14-7.

```
pulse_control_specparam ::= (From A.2.4)
        PATHPULSE$ = ( reject_limit_value [ , error_limit_value ] )
      | PATHPULSE$specify_input_terminal_descriptor$specify_output_terminal_descriptor
        = ( reject_limit_value [ , error_limit_value ] )
error_limit_value ::=
      limit_value
reject_limit_value ::=
      limit_value
limit_value ::=
      constant_mintypmax_expression
```

Syntax 14-7—Syntax for PATHPULSE$ pulse control

If only the reject limit value is specified, it shall apply to both the reject limit and the error limit.

The reject limit and error limit may be specified for a specific module path. When no module path is specified, the reject limit and error limit shall apply to all module paths defined in a module. If both path-specific **PATHPULSE$** specparams and a nonpath-specific **PATHPULSE$** specparam appear in the same module, then the path-specific specparams shall take precedence for the specified paths.

The module path input terminals and output terminals shall conform to the rules for module path inputs and outputs, with the following restriction: the terminals may not be a bit-select or part-select of a vector.

When a module path declaration declares multiple paths, the **PATHPULSE$** specparam shall only be specified for the first path input terminal and the first path output terminal. The reject limit and error limit specified shall apply to all other paths in the multiple path declaration. A **PATHPULSE$** specparam that specifies anything other than the first path input and path output terminals shall be ignored.

For example:

In the following example, the path `(clk=>q)` acquires a reject limit of `2` and an error limit of `9`, as defined by the first **PATHPULSE$** declaration. The paths `(clr*>q)` and `(pre*>q)` receive a reject limit of `0` and an error limit of `4`, as specified by the second **PATHPULSE$** declaration. The path `(data=>q)` is not explicitly defined in any of the **PATHPULSE$** declarations; therefore, it acquires reject and error limit of `3`, as defined by the last **PATHPULSE$** declaration.

```
specify
    (clk => q) = 12;
    (data => q) = 10;
    (clr, pre *> q) = 4;

    specparam
        PATHPULSE$clk$q = (2,9),
        PATHPULSE$clr$q = (0,4),
        PATHPULSE$ = 3;
endspecify
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

14.6.2 Global control of pulse limit values

Two invocation options can specify percentages applying globally to all module path transition delays. The error limit invocation option specifies the percentage of each module path transition delay used for its error limit value. The reject limit invocation option specifies the percentage of each module path transition delay used for its reject limit value. The percentage values shall be an integer between `0` and `100`.

The default values for both the reject and error limit invocation options are `100%`. When neither option is present, then `100%` of each module transition delay is used as the reject and error limits.

It is an error if the error limit percentage is smaller than the reject limit percentage. In such cases, the error limit percentage is set equal to the reject limit percentage.

When both **PATHPULSE$** and global pulse limit invocation options are present, the **PATHPULSE$** values shall take precedence.

14.6.3 SDF annotation of pulse limit values

SDF annotation can be used to specify the pulse limit values of module path transition delays. Clause 16 describes this in greater detail.

When **PATHPULSE$**, global pulse limit invocation options, and SDF annotation of pulse limit values are present, SDF annotation values shall take precedence.

14.6.4 Detailed pulse control capabilities

The default style of pulse filtering behavior has two drawbacks. First, pulse filtering to the `x` state may be insufficiently pessimistic with an `x` state duration too short to be useful. Second, unequal delays can result in pulse rejection whenever the trailing edge precedes the leading edge, leaving no indication that a pulse was rejected. This subclause introduces more detailed pulse control capabilities.

14.6.4.1 On-event versus on-detect pulse filtering

When an output pulse must be filtered to `x`, greater pessimism can be expressed if the module path output transitions immediately to `x` (*on-detect*) instead of at the already scheduled transition time of the leading edge of the pulse (*on-event*).

The on-event method of pulse filtering to `x` is the default. When an output pulse must be filtered to `x`, the leading edge of the pulse becomes a transition to `x`, and the trailing edge becomes a transition from `x`. The times of transition of the edges do not change.

Just like on-event, the on-detect method of pulse filtering changes the leading edge of the pulse into a transition to `x` and the trailing edge to a transition from `x`, but the time of the leading edge is changed to occur immediately upon detection of the pulse.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Figure 14-9 illustrates this behavior using a simple buffer with asymmetric rise/fall times and both the reject limits and error limits equal to 0. An output waveform is shown for both on-detect and on-event approaches.

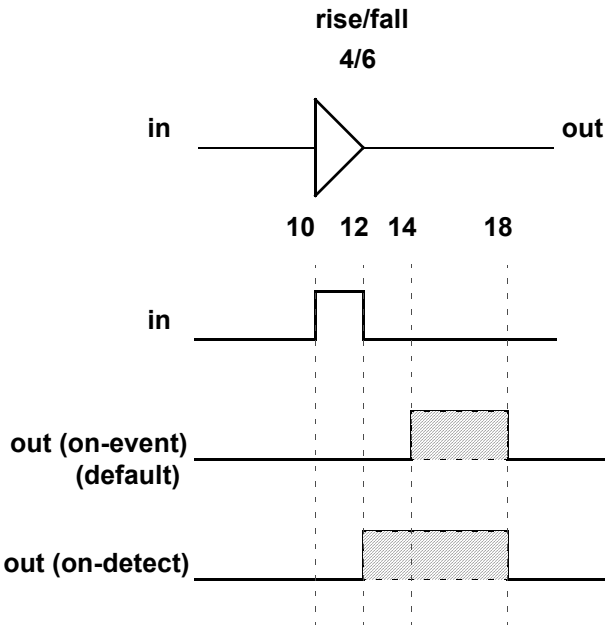


Figure 14-9—On-detect versus on-event

On-detect versus on-event behavior can be selected in two different ways. First, one may be selected globally for all module path outputs through use of the on-detect or on-event invocation option. Second, one may be selected locally through use of specify block pulse style declarations.

The syntax for *pulse* style declarations is shown in Syntax 14-8.

```
pulsestyle_declaration ::= (From A.7.1)
        pulsestyle_onevent list_of_path_outputs ;
      | pulsestyle_ondetect list_of_path_outputs ;
```

Syntax 14-8—Syntax for pulse style declarations

It is an error if a module path output appears in a pulse style declaration after it has already appeared in a module path declaration.

The pulse style invocation options take precedence over pulse style specify block declarations.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

14.6.4.2 Negative pulse detection

When the delays to a module path output are unequal, it is possible for the trailing edge of a pulse to be scheduled for a time earlier than the schedule time of the leading edge, yielding a pulse with a negative width. Under normal operation, if the schedule for a trailing pulse edge is earlier than the schedule for a leading pulse edge, then the leading edge is cancelled. No transition takes place when the initial and final states of the pulse are the same, leaving no indication a schedule was ever present.

Negative pulses can be indicated with the `x` state by use of the *showcancelled* style of behavior. When the trailing edge of a pulse would be scheduled before the leading edge, this style causes the leading edge to be scheduled to `x` and the trailing edge to be scheduled from `x`. With on-event pulse style, the schedule to `x` replaces the leading edge schedule. With on-detect pulse style, the schedule to `x` is made immediately upon detection of the negative pulse.

Showcancelled behavior can be enabled in two different ways. First, it may be enabled globally for all module path outputs through use of the **showcancelled** and **noshowcancelled** invocation options. Second, it may be enabled locally through use of specify block **showcancelled** declarations.

The syntax for showcancelled declarations is shown in Syntax 14-9.

```
showcancelled_declaration ::= (From A.7.1)
        showcancelled list_of_path_outputs ;
      | noshowcancelled list_of_path_outputs ;
```

Syntax 14-9—Syntax for showcancelled declarations

It is an error if a module path output appears in a showcancelled declaration after it has already appeared in a module path declaration. The showcancelled invocation options take precedence over the showcancelled specify block declarations.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The showcancelled behavior is illustrated in Figure 14-10, which shows a narrow pulse presented at the input to a buffer with unequal rise/fall delays. This causes the trailing edge of the pulse to be scheduled earlier than leading edge. The leading edge of the input pulse schedules an output event 6 units later at the point marked by A. The pulse trailing edge occurs one time unit later, which schedules an output event 4 units later marked by point B. This second schedule on the output is for a time prior to the already existing schedule for the leading output pulse edge.

The output waveform is shown for three different operating modes. The first waveform shows the default behavior with showcancelled behavior not enabled and with the default on-event style. The second waveform shows showcancelled behavior in conjunction with on-event. The last waveform shows showcancelled behavior in conjunction with on-detect.

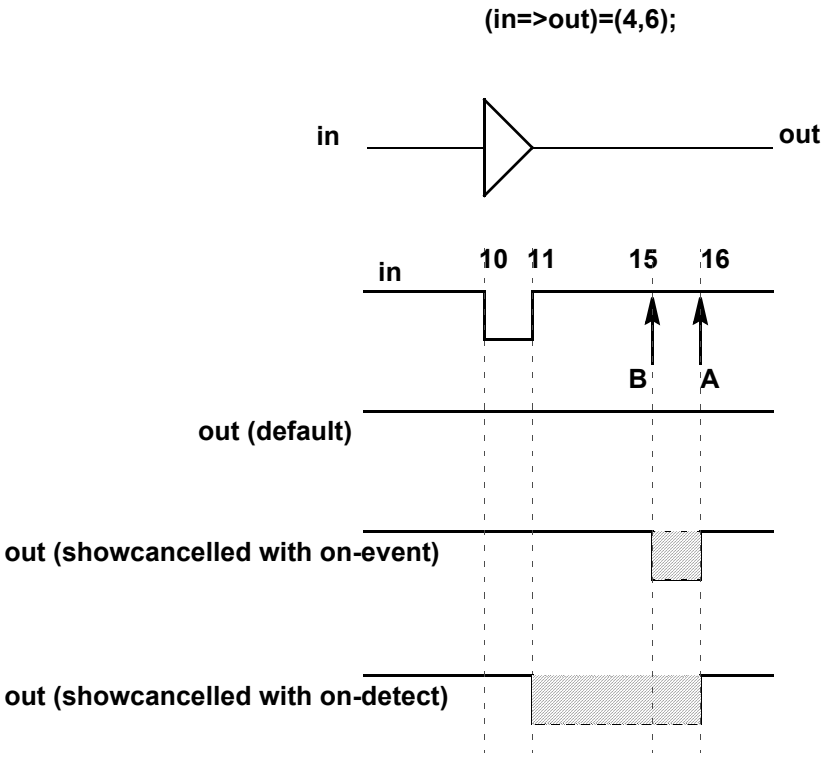


Figure 14-10—Current event cancellation problem and correction

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

This same situation can also arise with nearly simultaneous input transitions, which is defined as two inputs transitioning closer together in time than the difference in their respective delays to the output. Figure 14-11 shows waveforms for a two-input NAND gate where initially `A` is high and `B` is low. B transitions `0->1` at time `10`, causing a `1->0` output schedule at time `24`. A transitions `1->0` at time `12`, causing a `0->1` schedule at time `22`. Arrows mark the output transitions caused by the transitions on inputs `A` and `B`.

The output waveform is shown for three different operating modes. The first waveform shows the default behavior with showcancelled behavior not enabled and with the default on-event style. The second shows showcancelled behavior in conjunction with on-event. The third shows showcancelled behavior in conjunction with on-detect.

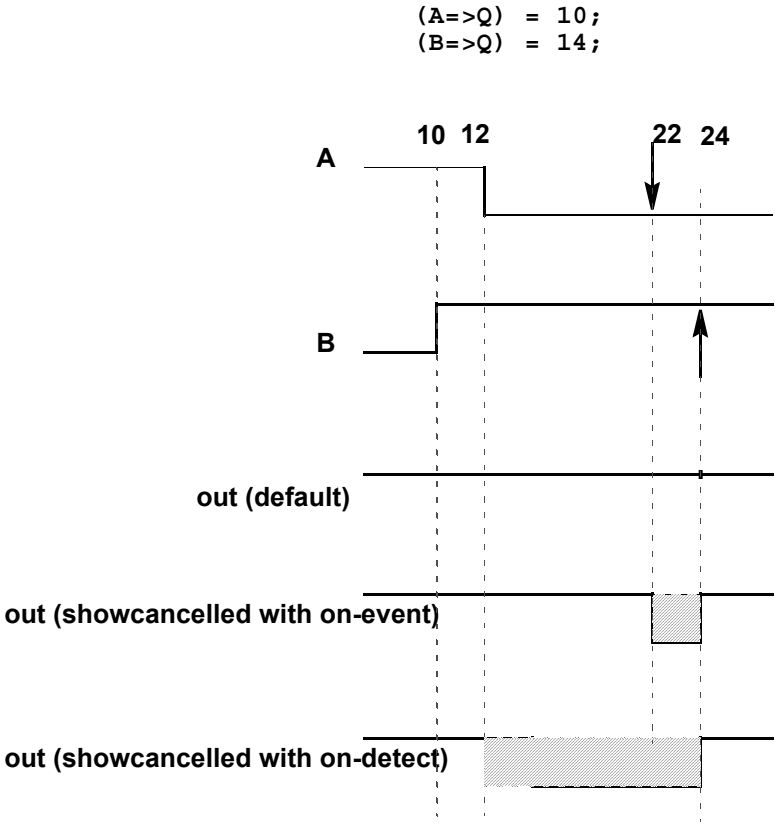


Figure 14-11—NAND gate with nearly simultaneous input switching where one event is scheduled prior to another that has not matured

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

One drawback of the on-event style with showcancelled behavior is that as the output pulse edges draw closer together, the duration of the resulting x state becomes smaller. Figure 14-12 illustrates how the on-detect style solves this problem.

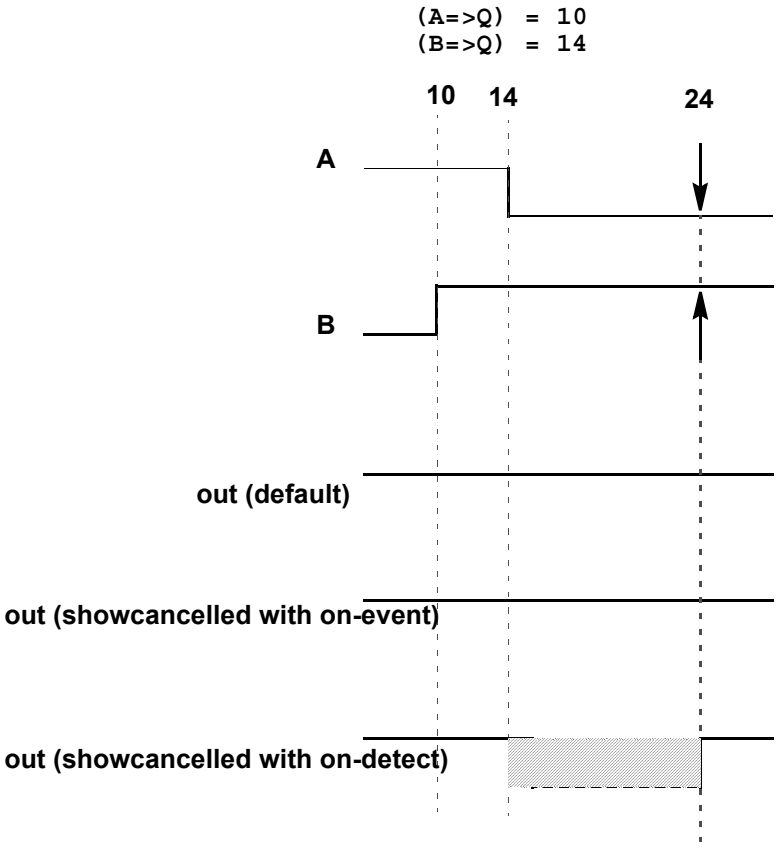


Figure 14-12—NAND gate with nearly simultaneous input switching with output event scheduled at same time

For example:

Example 1

```
specify
    (a=>out)=(2,3);
    (b =>out)=(3,4);
endspecify
```

Because no pulse style or showcancelled declarations appear within the specify block, the compiler applies the default modes of on-event and noshowcancelled.

Example 2

```
specify
    (a=>out)=(2,3);
    showcancelled out;
    (b =>out)=(3,4);
endspecify
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

This showcancelled declaration is in error because it follows use of out in a module path declaration. It would be contradictory for `out` to have noshowcancelled behavior from input `a`, but showcancelled behavior from input `b`.

Example 3—Both these specify blocks produce the same result. Outputs `out` and `out_b` are both declared showcancelled and on-detect.

```
specify
    showcancelled out;
    pulsestyle_ondetect out;
    (a => out) = (2,3);
    (b => out) = (4,5);
    showcancelled out_b;
    pulsestyle_ondetect out_b;
    (a => out_b) = (3,4);
    (b => out_b) = (5,6);
endspecify

specify
    showcancelled out,out_b;
    pulsestyle_ondetect out,out_b;
    (a => out) = (2,3);
    (b => out) = (4,5);
    (a => out_b) = (3,4);
    (b => out_b) = (5,6);
endspecify
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

15. Timing checks

This clause describes how timing checks are used in specify blocks to determine whether signals obey the timing constraints.

15.1 Overview

Timing checks can be placed in specify blocks to verify the timing performance of a design by making sure critical events occur within given time limits. The syntax for system timing checks is given in Syntax 15-1.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
system_timing_check ::= (From A.7.5.1)
        $setup_timing_check
      | $hold_timing_check
      | $setuphold_timing_check
      | $recovery_timing_check
      | $removal_timing_check
      | $recrem_timing_check
      | $skew_timing_check
      | $timeskew_timing_check
      | $fullskew_timing_check
      | $period_timing_check
      | $width_timing_check
      | $nochange_timing_check
$setup_timing_check ::=
      $setup ( data_event , reference_event , timing_check_limit [ , [ notifier ] ] ) ;
$hold_timing_check ::=
      $hold ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
$setuphold_timing_check ::=
      $setuphold ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notifier ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
        [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ] ) ;
$recovery_timing_check ::=
      $recovery ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
$removal_timing_check ::=
      $removal ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
$recrem_timing_check ::=
      $recrem ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notifier ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
        [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ] ) ;
$skew_timing_check ::=
      $skew ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
$timeskew_timing_check ::=
      $timeskew ( reference_event , data_event , timing_check_limit
        [ , [ notifier ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] ) ;
$fullskew_timing_check ::=
      $fullskew ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notifier ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] ) ;
$period_timing_check ::=
      $period ( controlled_reference_event , timing_check_limit [ , [ notifier ] ] ) ;
$width_timing_check ::=
      $width ( controlled_reference_event , timing_check_limit
        [ , threshold [ , notifier ] ] ) ;
$nochange_timing_check ::=
      $nochange ( reference_event , data_event , start_edge_offset ,
        end_edge_offset [ , [ notifier ] ] ) ;
```

Syntax 15-1—Syntax for system timing checks

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The syntax for check time conditions and timing check events is given in Syntax 15-2.

```
checktime_condition ::= (From A.7.5.2)
        mintypmax_expression
controlled_reference_event ::=
        controlled_timing_check_event
data_event ::=
        timing_check_event
delayed_data ::=
          terminal_identifier
        | terminal_identifier [ constant_mintypmax_expression ]
delayed_reference ::=
          terminal_identifier
        | terminal_identifier [ constant_mintypmax_expression ]
end_edge_offset ::= mintypmax_expression
event_based_flag ::= constant_expression
notifier ::= variable_identifier
reference_event ::= timing_check_event
remain_active_flag ::= constant_expression
stamptime_condition ::= mintypmax_expression
start_edge_offset ::= mintypmax_expression
threshold ::= constant_expression
timing_check_limit ::= expression
timing_check_event ::= (From A.7.5.3)
        [timing_check_event_control] specify_terminal_descriptor [ &&& timing_check_condition ]
controlled_timing_check_event ::=
        timing_check_event_control specify_terminal_descriptor [ &&& timing_check_condition ]
timing_check_event_control ::= posedge | negedge | edge_control_specifier
specify_terminal_descriptor ::=
          specify_input_terminal_descriptor
        | specify_output_terminal_descriptor
edge_control_specifier ::= edge [ edge_descriptor { , edge_descriptor } ]
edge_descriptor[a] ::= 01 | 10 | z_or_x zero_or_one | zero_or_one z_or_x
zero_or_one ::= 0 | 1
z_or_x ::= x | X | z | Z
timing_check_condition ::=
          scalar_timing_check_condition
        | ( scalar_timing_check_condition )
scalar_timing_check_condition ::=
          expression
        | ~ expression
        | expression == scalar_constant
        | expression === scalar_constant
        | expression != scalar_constant
        | expression !== scalar_constant
scalar_constant ::= 1'b0 | 1'b1 | 1'B0 | 1'B1 | 'b0 | 'b1 | 'B0 | 'B1 | 1 | 0
```

[a]Embedded spaces are illegal.

Syntax 15-2—Syntax for check time conditions and timing check events

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

For ease of presentation, the timing checks are divided into two groups. The first group of timing checks are described in terms of stability time windows:

$setup	**$hold**	**$setuphold**
$recovery	**$removal**	**$recrem**

The timing checks in the second group check clock and control signals and are described in terms of the difference in time between two events (the **$nochange** check involves three events):

$skew	**$timeskew**	**$fullskew**
$width	**$period**	**$nochange**

Although they begin with a **$**, timing checks are not system tasks. The leading **$** is present because of historical reasons, and timing checks shall not be confused with system tasks. In particular, no system task can appear in a specify block, and no timing check can appear in procedural code.

Some timing checks can accept negative limit values. This topic is discussed in detail in 15.8.

All timing checks have both a reference event and a data event, and boolean conditions can be associated with each. Some of the checks have two signal arguments, one of which is the reference event and the other is the data event. Other checks have only one signal argument, and the reference and data events are derived from it. Reference events and data events shall only be detected by timing checks when their associated conditions are true. See 15.6 for more information about conditions in timing checks.

Timing check evaluation is based upon the times of two events, called the *timestamp event* and the *timecheck event*. A transition on the timestamp event signal causes the simulator to record (stamp) the time of transition for future use in evaluating the timing check. A transition on the timecheck event signal causes the simulator to actually evaluate the timing check to determine whether a violation has taken place.

For some checks, the reference event is always the timestamp event, and the data event is always the timecheck event; while for other checks the reverse is true. And for yet other checks, the decision about which is the timestamp and which is the timecheck event is based upon factors that are discussed in greater detail in 15.2 and 15.3.

Every timing check can include an optional notifier that toggles whenever the timing check detects a violation. The model can use the notifier to make behavior a function of timing check violations. Notifiers are discussed in greater detail in 15.5.

Like expressions for module path delays, timing check limit values are constant expressions that can include specparams.

15.2 Timing checks using a stability window

These timing checks are discussed in this subclause:

$setup	**$hold**	**$setuphold**
$recovery	**$removal**	**$recrem**

These checks accept two signals, the reference event and the data event, and define a time window with respect to one signal while checking the time of transition of the other signal with respect to the window. In general, they all perform the following steps:

a) Define a time window with respect to the reference signal using the specified limit or limits.
b) Check the time of transition of the data signal with respect to the time window.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

c) Report a timing violation if the data signal transitions within the time window.

15.2.1 $setup

The **$setup** timing check syntax is shown in Syntax 15-3.

```
$setup_timing_check ::= (From A.7.5.1)
        $setup ( data_event , reference_event , timing_check_limit [ , [ notifier ] ] ) ;
data_event ::= (From A.7.5.2)
        timing_check_event
reference_event ::=
        timing_check_event
timing_check_limit ::=
        expression
```

Syntax 15-3—Syntax for $setup

Table 15-1 defines the **$setup** timing check.

Table 15-1—$setup arguments

Argument	Description
`data_event`	Timestamp event
`reference_event`	Timecheck event
`limit`	Non-negative constant expression
notifier (optional)	Reg

The data event is usually a data signal, while the reference event is usually a clock signal.

The end points of the time window are determined as follows:

```
(beginning of time window) = (timecheck time) - limit
(end of time window) = (timecheck time)
```

The **$setup** timing check reports a timing violation in the following case:

```
(beginning of time window) < (timestamp time) < (end of time window)
```

The end points of the time window are not part of the violation region. When the limit is zero, the **$setup** check shall never issue a violation.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

15.2.2 $hold

The **$hold** timing check syntax is shown in Syntax 15-4.

```
$hold_timing_check ::= (From A.7.5.1)
      $hold ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
data_event ::= (From A.7.5.2)
      timing_check_event
reference_event ::=
      timing_check_event
timing_check_limit ::=
      expression
```

Syntax 15-4—Syntax for $hold

Table 15-2 defines the **$hold** timing check.

Table 15-2—$hold arguments

Argument	Description
`reference_event`	Timestamp event
`data_event`	Timecheck event
`limit`	Non-negative constant expression
notifier (optional)	Reg

The data event is usually a data signal, while the reference event is usually a clock signal.

The end points of the time window are determined as follows:

```
(beginning of time window) = (timestamp time)
(end of time window) = (timestamp time) + limit
```

The **$hold** timing check reports a timing violation in the following case:

```
(beginning of time window) <= (timecheck time) < (end of time window)
```

Only the end of the time window is not part of the violation region. When the limit is zero, the **$hold** check shall never issue a violation.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

15.2.3 $setuphold

The **$setuphold** timing check syntax is shown in Syntax 15-5.

```
$setuphold_timing_check ::= (From A.7.5.1)
      $setuphold ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notifier ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
        [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ) ;
checktime_condition ::= (From A.7.5.2)
      mintypmax_expression
data_event ::=
      timing_check_event
delayed_data ::=
       terminal_identifier
      | terminal_identifier [ constant_mintypmax_expression ]
delayed_reference ::=
       terminal_identifier
      | terminal_identifier [ constant_mintypmax_expression ]
reference_event ::=
      timing_check_event
stamptime_condition ::=
      mintypmax_expression
timing_check_limit ::=
      expression
```

Syntax 15-5—Syntax for $setuphold

Table 15-3 defines the **$setuphold** timing check.

Table 15-3—$setuphold arguments

Argument	Description
`reference_event`	Timecheck or timestamp event when setup limit is positive Timestamp event when setup limit is negative
`data_event`	Timecheck or timestamp event when hold limit is positive Timestamp event when hold limit is negative
`setup_limit`	Constant expression
`hold_limit`	Constant expression
notifier (optional)	Reg
timestamp_cond (optional)	Timestamp condition for negative timing checks
timecheck_cond (optional)	Timecheck condition for negative timing checks
delayed_reference (optional)	Delayed reference signal for negative timing checks
delayed_data (optional)	Delayed data signal for negative timing checks

The **$setuphold** timing check can accept negative limit values. This is discussed in greater detail in 15.8.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The data event is usually a data signal, while the reference event is usually a clock signal.

When both the setup limit and the hold limit are positive, either the reference event or the data event can be the timecheck event. It shall depend upon which occurs first in the simulation.

When either the setup limit or the hold limit is negative, the restriction becomes as follows:

```
setup_limit + hold_limit > (simulation unit of precision)
```

The **$setuphold** timing check combines the functionality of the **$setup** and **$hold** timing checks into a single timing check. Therefore, the invocation

```
$setuphold( posedge clk, data, tSU, tHLD );
```

is equivalent in functionality to the following, if `tSU` and `tHLD` are not negative:

```
$setup( data, posedge clk, tSU );
$hold( posedge clk, data, tHLD );
```

When both setup and hold limits are positive and the data event occurs first, the end points of the time window are determined as follows:

```
(beginning of time window) = (timecheck time) - limit
(end of time window) = (timecheck time)
```

And the **$setuphold** timing check reports a timing violation in the following case:

```
(beginning of time window) < (timestamp time) <= (end of time window)
```

Only the beginning of the time window is not part of the violation region. The **$setuphold** check shall report a timing violation when the reference and data events occur simultaneously.

When both setup and hold limits are positive and the data event occurs second, the end points of the time window are determined as follows:

```
beginning of time window) = (timestamp time)
(end of time window) = (timestamp time) + limit
```

And the **$setuphold** timing check reports a timing violation in the following case:

```
(beginning of time window) <= (timecheck time) < (end of time window)
```

Only the end of the time window is not part of the violation region. The **$setuphold** check shall report a timing violation when the reference and data events occur simultaneously.

When both limits are zero, the **$setuphold** check shall never issue a violation.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

15.2.4 $removal

The **$removal** timing check syntax is shown in Syntax 15-6.

```
$removal_timing_check ::= (From A.7.5.1)
      $removal ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
data_event ::= (From A.7.5.2)
      timing_check_event
reference_event ::=
      timing_check_event
timing_check_limit ::=
      expression
```

Syntax 15-6—Syntax for $removal

Table 15-4 defines the **$removal** timing check.

Table 15-4—$removal arguments

Argument	Description
`reference_event`	Timecheck event
`data_event`	Timestamp event
`limit`	Non-negative constant expression
notifier (optional)	Reg

The reference event is usually a control signal like clear, reset, or set, while the data event is usually a clock signal.

The end points of the time window are determined as follows:

```
(beginning of time window) = (timecheck time) - limit
(end of time window) = (timecheck time)
```

The **$removal** timing check reports a timing violation in the following case:

```
(beginning of time window) < (timestamp time) < (end of time window)
```

The end points of the time window are not part of the violation region. When the limit is zero, the **$removal** check shall never issue a violation.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

15.2.5 $recovery

The **$recovery** timing check syntax is shown in Syntax 15-7.

```
$recovery_timing_check ::= (From A.7.5.1)
      $recovery ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
data_event ::= (From A.7.5.2)
      timing_check_event
reference_event ::=
      timing_check_event
timing_check_limit ::=
      expression
```

Syntax 15-7—Syntax for $recovery

Table 15-5 defines the **$recovery** timing check.

Table 15-5—$recovery arguments

Argument	Description
`reference_event`	Timestamp event
`data_event`	Timecheck event
`limit`	Non-negative constant expression
notifier (optional)	Reg

The reference event is usually a control signal like clear, reset, or set, while the data event is usually a clock signal.

The end points of the time window are determined as follows:

```
(beginning of time window) = (timestamp time)
(end of time window) = (timestamp time) + limit
```

The **$recovery** timing check reports a timing violation in the following case:

```
(beginning of time window) <= (timecheck time) < (end of time window)
```

Only the end of the time window is not part of the violation region. When the limit is zero, the **$recovery** check shall never issue a violation.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

15.2.6 $recrem

The **$recrem** timing check syntax is shown in Syntax 15-8.

```
$recrem_timing_check ::= (From A.7.5.1)
      $recrem ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notifier ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
        [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ) ;
checktime_condition ::= (From A.7.5.2)
      mintypmax_expression
data_event ::=
      timing_check_event
delayed_data ::=
        terminal_identifier
      | terminal_identifier [ constant_mintypmax_expression ]
delayed_reference ::=
        terminal_identifier
      | terminal_identifier [ constant_mintypmax_expression ]
reference_event ::=
      timing_check_event
stamptime_condition ::=
      mintypmax_expression
timing_check_limit ::=
      expression
```

Syntax 15-8—Syntax for $recrem

Table 15-6 defines the **$recrem** timing check.

Table 15-6—$recrem arguments

Argument	Description
`reference_event`	Timecheck or timestamp event when removal limit is positive Timestamp event when removal limit is negative
`data_event`	Timecheck or timestamp event when recovery limit is positive Timestamp event when recovery limit is negative
`recovery_limit`	Constant expression
`removal_limit`	Constant expression
notifier (optional)	Reg
timestamp_cond (optional)	Timestamp condition for negative timing checks
timecheck_cond (optional)	Timecheck condition for negative timing checks
delayed_reference (optional)	Delayed reference signal for negative timing checks
delayed_data (optional)	Delayed data signal for negative timing checks

The **$recrem** timing check can accept negative limit values. This is discussed in greater detail in 15.8.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

When both the removal limit and the recovery limit are positive, either the reference event or the data event can be the timecheck event. It shall depend upon which occurs first in the simulation.

When either the removal limit or the recovery limit is negative, the restriction becomes as follows:

```
removal_limit + recovery_limit > (simulation unit of precision)
```

The **$recrem** timing check combines the functionality of the **$removal** and **$recovery** timing checks into a single timing check. Therefore, the invocation

```
$recrem( posedge clear, posedge clk, tREC, tREM );
```

is equivalent in functionality to the following, if `tREC` and `tREM` are not negative:

```
$removal( posedge clear, posedge clk, tREM );
$recovery( posedge clear, posedge clk, tREC );
```

When both removal and recovery limits are positive and the data event occurs first, the end points of the time window are determined as follows:

```
(beginning of time window) = (timecheck time) - limit
(end of time window) = (timecheck time)
```

And the **$recrem** timing check reports a timing violation in the following case:

```
(beginning of time window) < (timestamp time) <= (end of time window)
```

Only the beginning of the time window is not part of the violation region. The **$recrem** check shall report a timing violation when the reference and data events occur simultaneously.

When both removal and recovery limits are positive and the data event occurs second, the end points of the time window are determined as follows:

```
(beginning of time window) = (timestamp time)
(end of time window) = (timestamp time) + limit
```

And the **$recrem** timing check reports a timing violation in the following case:

```
(beginning of time window) <= (timecheck time) < (end of time window)
```

Only the end of the time window is not part of the violation region. The **$recrem** check shall report a timing violation when the reference and data events occur simultaneously.

When both limits are zero, the **$recrem** check shall never issue a violation.

15.3 Timing checks for clock and control signals

The following timing checks are discussed in this subclause:

$skew **$timeskew** **$fullskew** **$period** **$width** **$nochange**

These checks accept one or two signals and verify that transitions on them are never separated by more than the limit. For checks specifying only one signal, the reference event and data event are derived from that one signal. In general, these checks all perform the following steps:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

a) Determine the elapsed time between two events.
b) Compare the elapsed time to the specified limit.
c) Report a timing violation if the elapsed time violates the limit.

The skew checks have two different violation detection mechanisms, *event-based* and *timer-based*. Event-based skew checking is performed only when a signal transitions, while timer-based skew checking takes place as soon as the simulation time equal to the skew limit has elapsed.

The **$nochange** check involves three events rather than two.

15.3.1 $skew

The **$skew** timing check syntax is shown in Syntax 15-9.

```
$skew_timing_check ::= (From A.7.5.1)
        $skew ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
data_event ::= (From A.7.5.2)
        timing_check_event
reference_event ::=
        timing_check_event
timing_check_limit ::=
        expression
```

Syntax 15-9—Syntax for $skew

Table 15-7 defines the **$skew** timing check.

Table 15-7—$skew arguments

Argument	Description
`reference_event`	Timestamp event
`data_event`	Timecheck event
`limit`	Non-negative constant expression
notifier (optional)	Reg

The **$skew** timing check reports a violation in the following case:

```
(timecheck time) - (timestamp time) > limit
```

Simultaneous transitions on the reference and data signals shall not cause **$skew** to report a timing violation, even when the skew limit value is zero.

The **$skew** timing check is event-based; it is evaluated only after a data event. If there is never a data event (i.e., the data event is infinitely late), the **$skew** timing check shall never be evaluated, and no timing violation shall ever be reported. In contrast, the **$timeskew** and **$fullskew** checks are timer-based by default, and they should be used if violation reports are absolutely required and the data event can be very late or even absent altogether. These checks are discussed in 15.3.2 and 15.3.3.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

$skew shall wait indefinitely for the data event once it has detected a reference event, and it shall not report a timing violation until the data event takes place. A second consecutive reference event shall cancel the old wait for the data event and begin a new one.

After a reference event, the **$skew** timing check shall never stop checking data events for a timing violation. **$skew** shall report timing violations for all data events occurring beyond the limit after a reference event.

15.3.2 $timeskew

The syntax for **$timeskew** is shown in Syntax 15-10.

```
$timeskew_timing_check ::= (From A.7.5.1)
      $timeskew ( reference_event , data_event , timing_check_limit
        [ , [ notifier ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] ) ;
data_event ::= (From A.7.5.2)
      timing_check_event
event_based_flag ::=
      constant_expression
reference_event ::=
      timing_check_event
remain_active_flag ::=
      constant_expression
timing_check_limit ::=
      expression
```

Syntax 15-10—Syntax for $timeskew

Table 15-8 defines the **$timeskew** timing check arguments.

Table 15-8—$timeskew arguments

Argument	Description
`reference_event`	Timestamp event
`data_event`	Timecheck event
`limit`	Non-negative constant expression
notifier (optional)	Reg
event_based_flag (optional)	Constant expression
remain_active_flag (optional)	Constant expression

The **$timeskew** timing check reports a violation only in the following case:

```
(timecheck time) - (timestamp time) > limit
```

Simultaneous transitions on the reference and data signals shall not cause **$timeskew** to report a timing violation, even when the skew limit value is zero. **$timeskew** shall also not report a violation if a new timestamp event occurs exactly at the expiration of the time limit.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

The default behavior for **$timeskew** is timer-based. A violation shall be reported immediately upon an elapse of time after the reference event equal to the limit, and the check shall become dormant and report no more violations (even in response to data events) until after the next reference event. However, if a data event occurs within the limit, then a violation shall not be reported, and the check shall become dormant immediately. This check shall also become dormant if it detects a conditioned reference event when its condition is false and the remain_active_flag is not set.

The **$timeskew** check's default timer-based behavior can be altered to event-based using the event_based_flag. It behaves like the **$skew** check when both the event_based_flag and the remain_active_flag are set. The **$timeskew** check behaves like the **$skew** check when only the event_based_flag is set, except that it becomes dormant after reporting the first violation or if it detects a conditioned reference event when its condition is false.

For example:

See Figure 15-1.

```
$timeskew (posedge CP &&& MODE, negedge CPN, 50,, event_based_flag,
remain_active_flag);
```

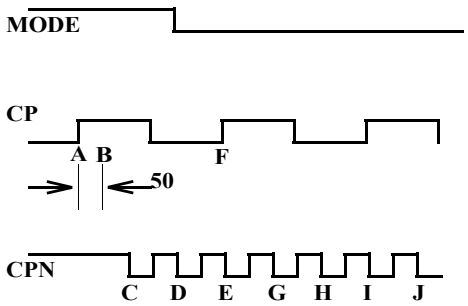


Figure 15-1—Sample $timeskew

Case 1: event_based_flag not set, remain_active_flag not set.

After the first reference event on `CP` at `A`, a violation is reported at `B` as soon as `50` time units have passed, turning the $timeskew check dormant, and no further violations are reported.

Case 2: event_based_flag set, remain_active_flag not set.

After the first reference event on `CP` at `A`, the negative transition on `CPN` at point `C` causes a timing violation, turning the **$timeskew** check dormant, and no further violations are reported. The second reference event at `F` occurs while `MODE` is false; therefore, the **$timeskew** check remains dormant.

Case 3: event_based_flag set, remain_active_flag set.

After the first reference event on `CP` at `A`, the first three negative transitions on `CPN` at points `C`, `D`, and `E` cause timing violations. The second reference event at `F` occurs while `MODE` is false, but because the remain_active_flag is set, the **$timeskew** check remains active. Therefore, additional violations are reported at `G`, `H`, `I`, and `J`. In other words, all negative transitions on `CPN` cause violations, which is identical to **$skew** behavior.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Case 4: event_based_flag not set, remain_active_flag set.

For the waveform depicted in Table 15-1, **$timeskew** has the same behavior in Case 4 as in Case 1. The difference between the two cases is illustrated by the waveform in Figure 15-2.

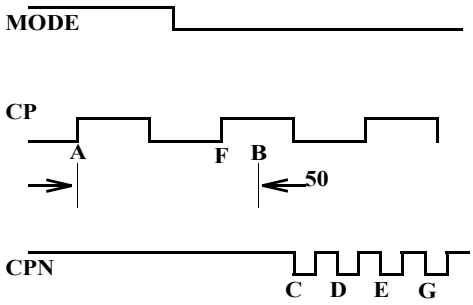


Figure 15-2—Sample $timeskew with remain_active_flag set

Although the reference event on `CP` at `F` occurs while `MODE` is false, it does not turn the **$timeskew** check dormant because the remain_active_flag is set. A violation will hence be reported at time `B`, whereas for Case 1, where the remain_active_flag is not set, the **$timeskew** check would turn dormant at `F`, and no violation would be reported.

15.3.3 $fullskew

The syntax for **$fullskew** is shown in Syntax 15-11.

```
$fullskew_timing_check ::= (From A.7.5.1)
      $fullskew ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notifier ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] ) ;
data_event ::= (From A.7.5.2)
      timing_check_event
event_based_flag ::=
      constant_expression
reference_event ::=
      timing_check_event
remain_active_flag ::=
      constant_expression
timing_check_limit ::=
      expression
```

Syntax 15-11—Syntax for $fullskew

Table 15-9 defines the **$fullskew** timing check arguments.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 15-9—$fullskew arguments

Argument	Description
`reference_event`	Timestamp or timecheck event
`data_event`	Timestamp or timecheck event
`limit 1`	Non-negative constant expression
`limit 2`	Non-negative constant expression
notifier (optional)	Reg
event_based_flag (optional)	Constant expression
remain_active_flag (optional)	Constant expression

$fullskew is similar to **$timeskew** except that the reference and data events can transition in either order. The first limit is the maximum time by which the data event should follow the reference event. The second limit is the maximum time by which the reference event should follow the data event.

The reference event is the timestamp event, and the data event is the timecheck event when the reference event precedes the data event. The data event is the timestamp event, and the reference event is the timecheck event when the data event precedes the reference event.

The **$fullskew** timing check reports a violation only in the following case, where limit is set to limit1 when the reference event transitions first and set to limit2 when the data event transitions first:

```
(timecheck time) - (timestamp time) > limit
```

Simultaneous transitions on the reference and data signals shall not cause **$fullskew** to report a timing violation, even when the skew limit value is zero. **$fullskew** shall also not report a violation if a new timestamp event occurs exactly at the expiration of the time limit.

The default behavior for **$fullskew** is timer-based (event_based_flag not set). A violation shall be reported immediately upon elapse of the time limit after the timestamp event if a timecheck event does not occur in this time, turning the timing check dormant. However, if a timecheck event does occur within the time limit, then no violation is reported, and the timing check turns dormant immediately.

A reference event or data event is a timestamp event and starts a new timing window, unless it is a timecheck event occurring within the time limit after a preceding timestamp event, in which case it turns the timing check dormant, as stated above.

In the timer-based mode, a second timestamp event that occurs within the time limit starts a new timing window that replaces the first one, unless the second timestamp event has an associated condition whose value is false. In such a case, the behavior of **$fullskew** depends on the remain_active_flag. If the flag is set, then the second timestamp event is simply ignored. If the flag is not set and if the timing check is active, then the timing check turns dormant.

The **$fullskew** check's default timer-based behavior can be altered to event-based using the event_based_flag. In this mode, **$fullskew** is similar to **$skew** in that a violation is reported not upon elapse of the time limit after the timestamp event (as in timer-based mode), but rather if a timecheck event occurs after the time limit. Such an event ends the first timing window and immediately begins a new timing window, where it acts as the timestamp event of the new window. A timecheck event within the time limit ends the timing window and turns the timing check dormant, and no violation is reported.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

In the event-based mode, a second timestamp event that occurs before a timecheck event has occurred starts a new timing window that replaces the first one, unless the second timestamp event has an associated condition whose value is false. In such a case, the behavior of **$fullskew** depends on the remain_active_flag. If the flag is set, then the second timestamp event is simply ignored. If the flag is not set and if the timing check is active, then the timing check turns dormant.

In both the timer-based and event-based modes, if the timestamp event has no condition or has a true condition and if the timing check is dormant, then the timing check is activated.

For example:

See Figure 15-3.

```
$fullskew (posedge CP &&& MODE, negedge CPN, 50, 70,, event_based_flag,
remain_active_flag);
```

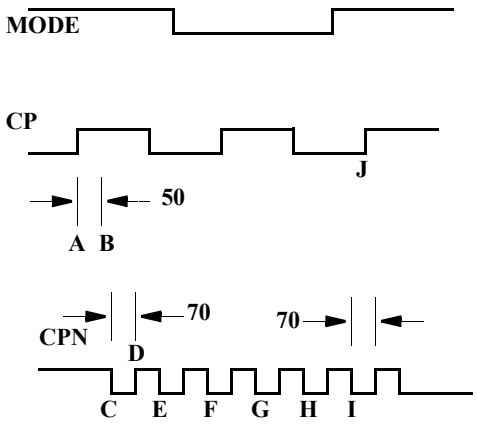


Figure 15-3—Sample $fullskew

Case 1: event_based_flag not set.

The transition at A of CP while MODE is true begins a wait for a negative transition on CPN, and a violation is reported at B as soon as a period of time equal to 50 time units has passed. This resets the check and readies it for the next active transition.

A negative transition on CPN occurs next at C, beginning a wait for a positive transition on CP while MODE is true. At D, a time equal to 70 time units has passed without a positive edge on CP while MODE is true; therefore, a violation is reported, and the check is again reset to await the next active transition.

A transition on CPN at E also results in a timing violation, as does the transition at F, because even though CP transitions, MODE is no longer true. Transitions at G and H also result in timing violations, but not the transition at I because it is followed by a positive transition on CP while MODE is true.

Case 2: event_based_flag set.

The transition at A of CP while MODE is true begins a wait for a negative transition on CPN, and a violation is reported at C on CPN because it occurs beyond the 50 time unit limit. This transition at C also begins a wait

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

of `70` time units for a positive transition on `CP` while `MODE` is true. But for transitions on `CPN` at `C` through `H`, there is no positive transition on `CP` while `MODE` is true; therefore, no timing violations are reported. The transition at `I` on `CPN` begins a wait of `70` time units, and this is satisfied by the positive transition on `CP` at `J` while `MODE` is true.

Although the waveform in this particular example does not show the role of the remain_active_flag, it should be recognized that this flag has a vital role in determining the behavior of the **$fullskew** timing check, just as it does for the **$timeskew** timing check.

15.3.4 $width

The **$width** timing check syntax is shown in Syntax 15-12.

```
$width_timing_check ::= (From A.7.5.1)
        $width ( controlled_reference_event , timing_check_limit
          [ , threshold [ , notifier ] ] ) ;
controlled_reference_event ::= (From A.7.5.2)
        controlled_timing_check_event
threshold ::=
        constant_expression
timing_check_limit ::=
        expression
```

Syntax 15-12—Syntax for $width

Table 15-10 defines the **$width** timing check.

Table 15-10—$width arguments

Argument	Description
`reference_event`	Timestamp edge triggered event
(data_event - implicit)	Timecheck edge triggered event
`limit`	Non-negative constant expression
threshold (optional)	Non-negative constant expression
notifier (optional)	Reg

The **$width** timing check monitors the width of signal pulses by measuring the time from the timestamp event to the timecheck event. Because a data event is not passed to **$width**, it is derived from the reference event, as follows:

```
data event = reference event signal with opposite edge
```

Because of the way the data event is derived for **$width**, an edge triggered event has to be passed as the reference event. A compilation error shall occur if the reference event is not an edge specification.

While the **$width** timing check can be defined in terms of a time window, it is simpler to express it as the difference between the timecheck and timestamp times. The **$width** timing check reports a violation in the following case:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
threshold < (timecheck time) - (timestamp time) < limit
```

The pulse width has to be greater than or equal to limit in order to avoid a timing violation, but no violation is reported for glitches smaller than the threshold.

The threshold argument shall be included if the notifier argument is required. It is permissible to not specify both the threshold and notifier arguments, making the default value for the threshold zero. If the notifier is present, a non-null value for the threshold shall also be present. Here is a legal **$width** check when the notifier is required and the threshold is not:

```
$width (posedge clk, 6, 0, ntfr_reg);
```

The data event and the reference event shall never occur at the same simulation time because these events are triggered by opposite transitions.

For example:

The following example demonstrates some examples of legal and illegal calls:

```
// Legal Calls
$width ( negedge clr, lim );
$width ( negedge clr, lim, thresh, notif );
$width ( negedge clr, lim, 0, notif );

// Illegal Calls
$width ( negedge clr, lim, , notif );
$width ( negedge clr, lim, notif );
```

15.3.5 $period

The **$period** timing check syntax is shown in Syntax 15-13.

```
$period_timing_check ::= (From A.7.5.1)
    $period ( controlled_reference_event , timing_check_limit [ , [ notifier ] ] ) ;
controlled_reference_event ::= (From A.7.5.2)
    controlled_timing_check_event
timing_check_limit ::=
    expression
```

Syntax 15-13—Syntax for $period

Table 15-11 defines the **$period** timing check.

Table 15-11—$period arguments

Argument	Description
`reference_event`	Timestamp edge triggered event
(data_event - implicit)	Timestamp edge triggered event
`limit`	Non-negative constant expression
notifier (optional)	Reg

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Because the data event is not passed as an argument to **$period**, it is derived from the reference event, as follows:

```
data event = reference event signal with the same edge
```

Because of the way the data event is derived for **$period**, an edge triggered event shall be passed as the reference event. A compilation error shall occur if the reference event is not an edge specification.

While the **$period** timing check can be defined in terms of a time window, it is simpler to express it as the difference between the timecheck and timestamp times. The **$period** timing check reports a violation in the following case:

```
(timecheck time) - (timestamp time) < limit
```

15.3.6 $nochange

The **$nochange** syntax is shown in Syntax 15-14.

```
$nochange_timing_check ::= (From A.7.5.1)
      $nochange ( reference_event , data_event , start_edge_offset ,
        end_edge_offset [ , [ notifier ] ] ) ;
data_event ::= (From A.7.5.2)
      timing_check_event
end_edge_offset ::=
      mintypmax_expression
reference_event ::=
      timing_check_event
start_edge_offset ::=
      mintypmax_expression
```

Syntax 15-14—Syntax for $nochange

Table 15-12 defines the **$nochange** timing check arguments.

Table 15-12—$nochange arguments

Argument	Description
`reference_event`	Edge triggered timestamp and/or timecheck event
`data_event`	Timestamp or timecheck event
`start_edge_offset`	Constant expression
`end_edge_offset`	Constant expression
notifier (optional)	Reg

The **$nochange** timing check reports a timing violation if the data event occurs during the specified level of the control signal (the *reference event*). The reference event can be specified with the **posedge** or the **negedge** keyword, but the edge-control specifiers (see 15.4) cannot be used.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The start edge and end edge offsets can expand or shrink the timing violation region, which is defined by the duration of the reference event signal after the edge. For example, if the reference event is a posedge, then the duration is the period during which the reference signal is high. A positive offset for start edge extends the region by starting the timing violation region earlier; a negative offset for start edge shrinks the region by starting the region later. Similarly, a positive offset for the end edge extends the timing violation region by ending it later, while a negative offset for the end edge shrinks the region by ending it earlier. If both the offsets are zero, the size of the region shall not change.

Unlike other timing checks, **$nochange** involves three, rather than two, transitions. The leading edge of the reference event defines the beginning of the time window, while the trailing edge of the reference event defines the end of the time window. A violation results if the data event occurs anytime within the time window.

The end points of the time window are determined as follows:

```
(beginning of time window) = (leading reference edge time) -
                                                    start_edge_offset
(end of time window) = (trailing reference edge time) + end_edge_offset
```

The **$nochange** timing check reports a timing violation in the following case:

```
(beginning of time window) < (data event time) < (end of time window)
```

The end points of the time window are not included. The values of `start_edge_offset` and `end_edge_offset` play a significant role in determining which signal, the reference event or the data event, is the timestamp or timecheck event.

For example:

```
$nochange ( posedge clk, data, 0, 0) ;
```

In this example, the **$nochange** timing check shall report a violation if the `data` signal changes while `clk` is high. It shall not be a violation if `posedge clk` and a transition on `data` occur simultaneously.

15.4 Edge-control specifiers

The edge-control specifiers can be used to control events in timing checks based on specific edge transitions between `0`, `1`, and `x`. Syntax 15-15 shows the syntax for edge-control specifiers.

```
edge_control_specifier ::= (From A.7.5.3)
    edge [ edge_descriptor { , edge_descriptor } ]
edge_descriptor[a] ::=
      01
    | 10
    | z_or_x zero_or_one
    | zero_or_one z_or_x
zero_or_one ::= 0 | 1
z_or_x ::= x | X | z | Z
```

[a]Embedded spaces are illegal.

Syntax 15-15—Syntax for edge-control specifier

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

Edge-control specifiers contain the keyword **edge** followed by a square-bracketed list of from one to six pairs of edge transitions between `0`, `1`, and `x`, as follows:

`01`	Transition from `0` to `1`
`0x`	Transition from `0` to `x`
`10`	Transition from `1` to `0`
`1x`	Transition from `1` to `x`
`x0`	Transition from `x` to `0`
`x1`	Transition from `x` to `1`

Edge transitions involving `z` are treated the same way as edge transitions involving `x`.

The **posedge** and **negedge** keywords can be used as a shorthand for certain edge-control specifiers. For example, the construct

```
posedge clr
```

is equivalent to the following:

```
edge[01, 0x, x1] clr
```

Similarly, the construct

```
negedge clr
```

is the same as the following:

```
edge[10, x0, 1x] clr
```

However, edge-control specifiers offer the flexibility to declare edge transitions other than **posedge** and **negedge**.

15.5 Notifiers: user-defined responses to timing violations

Timing check notifiers detect timing check violations behaviorally and, therefore, take an action as soon as a violation occurs. Such notifiers can be used to print an informative error message describing the violation or to propagate an `x` value at the output of the device that reported the violation.

The notifier is a reg, declared in the module where timing check tasks are invoked, that is passed as the last argument to a system timing check. Whenever a timing violation occurs, the timing check updates the value of the notifier.

The notifier is an optional argument to all system timing checks and can be omitted from the timing check call without adversely affecting its operation.

Table 15-13 shows how the notifier values are toggled when timing violations occur.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 15-13—Notifier value responses to timing violations

BEFORE violation	AFTER violation
x	Either 0 or 1
0	1
1	0
z	z

For example:

Example 1

```
$setup( data, posedge clk, 10, notifier ) ;
$width( posedge clk, 16, 0, notifier ) ;
```

Example 2—Consider a more complex example of how to use notifiers in a behavioral model. The following example uses a notifier to set the D flip-flop output to x when a timing violation occurs in an edge-sensitive UDP:

```
primitive posdff_udp(q, clock, data, preset, clear, notifier);
output q; reg q;
input clock, data, preset, clear, notifier;
table
//clock data   p c notifier state  q
//-------------------------------------
     r    0    1 1    ?    :  ?  : 0 ;
     r    1    1 1    ?    :  ?  : 1 ;

     p    1    ? 1    ?    :  1  : 1 ;
     p    0    1 ?    ?    :  0  : 0 ;

     n    ?    ? ?    ?    :  ?  : - ;
     ?    *    ? ?    ?    :  ?  : - ;

     ?    ?    0 1    ?    :  ?  : 1 ;
     ?    ?    * 1    ?    :  1  : 1 ;

     ?    ?    1 0    ?    :  ?  : 0 ;
     ?    ?    1 *    ?    :  0  : 0 ;
     ?    ?    ? ?    *    :  ?  : x ;  // At any notifier event
                                         // output x
endtable
endprimitive



module dff(q, qbar, clock, data, preset, clear);
output q, qbar;
input clock, data, preset, clear;
reg notifier;

and (enable, preset, clear);
not (qbar, ffout);
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
    buf (q, ffout);
    posdff_udp (ffout, clock, data, preset, clear, notifier);

    specify
        // Define timing check specparam values
        specparam tSU = 10, tHD = 1, tPW = 25, tWPC = 10, tREC = 5;
        // Define module path delay rise and fall min:typ:max values
        specparam tPLHc = 4:6:9 , tPHLc = 5:8:11;
        specparam tPLHpc = 3:5:6 , tPHLpc = 4:7:9;

        // Specify module path delays
        (clock *> q,qbar) = (tPLHc, tPHLc);
        (preset,clear *> q,qbar) = (tPLHpc, tPHLpc);

        // Setup time : data to clock, only when preset and clear are 1
        $setup(data, posedge clock &&& enable, tSU, notifier);

        // Hold time: clock to data, only when preset and clear are 1
        $hold(posedge clock, data &&& enable, tHD, notifier);

        // Clock period check
        $period(posedge clock, tPW, notifier);
        // Pulse width : preset, clear
        $width(negedge preset, tWPC, 0, notifier);
        $width(negedge clear, tWPC, 0, notifier);

        // Recovery time: clear or preset to clock
        $recovery(posedge preset, posedge clock, tREC, notifier);
        $recovery(posedge clear, posedge clock, tREC, notifier);
    endspecify
    endmodule
```

NOTE—This model applies to edge-sensitive UDPs only; for level-sensitive models, an additional UDP for `x` propagation has to be generated.

15.5.1 Requirements for accurate simulation

In order to accurately model negative value timing checks, the following requirements apply:

a) A timing violation shall be triggered if the signal changes in the violation window, exclusive of the end points. Violation windows smaller than two units of simulation precision cannot yield timing violations.
b) The value of the latched data shall be the one that is stable during the violation window, again, exclusive of the end points.

To facilitate these modeling requirements, delayed copies of the data and reference signals are generated in the timing checks, and these are used internally for timing check evaluation at run time. The setup and hold times used internally are adjusted to shift the violation window and make it overlap the reference signal.

Delayed data and reference signals can be declared within the timing check so they can be used in the model's functional implementation to ensure accurate simulation. If no delayed signals are declared in the timing check and if a negative setup or hold value is present, then implicit delayed signals are created. Because implicit delayed signals cannot be used in defining model behavior, such a model can possibly behave incorrectly.

For example:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Example 1

```
$setuphold(posedge CLK, DATA, -10, 20);
```

Implicit delayed signals shall be created for `CLK` and `DATA`, but it shall not be possible to access them. The **$setuphold** check shall be properly evaluated, but functional behavior shall not always be accurate. The old `DATA` value shall be incorrectly clocked in if `DATA` transitions between `posedge CLK` and `10` time units later.

Example 2

```
$setuphold(posedge CLK, DATA1, -10, 20);
$setuphold(posedge CLK, DATA2, -15, 18);
```

Implicit delayed signals shall be created for `CLK`, `DATA1`, and `DATA2`, one for each. Even though `CLK` is referenced in two different timing checks, only one implicit delayed signal is created, and it is used for both timing checks.

Example 3

If a given signal has a delayed signal in some timing checks but not in others, the delayed signal shall be used in both cases:

```
$setuphold(posedge CLK, DATA1, -10, 20,,,, del_CLK, del_DATA1);
$setuphold(posedge CLK, DATA2, -15, 18);
```

Explicit delayed signals of `del_CLK` and `del_DATA1` are created for `CLK` and `DATA1`, while an implicit delayed signal is created for `DATA2`. In other words, `CLK` has only one delayed signal created for it, `del_CLK`, rather than one explicit delayed signal for the first check and another implicit delayed signal for the second check.

The delayed versions of the signals, whether implicit or explicit, shall be used in the **$setup**, **$hold**, **$setuphold**, **$recovery**, **$removal**, **$recrem**, **$width**, **$period**, and **$nochange** timing checks; and these checks shall have their limits adjusted accordingly. This ensures the notifier shall be toggled at the proper moment. If the adjusted limit becomes less than or equal to `0`, the limit shall be set to `0`, and the simulator shall issue a warning.

The delayed versions of the signals shall not be used for the **$skew**, **$fullskew**, and **$timeskew** timing checks because it can possibly result in the reversal of the order of signal transitions. This causes the notifiers for these timing checks to toggle at the wrong time relative to the rest of the model, perhaps resulting in transitions to `X` due to a timing check violation being cancelled. This issue shall be addressed in the model, possibly by using separate notifiers for these checks.

It is possible for a set of negative timing check values to be mutually inconsistent and produce no solution for the delay values of delayed signals. In these situations, the simulator shall issue a warning message. The inconsistency shall be resolved by changing the smallest negative limit value to `0` and recalculating the delays for the delayed signals, and this shall be repeated until a solution is reached. This procedure shall always produce a solution because in the worst case all negative limit values become `0` and no delayed signals are needed.

The delayed timing check signals are only actually delayed when negative limit values are present. If a timing check signal becomes delayed by more than the propagation delay from that signal to an output, that output shall take longer than its propagation delay to change. It shall instead transition at the same time that the delayed timing check signal changes. Thus, the output shall behave as if its specify path delay were equal

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

to the delay applied to the timing check signal. This situation can only arise when unique setup/hold or removal/recovery times are given for each edge of the data signal.

For example:

```
(CLK = Q) = 6;
$setuphold (posedge CLK, posedge D, -3,  8, , , , dCLK, dD);
$setuphold (posedge CLK, negedge D, -7, 13, , , , dCLK, dD);
```

The setup time of `-7` (the larger in absolute value of `-3` and `-7`) creates a delay of `7` for `dCLK`; therefore, output `Q` shall not change until `7` time units after a positive edge on `CLK`, rather than the 6 time units given in the specify path.

15.5.2 Conditions in negative timing checks

Conditions can be associated with both the reference and data signals by using the `&&&` operator; but when either the setup or hold time is negative, the conditions need to be paired with reference and data signals in a more flexible way. This example illustrates why.

This pair of **$setup** and **$hold** checks works together to provide the same check as a single **$setuphold**:

```
$setup (data, clk &&& cond1, tsetup, ntfr);
$hold (clk, data &&& cond1, thold, ntfr);
```

`clk` is the timecheck event for the **$setup** check, while `data` is the timecheck event for the **$hold** check. This cannot be represented in a single **$setuphold** check; therefore, additional arguments are provided to make this possible. These arguments are timestamp_cond and timecheck_cond, and they immediately follow the notifier (see 15.2.3). The following **$setuphold** check is equivalent to the separate **$setup** and **$hold** checks shown above:

```
$setuphold( clk, data, tsetup, thold, ntfr, , cond1);
```

The timestamp_cond argument is null, while the timecheck_cond argument is `cond1`.

The timestamp_cond and timecheck_cond arguments are associated with either the reference or data signals based on which delayed version of these signals occurs first. timestamp_cond is associated with the delayed signal that transitions first, while timecheck_cond is associated with the delayed signal that transitions second.

Delayed signals are only created for the reference and data signals and not for any condition signals associated with them. Therefore, timestamp_cond and timecheck_cond are not implicitly delayed by the simulator. Delayed condition signals for the timestamp_cond and timecheck_cond fields can be created by making them a function of the delayed signals.

For example:

```
assign TE_cond_D  = (dTE !== 1'b1);
assign TE_cond_TI = (dTE !== 1'b0);
assign DXTI_cond  = (dTI !==   dD);

specify
   $setuphold(posedge CP, D, -10,  20, notifier, ,TE_cond_D,  dCP, dD);
   $setuphold(posedge CP, TI, 20, -10, notifier, ,TE_cond_TI, dCP, dTI);
   $setuphold(posedge CP, TE, -4,   8, notifier, ,DXTI_cond,  dCP, dTE);
endspecify
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The assign statements create condition signals that are functions of the delayed signals. Creating delayed signal conditions synchronizes the conditions with the delayed versions of the reference and data signals used to perform the checks.

The first **$setuphold** has a negative setup time; therefore, the timecheck condition `TE_cond_D` is associated with data signal `D`. The second **$setuphold** has a negative hold time; therefore, the timecheck condition `TE_cond_TI` is associated with reference signals `CP`. The third **$setuphold** has a negative setup time; therefore, the timecheck condition `DXTI_cond` is associated with data signal `TE`.

The violation windows for the example are shown in Figure 15-4.

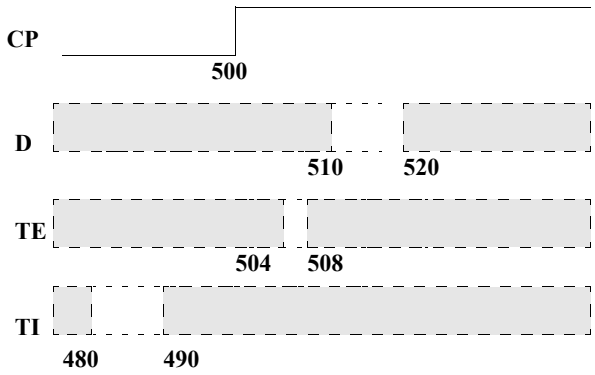


Figure 15-4—Timing check violation windows

These are the delay values calculated for the delayed signals:

```
dCP      10.01
dD        0.00
dTI      20.02
dTE       2.02
```

Use of delayed signals in creating the signals for the timestamp_cond and timecheck_cond arguments is not required, but it is usually closer to actual device behavior.

15.5.3 Notifiers in negative timing checks

Because the reference and data signals are delayed internally, the detection of the timing violation is also delayed. Notifier regs in negative timing checks shall be toggled when the timing check detects a timing violation, which occurs when the delayed signals as measured by the adjusted timing check values are in violation, not when the undelayed signals at the model inputs as measured by the original timing check values are in violation.

15.5.4 Option behavior

As already mentioned, the ability of Verilog simulators to handle negative values in **$setuphold** and **$recrem** timing checks shall be enabled with an invocation option. It is possible models written to accept negative timing check values with delayed reference and/or delayed data signals can be run without this invocation option enabled. In this circumstance, the delayed reference and data signals become copies of the original reference and data signals. The same occurs if an invocation option turning off all timing checks is used.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

15.6 Enabling timing checks with conditioned events

A construct called a *conditioned event* ties the occurrence of timing checks to the value of a conditioning signal. Syntax 15-16 shows the syntax for controlled timing check event.

```
timing_check_event ::= (From A.7.5.3)
      [timing_check_event_control] specify_terminal_descriptor [ &&& timing_check_condition ]
controlled_timing_check_event ::=
      timing_check_event_control specify_terminal_descriptor [ &&& timing_check_condition ]
timing_check_event_control ::=
        posedge
      | negedge
      | edge_control_specifier
specify_terminal_descriptor ::=
        specify_input_terminal_descriptor
      | specify_output_terminal_descriptor
timing_check_condition ::=
        scalar_timing_check_condition
      | ( scalar_timing_check_condition )
scalar_timing_check_condition ::=
        expression
      | ~ expression
      | expression == scalar_constant
      | expression === scalar_constant
      | expression != scalar_constant
      | expression !== scalar_constant
scalar_constant ::=
      1'b0 | 1'b1 | 1'B0 | 1'B1 | 'b0 | 'b1 | 'B0 | 'B1 | 1 | 0
```

Syntax 15-16—Syntax for controlled timing check event

The comparisons used in the condition can be deterministic, as in ===, !==, ~, or no operation, or nondeterministic, as in == or !=. When comparisons are deterministic, an `x` value on the conditioning signal shall not enable the timing check. For nondeterministic comparisons, an `x` on the conditioning signal shall enable the timing check.

The conditioning signal shall be a scalar net; if a vector net or an expression resulting in a multibit value is used, then the least significant bit of the vector net or the expression value is used.

If more than one conditioning signal is required for conditioning timing checks, appropriate logic shall be combined in a separate signal outside the specify block, which can be used as the conditioning signal.

For example:

Example 1—To illustrate the difference between conditioned and unconditioned timing check events, consider the following example with unconditioned timing check:

```
$setup( data, posedge clk, 10 );
```

Here, a setup timing check shall occur every time there is a positive edge on the signal `clk`.

To trigger the setup check on the positive edge on the signal `clk` only when the signal `clr` is high, rewrite the command as

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
$setup( data, posedge clk &&& clr, 10 ) ;
```

Example 2—This example shows two ways to trigger the same timing check as in Example 1 (on the positive `clk` edge) only when `clr` is low. The second method uses the === operator, which makes the comparison deterministic.

```
$setup( data, posedge clk &&& (~clr), 10 ) ;
$setup( data, posedge clk &&& (clr===0), 10 );
```

Example 3—To perform the previous sample setup check on the positive `clk` edge only when `clr` and `set` are high, add the following statement outside the specify block:

```
and new_gate( clr_and_set, clr, set );
```

Then add the condition to the timing check using the signal `clr_and_set` as follows:

```
$setup( data, posedge clk &&& clr_and_set, 10 );
```

15.7 Vector signals in timing checks

Either or both signals in a timing check can be a vector. This shall be interpreted as a single timing check where the transition of one or more bits of a vector is considered a single transition of that vector.

For example:

```
module DFF (Q, CLK, DAT);
input CLK;
input [7:0] DAT;
output [7:0] Q;
always @(posedge clk)
Q = DAT;
specify
$setup (DAT, posedge CLK, 10);
endspecify
endmodule
```

If `DAT` transitions from `'b00101110` to `'b01010011` at time `100` and if `CLK` transitions from `0` to `1` at time `105`, then the **$setup** timing check shall still only report a single timing violation.

Simulators may provide an option causing vectors in timing checks to result in the creation of multiple single-bit timing checks. For timing checks with only a single signal, such as **$period** or **$width**, a vector of width `N` results in `N` unique timing checks. For timing checks with two signals, such as **$setup**, **$hold**, **$setuphold**, **$skew**, **$timeskew**, **$fullskew**, **$recovery**, **$removal**, **$recrem**, and **$nochange**, where `M` and `N` are the widths of the signals, the result is `M*N` unique timing checks. If there is a notifier, all the timing checks trigger that notifier.

With such an option enabled, the above example yields six timing violation because 6 bits of `DAT` transitioned.

15.8 Negative timing checks

Both the **$setuphold** and **$recrem** timing checks can accept negative values when the negative timing check option is enabled. The behavior of these two timing checks is identical with respect to negative values. The

 Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

descriptions in this subclause are for the **$setuphold** timing check, but apply equally to the **$recrem** timing check.

The setup and hold timing check values define a timing violation window with respect to the reference signal edge during which the data shall remain constant. Any change of the data during the specified window causes a timing violation. The timing violation is reported, and through the notifier reg, other actions can take place in the model, such as forcing the output of a flip-flop to `x` when it detects a timing violation.

A positive value for both setup and hold times implies this violation window straddles the reference signal shown in Figure 15-5.

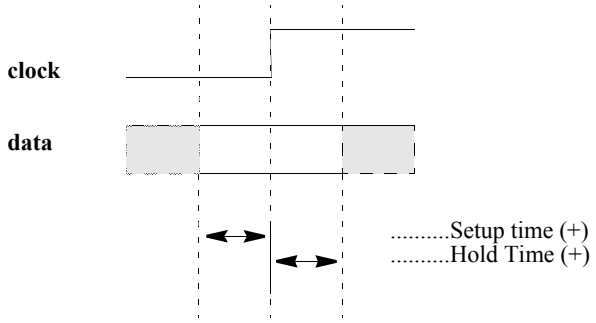


Figure 15-5—Data constraint interval, positive setup/hold

A negative hold or setup time means the violation window is shifted to either before or after the reference edge. This can happen in a real device because of disparate internal device delays between the internal clock and data signal paths. These internal device delays are illustrated in Figure 15-6 showing how significant differences in these delays can cause negative setup or hold values.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

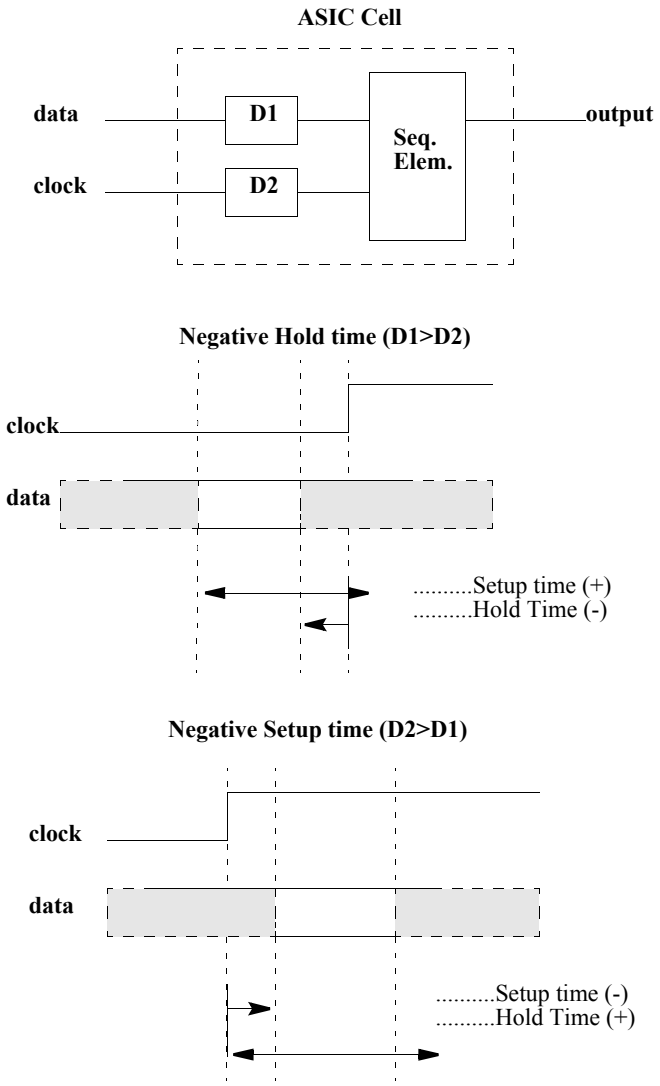


Figure 15-6—Data constraint interval, negative setup/hold

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

16. Backannotation using the standard delay format (SDF)

SDF files contain timing values for specify path delays, specparam values, timing check constraints, and interconnect delays. SDF files can also contain other information in addition to simulation timing, but these need not concern Verilog simulation. The timing values in SDF files usually come from application-specific integrated circuit (ASIC) delay calculation tools that take advantage of connectivity, technology, and layout geometry information.

Verilog backannotation is the process by which timing values from the SDF file update specify path delays, specparam values, timing constraint values, and interconnect delays.

All this information is covered further in IEEE Std 1497™-2001 [B1][9].

16.1 The SDF annotator

The term *SDF annotator* refers to any tool capable of backannotating SDF data to a Verilog simulator. It shall report a warning for any data it is unable to annotate.

An SDF file can contain many constructs that are not related to specify path delays, specparam values, timing check constraint values, or interconnect delays. An example is any construct in the `TIMINGENV` section of the SDF file. All constructs unrelated to Verilog timing shall be ignored without any warnings issued.

Any Verilog timing value for which the SDF file does not provide a value shall not be modified during the backannotation process, and its prebackannotation value shall be unchanged.

16.2 Mapping of SDF constructs to Verilog

SDF timing values appear within a `CELL` declaration, which can contain one or more of `DELAY`, `TIMINGCHECK`, and `LABEL` sections. The `DELAY` section contains propagation delay values for specify paths and interconnect delays. The `TIMINGCHECK` section contains timing check constraint values. The `LABEL` section contains new values for specparams. Backannotation into Verilog is done by matching SDF constructs to the corresponding Verilog declarations and then replacing the existing Verilog timing values with those from the SDF file.

16.2.1 Mapping of SDF delay constructs to Verilog declarations

When annotating `DELAY` constructs that are not interconnect delays (covered in 16.2.3), the SDF annotator looks for specify paths where the names and conditions match. When annotating `TIMINGCHECK` constructs, the SDF annotator looks for timing checks of the same type where the names and conditions match. Table 16-1 shows which Verilog structures can be annotated by each SDF construct in the `DELAY` section.

Table 16-1—Mapping of SDF delay constructs to Verilog declarations

SDF construct	Verilog annotated structure
(`PATHPULSE`...	Conditional and nonconditional specify path pulse limits
(`PATHPULSEPERCENT`...	Conditional and nonconditional specify path pulse limits

[9]The numbers in brackets correspond to those of the bibliography in Annex I.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 16-1—Mapping of SDF delay constructs to Verilog declarations *(continued)*

SDF construct	Verilog annotated structure
(IOPATH...	Conditional and nonconditional specify path delays/pulse limits
(IOPATH (RETAIN...	Conditional and nonconditional specify path delays/pulse limits, RETAIN may be ignored
(COND (IOPATH...	Conditional specify path delays/pulse limits
(COND (IOPATH (RETAIN...	Conditional specify path delays/pulse limits, RETAIN may be ignored
(CONDELSE (IOPATH...	ifnone
(CONDELSE (IOPATH (RETAIN...	ifnone, RETAIN may be ignored
(DEVICE...	All specify paths to module outputs. If no specify paths, all primitives driving module outputs.
(DEVICE port_instance...	If port_instance is a module instance, all specify paths to module outputs. If no specify paths, all primitives driving module outputs. If port_instance is a module instance output, all specify paths to that module output. If no specify path, all primitives driving that module output.

In the following example, the source SDF signal `sel` matches the source Verilog signal, and the destination SDF signal `zout` also matches the destination Verilog signal. Therefore, the rise/fall times of `1.3` and `1.7` are annotated to the specify path.

SDF file:

```
(IOPATH sel zout (1.3) (1.7))
```

Verilog specify path:

```
(sel => zout) = 0;
```

A conditional `IOPATH` delay between two ports shall annotate only to Verilog specify paths between those same two ports with the same condition. In the following example, the rise/fall times of `1.3` and `1.7` are annotated only to the second specify path:

SDF file:

```
(COND mode (IOPATH sel zout (1.3) (1.7)))
```

Verilog specify paths:

```
if (!mode) (sel => zout) = 0;
if (mode) (sel => zout) = 0;
```

A nonconditional `IOPATH` delay between two ports shall annotate to all Verilog specify paths between those same two ports. In the following example, the rise/fall times of `1.3` and `1.7` are annotated to both specify paths:

SDF file:

```
(IOPATH sel zout (1.3) (1.7))
```


Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Verilog specify paths:

```
if (!mode) (sel => zout) = 0;
if (mode) (sel => zout) = 0;
```

16.2.2 Mapping of SDF timing check constructs to Verilog

Table 16-2 shows which Verilog timing checks are annotated to by each type of SDF timing check. `v1` is the first value of a timing check, `v2` is the second value, while `x` indicates no value is annotated.

Table 16-2—Mapping of SDF timing check constructs to Verilog

SDF timing check	Annotated Verilog timing checks
(SETUP v1...	$setup(v1), $setuphold(v1,x)
(HOLD v1...	$hold(v1), $setuphold(x,v1)
(SETUPHOLD v1 v2...	$setup(v1), $hold(v2), $setuphold(v1,v2)
(RECOVERY v1...	$recovery(v1), $recrem(v1,x)
(REMOVAL v1...	$removal(v1), $recrem(x,v1)
(RECREM v1 v2...	$recovery(v1), $removal(v2), $recrem(v1,v2)
(SKEW v1...	$skew(v1)
(TIMESKEW v1...[a]	$timeskew(v1)
(FULLSKEW v1 v2...[a]	$fullskew(v1,v2)
(WIDTH v1...	$width(v1,x)
(PERIOD v1...	$period(v1)
(NOCHANGE v1 v2...	$nochange(v1,v2)

[a]Not part of current SDF standard

The reference and data signals of timing checks can have logical condition expressions and edges associated with them. An SDF timing check with no conditions or edges on any of its signals shall match all corresponding Verilog timing checks regardless of whether conditions are present. In the following example, the SDF timing check shall annotate to all the Verilog timing checks:

SDF file:

```
(SETUPHOLD data clk (3) (4))
```

Verilog timing checks:

```
$setuphold (posedge clk &&&  mode, data, 1, 1, ntfr);
$setuphold (negedge clk &&& !mode, data, 1, 1, ntfr);
```

When conditions and/or edges are associated with the signals in an SDF timing check, then they shall match those in any corresponding Verilog timing check before annotation shall happen. In the following example, the SDF timing check shall annotate to the first Verilog timing check, but not the second:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

SDF file:

```
(SETUPHOLD data (posedge clk) (3) (4))
```

Verilog timing checks:

```
$setuphold (posedge clk &&&  mode, data, 1, 1, ntfr); // Annotated
$setuphold (negedge clk &&& !mode, data, 1, 1, ntfr); // Not annotated
```

Here, the SDF timing check shall not annotate to any of the Verilog timing checks:

SDF file:

```
(SETUPHOLD data (COND !mode (posedge clk)) (3) (4))
```

Verilog timing checks:

```
$setuphold (posedge clk &&&  mode, data, 1, 1, ntfr); // Not annotated
$setuphold (negedge clk &&& !mode, data, 1, 1, ntfr); // Not annotated
```

16.2.3 SDF annotation of specparams

The SDF `LABEL` construct annotates to specparams. Any expression containing one or more specparams is reevaluated when annotated to from an SDF file.

The following example shows SDF `LABEL` constructs annotating to specparams in a Verilog module. The specparams are used in procedural delays to control when the clock transitions. The SDF `LABEL` construct annotates the values of `dhigh` and `dlow`, thereby setting the period and duty cycle of the clock.

SDF file:

```
(LABEL
   (ABSOLUTE
      (dhigh 60)
      (dlow 40)))
```

Verilog file:

```
module clock(clk);
output clk;
reg clk;
specparam dhigh=0, dlow=0;
initial clk = 0;
always
   begin
   #dhigh clk = 1;  // Clock remains low for time dlow
                    // before transitioning to 1
   #dlow  clk = 0;  // Clock remains high for time dhigh
                    // before transitioning to 0
   end;
endmodule
```

The following example shows a specparam in an expression of a specify path. The SDF `LABEL` construct can be used to change the value of the specparam and cause reevaluation of the expression.

```
specparam cap = 0;
...
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
specify
    (A => Z) = 1.4 * cap + 0.7;
endspecify
```

16.2.4 SDF annotation of interconnect delays

SDF interconnect delay annotation differs from annotation of other constructs described above in that there exists no corresponding Verilog declaration to which to annotate. In Verilog simulation, interconnect delays are an abstraction that represents the signal propagation delay from an output or inout module port to an input or inout module port. The `INTERCONNECT` construct includes a source, a load, and delay values, while the `PORT` and `NETDELAY` constructs include only a load and delay values. Interconnect delays can only be annotated between module ports, never between primitive pins. Table 16-3 shows how the SDF interconnect constructs in the `DELAY` section are annotated.

Table 16-3—SDF annotation of interconnect delays

SDF construct	Verilog annotated structure
(`PORT`...	Interconnect delay
(`NETDELAY` [a]	Interconnect delay
(`INTERCONNECT`...	Interconnect delay

[a]Only OVI SDF version 1.0, 2.0, and 2.1 and IEEE SDF version 4.0

Interconnect delays can be annotated to both single source and multisource nets.

When annotating a `PORT` construct, the SDF annotator shall search for the port and. if it exists, shall annotate an interconnect delay to that port that shall represent the delay from all sources on the net to that port.

When annotating a `NETDELAY` construct, the SDF annotator shall check to see if it is annotating to a port or a net. If it is a port, then the SDF annotator shall annotate an interconnect delay to that port. If it is a net, then it shall annotate an interconnect delay to all load ports connected to that net. If the port or net has more than one source, then the delay shall represent the delay from all sources. `NETDELAY` delays can only be annotated to input or inout module ports or to nets.

In the case of multisource nets, unique delays can be annotated between each source/load pair using the `INTERCONNECT` construct. When annotating this construct, the SDF annotator shall find the source port and the load port; and if both exist, it shall annotate an interconnect delay between the two. If the source port is not found or if the source port and the load port are not actually on the same net, then a warning message is issued, but the delay to the load port is annotated anyway. If this happens for a load port that is part of a multisource net, then the delay is treated as if it were the delay from all source ports, which is the same as the annotation behavior for a `PORT` delay. Source ports shall be output or inout ports, while load ports shall be input or inout ports.

Interconnect delays share many of the characteristics of specify path delays. The same rules for specify path delays for filling in missing delays and pulse limits also apply for interconnect delays. Interconnect delays have twelve transition delays, and unique reject and error pulse limits are associated with each of the twelve. An unlimited number of future schedules are permitted.

In a Verilog module, a reference to an annotated port, wherever it occurs, whether in `$monitor` and `$display` statements or in expressions, shall provide the delayed signal value. A reference to the source

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

shall yield the undelayed signal value, while a reference to the load shall yield the delayed signal value. In general, references to the signal value hierarchically before the load shall yield the undelayed signal value, while references to the signal at or hierarchically after the load shall yield the delayed signal value. An annotation to a hierarchical port shall affect all connected ports at higher or lower hierarchical levels, depending on the direction of annotation. An annotation from a source port shall be interpreted as being from all sources hierarchically higher or lower than that source port.

Up-hierarchy annotations shall be properly handled. This situation arises when the load is hierarchically above the source. The delay to all ports that are hierarchically above the load or that connect to the net at points hierarchically above the load is the same as the delay to that load.

Down-hierarchy annotation shall also be properly handled. This situation arises when the source is hierarchically above the load. The delay to the load is interpreted as being from all ports that are at or above the source or that connect to the net at points hierarchically above the source.

Hierarchically overlapping annotations are permitted. This occurs when annotations to or from the same port take place at different hierarchical levels and, therefore, do not correspond to the same hierarchical subset of ports. In the following example, the first `INTERCONNECT` statement annotates to all ports of the net that are at or hierarchically within `i53/selmode`, while the second annotates to a smaller subset of ports, only those at or hierarchically within `i53/u21/in`:

```
(INTERCONNECT i14/u5/out i53/selmode (1.43) (2.17))
(INTERCONNECT i14/u5/out i53/u21/in  (1.58) (1.92))
```

Overlapping annotations can occur in many different ways, particularly on multisource/multiload nets, and SDF annotation shall properly resolve all the interactions.

16.3 Multiple annotations

SDF annotation is an ordered process. The constructs from the SDF file are annotated in their order of occurrence. In other words, annotation of an SDF construct can be changed by annotation of a subsequent construct that either modifies `(INCREMENT)` or overwrites `(ABSOLUTE)` it. These do not have to be the same construct. The following example first annotates pulse limits to an `IOPATH` and then annotates the entire `IOPATH`, thereby overwriting the pulse limits that were just annotated:

```
(DELAY
   (ABSOLUTE
      (PATHPULSE A Z (2.1) (3.4))
      (IOPATH A Z (3.5) (6.1))
```

Overwriting the pulse limits can be avoided by using empty parentheses to hold the current values of the pulse limits:

```
(DELAY
   (ABSOLUTE
      (PATHPULSE A Z (2.1) (3.4))
      (IOPATH A Z ((3.5) () ()) ((6.1) () ()) )
```

The above annotation can be simplified into a single statement like this:

```
(DELAY
   (ABSOLUTE
      (IOPATH A Z ((3.5) (2.1) (3.4)) ((6.1) (2.1) (3.4)) )
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

A PORT annotation followed by an INTERCONNECT annotation to the same load shall cause only the delay from the INTERCONNECT source to be affected. For the following net with three sources and a single load, the delay from all sources except i13/out remains 6:

```
(DELAY
   (ABSOLUTE
      (PORT i15/in (6))
      (INTERCONNECT i13/out i15/in (5))
```

An INTERCONNECT annotation followed by a PORT annotation shall cause the INTERCONNECT annotation to be overwritten. Here, the delays from all sources to the load shall become 6:

```
(DELAY
   (ABSOLUTE
      (INTERCONNECT i13/out i15/in (5))
      (PORT i15/in (6))
```

16.4 Multiple SDF files

More than one SDF file can be annotated. Each call to the **$sdf_annotate** task annotates the design with timing information from an SDF file. Annotated values either modify (INCREMENT) or overwrite (ABSOLUTE) values from earlier SDF files. Different regions of a design can be annotated from different SDF files by specifying the region's hierarchy scope as the second argument to **$sdf_annotate**.

16.5 Pulse limit annotation

For SDF annotation of delays (not timing constraints), the default values annotated for pulse limits shall be calculated using the percentage settings for the reject and error limits. By default, these limits are 100%, but they can be modified through invocation options. For example, assuming invocation options have set the reject limit to 40% and the error limit to 80%, the following SDF construct shall annotate a delay of 5, a reject limit of 2, and an error limit of 4:

```
(DELAY
   (ABSOLUTE
      (IOPATH A Z (5))
```

Given that the specify path delay was originally 0, the following annotation results in a delay of 5 and pulse limits of 0:

```
(DELAY
   (ABSOLUTE
      (IOPATH A Z ((5) () ()) )
```

Annotations in INCREMENT mode can result in pulse limits less than 0, in which case they shall be adjusted to 0. For example, if the specify path pulse limits were both 3, the following annotation results in a 0 value for both pulse limits:

```
(DELAY
   (INCREMENT
      (IOPATH A Z (() (-4) (-5)) )
```

There are two SDF constructs that annotate only to pulse limits, PATHPULSE and PATHPULSEPERCENT. They do not affect the delay. When PATHPULSE sets the pulse limits to values greater than the delay, Verilog shall exhibit the same behavior as if the pulse limits had been set equal to the delay.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

16.6 SDF to Verilog delay value mapping

Verilog specify paths and interconnects can have unique delays for up to twelve state transitions (see 14.3.1). All other constructs, such as gate primitives and continuous assignments, can have only three state transition delays (see 7.14).

For Verilog specify path and interconnect delays, the number of transition delay values provided by SDF might be less than twelve.

Table 16-4 shows how fewer than twelve SDF delays are extended to be twelve delays. The Verilog transition types are shown down the left-hand side, while the number of SDF delays provided is shown across the top. The SDF values are given the names `v1` through `v12`.

Table 16-4—SDF to Verilog delay value mapping

Verilog transition	Number of SDF delay values provided				
	1 value	2 values	3 values	6 values	12 values
0 -> 1	v1	v1	v1	v1	v1
1 -> 0	v1	v2	v2	v2	v2
0 -> z	v1	v1	v3	v3	v3
z -> 1	v1	v1	v1	v4	v4
1 -> z	v1	v2	v3	v5	v5
z -> 0	v1	v2	v2	v6	v6
0 -> x	v1	v1	min(v1,v3)	min(v1,v3)	v7
x -> 1	v1	v1	v1	max(v1,v4)	v8
1 -> x	v1	v2	min(v2,v3)	min(v2,v5)	v9
x -> 0	v1	v2	v2	max(v2,v6)	v10
x -> z	v1	max(v1,v2)	v3	max(v3,v5)	v11
z -> x	v1	min(v1,v2)	min(v1,v2)	min(v4,v6)	v12

For other delays that can have at most three values, the expansion of less than three SDF delays into three Verilog delays is covered in Table 7-9. More than three SDF delays are reduced to three Verilog delays by simply ignoring the extra delays. The delay to the `x`-state is created from the minimum of the other three delays.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

17. System tasks and functions

This clause describes system tasks and functions that are considered part of the Verilog HDL. These system tasks and functions are divided into ten categories as follows:

Display tasks [17.1]

$display
$displayb
$displayh
$displayo
$strobe
$strobeb
$strobeh
$strobeo
$write
$writeb
$writeh
$writeo
$monitor
$monitorb
$monitorh
$monitoro
$monitoroff
$monitoron

File I/O tasks [17.2]

$fclose
$fdisplay
$fdisplayb
$fdisplayh
$fdisplayo
$fstrobe
$fstrobeb
$fstrobeh
$fstrobeo
$swrite
$swriteb
$swriteh
$swriteo
$fscanf
$fread
$fseek
$fflush
$feof
$sdf_annotate
$fopen
$fwrite
$fwriteb
$fwriteh
$fwriteo
$fmonitor
$fmonitorb
$fmonitorh
$fmonitoro
$sformat
$fgetc
$ungetc
$fgets
$sscanf
$rewind
$ftell
$ferror
$readmemb
$readmemh

Timescale tasks [17.3]

$printtimescale
$timeformat

Simulation control tasks [17.4]

$finish
$stop

PLA modeling tasks [17.5]

$async$and$array
$async$nand$array
$async$or$array
$async$and$plane
$async$nand$plane
$async$or$plane
$async$nor$array
$async$nor$plane
$sync$and$array
$sync$and$plane
$sync$nand$array
$sync$nand$plane
$sync$or$array
$sync$or$plane
$sync$nor$array
$sync$nor$plane

Stochastic analysis tasks [17.6]

$q_initialize
$q_remove
$q_exam
$q_add
$q_full

Simulation time functions [17.7]

$realtime
$time
$stime

Conversion functions [17.8]

$bitstoreal
$itor
$signed
$realtobits
$rtoi
$unsigned

Probabilistic distribution functions [17.9]

$random
$dist_erlang
$dist_normal
$dist_t
$dist_chi_square
$dist_exponential
$dist_poisson
$dist_uniform

Command line input [17.10]

$test$plusargs
$value$plusargs

Math functions [17.11]

$clog2
$ln
$log10
$exp
$sqrt
$pow
$floor
$ceil
$sin
$cos
$tan
$asin
$acos
$atan
$atan2
$hypot
$sinh
$cosh
$tanh
$asinh
$acosh
$atanh

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

These utility tasks and functions provide some broadly useful capabilities. The behavior of these tasks and functions is described in 17.1 through 17.11. Additional tasks for value change dump (VCD) are described in Clause 18.

17.1 Display system tasks

The display group of system tasks is divided into three categories: the display and write tasks, strobed monitoring tasks, and continuous monitoring tasks.

17.1.1 The display and write tasks

The syntax for **$display** and **$write** system tasks is shown in Syntax 17-1.

```
display_tasks ::=
        display_task_name [ ( list_of_arguments ) ] ;
display_task_name ::=
          $display | $displayb | $displayo | $displayh
        | $write   | $writeb   | $writeo   | $writeh
```

Syntax 17-1—Syntax for $display and $write system tasks

These are the main system task routines for displaying information. The two sets of tasks are identical except that **$display** automatically adds a newline character to the end of its output, whereas the **$write** task does not.

The **$display** and **$write** tasks display their arguments in the same order as they appear in the argument list. Each argument can be a quoted string, an expression that returns a value, or a null argument.

The contents of string arguments are output literally except when certain escape sequences are inserted to display special characters or to specify the display format for a subsequent expression.

Escape sequences are inserted into a string in three ways:

— The special character \ indicates that the character to follow is a literal or nonprintable character (see Table 17-1).
— The special character `%` indicates that the next character should be interpreted as a format specification that establishes the display format for a subsequent expression argument (see Table 17-2). For each `%` character (except `%m` and `%%`) that appears in a string, a corresponding expression argument shall be supplied after the string.
— The special character string `%%` indicates the display of the percent sign character `%` (see Table 17-1).

Any null argument produces a single space character in the display. (A null argument is characterized by two adjacent commas in the argument list.)

The **$display** task, when invoked without arguments, simply prints a newline character. A **$write** task supplied without arguments prints nothing at all.

17.1.1.1 Escape sequences for special characters

The escape sequences given in Table 17-1, when included in a string argument, cause special characters to be displayed.

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 17-1—Escape sequences for printing special characters

Argument	Description
\n	The newline character
\t	The tab character
\\	The \ character
\"	The " character
\ddd	A character specified in 1–3 octal digits ($0 \le d \le 7$). If fewer than three characters are used, the following character shall not be an octal digit. Implementations may issue an error if the character represented is greater than \377.
%%	The % character

For example:

```
module disp;
initial begin
  $display("\\\t\\\n\"\123");
end
endmodule
```

Simulating this example shall display the following:

```
\       \
"S
```

17.1.1.2 Format specifications

Table 17-2 shows the escape sequences used for format specifications. Each escape sequence, when included in a string argument, specifies the display format for a subsequent expression. For each `%` character (except `%m` and `%%`) that appears in a string, a corresponding expression shall follow the string in the argument list. The value of the expression replaces the format specification when the string is displayed.

Any expression argument that has no corresponding format specification is displayed using the default decimal format in **$display** and **$write**, binary format in **$displayb** and **$writeb**, octal format in **$displayo** and **$writeo**, and hexadecimal format in **$displayh** and **$writeh**.

Table 17-2—Escape sequences for format specifications

Argument	Description
%h or %H	Display in hexadecimal format
%d or %D	Display in decimal format
%o or %O	Display in octal format
%b or %B	Display in binary format
%c or %C	Display in ASCII character format

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 17-2—Escape sequences for format specifications *(continued)*

Argument	Description
%l or %L	Display library binding information
%v or %V	Display net signal strength
%m or %M	Display hierarchical name
%s or %S	Display as a string
%t or %T	Display in current time format
%u or %U	Unformatted 2 value data
%z or %Z	Unformatted 4 value data

The formatting specification `%l` (or `%L`) is defined for displaying the library information of the specific module. This information shall be displayed as "*library*.*cell*" corresponding to the library name from which the current module instance was extracted and the cell name of the current module instance. See Clause 13 for information on libraries and configuring designs.

The formatting specification `%u` (or `%U`) is defined for writing data without formatting (binary values). The application shall transfer the 2 value binary representation of the specified data to the output stream. This escape sequence can be used with any of the existing display system tasks, although `$fwrite` should be the preferred one to use. Any unknown or high-impedance bits in the source shall be treated as zero. This formatting specifier is intended to be used to support transferring data to and from external programs that have no concept of `x` and `z`. Applications that require preservation of `x` and `z` are encouraged to use the `%z` I/O format specification.

The data shall be written to the file in the native endian format of the underlying system (i.e., in the same endian order as if the PLI was used and the C language write (2) system call was used). The data shall be written in units of 32 bits with the word containing the LSB written first.

NOTE—For POSIX applications, it might be necessary to open files for unformatted I/O with the `wb`, `wb+`, or `w+b` specifiers to avoid the systems implementation of I/O altering patterns in the unformatted stream that match special characters.

The formatting specification `%z` (or `%Z`) is defined for writing data without formatting (binary values). The application shall transfer the 4 value binary representation of the specified data to the output stream. This escape sequence can be used with any of the existing display system tasks, although `$fwrite` should be the preferred one to use.

This formatting specifier is intended to be used to support transferring data to and from external programs that recognize and support the concept of `x` and `z`. Applications that do not require the preservation of `x` and `z` are encouraged to use the `%u` I/O format specification.

The data shall be written to the file in the native endian format of the underlying system [i.e., in the same endian order as if the PLI was used, the data were in a `s_vpi_vecval` structure (see Figure 27-8 in 27.14), and the C language `write(2)` system call was used to write the structure to disk]. The data shall be written in units of 32 bits with the structure containing the LSB written first.

NOTE—For POSIX applications, it might be necessary to open files for unformatted I/O with the `wb`, `wb+`, or `w+b` specifiers to avoid the systems implementation of I/O altering patterns in the unformatted stream that match special characters.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The format specifications in Table 17-3 are used with real numbers and have the full formatting capabilities available in the C language. For example, the format specification `%10.3g` specifies a minimum field width of 10 with 3 fractional digits.

Table 17-3—Format specifications for real numbers

Argument	Description
%e or %E	Display 'real' in an exponential format
%f or %F	Display 'real' in a decimal format
%g or %G	Display 'real' in exponential or decimal format, whichever format results in the shorter printed output

The net signal strength, hierarchical name, and string format specifications are described in 17.1.1.5 through 17.1.1.7.

The `%t` format specification works with the **$timeformat** system task to specify a uniform time unit, time precision, and format for reporting timing information from various modules that use different time units and precisions. The **$timeformat** task is described in 17.3.2.

For example:

```
module disp;
reg [31:0] rval;
pulldown (pd);
initial begin
   rval = 101;
   $display("rval = %h hex %d decimal",rval,rval);
   $display("rval = %o octal\nrval = %b bin",rval,rval);
   $display("rval has %c ascii character value",rval);
   $display("pd strength value is %v",pd);
   $display("current scope is %m");
   $display("%s is ascii value for 101",101);
   $display("simulation time is %t", $time);
end
endmodule
```

Simulating this example shall display the following:

```
rval = 00000065 hex         101 decimal
rval = 00000000145 octal
rval = 00000000000000000000000001100101 bin
rval has e ascii character value
pd strength value is StX
current scope is disp
e is ascii value for 101
simulation time is                    0
```

17.1.1.3 Size of displayed data

For expression arguments, the values written to the output file (or terminal) are sized automatically.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

For example, the result of a 12-bit expression would be allocated three characters when displayed in hexadecimal format and four characters when displayed in decimal format because the largest possible value for the expression is FFF (hexadecimal) and 4095 (decimal).

When displaying decimal values, leading zeros are suppressed and replaced by spaces. In other radices, leading zeros are always displayed.

The automatic sizing of displayed data can be overridden by inserting a zero between the `%` character and the letter that indicates the radix, as shown in the following example:

```
$display("d=%0h a=%0h", data, addr);
```

For example:

```
module printval;
reg [11:0] r1;
initial begin
   r1 = 10;
   $display( "Printing with maximum size - :%d: :%h:", r1,r1 );
   $display( "Printing with minimum size - :%0d: :%0h:", r1,r1 );
end
endmodule

Printing with maximum size - :  10: :00a:
Printing with minimum size - :10: :a:
```

In this example, the result of a 12-bit expression is displayed. The first call to **$display** uses the standard format specifier syntax and produces results requiring four and three columns for the decimal and hexadecimal radices, respectively. The second **$display** call uses the `%0` form of the format specifier syntax and produces results requiring two columns and one column, respectively.

17.1.1.4 Unknown and high-impedance values

When the result of an expression contains an unknown or high-impedance value, certain rules apply to displaying that value.

In decimal (`%d`) format, the rules are as follows:

- If all bits are at the unknown value, a single lowercase `x` character is displayed.
- If all bits are at the high-impedance value, a single lowercase `z` character is displayed.
- If some, but not all, bits are at the unknown value, the uppercase `X` character is displayed.
- If some, but not all, bits are at the high-impedance value, the uppercase `Z` character is displayed, unless there are also some bits at the unknown value, in which case the uppercase `X` character is displayed.
- Decimal numerals always appear right-justified in a fixed-width field.

In hexadecimal (`%h`) and octal (`%o`) formats, the rules are as follows:

- Each group of 4 bits is represented as a single hexadecimal digit; each group of 3 bits is represented as a single octal digit.
- If all bits in a group are at the unknown value, a lowercase `x` is displayed for that digit.
- If all bits in a group are at a high-impedance state, a lowercase `z` is printed for that digit.
- If some, but not all, bits in a group are unknown, an uppercase `X` is displayed for that digit.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

— If some, but not all, bits in a group are at a high-impedance state, then an uppercase `Z` is displayed for that digit, unless there are also some bits at the unknown value, in which case an uppercase `X` is displayed for that digit.

In binary (`%b`) format, each bit is printed separately using the characters `0`, `1`, `x`, and `z`.

For example:

```
   STATEMENT                                RESULT
$display("%d", 1'bx);                       x
$display("%h", 14'bx01010);                 xxXa
$display("%h %o", 12'b001xxx101x01,
                  12'b001xxx101x01);        XXX 1x5X
```

17.1.1.5 Strength format

The `%v` format specification is used to display the strength of scalar nets. For each `%v` specification that appears in a string, a corresponding scalar reference shall follow the string in the argument list.

The strength of a scalar net is reported in a three-character format. The first two characters indicate the strength. The third character indicates the current logic value of the scalar and can be any one of the values given in Table 17-4.

Table 17-4—Logic value component of strength format

Argument	Description
0	For a logic 0 value
1	For a logic 1 value
X	For an unknown value
Z	For a high-impedance value
L	For a logic 0 or high-impedance value
H	For a logic 1 or high-impedance value

The first two characters—the strength characters—are either a two-letter mnemonic or a pair of decimal digits. Usually, a mnemonic is used to indicate strength information; however, in less typical cases, a pair of decimal digits can be used to indicate a range of strength levels. Table 17-5 shows the mnemonics used to represent the various strength levels.

There are four driving strengths and three charge storage strengths. The driving strengths are associated with gate outputs and continuous assignment outputs. The charge storage strengths are associated with the **trireg** type net. (See Clause 7 for strength modeling.)

For the logic values `0` and `1`, a mnemonic is used when there is no range of strengths in the signal. Otherwise, the logic value is preceded by two decimal digits, which indicate the maximum and minimum strength levels.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 17-5—Mnemonics for strength levels

Mnemonic	Strength name	Strength level
Su	Supply drive	7
St	Strong drive	6
Pu	Pull drive	5
La	Large capacitor	4
We	Weak drive	3
Me	Medium capacitor	2
Sm	Small capacitor	1
Hi	High impedance	0

For the unknown value, a mnemonic is used when both the `0` and `1` strength components are at the same strength level. Otherwise, the unknown value `x` is preceded by two decimal digits, which indicate the `0` and `1` strength levels, respectively.

The high-impedance strength cannot have a known logic value; the only logic value allowed for this level is `z`.

For the values `L` and `H`, a mnemonic is always used to indicate the strength level.

For example:

```
always
#15 $display($time,,"group=%b signals=%v %v %v",{s1,s2,s3},s1,s2,s3);
```

The example below shows the output that might result from such a call, while Table 17-6 explains the various strength formats that appear in the output.

```
 0 group=111 signals=St1 Pu1 St1
15 group=011 signals=Pu0 Pu1 St1
30 group=0xz signals=520 PuH HiZ
45 group=0xx signals=Pu0 65X StX
60 group=000 signals=Me0 St0 St0
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 17-6—Explanation of strength formats

Argument	Description
St1	A strong driving 1 value
Pu0	A pull driving 0 value
HiZ	The high-impedance state
Me0	A 0 charge storage of medium capacitor strength
StX	A strong driving unknown value
PuH	A pull driving strength of 1 or high-impedance value
65X	An unknown value with a strong driving 0 component and a pull driving 1 component
520	An 0 value with a range of possible strength from pull driving to medium capacitor

17.1.1.6 Hierarchical name format

The `%m` format specifier does not accept an argument. Instead, it causes the display task to print the hierarchical name of the module, task, function, or named block that invokes the system task containing the format specifier. This is useful when there are many instances of the module that calls the system task. One obvious application is timing check messages in a flip-flop or latch module; the `%m` format specifier shall pinpoint the module instance responsible for generating the timing check message.

17.1.1.7 String format

The `%s` format specifier is used to print ASCII codes as characters. For each `%s` specification that appears in a string, a corresponding argument shall follow the string in the argument list. The associated argument is interpreted as a sequence of 8-bit hexadecimal ASCII codes, with each 8 bits representing a single character. If the argument is a variable, its value should be right-justified so that the rightmost bit of the value is the least significant bit of the last character in the string. No termination character or value is required at the end of a string, and leading zeros are never printed.

17.1.2 Strobed monitoring

The syntax for **$strobe** system task is shown in Syntax 17-2.

```
strobe_tasks ::=
        strobe_task_name [ ( list_of_arguments ) ] ;
strobe_task_name ::=
        $strobe | $strobeb | $strobeo | $strobeh
```

Syntax 17-2—Syntax for $strobe system tasks

The system task **$strobe** provides the ability to display simulation data at a selected time. That time is the end of the current simulation time, when all the simulation events have occurred for that simulation time, just before simulation time is advanced. The arguments for this task are specified in exactly the same manner as for the **$display** system task—including the use of escape sequences for special characters and format specifications (see 17.1.1).

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

For example:

```
forever @(negedge clock)
    $strobe ("At time %d, data is %h",$time,data);
```

In this example, **$strobe** writes the time and data information to the standard output and the log file at each negative edge of the clock. The action shall occur just before simulation time is advanced and after all other events at that time have occurred so that the data written are sure to be the correct data for that simulation time.

17.1.3 Continuous monitoring

The syntax for **$monitor** system task is shown in Syntax 17-3.

```
monitor_tasks ::=
        monitor_task_name [ ( list_of_arguments ) ] ;
      | $monitoron ;
      | $monitoroff ;
monitor_task_name ::=
      $monitor | $monitorb | $monitoro | $monitorh
```

Syntax 17-3—Syntax for $monitor system tasks

The **$monitor** task provides the ability to monitor and display the values of any variables or expressions specified as arguments to the task. The arguments for this task are specified in exactly the same manner as for the **$display** system task—including the use of escape sequences for special characters and format specifications (see 17.1.1).

When a **$monitor** task is invoked with one or more arguments, the simulator sets up a mechanism whereby each time a variable or an expression in the argument list changes value—with the exception of the **$time**, **$stime**, or **$realtime** system functions—the entire argument list is displayed at the end of the time step as if reported by the **$display** task. If two or more arguments change value at the same time, only one display is produced that shows the new values.

Only one **$monitor** display list can be active at any one time; however, a new **$monitor** task with a new display list can be issued any number of times during simulation.

The **$monitoron** and **$monitoroff** tasks control a monitor flag that enables and disables the monitoring. Use **$monitoroff** to turn off the flag and disable monitoring. The **$monitoron** system task can be used to turn on the flag so that monitoring is enabled and the most recent call to **$monitor** can resume its display. A call to **$monitoron** shall produce a display immediately after it is invoked, regardless of whether a value change has taken place; this is used to establish the initial values at the beginning of a monitoring session. By default, the monitor flag is turned on at the beginning of simulation.

17.2 File input-output system tasks and functions

The system tasks and functions for file-based operations are divided into the following categories:

— Functions and tasks that open and close files
— Tasks that output values into files
— Tasks that output values into variables
— Tasks and functions that read values from files and load into variables or memories

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

17.2.1 Opening and closing files

The syntax for **$fopen** and **$fclose** system tasks is shown in Syntax 17-4.

```
file_open_function ::=
        multi_channel_descriptor = $fopen ( " file_name " ) ;
      | fd = $fopen ( " file_name " , type ) ;
file_close_task ::=
        $fclose ( multi_channel_descriptor ) ;
      | $fclose ( fd ) ;
```

Syntax 17-4—Syntax for $fopen and $fclose system tasks

The function **$fopen** opens the file specified as the **filename** argument and returns either a 32-bit multichannel descriptor or a 32-bit file descriptor, determined by the absence or presence of the **type** argument.

filename is a character string or is a **reg** containing a character string that names the file to be opened.

type is a character string or is a **reg** containing a character string of one of the forms in Table 17-7 that indicates how the file should be opened. If **type** is omitted, the file is opened for writing, and a multichannel descriptor `mcd` is returned. If **type** is supplied, the file is opened as specified by the value of type, and a file descriptor `fd` is returned.

Table 17-7—Types for file descriptors

Argument	Description
"r" or "rb"	Open for reading
"w" or "wb"	Truncate to zero length or create for writing
"a" or "ab"	Append; open for writing at end of file (EOF), or create for writing
"r+", "r+b", or "rb+"	Open for update (reading and writing)
"w+", "w+b", or "wb+"	Truncate or create for update
"a+", "a+b", or "ab+"	Append; open or create for update at EOF

The multichannel descriptor `mcd` is a 32-bit **reg** in which a single bit is set indicating which file is opened. The least significant bit (bit `0`) of an `mcd` always refers to the standard output. Output is directed to two or more files opened with multichannel descriptors by bitwise OR-ing together their `mcd`s and writing to the resultant value.

The most significant bit (bit 31) of a multichannel descriptor is reserved and shall always be cleared, limiting an implementation to at most 31 files opened for output via multichannel descriptors.

The file descriptor `fd` is a 32-bit value. The most significant bit (bit 31) of a `fd` is reserved and shall always be set; this allows implementations of the file input and output functions to determine how the file was opened. The remaining bits hold a small number indicating what file is opened. Three file descriptors are pre-opened; they are **STDIN**, **STDOUT**, and **STDERR**, which have the values `32'h8000_0000`,

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

`32'h8000_0001`, and `32'h8000_0002`, respectively. **STDIN** is pre-opened for reading, and **STDOUT** and **STDERR** are pre-opened for append.

Unlike multichannel descriptors, file descriptors cannot be combined via bitwise OR in order to direct output to multiple files. Instead, files are opened via file descriptor for input, output, and both input and output, as well as for append operations, based on the value of **type**, according to Table 17-7.

If a file cannot be opened (either the file does not exist and the **type** specified is "r", "rb", "r+", "r+b", or "rb+", or the permissions do not allow the file to be opened at that path), a zero is returned for the `mcd` or `fd`. Applications can call **$ferror** to determine the cause of the most recent error (see 17.2.7).

The "b" in the above types exists to distinguish binary files from text files. Many systems (such as Unix) make no distinction between binary and text files, and on these systems the "b" is ignored. However, some systems (such as machines running Windows NT) perform data mappings on certain binary values written to and read from files that are opened for text access.

The **$fclose** system task closes the file specified by `fd` or closes the file(s) specified by the multichannel descriptor `mcd`. No further output to or input from any file descriptor(s) closed by **$fclose** is allowed. Active **$fmonitor** and/or **$fstrobe** operations on a file descriptor or multichannel descriptor are implicitly cancelled by an **$fclose** operation. The **$fopen** function shall reuse channels that have been closed.

NOTE—The number of simultaneous input and output channels that can be open at any one time is dependent on the operating system. Some operating systems do not support opening files for update.

17.2.2 File output system tasks

The syntax for **$display**, **$write**, **$monitor**, and **$strob** system tasks is shown in Syntax 17-5.

```
file_output_tasks ::=
        file_output_task_name ( multi_channel_descriptor [ , list_of_arguments ] ) ;
      | file_output_task_name ( fd [ , list_of_arguments ] ) ;
file_output_task_name ::=
        $fdisplay   | $fdisplayb   | $fdisplayh   | $fdisplayo
      | $fwrite     | $fwriteb     | $fwriteh     | $fwriteo
      | $fstrobe    | $fstrobeb    | $fstrobeh    | $fstrobeo
      | $fmonitor   | $fmonitorb   | $fmonitorh   | $fmonitoro
```

Syntax 17-5—Syntax for file output system tasks

Each of the four formatted display tasks—**$display**, **$write**, **$monitor**, and **$strobe**—has a counterpart that writes to specific files as opposed to the standard output. These counterpart tasks—**$fdisplay**, **$fwrite,** **$fmonitor**, and **$fstrobe**—accept the same type of arguments as the tasks upon which they are based, with one exception: The first argument shall be either a multichannel descriptor or a file descriptor, which indicates where to direct the file output. Multichannel descriptors are described in detail in 17.2.1. A multichannel descriptor is either a variable or the result of an expression that takes the form of a 32-bit unsigned integer value.

The **$fstrobe** and **$fmonitor** system tasks work just like their counterparts, **$strobe** and **$monitor**, except that they write to files using the multichannel descriptor for control. Unlike **$monitor**, any number of **$fmonitor** tasks can be set up to be simultaneously active. However, there is no counterpart to **$monitoron** and **$monitoroff** tasks. The task **$fclose** is used to cancel an active **$fstrobe** or **$fmonitor** task.

For example:

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The following example shows how to set up multichannel descriptors. In this example, three different channels are opened using the **$fopen** function. The three multichannel descriptors that are returned by the function are then combined in a bitwise OR operation and assigned to the integer variable `messages`. The `messages` variable can then be used as the first argument in a file output task to direct output to all three channels at once. To create a descriptor that directs output to the standard output as well, the `messages` variable is a bitwise OR with the constant `1`, which effectively enables channel `0`.

```
integer
    messages,        broadcast,
    cpu_chann,       alu_chann, mem_chann;
initial begin
    cpu_chann = $fopen("cpu.dat");
    if (cpu_chann == 0) $finish;
    alu_chann = $fopen("alu.dat");
    if (alu_chann == 0) $finish;
    mem_chann = $fopen("mem.dat");
    if (mem_chann == 0) $finish;
    messages = cpu_chann | alu_chann | mem_chann;
    // broadcast includes standard output
    broadcast = 1 | messages;
end
endmodule
```

The following file output tasks show how the channels opened in the preceding example might be used:

```
$fdisplay( broadcast, "system reset at time %d", $time );

$fdisplay( messages, "Error occurred on address bus",
              " at time %d, address = %h", $time, address );

forever @(posedge clock)
    $fdisplay( alu_chann, "acc= %h f=%h a=%h b=%h", acc, f, a, b );
```

17.2.3 Formatting data to a string

The syntax for the **$swrite** family of tasks and for **$sformat** system task is shown in Syntax 17-6.

```
string_output_tasks ::=
        string_output_task_name ( output_reg , list_of_arguments ) ;
string_output_task_name ::=
        $swrite | $swriteb | $swriteh | $swriteo
variable_format_string_output_task ::=
        $sformat ( output_reg , format_string , list_of_arguments ) ;
```

Syntax 17-6—Syntax for formatting data tasks

The **$swrite** family of tasks is based on the **$fwrite** family of tasks and accepts the same type of arguments as the tasks upon which it is based, with one exception: The first argument to **$swrite** shall be a **reg** variable to which the resulting string shall be written, instead of a variable specifying the file to which to write the resulting string.

The system task **$sformat** is similar to the system task **$swrite**, with one major difference.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Unlike the display and write family of output system tasks, **$sformat** always interprets its second argument, and only its second argument, as a format string. This format argument can be a static string, such as "data is %d" or can be a **reg** variable whose content is interpreted as the format string. No other arguments are interpreted as format strings. **$sformat** supports all the format specifiers supported by **$display**, as documented in 17.1.1.2.

The remaining arguments to **$sformat** are processed using any format specifiers in the *format_string*, until all such format specifiers are used up. If not enough arguments are supplied for the format specifiers or too many are supplied, then the application shall issue a warning and continue execution. The application, if possible, can statically determine a mismatch in format specifiers and number of arguments and issue a compile time error message.

NOTE—If the *format_string* is a **reg**, it might not be possible to determine its value at compile time.

The variable *output_reg* is assigned using the string assignment to variable rules, as specified in 5.2.3.

17.2.4 Reading data from a file

Files opened using file descriptors can be read from only if they were opened with either the `r` or `r+` **type** values. See 17.2.1 for more information about opening files.

17.2.4.1 Reading a character at a time

For example:

Example 1

```
c = $fgetc ( fd );
```

reads a byte from the file specified by `fd`. If an error occurs reading from the file, then *c* is set to EOF (`-1`). The code defines the width of the **reg** to be wider than 8 bits so that a return value from **$fgetc** of EOF (`-1`) can be differentiated from the character code `0xFF`. Applications can call **$ferror** to determine the cause of the most recent error (see 17.2.7).

Example 2

```
code = $ungetc ( c, fd );
```

inserts the character specified by `c` into the buffer specified by file descriptor `fd`. The character *c* shall be returned by the next **$fgetc** call on that file descriptor. The file itself is unchanged. If an error occurs pushing a character onto a file descriptor, then `code` is set to EOF. Otherwise, `code` is set to zero. Applications can call **$ferror** to determine the cause of the most recent error (see 17.2.7).

NOTE—The features of the underlying implementation of file I/O on the host system limit the number of characters that can be pushed back onto a stream. Operations like **$fseek** might erase any pushed back characters.

17.2.4.2 Reading a line at a time

For example:

```
integer code ;
code = $fgets ( str, fd );
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

reads characters from the file specified by `fd` into the **reg** `str` until `str` is filled, or a newline character is read and transferred to `str`, or an EOF condition is encountered. If `str` is not an integral number of bytes in length, the most significant partial byte is not used in order to determine the size.

If an error occurs reading from the file, then `code` is set to zero. Otherwise, the number of characters read is returned in `code`. Applications can call **$ferror** to determine the cause of the most recent error (see 17.2.7).

17.2.4.3 Reading formatted data

For example:

```
integer code ;
code = $fscanf ( fd, format, args );
code = $sscanf ( str, format, args );
```

$fscanf reads from the files specified by the file descriptor `fd`.

$sscanf reads from the **reg** `str`.

Both functions read characters, interpret them according to a format, and store the results. Both expect as arguments a control string, `format`, and a set of arguments specifying where to place the results. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are ignored.

If an argument is too small to hold the converted input, then, in general, the least significant bits are transferred. Arguments of any length that is supported by Verilog can be used. However, if the destination is a **real** or **realtime**, then the value +Inf (or -Inf) is transferred. The format can be a string constant or a **reg** containing a string constant. The string contains conversion specifications, which direct the conversion of input into the arguments. The control string can contain the following:

a) White space characters (blanks, tabs, newlines, or formfeeds) that, except in one case described below, cause input to be read up to the next nonwhite space character. For **$sscanf**, null characters shall also be considered white space.
b) An ordinary character (not %) that must match the next character of the input stream.
c) Conversion specifications consisting of the character `%`, an optional assignment suppression character `*`, a decimal digit string that specifies an optional numerical maximum field width, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable specified in the corresponding argument unless assignment suppression was indicated by the character `*`. In this case, no argument shall be supplied.

The suppression of assignment provides a way of describing an input field that is to be skipped. An *input field* is defined as a string of nonspace characters; it extends to the next inappropriate character or until the maximum field width, if one is specified, is exhausted. For all descriptors except the character c, white space leading an input field is ignored.

% A single % is expected in the input at this point; no assignment is done.

b Matches a binary number, consisting of a sequence from the set 0,1,X,x,Z,z,?, and _.

o Matches a octal number, consisting of a sequence of characters from the set 0,1,2,3,4,5,6,7,X,x,Z,z,?, and _.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

d Matches an optionally signed decimal number, consisting of the optional sign from the set + or -, followed by a sequence of characters from the set 0,1,2,3,4,5,6,7,8,9, and _, or a single value from the set x,X,z,Z,?.

h or x Matches a hexadecimal number, consisting of a sequence of characters from the set 0,1,2,3,4,5,6,7,8,9,a,A,b,B,c,C,d,D,e,E,f,F,x,X,z,Z,?, and _.

f, e, or g Matches a floating point number. The format of a floating point number is an optional sign (either + or -), followed by a string of digits from the set 0,1,2,3,4,5,6,7,8,9 optionally containing a decimal point character (.), followed by an optional exponent part including e or E, followed by an optional sign, followed by a string of digits from the set 0,1,2,3,4,5,6,7,8,9.

v Matches a net signal strength, consisting of a three-character sequence as specified in 17.1.1.5. This conversion is not extremely useful, as strength values are really only usefully assigned to nets and **$fscanf** can only assign values to regs (if assigned to regs, the values are converted to the 4 value equivalent).

t Matches a floating point number. The format of a floating point number is an optional sign (either + or -), followed by a string of digits from the set 0,1,2,3,4,5,6,7,8,9 optionally containing a decimal point character (.), followed by an optional exponent part including e or E, followed by an optional sign, followed by a string of digits from the set 0,1,2,3,4,5,6,7,8,9. The value matched is then scaled and rounded according to the current time scale as set by **$timeformat**. For example, if the timescale is `timescale 1ns/100ps and the time format is **$timeformat**(–3,2," ms",10);, then a value read with **$sscanf**("10.345", "%t", t) would return 10350000.0.

c Matches a single character, whose 8-bit ASCII value is returned.

s Matches a string, which is a sequence of nonwhite space characters.

u Matches unformatted (binary) data. The application shall transfer sufficient data from the input to fill the target **reg**. Typically, the data are obtained from a matching `$fwrite("%u",data)` or from an external application written in another programming language such as C, Perl, or FORTRAN.

The application shall transfer the 2 value binary data from the input stream to the destination **reg**, expanding the data to the 4 value format. This escape sequence can be used with any of the existing input system tasks, although `$fscanf` should be the preferred one to use. As the input data cannot represent `x` or `z`, it is not possible to obtain an `x` or `z` in the result **reg**. This formatting specifier is intended to be used to support transferring data to and from external programs that have no concept of `x` and `z`.

Applications that require preservation of `x` and `z` are encouraged to use the `%z` I/O format specification.

The data shall be read from the file in the native endian format of the underlying system (i.e., in the same endian order as if the PLI was used and the C language `read(2)` system call was used).

For POSIX applications, it might be necessary to open files for unformatted I/O with the "rb", "rb+", or "r+b" specifiers to avoid the systems implementation of I/O altering patterns in the unformatted stream that match special characters.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

z The formatting specification `%z` (or `%Z`) is defined for reading data without formatting (binary values). The application shall transfer the 4 value binary representation of the specified data from the input stream to the destination **reg**. This escape sequence can be used with any of the existing input system tasks, although `$fscanf` should be the preferred one to use.

This formatting specifier is intended to be used to support transferring data to and from external programs that recognize and support the concept of `x` and `z`. Applications that do not require the preservation of `x` and `z` are encouraged to use the `%u` I/O format specification.

The data shall be read from the file in the native endian format of the underlying system (i.e., in the same endian order as if the PLI was used, the data were in a `s_vpi_vecval` structure (see Figure 27-8 in 27.14), and the C language read(2) system call was used to read the data from disk).

For POSIX applications, it might be necessary to open files for unformatted I/O with the "rb", "rb+", or "r+b" specifiers to avoid the systems implementation of I/O altering patterns in the unformatted stream that match special characters.

m Returns the current hierarchical path as a string. Does not read data from the input file or str argument.

If an invalid conversion character follows the `%`, the results of the operation are implementation dependent.

If the format string or the `str` argument to **$sscanf** contains unknown bits (`x` or `z`), then the system task shall return EOF.

If EOF is encountered during input, conversion is terminated. If EOF occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure. Otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including newline characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable.

The number of successfully matched and assigned input items is returned in `code`; this number can be 0 in the event of an early matching failure between an input character and the control string. If the input ends before the first matching failure or conversion, EOF is returned. Applications can call **$ferror** to determine the cause of the most recent error (see 17.2.7).

17.2.4.4 Reading binary data

For example:

```
integer code ;
code = $fread( myreg, fd);
code = $fread( mem, fd);
code = $fread( mem, fd, start);
code = $fread( mem, fd, start, count);
code = $fread( mem, fd, , count);
```

reads a binary data from the file specified by `fd` into the **reg** `myreg` or the memory `mem`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

`start` is an optional argument. If present, `start` shall be used as the address of the first element in the memory to be loaded. If not present, the lowest numbered location in the memory shall be used.

`count` is an optional argument. If present, `count` shall be the maximum number of locations in `mem` that shall be loaded. If not supplied, the memory shall be filled with what data are available.

`start` and `count` are ignored if **$fread** is loading a **reg**.

If no addressing information is specified within the system task and no address specifications appear within the data file, then the default start address is the lowest address given in the declaration of the memory. Consecutive words are loaded toward the highest address until either the memory is full or the data file is completely read. If the start address is specified in the task without the finish address, then loading starts at the specified start address and continues toward the highest address given in the declaration of the memory.

`start` is the address in the memory. For `start` = 12 and the memory `up[10:20]`, the first data would be loaded at `up[12]`. For the memory `down[20:10]`, the first location loaded would be `down[12]`, then `down[13]`.

The data in the file shall be read byte by byte to fulfill the request. An 8-bit wide memory is loaded using 1 byte per memory word, while a 9-bit wide memory is loaded using 2 bytes per memory word. The data are read from the file in a big endian manner; the first byte read is used to fill the most significant location in the memory element. If the memory width is not evenly divisible by 8 (8, 16, 24, 32), not all data in the file are loaded into memory because of truncation.

The data loaded from the file are taken as "2 value" data. A bit set in the data is interpreted as a `1`, and bit not set is interpreted as a `0`. It is not possible to read a value of `x` or `z` using **$fread**.

If an error occurs reading from the file, then `code` is set to zero. Otherwise, the number of characters read is returned in code. Applications can call **$ferror** to determine the cause of the most recent error (see 17.2.7).

NOTE—There is not a "binary" mode and an "ASCII" mode; one can freely intermingle binary and formatted read commands from the same file.

17.2.5 File positioning

For example:

Example 1

```
integer pos ;
pos = $ftell ( fd );
```

returns in `pos` the offset from the beginning of the file of the current byte of the file `fd`, which shall be read or written by a subsequent operation on that file descriptor.

This value can be used by subsequent **$fseek** calls to reposition the file to this point. Any repositioning shall cancel any **$ungetc** operations. If an error occurs, EOF is returned. Applications can call **$ferror** to determine the cause of the most recent error (see 17.2.7).

Example 2

```
code = $fseek ( fd, offset, operation );
code = $rewind ( fd );
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

sets the position of the next input or output operation on the file specified by `fd`. The new position is at the signed distance offset bytes from the beginning, from the current position, or from the end of the file, according to an operation value of 0, 1, and 2 as follows:

— 0 sets position equal to offset bytes
— 1 sets position to current location plus offset
— 2 sets position to EOF plus offset

$rewind is equivalent to **$fseek** `(fd,0,0);`

Repositioning the current file position with **$fseek** or **$rewind** shall cancel any **$ungetc** operations.

$fseek() allows the file position indicator to be set beyond the end of the existing data in the file. If data are later written at this point, subsequent reads of data in the gap shall return zero until data are actually written into the gap. **$fseek**, by itself, does not extend the size of the file.

When a file is opened for append (that is, when `type` is "a" or "a+"), it is impossible to overwrite information already in the file. **$fseek** can be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output.

If an error occurs repositioning the file, then `code` is set to –1. Otherwise, `code` is set to 0. Applications can call **$ferror** to determine the cause of the most recent error (see 17.2.7).

17.2.6 Flushing output

For example:

```
$fflush ( mcd );
$fflush ( fd );
$fflush ( );
```

writes any buffered output to the file(s) specified by `mcd`, to the file specified by `fd`, or if **$fflush** is invoked with no arguments, to all open files.

17.2.7 I/O error status

Should any error be detected by one of the file I/O routines, an error code is returned. Often this is sufficient for normal operation (i.e., if the opening of an optional configuration file fails, the application typically would simply continue using default values). However, sometimes it is useful to obtain more information about the error for correct application operation. In this case, the **$ferror** function can be used:

```
integer errno ;
errno = $ferror ( fd, str );
```

A string description of type of error encountered by the most recent file I/O operation is written into `str`, which should be at least 640 bits wide. The integral value of the error code is returned in `errno`. If the most recent operation did not result in an error, then the value returned shall be zero, and the **reg** `str` shall be cleared.

17.2.8 Detecting EOF

For example:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
integer code;
code = $feof ( fd );
```

returns a nonzero value when EOF has previously been detected reading the input file `fd`. It returns zero otherwise.

17.2.9 Loading memory data from a file

The syntax for **$readmemb** and **$readmemh** system tasks is shown in Syntax 17-7.

```
load_memory_tasks ::=
        $readmemb ( " file_name " , memory_name [ , start_addr [ , finish_addr ] ] ) ;
      | $readmemh ( " file_name " , memory_name [ , start_addr [ , finish_addr ] ] ) ;
```

Syntax 17-7—Syntax for memory load system tasks

Two system tasks—**$readmemb** and **$readmemh**—read and load data from a specified text file into a specified memory. Either task can be executed at any time during simulation. The text file to be read shall contain only the following:

— White space (spaces, newlines, tabs, and formfeeds)
— Comments (both types of comment are allowed)
— Binary or hexadecimal numbers

The numbers shall have neither the length nor the base format specified. For **$readmemb**, each number shall be binary. For **$readmemh**, the numbers shall be hexadecimal. The unknown value (`x` or `X`), the high-impedance value (`z` or `Z`), and the underscore (_) can be used in specifying a number as in a Verilog HDL source description. White space and/or comments shall be used to separate the numbers.

In the following discussion, the term *address* refers to an index into the array that models the memory.

As the file is read, each number encountered is assigned to a successive word element of the memory. Addressing is controlled both by specifying start and/or finish addresses in the system task invocation and by specifying addresses in the data file.

When addresses appear in the data file, the format is an at character (`@`) followed by a hexadecimal number as follows:

```
@hh...h
```

Both uppercase and lowercase digits are allowed in the number. No white space is allowed between the `@` and the number. As many address specifications as needed within the data file can be used. When the system task encounters an address specification, it loads subsequent data starting at that memory address.

If no addressing information is specified within the system task and no address specifications appear within the data file, then the default start address shall be the lowest address in the memory. Consecutive words shall be loaded until either the highest address in the memory is reached or the data file is completely read. If the start address is specified in the task without the finish address, then loading shall start at the specified start address and shall continue upward toward the highest address in the memory. In both cases, loading shall continue upward even after an address specification in the data file.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

If both start and finish addresses are specified as arguments to the task, then loading shall begin at the start address and shall continue toward the finish address. If the start address is greater than the finish address, then the address will be decremented between consecutive loads rather than being incremented. Loading shall continue to follow this direction even after an address specification in the data file.

When addressing information is specified both in the system task and in the data file, the addresses in the data file shall be within the address range specified by the system task arguments; otherwise, an error message is issued, and the load operation is terminated.

A warning message shall be issued if the number of data words in the file differs from the number of words in the range implied by the start through finish addresses and no address specifications appear within the data file.

For example:

```
reg [7:0] mem[1:256];
```

Given this declaration, each of the following statements load data into `mem` in a different manner:

```
initial $readmemh("mem.data", mem);
initial $readmemh("mem.data", mem, 16);
initial $readmemh("mem.data", mem, 128, 1);
```

The first statement loads up the memory at simulation time `0` starting at the memory address `1`. The second statement begins loading at address 16 and continue on toward address 256. For the third and final statement, loading begins at address 128 and continue down toward address `1`.

In the third case, when loading is complete, a final check is performed to ensure that exactly 128 numbers are contained in the file. If the check fails, a warning message is issued.

17.2.10 Loading timing data from an SDF file

The syntax for the **$sdf_annotate** system task is shown in Syntax 17-8.

```
sdf_annotate_task ::=
    $sdf_annotate ("sdf_file" [ , [ module_instance ] [ , [ "config_file" ]
      [ , [ "log_file" ] [ , [ "mtm_spec" ]
      [ , [ "scale_factors" ] [ , [ "scale_type" ] ] ] ] ] ] );
```

Syntax 17-8—Syntax for $sdf_annotate system task

The **$sdf_annotate** system task reads timing data from an SDF file into a specified region of the design.

sdf_file Is a character string or is a **reg** containing a character string naming the file to be opened.

module_instance Is an optional argument specifying the scope to which to annotate the information in the SDF file. The SDF annotator uses the hierarchy level of the specified instance for running the annotation. Array indices are permitted. If the *module_instance* is not specified, the SDF annotator uses the module containing the call to the **$sdf_annotate** system task as the *module_instance* for annotation.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

config_file Is an optional character string argument providing the name of a configuration file. Information in this file can be used to provide detailed control over many aspects of annotation.

log_file Is an optional character string argument providing the name of the log file used during SDF annotation. Each individual annotation of timing data from the SDF file results in an entry in the log file.

mtm_spec Is an optional character string argument specifying which member of the `min/typ/max` triples shall be annotated. The legal values for this string are described in Table 17-8. This overrides any `MTM_SPEC` keywords in the configuration file.

Table 17-8—mtm spec argument

Keyword	Description
`MAXIMUM`	Annotates the maximum value
`MINIMUM`	Annotates the minimum value
`TOOL_CONTROL` (default)	Annotates the value as selected by the simulator
`TYPICAL`	Annotates the typical value

scale_factors Is an optional character string argument specifying the scale factors to be used while annotating timing values. For example, `"1.6:1.4:1.2"` causes minimum values to be multiplied by `1.6`, typical values by `1.4`, and maximum values by `1.2`. The default values are `1.0:1.0:1.0`. The *scale_factors* argument overrides any `SCALE_FACTORS` keywords in the configuration file.

scale_type Is an optional character string argument specifying how the scale factors should be applied to the `min/typ/max` triples. The legal values for this string are shown in Table 17-9. This overrides any `SCALE_TYPE` keywords in the configuration file.

Table 17-9—scale type argument

Keyword	Description
`FROM_MAXIMUM`	Applies scale factors to maximum value
`FROM_MINIMUM`	Applies scale factors to minimum value
`FROM_MTM` (default)	Applies scale factors to min/typ/max values
`FROM_TYPICAL`	Applies scale factors to typical value

17.3 Timescale system tasks

The following system tasks display and set timescale information:

a) **$printtimescale**
b) **$timeformat**

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

See 19.8 for a discussion of Verilog time scales and time units.

17.3.1 $printtimescale

The **$printtimescale** system task displays the time unit and precision for a particular module. The syntax for the system task is shown in Syntax 17-9.

```
printtimescale_task ::=
    $printtimescale [ ( hierarchical_identifier ) ] ;
```

Syntax 17-9—Syntax for $printtimescale

This system task can be specified with or without an argument.

— When no argument is specified, **$printtimescale** displays the time unit and precision of the module that is the current scope.
— When an argument is specified, **$printtimescale** displays the time unit and precision of the module passed to it.

The timescale information shall appear in the following format:

```
Time scale of (module_name) is unit / precision
```

For example:

```
`timescale 1 ms / 1 us
module a_dat;
initial
    $printtimescale(b_dat.c1);
endmodule

`timescale 10 fs / 1 fs
module b_dat;
   c_dat c1 ();
endmodule

`timescale 1 ns / 1 ns
module c_dat;
        .
        .
        .
endmodule
```

In this example, module `a_dat` invokes the **$printtimescale** system task to display timescale information about another module `c_dat`, which is instantiated in module `b_dat`.

The information about `c_dat` shall be displayed in the following format:

```
Time scale of (b_dat.c1) is 1ns / 1ns
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

17.3.2 $timeformat

The syntax for **$timeformat** system task is shown in Syntax 17-10.

```
timeformat_task ::=
    $timeformat [ ( units_number , precision_number , suffix_string , minimum_field_width ) ] ;
```

Syntax 17-10—Syntax for $timeformat

The **$timeformat** system task performs the following two functions:

— It specifies how the `%t` format specification reports time information for the **$write**, **$display**, **$strobe**, **$monitor**, **$fwrite**, **$fdisplay**, **$fstrobe**, and **$fmonitor** group of system tasks.

— It specifies the time unit for delays entered interactively.

The units number argument shall be an integer in the range from `0` to `-15`. This argument represents the time unit as shown in Table 17-10.

Table 17-10—$timeformat units_number arguments

Unit number	Time unit	Unit number	Time unit
0	1 s	–8	10 ns
–1	100 ms	–9	1 ns
–2	10 ms	–10	100 ps
–3	1 ms	–11	10 ps
–4	100 us	–12	1 ps
–5	10 us	–13	100 fs
–6	1 us	–14	10 fs
–7	100 ns	–15	1 fs

NOTE—While s, ms, ns, ps, and fs are the usual SI unit symbols for second, millisecond, nanosecond, picosecond, and femtosecond, due to lack of the Greek letter m (mu) in coding character sets, ‘us’ represents the SI unit symbol for microsecond, properly ms.

The **$timeformat** system task performs the following two operations:

— It sets the time unit for all later-entered delays entered interactively.

— It sets the time unit, precision number, suffix string, and minimum field width for all `%t` formats specified in all modules that follow in the source description until another **$timeformat** system task is invoked.

The default **$timeformat** system task arguments are given in Table 17-11.

For example:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 17-11—$timeformat default value for arguments

Argument	Default
units_number	The smallest time precision argument of all the **`timescale** compiler directives in the source description
precision_number	0
suffix_string	A null character string
minimum_field_width	20

The following example shows the use of %t with the **$timeformat** system task to specify a uniform time unit, time precision, and format for timing information.

```
`timescale 1 ms / 1 ns
module cntrl;
initial
    $timeformat(-9, 5, " ns", 10);
endmodule

`timescale 1 fs / 1 fs
module a1_dat;
reg in1;
integer file;
buf #10000000 (o1,in1);
initial begin
    file = $fopen("a1.dat");
    #00000000 $fmonitor(file,"%m: %t in1=%d o1=%h", $realtime,in1,o1);
    #10000000 in1 = 0;
    #10000000 in1 = 1;
end
endmodule

`timescale 1 ps / 1 ps
module a2_dat;
reg in2;
integer file2;
buf #10000 (o2,in2);
initial  begin
    file2=$fopen("a2.dat");
    #00000 $fmonitor(file2,"%m: %t in2=%d o2=%h",$realtime,in2,o2);
    #10000 in2 = 0;
    #10000 in2 = 1;
end
endmodule
```

The contents of file a1.dat are as follows:

```
a1_dat: 0.00000 ns in1= x o1=x
a1_dat: 10.00000 ns in1= 0 o1=x
a1_dat: 20.00000 ns in1= 1 o1=0
a1_dat: 30.00000 ns in1= 1 o1=1
```

The contents of file a2.dat are as follows:

Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
a2_dat: 0.00000 ns in2=x o2=x
a2_dat: 10.00000 ns in2=0 o2=x
a2_dat: 20.00000 ns in2=1 o2=0
a2_dat: 30.00000 ns in2=1 o2=1
```

In this example, the times of events written to the files by the **$fmonitor** system task in modules `a1_dat` and `a2_dat` are reported as multiples of `1 ns`—even though the time units for these modules are `1 fs` and `1 ps`, respectively—because the first argument of the **$timeformat** system task is `-9` and the `%t` format specification is included in the arguments to **$fmonitor**. This time information is reported after the module names with five fractional digits, followed by an `ns` character string in a space wide enough for 10 ASCII characters.

17.4 Simulation control system tasks

There are two simulation control system tasks:

a) **$finish**
b) **$stop**

17.4.1 $finish

Syntax 17-11 shows the syntax for **$finish** system task.

```
finish_task ::=
    $finish [ ( n ) ] ;
```

Syntax 17-11—Syntax for $finish

The **$finish** system task simply makes the simulator exit and pass control back to the host operating system. If an expression is supplied to this task, then its value (`0`, `1`, or `2`) determines the diagnostic messages that are printed before the prompt is issued (see Table 17-12). If no argument is supplied, then a value of 1 is taken as the default.

Table 17-12—Diagnostics for $finish

Argument value	Diagnostic message
0	Prints nothing
1	Prints simulation time and location
2	Prints simulation time, location, and statistics about the memory and central processing unit (CPU) time used in simulation

17.4.2 $stop

The syntax for the **$stop** system task is shown in Syntax 17-12.

```
stop_task ::=
    $stop [ ( n ) ] ;
```

Syntax 17-12—Syntax for $stop

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

The **$stop** system task causes simulation to be suspended. This task takes an optional expression argument (`0`, `1`, or `2`) that determines what type of diagnostic message is printed. The amount of diagnostic messages output increases with the value of the optional argument passed to **$stop**.

17.5 Programmable logic array (PLA) modeling system tasks

The modeling of PLA devices is provided in the Verilog HDL by a group of system tasks. This subclause describes the syntax and use of these system tasks and the formats of the logic array personality file. The syntax for PLA modeling system task is shown in Syntax 17-13.

```
pla_system_task ::=
    $array_type$logic$format ( memory_identifier , input_terms , output_terms ) ;
array_type ::=
    sync | async
logic ::=
    and | or | nand | nor
format ::=
    array | plane
memory_identifier ::=
    identifier
input_terms ::=
    expression
output_terms ::=
    variable_lvalue
```

Syntax 17-13 —Syntax for PLA modeling system task

The input terms can be nets or variables whereas the output terms shall only be variables.

The PLA syntax allows for the system tasks as shown in Table 17-13.

Table 17-13—PLA modeling system tasks

$async$and$array	**$sync$and$array**	**$async$and$plane**	**$sync$and$plane**
$async$nand$array	**$sync$nand$array**	**$async$nand$plane**	**$sync$nand$plane**
$async$or$array	**$sync$or$array**	**$async$or$plane**	**$sync$or$plane**
$async$nor$array	**$sync$nor$array**	**$async$nor$plane**	**$sync$nor$plane**

17.5.1 Array types

The modeling of both synchronous and asynchronous arrays is provided by the PLA system tasks. The synchronous forms control the time at which the logic array shall be evaluated and the outputs shall be updated. For the asynchronous forms, the evaluations are automatically performed whenever an input term changes value or any word in the personality memory is changed.

For both the synchronous and asynchronous forms, the output terms are updated without any delay.

For example:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

An example of an asynchronous system call is as follows:

```
wire        a1, a2, a3, a4, a5, a6, a7;
reg         b1, b2, b3;
wire [1:7] awire;
reg  [1:3] breg;

$async$and$array(mem,{a1,a2,a3,a4,a5,a6,a7},{b1,b2,b3});
or
$async$and$array(mem,awire, breg);
```

An example of a synchronous system call is as follows:

```
$sync$or$plane(mem,{a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
```

17.5.2 Array logic types

The logic arrays are modeled with and, or, nand, and nor logic planes. This applies to all array types and formats.

For example:

An example of a nor plane system call is as follows:

```
$async$nor$plane(mem,{a1,a2,a3,a4,a5,a6,a7},{b1,b2,b3});
```

An example of a nand plane system call is as follows:

```
$sync$nand$plane(mem,{a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
```

17.5.3 Logic array personality declaration and loading

The logic array personality is declared as an array of regs that is as wide as the number of input terms and as deep as the number of output terms.

The personality of the logic array is normally loaded into the memory from a text data file using the system tasks **$readmemb** or **$readmemh**. Alternatively, the personality data can be written directly into the memory using the procedural assignment statements. PLA personalities can be changed dynamically at any time during simulation simply by changing the contents of the memory. The new personality shall be reflected on the outputs of the logic array at the next evaluation.

For example:

The following example shows a logic array with `n` input terms and `m` output terms:

```
reg [1:n] mem[1:m];
```

As shown in the examples in 17.5, PLA input terms, output terms, and memory shall be specified in ascending order.

17.5.4 Logic array personality formats

Two separate personality formats are supported by the Verilog HDL and are differentiated by using either an array system call or a plane system call. The array system call allows for a `1` or `0` in the memory that has been declared. A `1` means take the input value, and a `0` means do not take the input value.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The plane system call complies with the University of California at Berkeley format for Espresso[10]. Each bit of the data stored in the array has the following meaning:

0 Take the complemented input value.
1 Take the true input value.
x Take the "worst case" of the input value.
z Do-not-care; the input value is of no significance.
? Same as *z*.

For example:

Example 1—The following example illustrates an array with logic equations:

```
b1 = a1 & a2
b2 = a3 & a4 & a5
b3 = a5 & a6 & a7
```

The PLA personality is as follows:

```
1100000 in mem[1]
0011100 in mem[2]
0000111 in mem[3]
```

The module for the PLA is as follows:

```
module async_array(a1,a2,a3,a4,a5,a6,a7,b1,b2,b3);
input a1, a2, a3, a4, a5, a6, a7 ;
output b1, b2, b3;
reg [1:7] mem[1:3]; // memory declaration for array personality
reg b1, b2, b3;
initial begin
    // set up the personality from the file array.dat
    $readmemb("array.dat", mem);
    // set up an asynchronous logic array with the input
    // and output terms expressed as concatenations
    $async$and$array(mem,{a1,a2,a3,a4,a5,a6,a7},{b1,b2,b3});
end
endmodule
```

Where the file `array.dat` contains the binary data for the PLA personality:

```
1100000
0011100
0000111
```

A synchronous version of this example has the following description:

```
module sync_array(a1,a2,a3,a4,a5,a6,a7,b1,b2,b3,clk);
input a1, a2, a3, a4, a5, a6, a7, clk;
output b1, b2, b3;
reg [1:7] mem[1:3]; // memory declaration
reg b1, b2, b3;
initial begin
```

[10]Information on Espresso can be found at http://embedded.eecs.berkeley.edu/pubs/downloads/espresso/index.htm.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
    // set up the personality
    $readmemb("array.dat", mem);
    // set up a synchronous logic array to be evaluated
    // when a positive edge on the clock occurs
    forever @(posedge clk)
        $async$and$array(mem,{a1,a2,a3,a4,a5,a6,a7},{b1,b2,b3});
end
endmodule
```

Example 2—An example of the usage of the plane format tasks follows. The logical function of this PLA is shown first, followed by the PLA personality in the new format, the Verilog HDL description using the **$async$and$plane** system task, and finally the result of running the simulation.

The logical function of the PLA is as follows:

```
b[1] = a[1] & ~a[2];
b[2] = a[3];
b[3] = ~a[1] & ~a[3];
b[4] = 1;
```

The PLA personality is as follows:

```
3'b10?
3'b??1
3'b0?0
3'b???
```

The Verilog HDL description using the **$async$and$plane** system task is as follows:

```
module pla;
`define rows 4
`define cols 3
reg [1:`cols] a, mem[1:`rows];
reg [1:`rows] b;
initial begin
    // PLA system call
    $async$and$plane(mem,a[1:3],b[1:4]);
    mem[1] = 3'b10?;
    mem[2] = 3'b??1;
    mem[3] = 3'b0?0;
    mem[4] = 3'b???;
    // stimulus and display
    #10 a = 3'b111;
    #10 $displayb(a, " -> ", b);
    #10 a = 3'b000;
    #10 $displayb(a, " -> ", b);
    #10 a = 3'bxxx;
    #10 $displayb(a, " -> ", b);
    #10 a = 3'b101;
    #10 $displayb(a, " -> ", b);
end
endmodule
```

The output is as follows:

```
111 -> 0101
000 -> 0011
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
xxx -> xxx1
101 -> 1101
```

17.6 Stochastic analysis tasks

This subclause describes a set of system tasks and functions that manage queues. These tasks facilitate implementation of stochastic queueing models.

The set of tasks and functions that create and manage queues follows:

$q_initialize (q_id**,** q_type**,** max_length**,** status**) ;**
$q_add (q_id**,** job_id**,** inform_id**,** status**) ;**
$q_remove (q_id**,** job_id**,** inform_id**,** status**) ;**
$q_full (q_id**,** status**)**
$q_exam (q_id**,** q_stat_code**,** q_stat_value**,** status**) ;**

17.6.1 $q_initialize

The **$q_initialize** system task creates new queues. The `q_id` argument is an integer input that shall uniquely identify the new queue. The `q_type` argument is an integer input. The value of the `q_type` argument specifies the type of the queue as shown in Table 17-14.

Table 17-14—Types of queues of $q_type values

q_type value	Type of queue
1	First-in, first-out
2	Last-in, first-out

The `max_length` argument is an integer input that specifies the maximum number of entries allowed on the queue. The success or failure of the creation of the queue is returned as an integer value in status. The error conditions and corresponding values of status are described in Table 17-16 (in 17.6.6).

17.6.2 $q_add

The **$q_add** system task places an entry on a queue. The `q_id` argument is an integer input that indicates to which queue to add the entry. The `job_id` argument is an integer input that identifies the job.

The `inform_id` argument is an integer input that is associated with the queue entry. Its meaning is user-defined. For example, the `inform_id` argument can represent execution time for an entry in a CPU model. The `status` code reports on the success of the operation or error conditions as described in Table 17-16.

17.6.3 $q_remove

The **$q_remove** system task receives an entry from a queue. The `q_id` argument is an integer input that indicates from which queue to remove. The `job_id` argument is an integer output that identifies the entry being removed. The `inform_id` argument is an integer output that the queue manager stored during **$q_add**. Its meaning is user-defined. The `status` code reports on the success of the operation or error conditions as described in Table 17-16.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

17.6.4 $q_full

The **$q_full** system function checks whether there is room for another entry on a queue. It returns `0` when the queue is not full and `1` when the queue is full. The `status` code reports on the success of the operation or error conditions as described in Table 17-16.

17.6.5 $q_exam

The **$q_exam** system task provides statistical information about activity at the queue `q_id`. It returns a value in `q_stat_value` depending on the information requested in `q_stat_code`. The values of `q_stat_code` and the corresponding information returned in `q_stat_value` are described in Table 17-15. The `status` code reports on the success of the operation or error conditions as described in Table 17-16.

Table 17-15—Argument values for $q_exam system task

Value requested in q_stat_code	Information received back from q_stat_value
1	Current queue length
2	Mean interarrival time
3	Maximum queue length
4	Shortest wait time ever
5	Longest wait time for jobs still in the queue
6	Average wait time in the queue

17.6.6 Status codes

All of the queue management tasks and functions return an output status code. The status code values and corresponding information are described in Table 17-16.

Table 17-16—Status code values

Status code values	What it means
0	OK
1	Queue full, cannot add
2	Undefined q_id
3	Queue empty, cannot remove
4	Unsupported queue type, cannot create queue
5	Specified length <= 0, cannot create queue
6	Duplicate q_id, cannot create queue
7	Not enough memory, cannot create queue

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

17.7 Simulation time system functions

The following system functions provide access to current simulation time:

$time **$stime** **$realtime**

17.7.1 $time

The syntax for **$time** system function is shown in Syntax 17-14.

```
time_function ::=
    $time
```

Syntax 17-14—Syntax for $time

The **$time** system function returns an integer that is a 64-bit time, scaled to the timescale unit of the module that invoked it.

For example:

```
`timescale 10 ns / 1 ns
module test;
reg set;
parameter p = 1.55;
initial begin
    $monitor($time,,"set=",set);
    #p set = 0;
    #p set = 1;
end
endmodule

// The output from this example is as follows:
// 0 set=x
// 2 set=0
// 3 set=1
```

In this example, the **reg** `set` is assigned the value `0` at simulation time 16 ns, and the value `1` at simulation time 32 ns. The time values returned by the **$time** system function are determined by the following steps:

a) The simulation times 16 ns and 32 ns are scaled to `1.6` and `3.2` because the time unit for the module is 10 ns; therefore, time values reported by this module are multiples of 10 ns.
b) The value `1.6` is rounded to `2`, and `3.2` is rounded to `3` because the **$time** system function returns an integer. The time precision does not cause rounding of these values.

NOTE—The times at which the assignments take place in this example do not match the times reported by **$time**.

17.7.2 $stime

The syntax for **$stime** system function is shown in Syntax 17-15.

```
stime_function ::=
    $stime
```

Syntax 17-15—Syntax for $stime

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The **$stime** system function returns an unsigned integer that is a 32-bit time, scaled to the timescale unit of the module that invoked it. If the actual simulation time does not fit in 32 bits, the low order 32 bits of the current simulation time are returned.

17.7.3 $realtime

The syntax for **$realtime** system function is shown in Syntax 17-16.

```
realtime_function ::= $realtime
```

Syntax 17-16—Syntax for $realtime

The **$realtime** system function returns a real number time that, like **$time**, is scaled to the time unit of the module that invoked it.

For example:

```
`timescale 10 ns / 1 ns
module test;
reg set;
parameter p = 1.55;
initial begin
    $monitor($realtime,,"set=",set);
    #p set = 0;
    #p set = 1;
end
endmodule

// The output from this example is as follows:
// 0 set=x
// 1.6 set=0
// 3.2 set=1
```

In this example, the event times in the **reg** `set` are multiples of 10 ns because 10 ns is the time unit of the module. They are real numbers because **$realtime** returns a real number.

17.8 Conversion functions

The conversion system functions may be used in constant expressions, as specified in Clause 5.

The following functions handle **real** values:

```
integer     $rtoi(real_val) ;
real        $itor(int_val) ;
[63:0]      $realtobits(real_val) ;
real        $bitstoreal(bit_val) ;
```

$rtoi Converts real values to integers by truncating the real value (for example, 123.45 becomes 123).

$itor Converts integers to real values (for example, 123 becomes 123.0).

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

$realtobits Passes bit patterns across module ports; converts from a real number to the 64-bit representation (vector) of that real number.

$bitstoreal Is the reverse of **$realtobits**; converts from the bit pattern to a real number.

The real numbers accepted or generated by these functions shall conform to the IEEE 754 representation of the real number. The conversion shall round the result to the nearest valid representation.

For example:

The following example shows how the **$realtobits** and **$bitstoreal** functions are used in port connections:

```
module driver (net_r);
output net_r;
real r;
wire [64:1] net_r = $realtobits(r);
endmodule

module receiver (net_r);
input net_r;
wire [64:1] net_r;
real r;
initial assign r = $bitstoreal(net_r);
endmodule
```

See 5.5 for a description of **$signed** and **$unsigned**.

17.9 Probabilistic distribution functions

There is a set of random number generators that return integer values distributed according to standard probabilistic functions.

17.9.1 $random function

The syntax for the system function **$random** is shown in Syntax 17-17.

```
random_function ::=
    $random [ ( seed ) ]
```

Syntax 17-17—Syntax for $random

The system function **$random** provides a mechanism for generating random numbers. The function returns a new 32-bit random number each time it is called. The random number is a signed integer; it can be positive or negative. For further information on probabilistic random number generators, see 17.9.2.

The `seed` argument controls the numbers that **$random** returns so that different seeds generate different random streams. The `seed` argument shall be either a **reg**, an **integer**, or a **time** variable. The seed value should be assigned to this variable prior to calling **$random**.

For example:

Example 1—Where b is greater than `0`, the expression `(`**$random** % `b)` gives a number in the following range: [`(-b+1): (b-1)`].

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The following code fragment shows an example of random number generation between –59 and 59:

```
reg [23:0] rand;
rand = $random % 60;
```

Example 2—The following example shows how adding the concatenation operator to the preceding example gives `rand` a positive value from `0` to `59`:

```
reg [23:0] rand;
rand = {$random} % 60;
```

17.9.2 $dist_ functions

The syntax for the probablisitic distribution functions is shown in Syntax 17-18.

```
dist_functions ::=
        $dist_uniform ( seed , start , end )
      | $dist_normal ( seed , mean , standard_deviation )
      | $dist_exponential ( seed , mean )
      | $dist_poisson ( seed , mean )
      | $dist_chi_square ( seed , degree_of_freedom )
      | $dist_t ( seed , degree_of_freedom )
      | $dist_erlang ( seed , k_stage , mean )
```

Syntax 17-18—Syntax for probabilistic distribution functions

All arguments to the system functions are integer values. For the `exponential`, `poisson`, `chi-square`, `t`, and `erlang` functions, the arguments `mean`, `degree_of_freedom`, and `k_stage` shall be greater than `0`.

Each of these functions returns a pseudo-random number whose characteristics are described by the function name. In other words, **$dist_uniform** returns random numbers uniformly distributed in the interval specified by its arguments.

For each system function, the `seed` argument is an inout argument; that is, a value is passed to the function, and a different value is returned. The system functions shall always return the same value given the same `seed`. This facilitates debugging by making the operation of the system repeatable. The `seed` argument should be an integer variable that is initialized by the user and only updated by the system function. This ensures the desired distribution is achieved.

In the **$dist_uniform** function, the `start` and `end` arguments are integer inputs that bound the values returned. The `start` value should be smaller than the `end` value.

The `mean` argument, used by **$dist_normal**, **$dist_exponential**, **$dist_poisson**, and **$dist_erlang**, is an integer input that causes the average value returned by the function to approach the value specified.

The `standard_deviation` argument used with the **$dist_normal** function is an integer input that helps determine the shape of the density function. Larger numbers for `standard_deviation` spread the returned values over a wider range.

The `degree_of_freedom` argument used with the **$dist_chi_square** and **$dist_t** functions is an integer input that helps determine the shape of the density function. Larger numbers spread the returned values over a wider range.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

17.9.3 Algorithm for probabilistic distribution functions

Table 17-17 shows the Verilog probabilistic distribution functions listed with their corresponding C functions.

Table 17-17—Verilog to C function cross-listing

Verilog function	C function
$dist_uniform	rtl_dist_uniform
$dist_normal	rtl_dist_normal
$dist_exponential	rtl_dist_exponential
$dist_poisson	rtl_dist_poisson
$dist_chi_square	rtl_dist_chi_square
$dist_t	rtl_dist_t
$dist_erlang	rtl_dist_erlang
$random	rtl_dist_uniform (seed, LONG_MIN, LONG_MAX)

The algorithm for these functions is defined by the following C code:

```
/*
*  Algorithm for probabilistic distribution functions.
*
*  IEEE Std 1364-2005 Verilog Hardware Description Language (HDL)
*/

#include <limits.h>

static double uniform( long *seed, long start, long end );
static double normal( long *seed, long mean, long deviation);
static double exponential( long *seed, long mean);
static long poisson( long *seed, long mean);
static double chi_square( long *seed, long deg_of_free);
static double t( long *seed, long deg_of_free);
static double erlangian( long *seed, long k, long mean);

long
rtl_dist_chi_square( seed, df )
        long *seed;
        long df;
{
    double r;
    long i;

    if(df>0)
    {
       r=chi_square(seed,df);
       if(r>=0)
       {
          i=(long)(r+0.5);
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
        }
                else
        {
             r = -r;
             i=(long)(r+0.5);
             i = -i;
         }
    }
         else

    {
            print_error("WARNING: Chi_square distribution must ",
                    "have positive degree of freedom\n");
            i=0;
    }

    return (i);
}

long
rtl_dist_erlang( seed, k, mean )
        long *seed;
        long k, mean;
{
    double r;
    long i;

    if(k>0)
    {
        r=erlangian(seed,k,mean);
        if(r>=0)
        {
            i=(long)(r+0.5);
        }
        else
        {
            r = -r;
            i=(long)(r+0.5);
            i = -i;
        }
    }
    else
    {
         print_error("WARNING: k-stage erlangian distribution ",
                "must have positive k\n");
        i=0;
    }

return (i);
}

long
rtl_dist_exponential( seed, mean )
        long *seed;
        long mean;
{
    double r;
    long i;
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
    if(mean>0)
    {
        r=exponential(seed,mean);
        if(r>=0)


        {
            i=(long)(r+0.5);
        }
        else

        {
            r = -r;
            i=(long)(r+0.5);
            i = -i;
        }
     }
    else
    {
        print_error("WARNING: Exponential distribution must ",
                "have a positive mean\n");
        i=0;
    }

    return (i);
}

long
rtl_dist_normal( seed, mean, sd )
        long *seed;
        long mean, sd;
{
    double r;
    long i;

    r=normal(seed,mean,sd);
    if(r>=0)
    {
        i=(long)(r+0.5);
    }
    else
    {
        r = -r;
        i=(long)(r+0.5);
        i = -i;
    }

    return (i);
}

long
rtl_dist_poisson( seed, mean )
        long *seed;
        long mean;
{
    long i;

    if(mean>0)
    {
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
            i=poisson(seed,mean);
        }
        else
        {
            print_error("WARNING: Poisson distribution must have a ",
                    "positive mean\n");
            i=0;
        }
        return (i);
    }

    long
    rtl_dist_t( seed, df )
            long *seed;
            long df;
    {
        double r;
        long i;

        if(df>0)
        {
            r=t(seed,df);
            if(r>=0)
            {
                i=(long)(r+0.5);
            }
            else
            {
                r = -r;
                i=(long)(r+0.5);
                i = -i;
            }
        }
        else
        {
            print_error("WARNING: t distribution must have positive ",
                    "degree of freedom\n");
            i=0;
        }
        return (i);
    }

    long
    rtl_dist_uniform(seed, start, end)
            long *seed;
            long start, end;
    {
            double r;
            long i;

            if (start >= end) return(start);

            if (end != LONG_MAX)
            {
                    end++;
                    r = uniform( seed, start, end );
                    if (r >= 0)
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
                {
                        i = (long) r;
                }
                else
                {
                        i = (long) (r-1);
                }
                if (i<start) i = start;
                if (i>=end) i = end-1;
        }
        else if (start!=LONG_MIN)
        {
                start--;
                r = uniform( seed, start, end) + 1.0;
                if (r>=0)
                {
                        i = (long) r;
                }
                else
                {
                        i = (long) (r-1);
                }
                if (i<=start) i = start+1;
                if (i>end) i = end;
        }
        else
        {
                r =(uniform(seed,start,end)+
                        2147483648.0)/4294967295.0);
                r = r*4294967296.0-2147483648.0;
                if (r>=0)
                {
                        i = (long) r;
                }
                else
                {
                        i = (long) (r-1);
                }
        }

        return (i);
}

static double
uniform( seed, start, end )
        long *seed, start, end;
{
        union u_s
        {
                float s;
                unsigned stemp;
        } u;

        double d = 0.00000011920928955078125;
        double a,b,c;


        if ((*seed) == 0)
                *seed = 259341593;
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
        if (start >= end)
        {
                a = 0.0;
                b = 2147483647.0;
        }
        else
        {
                a = (double) start;
                b = (double) end;
        }
        *seed = 69069 * (*seed) + 1;
        u.stemp = *seed;

        /*
         * This relies on IEEE floating point format
         */

        u.stemp = (u.stemp >> 9) | 0x3f800000;

        c = (double) u.s;

        c = c+(c*d);
        c = ((b - a) * (c - 1.0)) + a;

        return (c);
}


static double
normal(seed,mean,deviation)
long *seed,mean,deviation;
{
    double v1,v2,s;
    double log(), sqrt();

    s = 1.0;
    while((s >= 1.0) || (s == 0.0))
    {
        v1 = uniform(seed,-1,1);
        v2 = uniform(seed,-1,1);
        s = v1 * v1 + v2 * v2;
    }
    s = v1 * sqrt(-2.0 * log(s) / s);
    v1 = (double) deviation;
    v2 = (double) mean;
    return(s * v1 + v2);
}

static double
exponential(seed,mean)
long *seed,mean;
{
    double log(),n;
    n = uniform(seed,0,1);
    if(n != 0)
    {
        n = -log(n) * mean;
    }
```


--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
    return(n);
}

static long
poisson(seed,mean)
long *seed,mean;
{
    long n;
    double p,q;
    double exp();

    n = 0;
    q = -(double)mean;
    p = exp(q);
    q = uniform(seed,0,1);
    while(p < q)
    {
        n++;
        q = uniform(seed,0,1) * q;
    }
    return(n);
}

static double
chi_square(seed,deg_of_free)
long *seed,deg_of_free;
{
    double x;
    long k;
    if(deg_of_free % 2)
    {
        x = normal(seed,0,1);
        x = x * x;
    }
    else
    {
        x = 0.0;
    }
    for(k = 2; k <= deg_of_free; k = k + 2)
    {
        x = x + 2 * exponential(seed,1);
    }
    return(x);
}

static double
t(seed,deg_of_free)
long *seed,deg_of_free;

{
    double sqrt(),x;
    double chi2 = chi_square(seed,deg_of_free);
    double div = chi2 / (double)deg_of_free;
    double root = sqrt(div);
    x = normal(seed,0,1) / root;
    return(x);
}

static double
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
erlangian(seed,k,mean)
long *seed,k,mean;
{
    double x,log(),a,b;
    long i;

    x=1.0;
    for(i=1;i<=k;i++)


    {
        x = x * uniform(seed,0,1);
    }
    a=(double)mean;
    b=(double)k;
    x= -a*log(x)/b;
    return(x);
}
```

17.10 Command line input

An alternative to reading a file to obtain information for use in the simulation is specifying information with the command to invoke the simulator. This information is in the form of an optional argument provided to the simulation. These arguments are visually distinguished from other simulator arguments by their starting with the plus (+) character.

These arguments, referred to below as *plusargs*, are accessible through the system functions described in 17.10.1 and 17.10.2.

17.10.1 $test$plusargs (string)

The **$test$plusarg** system function searches the list of *plusargs* for a user specified *plusarg_string*. The string is specified in the argument to the system function as either a string or a nonreal variable that is interpreted as a string. This string shall not include the leading plus sign of the command line argument. The *plusargs* present on the command line are searched in the order provided. If the prefix of one of the supplied *plusargs* matches all characters in the provided string, the function returns a nonzero integer. If no *plusarg* from the command line matches the string provided, the function returns the integer value zero.

For example:

Run simulator with command: `+HELLO`

The Verilog code is as follows:

```
initial  begin
  if (($test$plusargs("HELLO"))      $display("Hello argument found.")
  if (($test$plusargs("HE")) $display("The HE subset string is detected.");
  if (($test$plusargs("H"))  $display("Argument starting with H found.");
  if (($test$plusargs("HELLO_HERE"))$display("Long argument.");
  if (($test$plusargs("HI"))         $display("Simple greeting.");
  if (($test$plusargs("LO"))         $display("Does not match.");
end
```

This would produce the following output:

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
Hello argument found.
The HE subset string is detected.
Argument starting with H found.
```

17.10.2 $value$plusargs (user_string, variable)

The **$value$plusarg** system function searches the list of *plusargs* (like the **$test$plusargs** system function) for a user-specified *plusarg_string*. The string is specified in the first argument to the system function as either a string or a nonreal variable that is interpreted as a string. This string shall not include the leading plus sign of the command line argument. The *plusargs* present on the command line are searched in the order provided. If the prefix of one of the supplied *plusargs* matches all characters in the provided string, the function returns a nonzero integer, the remainder of the string is converted to the type specified in the *user_string*, and the resulting value is stored in the variable provided. If no string is found matching, the function returns the integer value zero, and the variable provided is not modified. No warnings shall be generated when the function returns zero (`0`).

The *user_string* shall be of the following form: **"***plusarg_string format_string***"**. The format strings are the same as the **$display** system tasks. These are the only valid ones (uppercase and lowercase as well as leading `0` forms are valid):

- `%d` decimal conversion
- `%o` octal conversion
- `%h` hexadecimal conversion
- `%b` binary conversion
- `%e` real exponential conversion
- `%f` real decimal conversion
- `%g` real decimal or exponential conversion
- `%s` string (no conversion)

The first string from the list of *plusargs* provided to the simulator, which matches the *plusarg_string* portion of the *user_string* specified shall be the *plusarg* string available for conversion. The remainder string of the matching *plusarg* (the remainder is the part of the *plusarg* string after the portion that matches the user's *plusarg_string*) shall be converted from a string into the format indicated by the format string and stored in the variable provided. If there is no remaining string, the value stored into the variable shall be either a zero or an empty string value.

If the size of the variable is larger than the value after conversion, the value stored is zero-padded to the width of the variable. If the variable cannot contain the value after conversion, the value shall be truncated. If the value is negative, the value shall be considered larger than the variable provided. If characters exist in the string available for conversion that are illegal for the specified conversion, the variable shall be written with the value `'bx`.

Given the Verilog HDL

```
`define STRING reg [1024 * 8:1]

module goodtasks;
  `STRING str;
  integer int;
  reg [31:0] vect;
  real realvar;

  initial
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
    begin
      if ($value$plusargs("TEST=%d",int))
        $display("value was %d",int);
      else
        $display("+TEST= not found");
      #100 $finish;
    end
endmodule

module ieee1364_example;
  real frequency;
  reg [8*32:1] testname;
  reg [64*8:1] pstring;
  reg clk;

  initial
    begin
      if ($value$plusargs("TESTNAME=%s",testname))
        begin
          $display(" TESTNAME= %s.",testname);
          $finish;
        end

      if (!($value$plusargs("FREQ+%0F",frequency)))
        frequency = 8.33333; // 166 MHz
      $display("frequency = %f",frequency);

      pstring = "TEST%d";
      if ($value$plusargs(pstring, testname))
        $display("Running test number %0d.",testname);
    end
endmodule
```

and adding to the tool's command line the *plusarg*

```
+TEST=5
```

will result in the following output:

```
value was           5
frequency = 8.333330
Running text number x.
```

Adding to the tool's command line the *plusarg*

```
+TESTNAME=bar
```

will result in the following output:

```
+TEST= not found
 TESTNAME=                                bar.
```

Adding to the tool's command line the *plusarg*

```
+FREQ+9.234
```

will result in the following output:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
+TEST= not found
frequency = 9.234000
```

Adding to the tool's command line the *plusarg*

```
+TEST23
```

will result in the following output:

```
+TEST= not found
frequency = 8.333330
Running test number 23.
```

17.11 Math functions

There are integer and real math functions. The math system functions may be used in constant expressions, as specified in Clause 5.

17.11.1 Integer math functions

For example:

```
integer result;
result = $clog2(n);
```

The system function **$clog2** shall return the ceiling of the log base 2 of the argument (the log rounded up to an integer value). The argument can be an integer or an arbitrary sized vector value. The argument shall be treated as an unsigned value, and an argument value of 0 shall produce a result of 0.

This system function can be used to compute the minimum address width necessary to address a memory of a given size or the minimum vector width necessary to represent a given number of states.

17.11.2 Real math functions

The system functions in Table 17-18 shall accept real arguments and return a real result. Their behavior shall match the equivalent C language standard math library function indicated.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 17-18—Verilog to C real math function cross-listing

Verilog function	Equivalent C function	Description
$ln(x)	log(x)	Natural logarithm
$log10(x)	log10(x)	Decimal logarithm
$exp(x)	exp(x)	Exponential
$sqrt(x)	sqrt(x)	Square root
$pow(x,y)	pow(x,y)	x**y
$floor(x)	floor(x)	Floor
$ceil(x)	ceil(x)	Ceiling
$sin(x)	sin(x)	Sine
$cos(x)	cos(x)	Cosine
$tan(x)	tan(x)	Tangent
$asin(x)	asin(x)	Arc-sine
$acos(x)	acos(x)	Arc-cosine
$atan(x)	atan(x)	Arc-tangent
$atan2(x,y)	atan2(x,y)	Arc-tangent of x/y
$hypot(x,y)	hypot(x,y)	sqrt(x*x+y*y)
$sinh(x)	sinh(x)	Hyperbolic sine
$cosh(x)	cosh(x)	Hyperbolic cosine
$tanh(x)	tanh(x)	Hyperbolic tangent
$asinh(x)	asinh(x)	Arc-hyperbolic sine
$acosh(x)	acosh(x)	Arc-hyperbolic cosine
$atanh(x)	atanh(x)	Arc-hyperbolic tangent

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

18. Value change dump (VCD) files

A *VCD file* contains information about value changes on selected variables in the design stored by VCD system tasks. Two types of VCD files exist:

a) *Four-state:* to represent variable changes in `0`, `1`, `x`, and `z` with no strength information.
b) *Extended:* to represent variable changes in all states and strength information.

This clause describes how to generate both types of VCD files and their format.

18.1 Creating four-state VCD file

The steps involved in creating the four-state VCD file are listed below and illustrated in Figure 18-1.

a) Insert the VCD system tasks in the Verilog source file to define the dump file name and to specify the variables to be dumped.
b) Run the simulation.

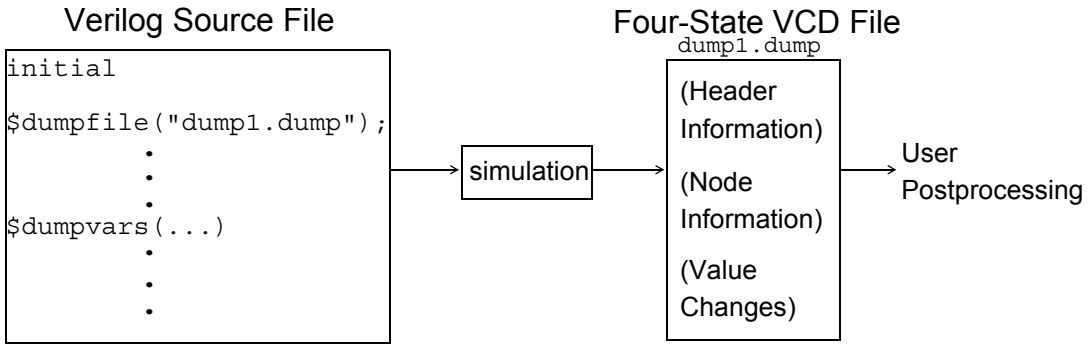


Figure 18-1—Creating the four-state VCD file

A VCD file is an ASCII file that contains header information, variable definitions, and the value changes for all variables specified in the task calls.

Several system tasks can be inserted in the source description to create and control the VCD file.

18.1.1 Specifying name of dump file ($dumpfile)

The **$dumpfile** task shall be used to specify the name of the VCD file. The syntax for the task is given in Syntax 18-1.

```
dumpfile_task ::=
    $dumpfile ( filename ) ;
```

Syntax 18-1—Syntax for $dumpfile task

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The *filename* syntax is given in Syntax 18-2.

```
filename ::=
        literal_string
      | variable
      | expression
```

Syntax 18-2—Syntax for filename

The *filename* is optional and defaults to the literal string `dump.vcd` if not specified.

For example:

```
initial  $dumpfile ("module1.dump") ;
```

18.1.2 Specifying variables to be dumped ($dumpvars)

The **$dumpvars** task shall be used to list which variables to dump into the file specified by **$dumpfile**. The **$dumpvars** task can be invoked as often as desired throughout the model (for example, within various blocks), but the execution of all the **$dumpvars** tasks shall be at the same simulation time.

The **$dumpvars** task can be used with or without arguments. The syntax for the **$dumpvars** task is given in Syntax 18-3.

```
dumpvars_task ::= (Not in the Annex A BNF)
        $dumpvars ;
      | $dumpvars ( levels [ , list_of_modules_or_variables ] ) ;
list_of_modules_or_variables ::= (Not in the Annex A BNF)
      module_or_variable { , module_or_variable }
module_or_variable ::=
        module_identifier
      | variable_identifier
```

Syntax 18-3—Syntax for $dumpvars task

When invoked with no arguments, **$dumpvars** dumps all the variables in the model to the VCD file.

When the **$dumpvars** task is specified with arguments, the first argument indicates how many levels of the hierarchy below each specified module instance to dump to the VCD file. Subsequent arguments specify which scopes of the model to dump to the VCD file. These arguments can specify entire modules or individual variables within a module.

Setting the first argument to `0` causes a dump of all variables in the specified module and in all module instances below the specified module. The argument `0` applies only to subsequent arguments that specify module instances, and not to individual variables.

For example:

Example 1

```
$dumpvars (1, top);
```


--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Because the first argument is a `1`, this invocation dumps all variables within the module `top`; it does not dump variables in any of the modules instantiated by module `top`.

Example 2

```
$dumpvars (0, top);
```

In this example, the **$dumpvars** task shall dump all variables in the module `top` and in all module instances below module `top` in the hierarchy.

Example 3—This example shows how the **$dumpvars** task can specify both modules and individual variables.

```
$dumpvars (0, top.mod1, top.mod2.net1);
```

This call shall dump all variables in module `mod1` and in all module instances below `mod1`, along with variable `net1` in module `mod2`. The argument `0` applies only to the module instance `top.mod1` and not to the individual variable `top.mod2.net1`.

18.1.3 Stopping and resuming the dump ($dumpoff/$dumpon)

Executing the **$dumpvars** task causes the value change dumping to start at the end of the current simulation time unit. To suspend the dump, the **$dumpoff** task can be invoked. To resume the dump, the **$dumpon** task can be invoked. The syntax of these two tasks is given in Syntax 18-4.

```
dumpoff_task ::=
    $dumpoff ;
dumpon_task ::=
    $dumpon ;
```

Syntax 18-4—Syntax for $dumpoff and $dumpon tasks

When the **$dumpoff** task is executed, a checkpoint is made in which every selected variable is dumped as an `x` value. When the **$dumpon** task is later executed, each variable is dumped with its value at that time. In the interval between **$dumpoff** and **$dumpon**, no value changes are dumped.

The **$dumpoff** and **$dumpon** tasks provide the mechanism to control the simulation period during which the dump shall take place.

For example:

```
initial  begin
  #10      $dumpvars( . . . );

  #200     $dumpoff;

  #800     $dumpon;

  #900     $dumpoff;
end
```

This example starts the VCD after 10 time units, stops it 200 time units later (at time 210), restarts it again 800 time units later (at time 1010), and stops it 900 time units later (at time 1910).

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

18.1.4 Generating a checkpoint ($dumpall)

The **$dumpall** task creates a checkpoint in the VCD file that shows the current value of all selected variables. The syntax is given in Syntax 18-5.

```
dumpall_task ::=
    $dumpall ;
```

Syntax 18-5—Syntax for $dumpall task

When dumping is enabled, the value change dumper records the values of the variables that change during each time increment. Values of variables that do not change during a time increment are not dumped.

18.1.5 Limiting size of dump file ($dumplimit)

The **$dumplimit** task can be used to set the size of the VCD file. The syntax for this task is given in Syntax 18-6.

```
dumplimit_task ::=
    $dumplimit ( filesize ) ;
```

Syntax 18-6—Syntax for $dumplimit task

The *filesize* argument specifies the maximum size of the VCD file in bytes. When the size of the VCD file reaches this number of bytes, the dumping stops, and a comment is inserted in the VCD file indicating the dump limit was reached.

18.1.6 Reading dump file during simulation ($dumpflush)

The **$dumpflush** task can be used to empty the VCD file buffer of the operating system to ensure all the data in that buffer are stored in the VCD file. After executing a **$dumpflush** task, dumping is resumed as before so no value changes are lost. The syntax for the task is given in Syntax 18-7.

```
dumpflush_task ::=
    $dumpflush ;
```

Syntax 18-7—Syntax for $dumpflush task

A common application is to call **$dumpflush** to update the dump file so an application program can read the VCD file during a simulation.

For example:

Example 1—This example shows how the **$dumpflush** task can be used in a Verilog HDL source file.

```
initial begin
    $dumpvars ;
            .
            .
            .
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
        $dumpflush ;

        $(applications program) ;

    end
```

Example 2—The following is a simple source description example to produce a VCD file:

In this example, the name of the dump file is `verilog.dump`. It dumps value changes for all variables in the model. Dumping begins when an event `do_dump` occurs. The dumping continues for 500 clock cycles and then stops and waits for the event `do_dump` to be triggered again. At every 10000 time steps, the current values of all VCD variables are dumped.

```
module dump;
    event do_dump;

    initial $dumpfile("verilog.dump");
    initial @do_dump
        $dumpvars;          //dump variables in the design

    always @do_dump         //to begin the dump at event do_dump
    begin
        $dumpon;            //no effect the first time through
        repeat (500) @(posedge clock); //dump for 500 cycles
        $dumpoff;           //stop the dump
    end

    initial @(do_dump)
        forever #10000 $dumpall; //checkpoint all variables
endmodule
```

18.2 Format of four-state VCD file

The dump file is structured in a free format. White space is used to separate commands and to make the file easily readable by a text editor.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

18.2.1 Syntax of four-state VCD file

The syntax of the four-state VCD file is given in Syntax 18-8.

```
value_change_dump_definitions ::=
        { declaration_command } { simulation_command }
declaration_command ::=
        declaration_keyword
        [ command_text ]
        $end
simulation_command ::=
          simulation_keyword { value_change } $end
        | $comment [ comment_text ] $end
        | simulation_time
        | value_change
declaration_keyword ::=
          $comment | $date | $enddefinitions | $scope | $timescale | $upscope
        | $var | $version
simulation_keyword ::=
        $dumpall | $dumpoff | $dumpon | $dumpvars
simulation_time ::=
        # decimal_number
value_change ::=
          scalar_value_change
        | vector_value_change
scalar_value_change ::=
        value identifier_code
value ::=
        0 | 1 | x | X | z | Z
vector_value_change ::=
          b binary_number identifier_code
        | B binary_number identifier_code
        | r real_number identifier_code
        | R real_number identifier_code
identifier_code ::=
        { ASCII character }
```

Syntax 18-8—Syntax for output four-state VCD file

The VCD file starts with header information giving the date, the version number of the simulator used for the simulation, and the timescale used. Next, the file contains definitions of the scope and type of variables being dumped, followed by the actual value changes at each simulation time increment. Only the variables that change value during a time increment are listed.

The simulation time recorded in the VCD file is the absolute value of the simulation time for the changes in variable values that follow.

Value changes for real variables are specified by real numbers. Value changes for all other variables are specified in binary format by `0`, `1`, `x`, or `z` values. Strength information and memories are not dumped.

A real number is dumped using a `%.16g printf()` format. This preserves the precision of that number by outputting all 53 bits in the mantissa of a 64-bit IEEE 754 double-precision number. Application programs can read a real number using a `%g` format to `scanf()`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The value change dumper generates character identifier codes to represent variables. The identifier code is a code composed of the printable characters, which are in the ASCII character set from ! to ~ (decimal 33 to 126).

The VCD format does not support a mechanism to dump part of a vector. For example, bits 8 to 15 (`[8:15]`) of a 16-bit vector cannot be dumped in VCD file; instead, the entire vector (`[0:15]`) has to be dumped. In addition, expressions, such as a + b, cannot be dumped in the VCD file.

Data in the VCD file are case sensitive.

18.2.2 Formats of variable values

Variables can be either scalars or vectors. Each type is dumped in its own format. Dumps of value changes to scalar variables shall not have any white space between the value and the identifier code.

Dumps of value changes to vectors shall not have any white space between the base letter and the value digits, but they shall have one white space between the value digits and the identifier code.

The output format for each value is right-justified. Vector values appear in the shortest form possible: redundant bit values that result from left-extending values to fill a particular vector size are eliminated.

The rules for left-extending vector values are given in Table 18-1.

Table 18-1—Rules for left-extending vector values

When the value is	VCD left-extends with
1	0
0	0
Z	Z
X	X

Table 18-2 shows how the VCD can shorten values.

Table 18-2—How the VCD can shorten values

Binary value	Extends to fill a 4-bit reg as	Appears in the VCD file as
10	0010	b10
X10	XX10	bX10
ZX0	ZZX0	bZX0
0X10	0X10	b0X10

Events are dumped in the same format as scalars; for example, `1*%`. For events, however, the value (1 in this example) is irrelevant. Only the identifier code (`*%` in this example) is significant. It appears in the VCD file as a marker to indicate the event was triggered during the time step.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

For example:

```
1*@     No space between the value 1 and the identifier code *@

b1100x01z (k        No space between the b and 1100x01z,
                    but a space between b1100x01z and (k
```

18.2.3 Description of keyword commands

The general information in the VCD file is presented as a series of sections surrounded by keywords. Keyword commands provide a means of inserting information in the VCD file. Keyword commands can be inserted either by the dumper or manually.

This subclause deals with the keyword commands given in Table 18-3.

Table 18-3—Keyword commands

Declaration keywords		Simulation keywords
$comment	**$timescale**	**$dumpall**
$date	**$upscope**	**$dumpoff**
$enddefinitions	**$var**	**$dumpon**
$scope	**$version**	**$dumpvars**

18.2.3.1 $comment

The **$comment** section provides a means of inserting a comment in the VCD file. The syntax for the section is given in Syntax 18-9.

```
vcd_declaration_comment ::=
    $comment comment_text $end
```

Syntax 18-9—Syntax for $comment section

For example:

```
$comment This is a single-line comment      $end
$comment This is a
multiple-line comment
$end
```

18.2.3.2 $date

The **$date** section indicates the date on which the VCD file was generated. The syntax for the section is given in Syntax 18-10.

```
vcd_declaration_date ::=
    $date date_text $end
```

Syntax 18-10—Syntax for $date section

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

For example:

```
$date
        June 25, 1989 09:24:35
$end
```

18.2.3.3 $enddefinitions

The **$enddefinitions** section marks the end of the header information and definitions. The syntax for the section is given in Syntax 18-11..

```
vcd_declaration_enddefinitions ::=
    $enddefinitions $end
```

Syntax 18-11—Syntax for $enddefinitions section

18.2.3.4 $scope

The **$scope** section defines the scope of the variables being dumped. The syntax for the section is given in Syntax 18-12.

```
vcd_declaration_scope ::=
    $scope scope_type scope_identifier $end
scope_type ::=
      begin
    | fork
    | function
    | module
    | task
```

Syntax 18-12—Syntax for $scope section

The scope type indicates one of the following scopes:

module	Top-level module and module instances
task	Tasks
function	Functions
begin	Named sequential blocks
fork	Named parallel blocks

For example:

```
$scope
        module top
$end
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

18.2.3.5 $timescale

The **$timescale** keyword specifies what timescale was used for the simulation. The syntax for the keyword is given in Syntax 18-13.

```
vcd_declaration_timescale ::=
      $timescale time_number time_unit $end
time_number ::=
      1 | 10 | 100
time_unit ::=
      s | ms | us | ns | ps | fs
```

Syntax 18-13—Syntax for $timescale

For example:

```
$timescale   10   ns   $end
```

18.2.3.6 $upscope

The **$upscope** section indicates a change of scope to the next higher level in the design hierarchy. The syntax for the section is given in Syntax 18-14.

```
vcd_declaration_upscope ::=
      $upscope $end
```

Syntax 18-14—Syntax for $upscope section

18.2.3.7 $var

The **$var** section prints the names and identifier codes of the variables being dumped. The syntax for the section is given in Syntax 18-15.

```
vcd_declaration_vars ::=
      $var var_type size identifier_code reference $end
var_type ::=
        event | integer | parameter | real | realtime | reg | supply0 | supply1 | time
      | tri | triand | trior | trireg | tri0 | tri1 | wand | wire | wor
size ::=
      decimal_number
reference ::=
        identifier
      | identifier [ bit_select_index ]
      | identifier [ msb_index : lsb_index ]
index ::=
      decimal_number
```

Syntax 18-15—Syntax for $var section

Size specifies how many bits are in the variable.

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The identifier code specifies the name of the variable using printable ASCII characters, as previously described.

a) The msb index indicates the most significant index; the lsb index indicates the least significant index.
b) More than one reference name can be mapped to the same identifier code. For example, net10 and net15 can be interconnected in the circuit and, therefore, have the same identifier code.
c) The individual bits of vector nets can be dumped individually.
d) The identifier is the name of the variable being dumped in the model.

In the **$var** section, a net of net type **uwire** shall have a variable type of **wire**.

For example:

```
$var
        integer 32 (2 index
$end
```

18.2.3.8 $version

The **$version** section indicates which version of the VCD writer was used to produce the VCD file and the **$dumpfile** system task used to create the file. If a variable or an expression was used to specify the *filename* within **$dumpfile**, the unevaluated variable or expression literal shall appear in the **$version** string. The syntax for the **$version** section is given in Syntax 18-16.

```
vcd_declaration_version ::=
    $version version_text system_task $end
```

Syntax 18-16—Syntax for $version section

For example:

```
$version
    VERILOG-SIMULATOR 1.0a
      $dumpfile("dump1.dump")
$end
```

18.2.3.9 $dumpall

The **$dumpall** keyword specifies current values of all variables dumped. The syntax for the keyword is given in Syntax 18-17.

```
vcd_simulation_dumpall ::=
    $dumpall { value_changes } $end
```

Syntax 18-17—Syntax for $dumpall keyword

For example:

```
$dumpall    1*@   x*#    0*$    bx    (k    $end
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

18.2.3.10 $dumpoff

The **$dumpoff** keyword indicates all variables dumped with X values. The syntax for the keyword is given in Syntax 18-18.

```
vcd_simulation_dumpoff ::=
    $dumpoff { value_changes } $end
```

Syntax 18-18—Syntax for $dumpoff keyword

For example:

```
$dumpoff   x*@   x*#    x*$    bx    (k    $end
```

18.2.3.11 $dumpon

The **$dumpon** keyword indicates resumption of dumping and lists current values of all variables dumped. The syntax for the keyword is given in Syntax 18-19.

```
vcd_simulation_dumpon ::=
    $dumpon { value_changes } $end
```

Syntax 18-19—Syntax for $dumpon keyword

For example:

```
$dumpon    x*@   0*#    x*$    b1    (k    $end
```

18.2.3.12 $dumpvars

The section beginning with **$dumpvars** keyword lists initial values of all variables dumped. The syntax for the keyword is given in Syntax 18-20.

```
vcd_simulation_dumpvars ::=
    $dumpvars { value_changes } $end
```

Syntax 18-20—Syntax for $dumpvars keyword

For example:

```
$dumpvars    x*@    z*$    b0    (k    $end
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

18.2.4 Four-state VCD file format example

The following example illustrates the format of the four-state VCD file.

```
$date June 26, 1989 10:05:41
$end
$version VERILOG-SIMULATOR 1.0a
$end
$timescale 1 ns
$end
$scope module top $end
$scope module m1  $end
$var trireg 1 *@ net1 $end
$var trireg 1 *# net2 $end
$var trireg 1 *$ net3 $end
$upscope $end
$scope task t1 $end
$var reg 32 (k accumulator[31:0] $end
$var integer 32 {2 index  $end
$upscope $end
$upscope $end
$enddefinitions $end
$comment
  $dumpvars was executed at time '#500'.
  All initial values are dumped at this time.
$end

#500
$dumpvars
x*@
x*#
x*$
bx (k
bx {2
$end
#505
0*@
1*#
1*$
b10zx1110x11100 (k
b1111000101z01x {2
#510
0*$
#520
1*$
#530
0*$
bz (k
#535
$dumpall   0*@   1*#   0*$
```

(Continued in right column)

(Continued from left column)

```
bz (k
b1111000101z01x {2
$end
#540
1*$
#1000
$dumpoff
x*@
x*#
x*$
bx (k
bx {2
$end
#2000
$dumpon
z*@
1*#
0*$
b0 (k
bx {2
$end
#2010
1*$
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

18.3 Creating extended VCD file

The steps involved in creating the extended VCD file are listed below and illustrated in Figure 18-2.

a) Insert the extended VCD system tasks in the Verilog source file to define the dump file name and to specify the variables to be dumped.
b) Run the simulation.

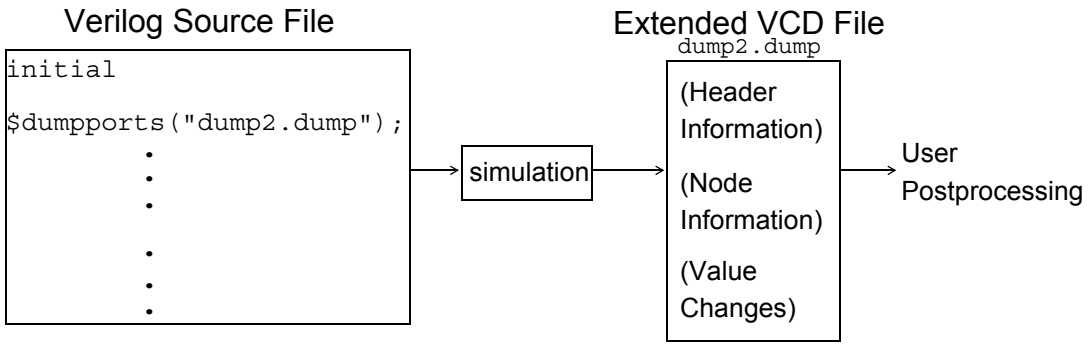


Figure 18-2—Creating the extended VCD file

The four-state VCD file rules and syntax apply to the extended VCD file unless otherwise stated in this subclause.

18.3.1 Specifying dump file name and ports to be dumped ($dumpports)

The **$dumpports** task shall be used to specify the name of the VCD file and the ports to be dumped. The syntax for the task is given in Syntax 18-21.

```
dumpports_task ::=
      $dumpports ( scope_list , file_pathname ) ;
scope_list ::=
      module_identifier { , module_identfier }
file_pathname ::=
       literal_string
      | variable
      | expression
```

Syntax 18-21—Syntax for $dumpports task

The arguments are optional and are defined as follows:

scope_list — One or more module identifiers. Only modules are allowed (not variables). If more than one *module_identifier* is specified, they shall be separated by a comma. Pathnames to modules are allowed, using the period hierarchy separator. Literal strings are not allowed for the *module_identifier*.

If no *scope_list* value is provided, the scope shall be the module from which **$dumpports** is called.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

file_pathname — Can be a double quoted pathname (literal string), a reg type variable, or an expression that denotes the file which shall contain the port VCD information. If no *file_pathname* is provided, the file shall be written to the current working directory with the name *dumpports.vcd*. If that file already exists, it shall be silently overwritten. All file-writing checks shall be made by the simulator (e.g., write rights, correct pathname) and appropriate errors or warnings issued.

The following rules apply to the use of the $dumpports system task:

- All the ports in the model from the point of the **$dumpports** call are considered primary I/O pins and shall be included in the VCD file. However, any ports that exist in instantiations below *scope_list* are not dumped.
- If no arguments are specified for the task, **$dumpports;** and **$dumpports();** are allowed. In both of these cases, the default values for the arguments shall be used.
- If the first argument is null, a comma shall be used before specifying the second argument in the argument list.
- Each scope specified in the *scope_list* shall be unique. If multiple calls to **$dumpports** are specified, the *scope_list* values in these calls shall also be unique.
- The **$dumpports** task can be used in source code that also contains the **$dumpvars** task.
- When **$dumpports** executes, the associated value change dumping shall start at the end of the current simulation time unit.
- The **$dumpports** task can be invoked multiple times throughout the model, but the execution of all **$dumpports** tasks shall be at the same simulation time. Specifying the same *file_pathname* multiple times is not allowed.

18.3.2 Stopping and resuming the dump ($dumpportsoff/$dumpportson)

The **$dumpportsoff** and **$dumpportson** system tasks provide a means to control the simulation period for dumping port values. The syntax for these system tasks is given in Syntax 18-22.

```
dumpportsoff_task ::=
      $dumpportsoff ( file_pathname ) ;
dumpportson_task ::=
      $dumpportson ( file_pathname ) ;
file_pathname ::=
       literal_string
      | variable
      | expression
```

Syntax 18-22—Syntax for $dumpportsoff and $dumpportson system tasks

The *file_pathname* argument can be a double quoted pathname (literal string), a reg type variable, or an expression that denotes the *file_pathname* specified in the **$dumpports** system task.

When the **$dumpportsoff** task is executed, a checkpoint is made in the *file_pathname* where each specified port is dumped with an X value. Port values are no longer dumped from that simulation time forward. If *file_pathname* is not specified, all dumping to files opened by **$dumpports** calls shall be suspended.

When the **$dumpportson** task is executed, all ports specified by the associated **$dumpports** call shall have their values dumped. This system task is typically used to resume dumping after the execution of

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

$dumpportsoff. If *file_pathname* is not specified, dumping shall resume for all files specified by **$dumpports** calls, if dumping to those files was stopped.

If **$dumpportson** is executed while ports are already being dumped to *file_pathname*, the system task is ignored. If **$dumpportsoff** is executed while port dumping is already suspended for *file_pathname*, the system task is ignored.

18.3.3 Generating a checkpoint ($dumpportsall)

The **$dumpportsall** system task creates a checkpoint in the VCD file that shows the value of all selected ports at that time in the simulation, regardless of whether the port values have changed since the last time step. The syntax for this system task is given in Syntax 18-23.

```
dumpportsall_task ::=
    $dumpportsall ( file_pathname ) ;
file_pathname ::=
     literal_string
    | variable
    | expression
```

Syntax 18-23—Syntax for $dumpportsall system task

The *file_pathname* argument can be a double quoted pathname (literal string), a reg type variable, or an expression that denotes the *file_pathname* specified in the **$dumpports** system task.

If the *file_pathname* is not specified, checkpointing occurs for all files opened by calls to **$dumpports**.

18.3.4 Limiting size of dump file ($dumpportslimit)

The **$dumpportslimit** system task allows control of the VCD file size. The syntax for this system task is given in Syntax 18-24.

```
dumpportslimit_task ::=
    $dumpportslimit ( filesize , file_pathname ) ;
file_size ::=
    integer
file_pathname ::=
     literal_string
    | variable
    | expression
```

Syntax 18-24—Syntax for $dumpportslimit system task

The *filesize* argument is required, and it specifies the maximum size in bytes for the associated *file_pathname*. When this *filesize* is reached, the dumping stops, and a comment is inserted into *file_pathname* indicating the size limit was attained.

The *file_pathname* argument can be a double quoted pathname (literal string), a reg type variable, or an expression that denotes the *file_pathname* specified in the **$dumpports** system task.

If the *file_pathname* is not specified, the *filesize* limit applies to all files opened for dumping due to calls to **$dumpports**.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

18.3.5 Reading dump file during simulation ($dumpportsflush)

To facilitate performance, simulators often buffer VCD output and write to the file at intervals, instead of line by line. The **$dumpportsflush** system task writes all port values to the associated file, clearing a simulator's VCD buffer.

The syntax for this system task is given in Syntax 18-25.

```
dumpportsflush_task ::=
      $dumpportsflush ( file_pathname ) ;
file_pathname ::=
       literal_string
      | variable
      | expression
```

Syntax 18-25—Syntax for $dumpportsflush system task

The *file_pathname* argument can be a double quoted pathname (literal string), a reg type variable, or an expression that denotes the *file_pathname* specified in the **$dumpports** system task.

If the *file_pathname* is not specified, the VCD buffers shall be flushed for all files opened by calls to **$dumpports**.

18.3.6 Description of keyword commands

The general information in the extended VCD file is presented as a series of sections surrounded by keywords. Keyword commands provide a means of inserting information in the extended VCD file. Keyword commands can be inserted either by the dumper or manually. Extended VCD provides one additional keyword command to that of the four-state VCD.

18.3.6.1 $vcdclose

The **$vcdclose** keyword indicates the final simulation time at the time the extended VCD file is closed. This allows accurate recording of the end simulation time, regardless of the state of signal changes, in order to assist parsers that require this information. The syntax for the keyword is given in Syntax 18-26.

```
vcdclose_task ::=
      $vcdclose final_simulation_time $end
```

Syntax 18-26—Syntax for $vcdclose keyword

For example:

```
$vcdclose #13000 $end
```

18.3.7 General rules for extended VCD system tasks

For each extended VCD system task, the following rules apply:

— If a *file_pathname* is specified that does not match a *file_pathname* specified in a **$dumpports** call, the control task shall be ignored.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

— If no arguments are specified for the tasks that have only optional arguments, the system task name can be used with no arguments or the name followed by () can be specified, for example, **$dumpportsflush** or **$dumpportsflush()**. In both of these cases, the default actions for the arguments shall be executed.

18.4 Format of extended VCD file

The format of the extended VCD file is similar to that of the four-state VCD file, as it is also structured in a free format. White space is used to separate commands and to make the file easily readable by a text editor.

18.4.1 Syntax of extended VCD file

The syntax of the extended VCD file is given in Syntax 18-27. A four-state VCD construct name that matches an extended VCD construct shall be considered equivalent, except if preceded by an `*`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
value_change_dump_definitions ::={declaration_command} {simulation_command}
declaration_command ::= declaration_keyword [command_text] $end
simulation_command ::= (Not in the Annex A BNF)
        simulation_keyword { value_change } $end
      | $comment [comment_text] $end
      | simulation_time
      | value_change
* declaration_keyword ::=
        $comment | $date | $enddefinitions | $scope | $timescale | $upscope | $var
      | $vcdclose | $version
command_text ::=
        comment_text | close_text | date_section | scope_section | timescale_section
      | var_section | version_section
* simulation_keyword ::= $dumpports | $dumpportsoff | $dumpportson |
      $dumpportsall
simulation_time ::= #decimal_number
value_change ::= value identifier_code
value ::= pport_value 0_strength_component 1_strength_component
port_value ::= input_value | output_value | unknown_direction_value
input_value ::= D | U | N | Z | d | u
output_value ::= L | H | X | T | l | h
unknown_direction_value ::= 0 | 1 | ? | F | A | a | B | b | C | c | f
strength_component ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
* identifier_code ::= <{integer}
comment_text ::= {ASCII_character}
close_text ::= final_simulation_time
date_section ::= date_text
date_text :: = day month date time year
scope_section ::= scope_type scope_identifier
* scope_type ::= module
timescale_section ::= number time_unit
number ::= 1 | 10 | 100
time_unit ::= fs | ps | ns | us | ms | s
var_section ::= var_type size identifier_code reference
* var_type ::= port
* size ::= 1 | vector_index
vector_index ::= [ msb_index : lsb_index ]
index ::= decimal_number
* reference ::= port_identifier
identifier ::= {printable_ASCII_character}
version_section ::= version_text
* version_text ::= version_identifier {dumpports_command}
dumpports_command ::=
        $dumpports (scope_identifier , string_literal
      | variable
      | expression )
```

Syntax 18-27—Syntax for output extended VCD file

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The extended VCD file starts with header information giving the date, the version number of the simulator used for the simulation, and the timescale used. Next, the file contains definitions of the scope of the ports being dumped, followed by the actual value changes at each simulation time increment. Only the ports that change value during a time increment are listed.

The simulation time recorded in the extended VCD file is the absolute value of the simulation time for the changes in port values that follow.

Value changes for all ports are specified in binary format by `0`, `1`, `x`, or `z` values and include strength information.

A real number is dumped using a `%.16g printf()` format. This preserves the precision of that number by outputting all 53 bits in the mantissa of a 64-bit IEEE 754 double-precision number. Application programs can read a real number using a `%g` format to `scanf()`.

The extended VCD format does not support a mechanism to dump part of a vector. For example, bits 8 to 15 (`[8:15]`) of a 16-bit vector cannot be dumped in VCD file; instead, the entire vector (`[0:15]`) has to be dumped. In addition, expressions, such as a + b, cannot be dumped in the VCD file.

Data in the extended VCD file are case sensitive.

18.4.2 Extended VCD node information

The node information section (also referred to as the *variable definitions section*) is affected by the **$dumpports** task as Syntax 18-28 shows.

```
$var var_type size < identifier_code reference $end
var_type ::=
      port
size ::=
       1
      | vector_index
vector_index ::=
      [msb_index : lsb_index]
index ::=
      decimal_number
identifier_code ::=
      integer
reference ::=
      port_identifier
```

Syntax 18-28—Syntax for extended VCD node information

The constructs are defined as follows:

var_type	The keyword **port**. No other keyword is allowed.
size	A decimal number indicating the number of bits in the port. If the port is a single bit, the value shall be `1`. If the port is a bus, the actual index is printed. The *msb* indicates the most significant index; *lsb*, the least significant index.
identifier_code	An integer preceded by `<`, which starts at zero and ascends in one-unit increments for each port, in the order found in the module declaration.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

reference Identifier indicating the port name.

For example:

```
module test_device(count_out, carry, data, reset)
output count_out, carry ;
input [0:3] data;
input reset;
. . .
initial
    begin
        $dumpports(testbench.DUT, "testoutput.vcd");
. . .
end
```

This example produces the following node information in the VCD file:

```
$scope module testbench.DUT $end
$var port        1 <0          count_out    $end
$var port        1 <1              carry    $end
$var port     [0:3] <2               data   $end
$var port        1 <3              reset    $end
$upscope $end
```

At least one space shall separate each syntactical element. However, the formatting of the information is the choice of the simulator vendor. All four-state VCD syntax rules for the *vector_index* apply.

If the *vector_index* appears in the port declaration, this shall be the index dumped. If the *vector_index* is not in the port declaration, the *vector_index* in the net or reg declaration matching the port name shall be dumped. If no *vector_index* is found, the port is considered scalar (1 bit wide).

Concatenated ports shall appear in the extended VCD file as separate entries.

For example:

```
module addbit ({A, b}, ci, sum, co);
    input    A, b, ci;
    output   sum, co;
. . .
```

The VCD file output looks like the following:

```
$scope module addbit $end
$var port 1 <0 A $end
$var port 1 <1 b $end
$var port 1 <2 ci $end
$enddefinitions $end
. . .
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

18.4.3 Value changes

The value change section of the VCD file is also affected by **$dumpports**, as Syntax 18-29 shows.

```
pport_value  0_strength_component  1_strength_component  identifier_code
```

Syntax 18-29—Syntax for value change section

The constructs are defined as follows:

p	Key character that indicates a port. There is no space between the **p** and the *port_value*.
port_value	State character (described below).
0_strength_component	One of the eight Verilog strengths that indicates the *strength0* specification for the port.
1_strength_component	One of the eight Verilog strengths that indicates the *strength1* specification for the port.

The Verilog strength values are as follows (append keyword with `0` or `1` as appropriate for the strength component):

`0`	highz
`1`	small
`2`	medium
`3`	weak
`4`	large
`5`	pull
`6`	strong
`7`	supply
identifier_code	the integer preceded by the `<` character as defined in the `$var` construct for the port.

18.4.3.1 State characters

The following state information is listed in terms of input values from a test fixture, the output values of the device under test (DUT), and the states representing unknown direction:

INPUT (TESTFIXTURE):

`D`	low
`U`	high
`N`	unknown
`Z`	three-state
`d`	low (two or more drivers active)
`u`	high (two or more drivers active)

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

OUTPUT (DUT):

`L`	low
`H`	high
`X`	unknown (do-not care)
`T`	three-state
`l`	low (two or more drivers active)
`h`	high (two or more drivers active)

UNKNOWN DIRECTION:

`0`	low (both input and output are active with `0` value)
`1`	high (both input and output are active with `1` value)
`?`	unknown
`F`	three-state (input and output unconnected)
`A`	unknown (input `0` and output `1`)
`a`	unknown (input `0` and output `X`)
`B`	unknown (input `1` and output `0`)
`b`	unknown (input `1` and output `X`)
`C`	unknown (input `X` and output `0`)
`c`	unknown (input `X` and output `1`)
`f`	unknown (input and output three-stated)

18.4.3.2 Drivers

Drivers are considered only in terms of primitives, continuous assignments, and procedural continuous assignments. Value `0/1` means both input and output are active with value `0/1`. `0` and `1` are conflict states. The following rules apply to conflicts:

— If both input and output are driving the same value with the same range of strength, then this is a conflict. The resolved value is `0/1`, and the strength is the stronger of the two.
— If the input is driving a strong strength (range) and the output is driving a weak strength (range), the resolved value is d/u, and the strength is the strength of the input.
— If the input is driving a weak strength (range) and the output is driving a strong strength (range), then the resolved value is l/h, and the strength is the strength of the output.

Range is as follows:

— Strength supply `7` to `5` (large): strong strength
— Strength `4` to `1`: weak strength

18.4.4 Extended VCD file format example

The following example illustrates the format of the extended VCD file.

A module declaration:

```
module adder(data0, data1, data2, data3, carry, as, rdn, reset,
             test, write);
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
    inout data0, data1, data2, data3;
    output carry;
    input as, rdn, reset, test, write;
. . .
```

and the resulting VCD fragment:

```
$scope module testbench.adder_instance $end
$var port       1 <0         data0 $end
$var port       1 <1         data1 $end
$var port       1 <2         data2 $end
$var port       1 <3         data3 $end
$var port       1 <4         carry $end
$var port       1 <5            as $end
$var port       1 <6           rdn $end
$var port       1 <7         reset $end
$var port       1 <8          test $end
$var port       1 <9         write $end
$upscope $end
$enddefinitions $end

#0
$dumpports
pX  6  6    <0
pX  6  6    <1
pX  6  6    <2
pX  6  6    <3
pX  6  6    <4
pN  6  6    <5
pN  6  6    <6
pU  0  6    <7
pD  6  0    <8
pN  6  6    <9
$end
#180
pH  0  6    <4
#200000
pD  6  0    <5
pU  0  6    <6
pD  6  0    <9
#200500
pf  0  0    <0
pf  0  0    <1
pf  0  0    <2
pf  0  0    <3
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

19. Compiler directives

All Verilog compiler directives are preceded by the (`) character. This character is called *grave accent* (ASCII 0x60). It is different from the character (**'**), which is the *apostrophe* character (ASCII 0x27). The scope of a compiler directive extends from the point where it is processed, across all files processed, to the point where another compiler directive supersedes it or the processing completes.

This clause describes the following compiler directives:

`begin_keywords [19.11]
`celldefine [19.1]
`default_nettype [19.2]
`define [19.3]
`else [19.4]
`elsif [19.4]
`end_keywords [19.11]
`endcelldefine [19.1]
`endif [19.4]
`ifdef [19.4]
`ifndef [19.4]
`include [19.5]
`line [19.7]
`nounconnected_drive [19.9]
`pragma [19.10]
`resetall [19.6]
`timescale [19.8]
`unconnected_drive [19.9]
`undef [19.3]

19.1 `celldefine and `endcelldefine

The directives **`celldefine** and **`endcelldefine** tag modules as cell modules. Cells are used by certain PLI routines for applications, such as delay calculations. It is advisable to pair each **`celldefine** with an **`endcelldefine**, but it is not required. The latest occurrence of either directive in the source controls whether modules are tagged as cell modules. More than one of these pairs may appear in a single source description.

These directives may appear anywhere in the source description, but it is recommended that the directives be specified outside the module definition.

The **`resetall** directive includes the effects of a **`endcelldefine** directive.

19.2 `default_nettype

The directive **`default_nettype** controls the net type created for implicit net declarations (see 4.5). It can be used only outside of module definitions. Multiple **`default_nettype** directives are allowed. The latest occurrence of this directive in the source controls the type of nets that will be implicitly declared. Syntax 19-1 contains the syntax of the directive.

When no **`default_nettype** directive is present or if the **`resetall** directive is specified, implicit nets are of type **wire**. When the **`default_nettype** is set to **none**, all nets shall be explicitly declared. If a net is not explicitly declared, an error is generated.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
default_nettype_compiler_directive ::=
    `default_nettype default_nettype_value
default_nettype_value ::= wire | tri | tri0 | tri1 | wand | triand | wor | trior | trireg |
    uwire | none
```

Syntax 19-1—Syntax for default_nettype compiler directive

19.3 `define and `undef

A text macro substitution facility has been provided so that meaningful names can be used to represent commonly used pieces of text. For example, in the situation where a constant number is repetitively used throughout a description, a text macro would be useful in that only one place in the source description would need to be altered if the value of the constant needed to be changed.

The text macro facility is not affected by the compiler directive **`resetall**.

19.3.1 `define

The directive **`define** creates a macro for text substitution. This directive can be used both inside and outside module definitions. After a text macro is defined, it can be used in the source description by using the (`) character, followed by the macro name. The compiler shall substitute the text of the macro for the string `text_macro_name and any actual arguments that follow it. All compiler directives shall be considered predefined macro names; it shall be illegal to redefine a compiler directive as a macro name.

A text macro can be defined with arguments. This allows the macro to be customized for each use individually.

The syntax for text macro definitions is given in Syntax 19-2.

```
text_macro_definition ::=
    `define text_macro_name macro_text
text_macro_name ::=
    text_macro_identifier [ ( list_of_formal_arguments ) ]
list_of_formal_arguments ::=
    formal_argument_identifier { , formal_argument_identifier }
formal_argument_identifier ::=
    simple_identifier
text_macro_identifier ::= (From A.9.3)
    identifier
```

Syntax 19-2—Syntax for text macro definition

The macro text can be any arbitrary text specified on the same line as the text macro name. If more than one line is necessary to specify the text, the newline shall be preceded by a backslash (\). The first newline not preceded by a backslash shall end the macro text. The newline preceded by a backslash shall be replaced in the expanded macro with a newline (but without the preceding backslash character).

When formal arguments are used to define a text macro, the scope of the formal argument shall extend up to the end of the macro text. A formal argument can be used in the macro text in the same manner as an identifier.

--`,,```,,,Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

If formal arguments are used, the list of formal argument names shall be enclosed in parentheses following the name of the macro. The formal argument names shall be *simple_identifiers*, separated by commas and optionally whitespace. The left parenthesis shall follow the text macro name immediately, with no space in between.

If a one-line comment (that is, a comment specified with the characters //) is included in the text, then the comment shall not become part of the substituted text. The macro text can be blank, in which case the text macro is defined to be empty and no text is substituted when the macro is used.

The syntax for using a text macro is given in Syntax 19-3.

```
text_macro_usage ::=
      `text_macro_identifier [ ( list_of_actual_arguments ) ]
list_of_actual_arguments ::=
      actual_argument { , actual_argument }
actual_argument ::=
      expression
```

Syntax 19-3—Syntax for text macro usage

For a macro without arguments, the text shall be substituted as is for every occurrence of `text_macro_name. However, a text macro with one or more arguments shall be expanded by substituting each formal argument with the expression used as the actual argument in the macro usage.

To use a macro defined with arguments, the name of the text macro shall be followed by a list of actual arguments in parentheses, separated by commas. White space shall be allowed between the text macro name and the left parenthesis. The number of actual arguments shall match the number of formal arguments.

Once a text macro name has been defined, it can be used anywhere in a source description; that is, there are no scope restrictions. Text macros can be defined and used interactively.

The text specified for macro text shall not be split across the following lexical tokens:

— Comments
— Numbers
— Strings
— Identifiers
— Keywords
— Operators

For example:

```
`define wordsize 8
reg [1:`wordsize] data;

//define a nand with variable delay
`define var_nand(dly) nand #dly

`var_nand(2) g121 (q21, n10, n11);
`var_nand(5) g122 (q22, n10, n11);
```

The following is illegal syntax because it is split across a string:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
`define first_half "start of string
$display(`first_half end of string");
```

Each actual argument is substituted for the corresponding formal argument literally. Therefore, when an expression is used as an actual argument, the expression will be substituted in its entirety. This may cause an expression to be evaluated more than once if the formal argument was used more than once in the macro text. For example:

```
`define max(a,b)((a) > (b) ? (a) : (b))
n = `max(p+q, r+s) ;
```

will expand as

```
n = ((p+q) > (r+s)) ? (p+q) : (r+s) ;
```

Here, the larger of the two expressions p + q and r + s will be evaluated twice.

The word define is known as a compiler directive keyword, and it is not part of the normal set of keywords. Thus, normal identifiers in a Verilog HDL source description can be the same as compiler directive keywords (although this is not recommended). The following problems should be considered:

a) Text macro names may not be the same as compiler directive keywords.
b) Text macro names can reuse names being used as ordinary identifiers. For example, `signal_name` and `` `signal_name `` are different.
c) Redefinition of text macros is allowed; the latest definition of a particular text macro read by the compiler prevails when the macro name is encountered in the source text.

The macro text can contain usages of other text macros. Such usages shall be substituted after the original macro is substituted, not when it is defined. It shall be an error for a macro to expand directly or indirectly to text containing another usage of itself (a recursive macro).

19.3.2 `undef

The directive **`undef** shall undefine a previously defined text macro. An attempt to undefine a text macro that was not previously defined using a **`define** compiler directive can result in a warning. The syntax for **`undef** compiler directive is given in Syntax 19-4.

```
undefine_compiler_directive ::=
    `undef text_macro_identifier
```

Syntax 19-4—Syntax for undef compiler directive

An undefined text macro has no value, just as if it had never been defined.

19.4 `ifdef, `else, `elsif, `endif, `ifndef

These conditional compilation compiler directives are used to include optionally lines of a Verilog HDL source description during compilation. The **`ifdef** compiler directive checks for the definition of a `text_macro_name`. If the `text_macro_name` is defined, then the lines following the **`ifdef** directive are included. If the `text_macro_name` is not defined and an **`else** directive exists, then this source is compiled. The **`ifndef** compiler directive checks for the definition of a `text_macro_name`. If the `text_macro_name`

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

is not defined, then the lines following the **`ifndef** directive are included. If the `text_macro_name` is defined and an **`else** directive exists, then this source is compiled.

If the **`elsif** directive exists (instead of the **`else**), the compiler checks for the definition of the `text_macro_name`. If the name exists, the lines following the **`elsif** directive are included. The **`elsif** directive is equivalent to the compiler directive sequence **`else `ifdef** ... **`endif**. This directive does not need a corresponding **`endif** directive. This directive shall be preceded by an **`ifdef** or **`ifndef** directive.

These directives may appear anywhere in the source description.

Situations where the **`ifdef**, **`else**, **`elsif**, **`endif**, and **`ifndef** compiler directives may be useful include the following:

— Selecting different representations of a module such as behavioral, structural, or switch level
— Choosing different timing or structural information
— Selecting different stimulus for a given run

The **`ifdef**, **`else**, **`elsif**, **`endif**, and **`ifndef** compiler directives have the syntax shown in Syntax 19-5.

```
conditional_compilation_directive ::=
        ifdef_directive
      | ifndef_directive
ifdef_directive ::=
      `ifdef text_macro_identifier
      ifdef_group_of_lines
      { `elsif text_macro_identifier elsif_group_of_lines }
      [ `else else_group_of_lines ]
      `endif
ifndef_directive ::=
      `ifndef text_macro_identifier
      ifndef_group_of_lines
      { `elsif text_macro_identifier elsif_group_of_lines }
      [ `else else_group_of_lines ]
      `endif
```

Syntax 19-5—Syntax for conditional compilation directives

The `text_macro_identifier` is a Verilog HDL *identifier*. The `ifdef_group_of_lines`, `ifndef_group_of_lines`, `elsif_group_of_lines`, and the `else_group_of_lines` are parts of a Verilog HDL source description. The **`else** and **`elsif** compiler directives and all of the groups of lines are optional.

The **`ifdef**, **`else**, **`elsif**, and **`endif** compiler directives work together in the following manner:

— When an **`ifdef** is encountered, the **`ifdef** text macro identifier is tested to see whether it is defined as a text macro name using **`define** within the Verilog HDL source description.
— If the **`ifdef** text macro identifier is defined, the **`ifdef** group of lines is compiled as part of the description; and if there are **`else** or **`elsif** compiler directives, these compiler directives and corresponding groups of lines are ignored.
— If the **`ifdef** text macro identifier has not been defined, the **`ifdef** group of lines is ignored.
— If there is an **`elsif** compiler directive, the **`elsif** text macro identifier is tested to see whether it is defined as a text macro name using **`define** within the Verilog HDL source description.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

— If the **`elsif** text macro identifier is defined, the **`elsif** group of lines is compiled as part of the description; and if there are other **`elsif** or **`else** compiler directives, the other **`elsif** or **`else** directives and corresponding groups of lines are ignored.
— If the first **`elsif** text macro identifier has not been defined, the first **`elsif** group of lines is ignored.
— If there are multiple **`elsif** compiler directives, they are evaluated like the first **`elsif** compiler directive in the order they are written in the Verilog HDL source description.
— If there is an **`else** compiler directive, the **`else** group of lines is compiled as part of the description.

The **`ifndef**, **`else**, **`elsif**, and **`endif** compiler directives work together in the following manner:

— When an **`ifndef** is encountered, the **`ifndef** text macro identifier is tested to see whether it is defined as a text macro name using **`define** within the Verilog HDL source description.
— If the **`ifndef** text macro identifier is not defined, the **`ifndef** group of lines is compiled as part of the description; and if there are **`else** or **`elsif** compiler directives, these compiler directives and corresponding groups of lines are ignored.
— If the **`ifndef** text macro identifier is defined, the **`ifndef** group of lines is ignored.
— If there is an **`elsif** compiler directive, the **`elsif** text macro identifier is tested to see whether it is defined as a text macro name using **`define** within the Verilog HDL source description.
— If the **`elsif** text macro identifier is defined, the **`elsif** group of lines is compiled as part of the description; and if there are other **`elsif** or **`else** compiler directives, the other **`elsif** or **`else** directives and corresponding groups of lines are ignored.
— If the first **`elsif** text macro identifier has not been defined, the first **`elsif** group of lines is ignored.
— If there are multiple **`elsif** compiler directives, they are evaluated like the first **`elsif** compiler directive in the order they are written in the Verilog HDL source description.
— If there is an **`else** compiler directive, the **`else** group of lines is compiled as part of the description.

Although the names of compiler directives are contained in the same name space as text macro names, the names of compiler directives are considered not to be defined by **`ifdef**, **`ifndef**, and **`elseif**.

Nesting of **`ifdef**, **`ifndef**, **`else**, **`elsif**, and **`endif** compiler directives shall be permitted.

Any group of lines that the compiler ignores shall still follow the Verilog HDL lexical conventions for white space, comments, numbers, strings, identifiers, keywords, and operators.

For example:

Example 1—The example below shows a simple usage of an **`ifdef** directive for conditional compilation. If the identifier `behavioral` is defined, a continuous net assignment will be compiled in; otherwise, an **and** gate will be instantiated.

```
module and_op (a, b, c);
output a;
input b, c;

`ifdef behavioral
    wire a = b & c;
`else
    and a1 (a,b,c);
`endif

endmodule
```


--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Example 2—The following example shows usage of nested conditional compilation directive:

```
module test(out);
output out;
`define wow
`define nest_one
`define second_nest
`define nest_two
    `ifdef wow
        initial $display("wow is defined");
        `ifdef nest_one
            initial $display("nest_one is defined");
            `ifdef nest_two
                initial $display("nest_two is defined");
            `else
                initial $display("nest_two is not defined");
            `endif
        `else
            initial $display("nest_one is not defined");
        `endif
    `else
        initial $display("wow is not defined");
        `ifdef second_nest
            initial $display("second_nest is defined");
        `else
            initial $display("second_nest is not defined");
        `endif
    `endif
endmodule
```

Example 3—The following example shows usage of chained nested conditional compilation directives:

```
module test;
    `ifdef first_block
        `ifndef second_nest
            initial $display("first_block is defined");
        `else
            initial $display("first_block and second_nest defined");
        `endif
    `elsif second_block
        initial $display("second_block defined, first_block is not");
    `else
        `ifndef last_result
            initial $display("first_block, second_block,"
                " last_result not defined.");
        `elsif real_last
            initial $display("first_block, second_block not defined,"
                " last_result and real_last defined.");
        `else
            initial $display("Only last_result defined!");
        `endif
    `endif
endmodule
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

19.5 `include

The file inclusion (**`include**) compiler directive is used to insert the entire contents of a source file in another file during compilation. The result is as though the contents of the included source file appear in place of the **`include** compiler directive. The **`include** compiler directive can be used to include global or commonly used definitions and tasks without encapsulating repeated code within module boundaries.

Advantages of using the **`include** compiler directive include the following:

— Providing an integral part of configuration management
— Improving the organization of Verilog HDL source descriptions
— Facilitating the maintenance of Verilog HDL source descriptions

The syntax for the **`include** compiler directive is given in Syntax 19-6.

```
include_compiler_directive ::=
    `include "filename"
```

Syntax 19-6—Syntax for include compiler directive

The compiler directive **`include** can be specified anywhere within the Verilog HDL description. The *filename* is the name of the file to be included in the source file. The *filename* can be a full or relative path name.

Only white space or a comment may appear on the same line as the **`include** compiler directive.

A file included in the source using the **`include** compiler directive may contain other **`include** compiler directives. The number of nesting levels for include files shall be finite.

For example:

Examples of **`include** compiler directives are as follows:

```
`include "parts/count.v"
`include "fileB"
`include "fileB" // including fileB
```

Implementations may limit the maximum number of levels to which include files can be nested, but the limit shall be at least 15.

19.6 `resetall

When **`resetall** compiler directive is encountered during compilation, all compiler directives are set to the default values. This is useful for ensuring that only directives that are desired in compiling a particular source file are active.

The recommended usage is to place **`resetall** at the beginning of each source text file, followed immediately by the directives desired in the file.

It shall be illegal for the **`resetall** directive to be specified within a module or UDP declaration.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

19.7 `line

It is important for Verilog tools to keep track of the filenames of the Verilog source files and the line numbers in the files. This information can be used for error messages or source code debugging and can be accessed by the Verilog PLI.

In many cases, however, the Verilog source is preprocessed by some other tool, and the line and file information of the original source file can be lost because the preprocessor might add additional lines to the source code file, combine multiple source code lines into one line, concatenate multiple source files, and so on.

The **`line** compiler directive can be used to specify the original source code line number and filename. This allows the location in the original file to be maintained if another process modifies the source. After the newline number and filename are specified, the compiler can correctly refer to the original source location. However, a tool is not required to produce **`line** directives. These directives are not intended to be inserted manually into the code, although they can be.

The compiler shall maintain the current line number and filename of the file being compiled. The **`line** directive shall set the line number and filename of the following line to those specified in the directive. The directive can be specified anywhere within the Verilog HDL source description. However, only white space may appear on the same line as the **`line** directive. Comments are not allowed on the same line as a **`line** directive. All parameters in the **`line** directive are required. The results of this directive are not affected by the **`resetall** directive.

The syntax for the **`line** compiler directive is given in Syntax 19-7.

```
line_compiler_directive ::=
    `line number "filename" level
```

Syntax 19-7—Syntax for line compiler directive

The *number* parameter shall be a positive integer that specifies the newline number of the following text line. The *filename* parameter shall be a string constant that is treated as the new name of the file. The filename can also be a full or relative path name. The *level* parameter shall be 0, 1, or 2. The value 1 indicates that the following line is the first line after an include file has been entered. The value 2 indicates that the following line is the first line after an include file has been exited. The value 0 indicates any other line.

For example:

```
`line 3 "orig.v" 2
// This line is line 3 of orig.v after exiting include file
```

As the compiler processes the remainder of the file and new files, the line number shall be incremented as each line is read, and the name shall be updated to the new current file being processed. The line number shall be reset to 1 at the beginning of each file. When beginning to read include files, the current line and filename shall be stored for restoration at the termination of the include file. The updated line number and filename information shall be available for PLI access. The mechanism of library searching is not affected by the effects of the **`line** compiler directive.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

19.8 `timescale

This directive specifies the time unit and time precision of the modules that follow it. The time unit is the unit of measurement for time values such as the simulation time and delay values.

To use modules with different time units in the same design, the following timescale constructs are useful:

— The **`timescale** compiler directive to specify the unit of measurement for time and precision of time in the modules in the design
— The **$printtimescale** system task to display the time unit and precision of a module
— The **$time** and **$realtime** system functions, the **$timeformat** system task, and the `%t` format specification to specify how time information is reported

The **`timescale** compiler directive specifies the unit of measurement for time and delay values and the degree of accuracy for delays in all modules that follow this directive until another **`timescale** compiler directive is read. If there is no **`timescale** specified or it has been reset by a **`resetall** directive, the time unit and precision are simulator-specific. It shall be an error if some modules have a **`timescale** specified and others do not.

The syntax for the **`timescale** directive is given in Syntax 19-8.

```
timescale_compiler_directive ::=
    `timescale time_unit / time_precision
```

Syntax 19-8—Syntax for timescale compiler directive

The `time_unit` argument specifies the unit of measurement for times and delays.

The `time_precision` argument specifies how delay values are rounded before being used in simulation. The values used are accurate to within the unit of time specified here, even if there is a smaller `time_precision` argument elsewhere in the design. The smallest `time_precision` argument of all the **`timescale** compiler directives in the design determines the precision of the time unit of the simulation.

The `time_precision` argument shall be at least as precise as the `time_unit` argument; it cannot specify a longer unit of time than `time_unit`.

The integers in these arguments specify an order of magnitude for the size of the value; the valid integers are 1, 10, and 100. The character strings represent units of measurement; the valid character strings are **s**, **ms**, **us**, **ns**, **ps**, and **fs**.

The units of measurement specified by these character strings are given in Table 19-1.

NOTE—While s, ms, ns, ps and fs are the usual SI unit symbols for second, millisecond, nanosecond, picosecond and femtosecond, due to lack of the Greek letter μ (mu) in coding character sets, “us” represents the SI unit symbol for microsecond, properly ms.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 19-1—Arguments of time_precision

Character string	Unit of measurement
s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds
ps	picoseconds
fs	femtoseconds

For example:

The following example shows how this directive is used**:**

```
`timescale 1 ns / 1 ps
```

Here, all time values in the modules that follow the directive are multiples of 1 ns because the `time_unit` argument is "1 ns." Delays are rounded to real numbers with three decimal places—or precise to within one thousandth of a nanosecond—because the `time_precision` argument is "1 ps," or one thousandth of a nanosecond.

Consider the following example:

```
`timescale 10 us / 100 ns
```

The time values in the modules that follow this directive are multiples of `10 us` because the `time_unit` argument is "10 us." Delays are rounded to within one tenth of a microsecond because the `time_precision` argument is "100 ns," or one tenth of a microsecond.

The following example shows a **`timescale** directive in the context of a module:

```
`timescale 10 ns / 1 ns
module test;
reg set;
parameter d = 1.55;

initial begin
    #d set = 0;
    #d set = 1;
end
endmodule
```

The **`timescale** `10 ns / 1 ns` compiler directive specifies that the time unit for module test is 10 ns. As a result, the time values in the module are multiples of 10 ns, rounded to the nearest 1 ns; therefore, the value stored in parameter `d` is scaled to a delay of 16 ns. In other words, the value 0 is assigned to reg `set` at simulation time 16 ns (1.6 × 10 ns), and the value 1 at simulation time 32 ns.

Parameter `d` retains its value no matter what timescale is in effect.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

These simulation times are determined as follows:

a) The value of parameter `d` is rounded from 1.55 to 1.6 according to the time precision.
b) The time unit of the module is 10 ns, and the precision is 1 ns; therefore, the delay of parameter `d` is scaled from 1.6 to 16.
c) The assignment of 0 to reg `set` is scheduled at simulation time 16 ns, and the assignment of 1 at simulation time 32 ns. The time values are not rounded when the assignments are scheduled.

19.9 `unconnected_drive and `nounconnected_drive

All unconnected input ports of a module appearing between the directives **`unconnected_drive** and **`nounconnected_drive** are pulled up or pulled down instead of the normal default.

The directive **`unconnected_drive** takes one of two arguments—**pull1** or **pull0**. When **pull1** is specified, all unconnected input ports are automatically pulled up. When **pull0** is specified, unconnected ports are pulled down. It is advisable to pair each **`unconnected_drive** with a **`nounconnected_drive**, but it is not required. The latest occurrence of either directive in the source controls what happens to unconnected ports. These directives shall be specified in pairs outside of the module declarations.

The **`resetall** directive includes the effects of a **`nounconnected_drive** directive.

19.10 `pragma

The **`pragma** directive is a structured specification that alters interpretation of the Verilog source. The specification introduced by this directive is referred to as a *pragma*. The effect of pragmas other than those specified in this standard is implementation-specified. The syntax for the **`pragma** directive is given in Syntax 19-9.

```
pragma ::=
      `pragma pragma_name [ pragma_expression { , pragma_expression } ]
pragma_name ::= simple_identifier
pragma_expression ::=
        pragma_keyword
      | pragma_keyword = pragma_value
      | pragma_value
pragma_value ::=
        ( pragma_expression { , pragma_expression } )
      | number
      | string
      | identifier
pragma_keyword ::= simple_identifier
```

Syntax 19-9—Syntax for pragma compiler directive

The pragma specification is identified by the *pragma_name*, which follows the **`pragma** directive. The *pragma_name* is followed by an optional list of *pragma_expressions*, which qualify the altered interpretation indicated by the *pragma_name*. Unless otherwise specified, pragma directives for *pragma_names* that are not recognized by an implementation shall have no effect on interpretation of the Verilog source text.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

19.10.1 Standard pragmas

The **reset** and **resetall** pragmas shall restore the default values and state of *pragma_keywords* associated with the affected pragmas. These default values shall be the values that the tool defines before any Verilog text has been processed. The **reset** pragma shall reset the state for all *pragma_names* that appear as *pragma_keywords* in the directive. The **resetall** pragma shall reset the state of all *pragma_names* recognized by the implementation.

19.11 `begin_keywords, `end_keywords

A pair of directives, **`begin_keywords** and **`end_keywords**, can be used to specify what identifiers are reserved as keywords within a block of source code, based on a specific version of IEEE Std 1364. The **`begin_keywords** and **`end_keywords** directives only specify the set of identifiers that are reserved as keywords. The directives do not affect the semantics, tokens, and other aspects of the Verilog language.

The syntax of the **`begin_keywords** and **`end_keywords** directives is in Syntax 19-10.

```
keywords_directive ::= `begin_keywords "version_specifier"
version_specifier ::=
        1364-1995
      | 1364-2001
      | 1364-2001-noconfig
      | 1364-2005
endkeywords_directive ::= `end_keywords
```

Syntax 19-10—Syntax for begin keywords and end keywords compiler directives

Implementations and other standards are permitted to extend the **`begin_keywords** directive with custom version specifiers. It shall be an error if an implementation does not recognize the *version_specifier* used with the **`begin_keywords** directive.

The **`begin_keywords** and **`end_keywords** directives can only be specified outside of a design element (`module`, `primitive`, or `configuration`). The **`begin_keywords** directive affects all source code that follow the directive, even across source code file boundaries, until the matching **`end_keywords** directive is encountered.

Each **`begin_keywords** directive must be paired with an **`end_keywords** directive. The pair of directives define a region of source code to which a specified *version_specifier* applies.

The **`begin_keywords...`end_keywords** directive pair can be nested. Each nested pair is stacked so that when an **`end_keywords** directive is encountered, the implementation returns to using the *version_ specifier* that was in effect prior to the matching **`begin_keywords** directive.

If no **`begin_keywords** directive is specified, then the reserved keyword list shall be the implementation's default set of keywords. The default set of reserved keywords used by an implementation shall be implementation dependent. For example, an implementation based on IEEE Std 1364-2005 would most likely use the 1364-2005 set of reserved keywords as its default, whereas an implementation based on IEEE Std 1364-2001 would most likely use the 1364-2001 set of reserved keywords as its default. Implementations may provide other mechanisms for specifying the set of reserved keywords to be used as the default. One possible use model might be for an implementation to use invocation options to specify its default set of reserved keywords. Another possible use model might be the use of source file name extensions for determining a default set of reserved keywords to be used for each source file.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The *version_specifier* `"1364-1995"` specifies that only the identifiers listed as reserved keywords in IEEE Std 1364-1995 are considered to be reserved words. These identifiers are listed in Table 19-2.

Table 19-2—IEEE 1364-1995 reserved keywords

always	for	output	supply0
and	force	parameter	supply1
assign	forever	pmos	table
begin	fork	posedge	task
buf	function	primitive	time
bufif0	highz0	pull0	tran
bufif1	highz1	pull1	tranif0
case	if	pullup	tranif1
casex	ifnone	pulldown	tri
casez	initial	rcmos	tri0
cmos	inout	real	tri1
deassign	input	realtime	triand
default	integer	reg	trior
defparam	join	release	trireg
disable	large	repeat	vectored
edge	macromodule	rnmos	wait
else	medium	rpmos	wand
end	module	rtran	weak0
endcase	nand	rtranif0	weak1
endmodule	negedge	rtranif1	while
endfunction	nmos	scalared	wire
endprimitive	nor	small	wor
endspecify	not	specify	xnor
endtable	notif0	specparam	xor
endtask	notif1	strong0	
event	or	strong1	

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The *version_specifier* `"1364-2001"` specifies that only the identifiers listed as reserved keywords in IEEE Std 1364-2001 are considered to be reserved words. These identifiers are listed in Table 19-3.

Table 19-3—IEEE 1364-2001 reserved keywords

always	event	noshowcancelled	specify
and	for	not	specparam
assign	force	notif0	strong0
automatic	forever	notif1	strong1
begin	fork	or	supply0
buf	function	output	supply1
bufif0	generate	parameter	table
bufif1	genvar	pmos	task
case	highz0	posedge	time
casex	highz1	primitive	tran
casez	if	pull0	tranif0
cell	ifnone	pull1	tranif1
cmos	incdir	pulldown	tri
config	include	pullup	tri0
deassign	initial	pulsestyle_onevent	tri1
default	inout	pulsestyle_ondetect	triand
defparam	input	rcmos	trior
design	instance	real	trireg
disable	integer	realtime	unsigned
edge	join	reg	use
else	large	release	vectored
end	liblist	repeat	wait
endcase	library	rnmos	wand
endconfig	localparam	rpmos	weak0
endfunction	macromodule	rtran	weak1
endgenerate	medium	rtranif0	while
endmodule	module	rtranif1	wire
endprimitive	nand	scalared	wor
endspecify	negedge	showcancelled	xnor
endtable	nmos	signed	xor
endtask	nor	small	

The *version_specifier* `"1364-2001-noconfig"` behaves similarly to the `"1364-2001"` version_specifier, with the exception that the following identifiers are excluded from the reserved list in Table 19-3:

```
cell
config
design
endconfig
incdir
include
instance
liblist
library
use
```

Because these identifiers are not reserved when using the `"1364-2001-noconfig"` version_specifier, they may be used as normal Verilog identifiers within the corresponding **`begin_keywords...`end_ keywords** region.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The *version_specifier* `"1364-2005"` specifies that only the identifiers listed as reserved keywords in IEEE Std 1364-2005 are considered to be reserved words. These identifiers are listed in Table 19-4.

Table 19-4—IEEE 1364-2005 reserved keywords

always	event	noshowcancelled	specify
and	for	not	specparam
assign	force	notif0	strong0
automatic	forever	notif1	strong1
begin	fork	or	supply0
buf	function	output	supply1
bufif0	generate	parameter	table
bufif1	genvar	pmos	task
case	highz0	posedge	time
casex	highz1	primitive	tran
casez	if	pull0	tranif0
cell	ifnone	pull1	tranif1
cmos	incdir	pulldown	tri
config	include	pullup	tri0
deassign	initial	pulsestyle_onevent	tri1
default	inout	pulsestyle_ondetect	triand
defparam	input	rcmos	trior
design	instance	real	trireg
disable	integer	realtime	unsigned
edge	join	reg	use
else	large	release	uwire
end	liblist	repeat	vectored
endcase	library	rnmos	wait
endconfig	localparam	rpmos	wand
endfunction	macromodule	rtran	weak0
endgenerate	medium	rtranif0	weak1
endmodule	module	rtranif1	while
endprimitive	nand	scalared	wire
endspecify	negedge	showcancelled	wor
endtable	nmos	signed	xnor
endtask	nor	small	xor

In the example below, it is assumed that the definition of module `m1` does not have a **`begin_keywords** directive specified prior to the module definition. Without this directive, the set of reserved keywords in effect for this module shall be the implementation's default set of reserved keywords.

```
module m1;  // module definition with no `begin_keywords directive
  ...
endmodule
```

The following example specifies a **`begin_keywords** `"1364-2001"` directive. The source code within the module uses the identifier **uwire** as a net name. The **`begin_keywords** directive would be necessary in this example if an implementation uses IEEE Std 1364-2005 as its default set of keywords because **uwire** is a reserved keyword in this standard. Specifying that the `"1364-1995"` Verilog keyword lists should be used would also work with this example.

```
`begin_keywords "1364-2001"   // use IEEE Std 1364-2001 Verilog keywords
module m2 (...);
  wire [63:0] uwire;          // OK: "uwire" is not a keyword in 1364-2001
  ...
endmodule
`end_keywords
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The next example is the same code as the previous example, except that it explicitly specifies that the IEEE Std 1364-2005 Verilog keywords should be used. This example shall result in an error because **uwire** is reserved as a keyword in this standard.

```
‘begin_keywords "1364-2005"   // use IEEE Std 1364-2005 Verilog keywords
module m2 (...);
  wire [63:0] uwire;          // ERROR: "uwire" is a keyword in 1364-2005
  ...
endmodule
‘end_keywords
```

Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

20. Programming language interface (PLI) overview

20.1 PLI purpose and history

IEEE Std 1364-2005 has deprecated the task/function (TF) and access (ACC) routines, which were specified previously in Clause 21 through Clause 25, Annex E, and Annex F of IEEE Std 1364-2001. Clause 20 has been modified to reflect this change. The text of deprecated clauses and annexes has been removed from this version of the standard, but the clause headings have been retained. See the corresponding clauses in IEEE Std 1364-2001 for the deprecated text.

Clause 26, Clause 27, and Annex G describe the C language procedural interface standard and interface mechanisms that are part of the Verilog HDL. This procedural interface, known as the PLI, provides a means for Verilog HDL users to access and modify data in an instantiated Verilog HDL data structure dynamically. An instantiated Verilog HDL data structure is the result of compiling Verilog HDL source descriptions and generating the hierarchy modeled by module instances, primitive instances, and other Verilog HDL constructs that represent scope. The PLI procedural interface provides a library of C language functions that can directly access data within an instantiated Verilog HDL data structure.

A few of the many possible applications for the PLI procedural interface are as follows:

— C language delay calculators for Verilog model libraries that can dynamically scan the data structure of a Verilog software product and then dynamically modify the delays of each instance of models from the library
— C language applications that dynamically read test vectors or other data from a file and pass the data into a Verilog software product
— Custom graphical waveform and debugging environments for Verilog software products
— Source code decompilers that can generate Verilog HDL source code from the compiled data structure of a Verilog software product
— Simulation models written in the C language and dynamically linked into Verilog HDL simulations
— Interfaces to actual hardware, such as a hardware modeler, that dynamically interact with simulations

There are three primary generations of the Verilog PLI:

a) *Task/function* routines, called *TF* routines, made up the first generation of the PLI. These routines, most of which started with the characters **tf_**, were primarily used for operations involving user-defined system task/function arguments, along with utility functions, such as setting up call-back mechanisms and writing data to output devices. The TF routines were sometimes referred to as *utility* routines

   NOTE—The TF routines have been deprecated from this version of the standard (see 1.6).

b) *Access* routines, called *ACC* routines, formed the second generation of the PLI. These routines, which all started with the characters **acc_**, provided an object-oriented access directly into a Verilog HDL structural description. ACC routines were used to access and modify information, such as delay values and logic values, on a wide variety of objects that exist in a Verilog HDL description. There was some overlap in functionality between ACC routines and TF routines.

   NOTE—The ACC routines have been deprecated from this version of the standard (see 1.6).

c) *Verilog procedural interface* routines, called *VPI* routines, are the third generation of the PLI. These routines, all of which start with the characters **vpi_**, provide an object-oriented access for both Verilog HDL structural and behavioral objects. The VPI routines are a superset of the functionality of the TF routines and ACC routines.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

20.2 User-defined system task/function names

A user-defined system task/function name is the name that will be used within a Verilog HDL source file to invoke specific PLI applications. The name shall adhere to the following rules:

— The first character of the name shall be the dollar sign ($).
— The remaining characters shall be letters, digits, the underscore character (_), or the dollar sign ($).
— Uppercase and lowercase letters shall be considered to be unique—the name is case sensitive.
— The name can be any size, and all characters are significant.

20.3 User-defined system task/function types

The type of a user-defined system task/function determines how a PLI application is called from the Verilog HDL source code. The types are as follows:

— A user *task* can be used in the same places a Verilog HDL task can be used (see 10.2). A user-defined system task can read and modify the arguments of the task, but does not return any value.
— A user *function* can be used in the same places a Verilog HDL function can be used (see 10.4). A user-defined system function can read and modify the arguments of the function, and it returns a value. The bit width of a vector shall be determined by a user-supplied *sizetf* application (see 27.34).

20.4 Overriding built-in system task/function names

Clause 17 defines a number of built-in system tasks and functions that are part of the Verilog language. In addition, software products can include other built-in system tasks and functions specific to the product. These built-in system task/function names begin with the dollar sign ($) just as user-defined system task/function names do.

If a user-provided PLI application is associated with the same name as a built-in system task/function (using the PLI mechanism), the user-provided C application shall override the built-in system task/function, replacing its functionality with that of the user-provided C application. For example, a user could write a random number generator as a PLI application and then associate the application with the name **$random**, thereby overriding the built-in **$random** function with the user's application.

Verilog timing checks, such as **$setup**, are not system tasks and cannot be overridden.

The system functions **$signed** and **$unsigned** can be overridden. These system functions are unique in the Verilog HDL in that the return width is based on the width of their argument. If overridden, the PLI version shall have the same return width for all instances of the system function. The PLI return width is defined by the PLI *sizetf* routine.

20.5 User-supplied PLI applications

User-supplied PLI applications are C language functions that utilize the library of PLI C functions to access and interact dynamically with Verilog HDL software implementations as the Verilog HDL source code is executed.

These PLI applications are not independent C programs. They are C functions that are linked into a software product and become part of the product. This allows the PLI application to be called when the user-defined system task/function **$** name is compiled or executed in the Verilog HDL source code (see 26.1).

Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

20.6 PLI mechanism

The PLI mechanism provides a means to have PLI applications called for various reasons when the associated system task/function **$** name is encountered in the Verilog HDL source description. For example, when a Verilog HDL simulator first compiles the Verilog HDL source description, a specific PLI routine can be called that performs syntax checking to ensure the user-defined system task/function is being used correctly. Then, as simulation is executing, a different PLI routine can be called to perform the operations required by the PLI application. Other PLI routines can be automatically called by the simulator for miscellaneous reasons, such as the end of a simulation time step or a logic value change on a specific signal (see 26.1).

20.7 User-defined system task/function arguments

When a user-defined system task/function is used in a Verilog HDL source file, it can have arguments that can be used by the PLI applications associated with the system task/function. In the following example, the user-defined system task `$get_vector` has two arguments:

```
$get_vector("test_vector.pat", input_bus);
```

The arguments to a system task/function are referred to as *task/function arguments* (often abbreviated as *tfargs*). These arguments are not the same as C language arguments. When the PLI applications associated with a user-defined system task/function are called, the task/function arguments are not passed to the PLI application. Instead, a number of PLI routines are provided that allow the PLI applications to read and write to the task/function arguments. See Clause 27 for information on specific routines that work with task/function arguments.

20.8 PLI include files

The libraries of PLI functions are defined in a C include file, which is a normative part of this standard. This file also defines constants, structures, and other data used by the library of PLI routines and the interface mechanisms. The file is `vpi_user.h` (listed in Annex G). PLI applications that use the VPI routines shall include the file `vpi_user.h`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

21. PLI TF and ACC interface mechanism (deprecated)

This clause has been deprecated (see 1.6).

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

22. Using ACC routines (deprecated)

This clause has been deprecated (see 1.6).

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

23. ACC routine definitions (deprecated)

This clause has been deprecated (see 1.6).

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

24. Using TF routines (deprecated)

This clause has been deprecated (see 1.6).

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

25. TF routine definitions (deprecated)

This clause has been deprecated (see 1.6).

Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26. Using Verilog procedural interface (VPI) routines

Clause 26 and Clause 27 specify the VPI for the Verilog HDL. This clause describes how the VPI routines are used, and Clause 27 defines each of the routines in alphabetical order.

26.1 VPI system tasks and functions

User-defined system tasks and functions are created using the routine **vpi_register_systf()** (see 27.34). The registration of system tasks must occur prior to elaboration or the resolution of references.

The intended use model would be to place a reference to a routine within the **vlog_startup_routines[]** array. This routine would register all user-defined system tasks and functions when it is called.

Through VPI, an application can perform the following:

— Specify a user-defined system task/function name that can be included in Verilog HDL source descriptions; the user-defined system task/function name shall begin with a dollar sign ($), such as `$get_vector`.
— Provide one or more PLI C applications to be called by a software product (such as a logic simulator).
— Define which PLI C applications are to be called—and when the applications should be called—when the user-defined system task/function name is encountered in the Verilog HDL source description.
— Define whether the PLI applications should be treated as *function*s (which return a value) or *task*s (analogous to subroutines in other programming languages).
— Define a data argument to be passed to the PLI applications each time they are called.

It is also possible through the callback mechanisms in VPI to create applications that are not directly tied to a user-defined system task/function.

VPI-based system tasks have *sizetf*, *compiletf*, and *calltf* routines, which perform specific actions for the task/function. The *sizetf*, *compiletf*, and *calltf* routines are called during specific periods during processing. The purpose of each of these routines is explained in 26.1.1 through 26.1.4.

26.1.1 sizetf VPI application routine

A *sizetf* VPI application routine can be used in conjunction with user-defined system functions. A function shall return a value, and software products that execute the system function need to determine how many bits wide that return value shall be. When *sizetf* shall be called is described in 26.2.4 and 27.34.1. Each *sizetf* routine shall be called at most once. It shall be called if its associated system function appears in the design. The value returned by the *sizetf* routine shall be the number of bits that the *calltf* routine shall provide as the return value for the system function. If no *sizetf* routine is specified, a user-defined system function shall return 32 bits. The *sizetf* routine shall not be called for user-defined system tasks or for functions whose *sysfunctype* is set to **vpiRealFunc**.

26.1.2 compiletf VPI application routine

A *compiletf* VPI application routine shall be called when the user-defined system task/function name is encountered during parsing or compiling the Verilog HDL source code. This routine is typically used to check the correctness of any arguments passed to the user-defined system task/function in the Verilog HDL source code. The *compiletf* routine shall be called one time for each instance of a system task/function in the source description. Providing a *compiletf* routine is optional, but it is recommended that any arguments used

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

with the system task/function be checked for correctness to avoid problems when the *calltf* or other PLI routines read and perform operations on the arguments. When the *compiletf* is called is described in 26.2.4 and 27.34.1.

26.1.3 calltf VPI application routine

A *calltf* VPI application routine shall be called each time the associated user-defined system task/function is executed within the Verilog HDL source code. For example, the following Verilog loop would call the *calltf* routine that is associated with the `$get_vector` user-defined system task name 1024 times:

```
for (i = 1; i <= 1024; i = i + 1)
   @(posedge clk) $get_vector("test_vector.pat", input_bus);
```

In this example, the *calltf* might read a test vector from a file called `test_vector.pat` (the first task/function argument), perhaps manipulate the vector to put it in a proper format for Verilog, and then assign the vector value to the second task/function argument called `input_bus`.

26.1.4 Arguments to sizetf, compiletf, and calltf application routines

The *sizetf*, *compiletf*, and *calltf* routines all take one argument. When the software product calls these routines, it will pass to them the value supplied in the `s_vpi_systf_data` structure's *user_data* field when the user-defined system task/function was registered. See 27.34.

26.2 VPI mechanism

VPI provides routines that allow application developers to access information contained in a Verilog design and that allow facilities to interact dynamically with a software product. Applications of VPI can include delay calculators and annotators, connecting a Verilog simulator with other simulation and CAE systems, and customized debugging tasks.

The functions of VPI can be grouped into two main areas:

— Dynamic software product interaction using VPI callbacks
— Access to Verilog HDL objects and simulation-specific objects

26.2.1 VPI callbacks

Dynamic software product interaction shall be accomplished with a registered callback mechanism. VPI callbacks shall allow an application to request that a Verilog HDL software product, such as a logic simulator, call a user-defined application when a specific activity occurs. For example, the application can request that the application routine `my_monitor()` be called when a particular net changes value or that `my_cleanup()` be called when the software product execution has completed.

The VPI callback facility shall provide the application with the means to interact dynamically with a software product, detecting the occurrence of value changes, advancement of time, end of simulation, etc. This feature allows integration with other simulation systems, specialized timing checks, complex debugging features, etc.

The reasons for which callbacks shall be provided can be separated into four categories:

— *Simulation event* (e.g., a value change on a net or a behavioral statement execution)
— *Simulation time* (e.g., the end of a time queue or after certain amount of time)

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

— *Simulator action or feature* (e.g., the end of compile, end of simulation, restart, or enter interactive mode)
— *User-defined system task/function execution*

VPI callbacks shall be registered by the application with the functions **vpi_register_cb()** and **vpi_register_systf()**. These routines indicate the specific reason for the callback, the application routines to be called, and what system and *user_data* shall be passed to the callback application when the callback occurs. A facility is also provided to call the callback functions when a Verilog HDL product is first invoked. A primary use of this facility shall be for registration of user-defined system tasks and functions.

26.2.2 VPI access to Verilog HDL objects and simulation objects

Accessible Verilog HDL objects and simulation objects and their relationships and properties are described using data model diagrams. These diagrams are presented in 26.6. The data model diagrams indicate the routines and constants that are required to access and manipulate objects within an application environment. An associated set of routines to access these objects is defined in Clause 27.

VPI also includes a set of utility routines for functions such as handle comparison, file handling, and redirected printing, which are described in Table 26-9 (in 26.4).

VPI routines provide access to objects in an *instantiated* Verilog design. An instantiated design is one where each instance of an object is uniquely accessible. For instance, if a module `m` contains wire `w` and is instantiated twice as `m1` and `m2`, then `m1.w` and `m2.w` are two distinct objects, each with its own set of related objects and properties.

VPI is designed as a *simulation* interface, with access to both Verilog HDL objects and specific simulation objects. This simulation interface is different from a hierarchical language interface, which would provide access to HDL information, but would not provide information about simulation objects.

26.2.3 Error handling

To determine whether an error occurred, the routine **vpi_chk_error()** shall be provided. The **vpi_chk_error()** routine shall return a nonzero value if an error occurred in the previously called VPI routine. Callbacks can be set up for when an error occurs as well. The **vpi_chk_error()** routine can provide detailed information about the error.

26.2.4 Function availability

Certain features of VPI must occur early in the execution of a tool. In order to allow this process to occur in an orderly manner, some functionality must be restricted in these early stages. Specifically, when the routines within the **vlog_startup_routines[ ]** array are executed, there is very little functionality available. Only two routines can be called at this time:

— **vpi_register_systf()**
— **vpi_register_cb()**

In addition, the **vpi_register_cb()** routine can only be called for the following reasons:

— **cbEndOfCompile**
— **cbStartOfSimulation**
— **cbEndOfSimulation**
— **cbUnresolvedSystf**

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

— **cbError**
— **cbPLIError**

See 27.34 for a further explanation of the use of the **vlog_startup_routines**[ ] array.

The next earliest phase is when the *sizetf* routines are called for the user-defined system functions. At this phase, no additional access is permitted. After the *sizetf* routines are called, the routines registered for reason **cbEndOfCompile** are called. At this point, and continuing until the tool has finished execution, all functionality is available.

26.2.5 Traversing expressions

The VPI routines provide access to any expression that can be written in the HDL. Dealing with these expressions can be complex because very complex expressions can be written in the HDL. Expressions with multiple operands will result in a handle of type **vpiOperation**. To determine how many operands, access the property **vpiOpType**. This operation will be evaluated after its subexpressions. Therefore, it has the least precedence in the expression.

An example of a routine that traverses an entire complex expression is listed below:

```
void traverseExpr(vpiHandle expr)
{
    vpiHandle subExprI, subExprH;

    switch (vpi_get(vpiExpr,expr))
    {
        case vpiOperation:
            subExprI = vpi_iterate(vpiOperand, expr);
            if (subExprI)
                while (subExprH = vpi_scan(subExprI))
                    traverseExpr(subExprH);
            /* else it is of op type vpiNullOp */
            break;
        default:
            /* Do whatever to the leaf object. */
            break;
    }
}
```

26.3 VPI object classifications

VPI objects are classified using data model diagrams. These diagrams provide a graphical representation of those objects within a Verilog design to which the VPI routines shall provide access. The diagrams shall show the relationships between objects and the properties of each object. Objects with sufficient commonality are placed in groups. Group relationships and properties apply to all the objects in the group.

As an example, the simplified diagram in Figure 26-1 shows that there is a one-to-many relationship from objects of type **module** to objects of type **net** and a one-to-one relationship from objects of type **net** to objects of type **module**. Objects of type **net** have properties **vpiName**, **vpiVector**, and **vpiSize** with data types string, boolean, and integer, respectively.

The VPI data model diagrams are presented in 26.6.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

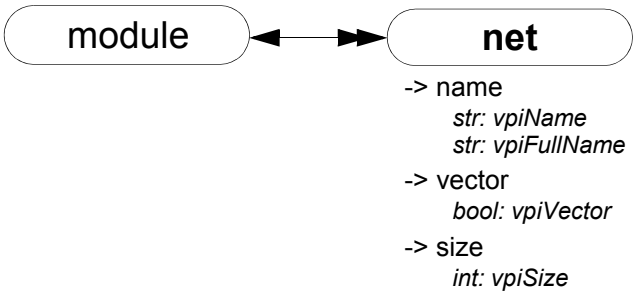


Figure 26-1—Example of object relationships diagram

For object relationships (unless a special tag is shown in the diagram), the type used for access is determined by adding "vpi" to the beginning of the word within the enclosure with each word's first letter being a capital. Using the above example, if an application has a handle to a net and wants to go to the module instance where the net is defined, the call would be as follows:

```
modH = vpi_handle(vpiModule,netH);
```

where netH is a handle to the net. As another example, to access a "named event" object, use the type **vpiNamedEvent**.

26.3.1 Accessing object relationships and properties

VPI defines the C data type of **vpiHandle**. All objects are manipulated via a **vpiHandle** variable. Object handles can be accessed from a relationship with another object or from a hierarchical name as the following example demonstrates:

```
vpiHandle net;
net = vpi_handle_by_name("top.m1.w1", NULL);
```

This example call retrieves a handle to wire `top.m1.w1` and assigns it to the **vpiHandle** variable `net`. The `NULL` second argument directs the routine to search for the name from the top level of the design.

VPI provides generic functions for tasks, such as traversing relationships and determining property values. One-to-one relationships are traversed with routine **vpi_handle()**. In the following example, the module that contains `net` is derived from a handle to that net:

```
vpiHandle net, mod;
net = vpi_handle_by_name("top.m1.w1", NULL);
mod = vpi_handle(vpiModule, net);
```

The call to **vpi_handle()** in the above example shall return a handle to module `top.m1`.

Sometimes it is necessary to access a class of objects that do not have a name or whose name is ambiguous with another class of objects that can be accessed from the reference handle. *Tags* are used in this situation, as shown in Figure 26-2.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

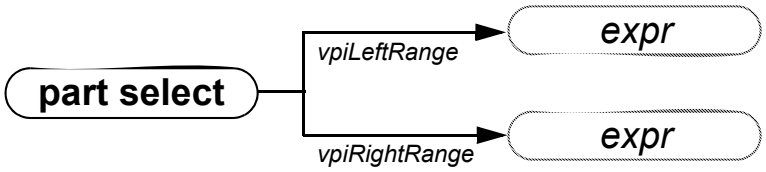


Figure 26-2—Accessing a class of objects using tags

In this example, the tags **vpiLeftRange** and **vpiRightRange** are used to access the expressions that make up the range of the part-select. These tags are used instead of **vpiExpr** to get to the expressions. Without the tags, VPI would not know which expression should be accessed. For example:

```
vpi_handle(vpiExpr, part_select_handle)
```

would be illegal when the reference handle (part_select_handle) is a handle to a part-select because the part-select can refer to two expressions, a left-range and a right-range.

Properties of objects shall be derived with routines in the **vpi_get** family. The routine **vpi_get()** returns integer and boolean properties. Integer and boolean properties shall be defined to be of type `PLI_INT32`. For boolean properties, a value of `1` shall represent `TRUE` and a value of `0` shall represent `FALSE`. The routine **vpi_get_str()** accesses string properties. String properties shall be defined to be of type `PLI_BYTE8 *`. For example, to retrieve a pointer to the full hierarchical name of the object referenced by handle `mod`, the following call would be made:

```
PLI_BYTE8 *name = vpi_get_str(vpiFullName, mod);
```

In the above example, the pointer `name` shall now point to the string `"top.m1"`.

One-to-many relationships are traversed with an iteration mechanism. The routine **vpi_iterate()** creates an object of type **vpiIterator**, which is then passed to the routine **vpi_scan()** to traverse the desired objects. In the following example, each net in module `top.m1` is displayed:

```
vpiHandle itr;
itr = vpi_iterate(vpiNet,mod);
while (net = vpi_scan(itr) )
       vpi_printf("\t%s\n", vpi_get_str(vpiFullName, net) );
```

As the above examples illustrate, the routine naming convention is a ‘`vpi`’ prefix with ‘_’ word delimiters (with the exception of callback-related defined values, which use the ‘`cb`’ prefix). Macro-defined types and properties have the ‘`vpi`’ prefix, and they use capitalization for word delimiters.

The routines for traversing Verilog HDL structures and accessing objects are described in Clause 27.

26.3.2 Object type properties

All objects have a **vpiType** property, which is not shown in the data model diagrams.

-> type
int: vpiType

Using **vpi_get(vpiType, <object_handle>)** returns an integer constant that represents the type of the object.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Using **vpi_get_str(vpiType, <object_handle>)** returns a pointer to a string containing the name of the type constant. The name of the type constant is derived from the name of the object as it is shown in the data model diagram (see 26.3 for a description of how type constant names are derived from object names).

Some objects have additional type properties that are shown in the data model diagrams: **vpiDelayType**, **vpiNetType**, **vpiOpType**, **vpiPrimType**, **vpiResolvedNetType**, and **vpiTchkType**. Using **vpi_get(<type_property>, <object_handle>)** returns an integer constant that represents the additional type of the object. See `vpi_user.h` in Annex G for the types that can be returned for these additional type properties. The constant names of the types returned for these additional type properties can be accessed using **vpi_get_str()**.

26.3.3 Object file and line properties

Most objects have two location properties, which are not shown in the data model diagrams:

-> location
int: vpiLineNo
str: vpiFile

The properties **vpiLineNo** and **vpiFile** can be affected by the **`line** compiler directive. See 19.7 for more details on the **`line** compiler directive. These properties are applicable to every object that corresponds to some object within the HDL. The exceptions are objects of the following types:

- **vpiCallback**
- **vpiDelayTerm**
- **vpiDelayDevice**
- **vpiInterModPath**
- **vpiIterator**
- **vpiTimeQueue**
- **vpiGenScopeArray**
- **vpiGenScope**

26.3.4 Delays and values

Most properties are of type integer, boolean, or string. Delay and logic value properties, however, are more complex and require specialized routines and associated structures. The routines **vpi_get_delays()** and **vpi_put_delays()** use structure pointers, where the structure contains the pertinent information about delays. Similarly, simulation values are also handled with the routines **vpi_get_value()** and **vpi_put_value()**, along with an associated set of structures.

The routines, C structures, and some examples for handling delays and logic values are presented in Clause 27. See 27.14 for **vpi_get_value()**, 27.32 for **vpi_put_value()**, 27.9 for **vpi_get_delays()**, and 27.30 for **vpi_put_delays()**.

Nets, primitives, module paths, timing checks, and continuous assignments can have delays specified within the HDL. Additional delays may exist, such as module input port delays or inter-module path delays, that do not appear within the HDL. To access the delay expressions that are specified within the HDL, use the method **vpiDelay**. These expressions shall be either an expression that evaluates to a constant if there is only one delay specified or an operation if there are more than one delay specified. If multiple delays are specified, then the operation's **vpiOpType** shall be **vpiListOp**. To access the actual delays being used by the tool, use the routine **vpi_get_delays()** on any of these objects.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.3.5 Object protection properties

All objects have a **vpiIsProtected** property, which is not shown in the data model diagrams.

-> IsProtected
bool: vpiIsProtected

Using **vpi_get(vpiIsProtected,** *object_handle***)** returns a boolean constant that indicates whether the object represents code contained in a decryption envelope. The **vpiIsProtected** property shall be TRUE if the *object_handle* represents code that is protected; otherwise, it shall be FALSE. Unless otherwise specified, access to relationships and properties of a protected object shall be an error. Restrictions on access to complex properties are specified in the function reference descriptions for the corresponding VPI functions. Access to the **vpiType** property and the **vpiIsProtected** property of a protected object shall be permitted for all objects.

NOTE—Protected objects can be returned through object relationships or by direct lookup using VPI functions that return handles.

26.4 List of VPI routines by functional category

The VPI routines can be divided into groups based on primary functionality:

- Simulation-related callbacks
- System task/function callbacks
- Traversing Verilog HDL hierarchy
- Accessing properties of objects
- Accessing objects from properties
- Delay processing
- Logic and strength value processing
- Simulation time processing
- Miscellaneous utilities

Table 26-1 through Table 26-9 list the VPI routines by major category. Clause 27 defines each of the VPI routines, listed in alphabetical order.

Table 26-1—VPI routines for simulation-related callbacks

To	Use
Register a simulation-related callback	**vpi_register_cb()**
Remove a simulation-related callback	**vpi_remove_cb()**
Get information about a simulation-related callback	**vpi_get_cb_info()**

Table 26-2—VPI routines for system task/function callbacks

To	Use
Register a system task/function callback	**vpi_register_systf()**
Get information about a system task/function callback	**vpi_get_systf_info()**

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 26-3—VPI routines for traversing Verilog HDL hierarchy

To	Use
Obtain a handle for an object with a one-to-one relationship	**vpi_handle()**
Obtain handles for objects in a one-to-many relationship	**vpi_iterate()** **vpi_scan()**
Obtain a handle for an object in a many-to-one relationship	**vpi_handle_multi()**

Table 26-4—VPI routines for accessing properties of objects

To	Use
Get the value of objects with types of `int` or `bool`	**vpi_get()**
Get the value of objects with types of string	**vpi_get_str()**

Table 26-5—VPI routines for accessing objects from properties

To	Use
Obtain a handle for a named object	**vpi_handle_by_name()**
Obtain a handle for an indexed object	**vpi_handle_by_index()**
Obtain a handle to a word or bit in an array	**vpi_handle_by_multi_index()**

Table 26-6—VPI routines for delay processing

To	Use
Retrieve delays or timing limits of an object	**vpi_get_delays()**
Write delays or timing limits to an object	**vpi_put_delays()**

Table 26-7—VPI routines for logic and strength value processing

To	Use
Retrieve logic value or strength value of an object	**vpi_get_value()**
Write logic value or strength value to an object	**vpi_put_value()**

Table 26-8—VPI routines for simulation time processing

To	Use
Find the current simulation time or the scheduled time of future events	**vpi_get_time()**

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

Table 26-9—VPI routines for miscellaneous utilities

To	Use
Write to the output channel of the software product that invoked the PLI application and the current log file	**vpi_printf()**
Write to the output channel of the software product that invoked the PLI application and the current log file using varargs	**vpi_vprintf()**
Flush data from the current simulator output buffers	**vpi_flush()**
Open a file for writing	**vpi_mcd_open()**
Close one or more files	**vpi_mcd_close()**
Write to one or more files	**vpi_mcd_printf()**
Write to one or more open files using varargs	**vpi_mcd_vprintf()**
Flush data from a given *mcd* output buffer	**vpi_mcd_flush()**
Retrieve the name of an open file	**vpi_mcd_name()**
Retrieve data about product invocation options	**vpi_get_vlog_info()**
See whether two handles refer to the same object	**vpi_compare_objects()**
Obtain error status and error information about the previous call to a VPI routine	**vpi_chk_error()**
Free memory allocated by VPI routines	**vpi_free_object()**
Add application-allocated storage to application saved data	**vpi_put_data()**
Retrieve application-allocated storage from application saved data	**vpi_get_data()**
Store user data in VPI work area	**vpi_put_userdata()**
Retrieve user data from VPI work area	**vpi_get_userdata()**
Control simulation execution (e.g., stop, finish)	**vpi_sim_control()**

26.5 Key to data model diagrams

This subclause contains the keys to the symbols used in the data model diagrams. Keys are provided for objects and classes, traversing relationships, and accessing properties.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.5.1 Diagram key for objects and classes

obj defn	Object definition**:** **Bold** letters in a solid enclosure indicate an object definition. The properties of the object are defined in this location.
object	Object reference: Normal letters in a solid enclosure indicate an object reference.
class defn *class* **obj defn** object	Class definition: ***Bold italic*** letters in a dotted enclosure indicate a class definition, where the class groups other objects and classes. Properties of the class are defined in this location. The class definition can contain an object definition.
class	Class reference: *Italic* letters in a dotted enclosure indicate a class reference.
obj1 obj2	Unnamed class: A dotted enclosure with no name is an unnamed class. It is sometimes convenient to group objects although they shall not be referenced as a group elsewhere; therefore, a name is not indicated.

26.5.2 Diagram key for accessing properties

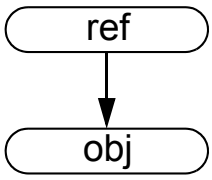
obj
-> vector
bool: vpiVector
-> size
int: vpiSize
	Integer and boolean properties are accessed with the routine **vpi_get()**. These properties are of type `PLI_INT32`.
	
	For example: Given handle `obj_h` to an object of type **vpiObj**, test if the object is a vector, and get the size of the object.
	
	`PLI_INT32 vect_flag = vpi_get(vpivector, obj_h);`
	`PLI_INT32 size = vpi_get(vpiSize, obj_h);`

obj -> name *str: vpiName* *str: vpiFullName*	String properties are accessed with routine **vpi_get_str()**. String properties are of type `PLI_BYTE8 *`. For example: `PLI_BYTE8 *name = vpi_get_str(vpiName, obj_h);`
object -> complex *func1()* *func2()*	Complex properties for time and logic value are accessed with the indicated routines. See the descriptions of the routines for usage.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

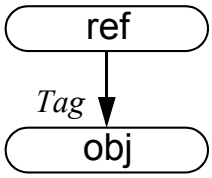
26.5.3 Diagram key for traversing relationships



A single arrow indicates a one-to-one relationship accessed with the routine **vpi_handle()**.

For example: Given **vpiHandle** variable `ref_h` of type `ref`, access `obj_h` of type *Obj*:

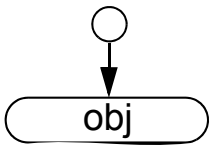
```
obj_h = vpi_handle(Obj, ref_h);
```



A tagged one-to-one relationship is traversed similarly, using *Tag* instead of *Obj*.

For example:

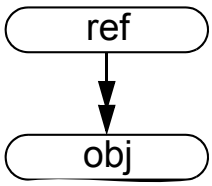
```
obj_h = vpi_handle(Tag, ref_h);
```



A one-to-one relationship which originates from a circle is traversed using `NULL` for the `ref_h`.

For example:

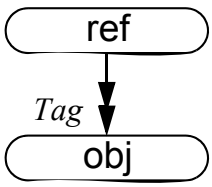
```
obj_h = vpi_handle(Obj, NULL);
```



A double arrow indicates a one-to-many relationship accessed with the routine **vpi_scan()**.

For example: Given **vpiHandle** variable `ref_h` of type `ref`, scan objects of type *Obj*:

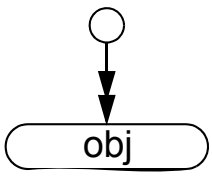
```
itr = vpi_iterate(Obj, ref_h);
while (obj_h = vpi_scan(itr) )
  /* process 'obj_h' */
```



A tagged one-to-many relationship is traversed similarly, using *Tag* instead of *Obj*.

For example:

```
itr = vpi_iterate(Tag, ref_h);
while (obj_h = vpi_scan(itr) )
  /* process 'obj_h' */
```



A one-to-many relationship that originates from a circle is traversed using `NULL` for the `ref_h`.

For example:

```
itr = vpi_iterate(Obj, NULL);
while (obj_h = vpi_scan(itr) )
  /* process 'obj_h' */
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

For relationships that do not have a tag, the type used for access is determined by adding “vpi” to the beginning of the word within the enclosure with each word’s first letter being a capital. See 26.3 for more details on VPI access constant names.

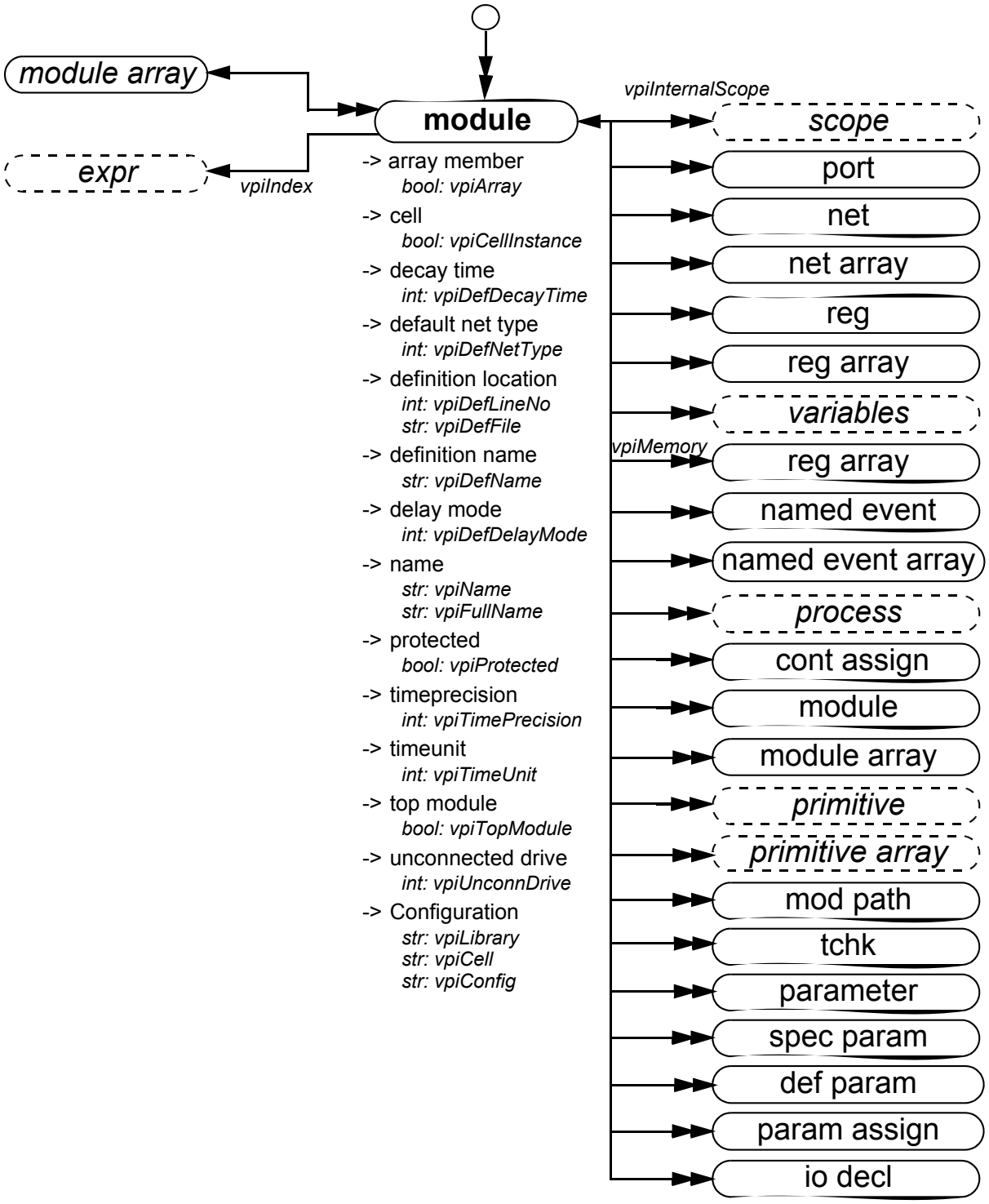
26.6 Object data model diagrams

Subclauses 26.6.1 through 26.6.43 contain the data model diagrams that define the accessible objects and groups of objects, along with their relationships and properties.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.1 Module



Details:

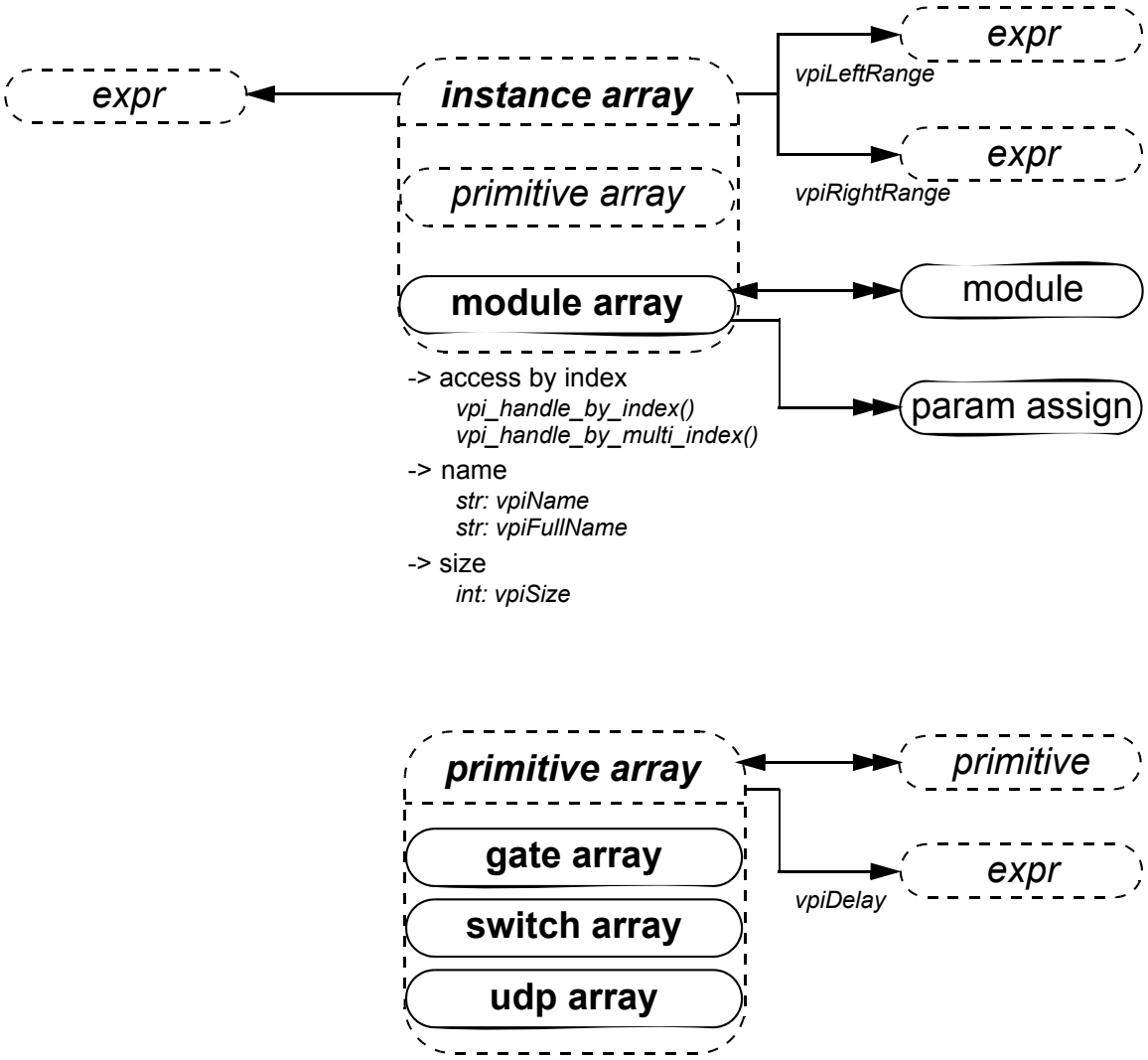
a) Top-level modules shall be accessed using **vpi_iterate()** with a `NULL` reference object.
b) Passing a `NULL` handle to **vpi_get()** with properties **vpiTimePrecision** or **vpiTimeUnit** shall return the smallest time precision of all modules in the instantiated design.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

c) The properties **vpiDefLineNo** and **vpiDefFile** can be affected by the **`line** compiler directive. See 19.7 for more details on the **`line** compiler directive.
d) If a module is an element within a module array, the **vpiIndex** transition is used to access the index within the array. If a module is not part of a module array, this transition shall return `NULL`.

26.6.2 Instance arrays

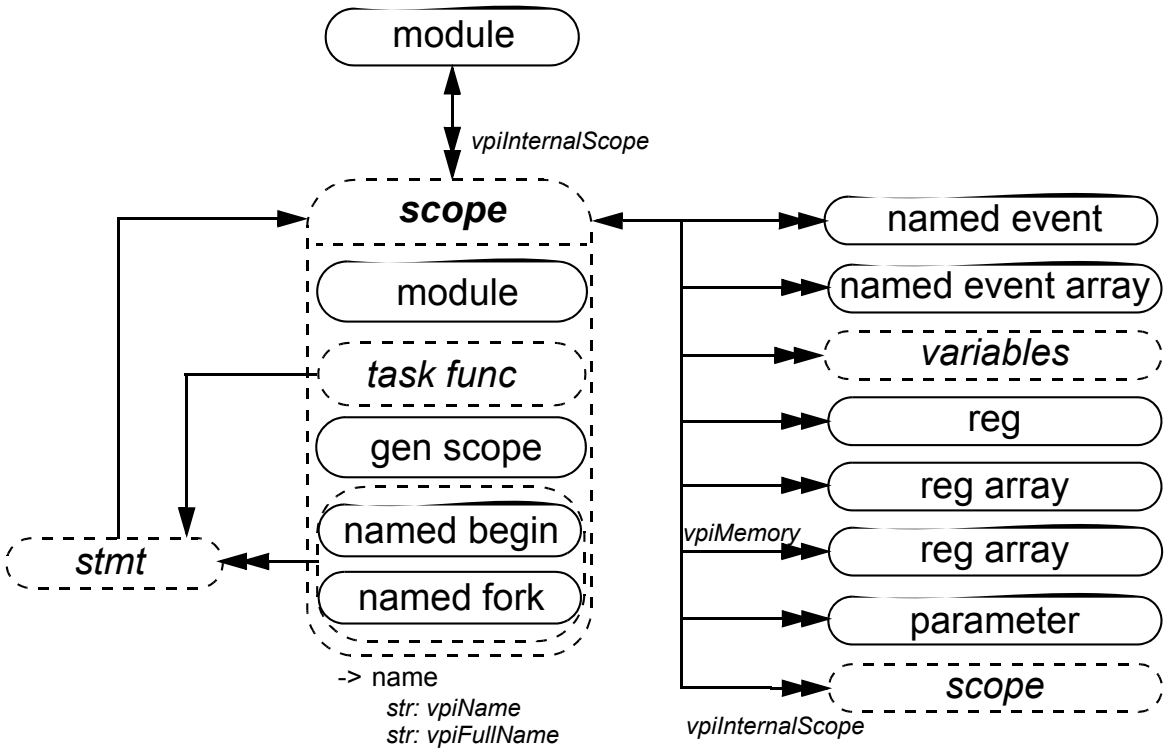


Details:

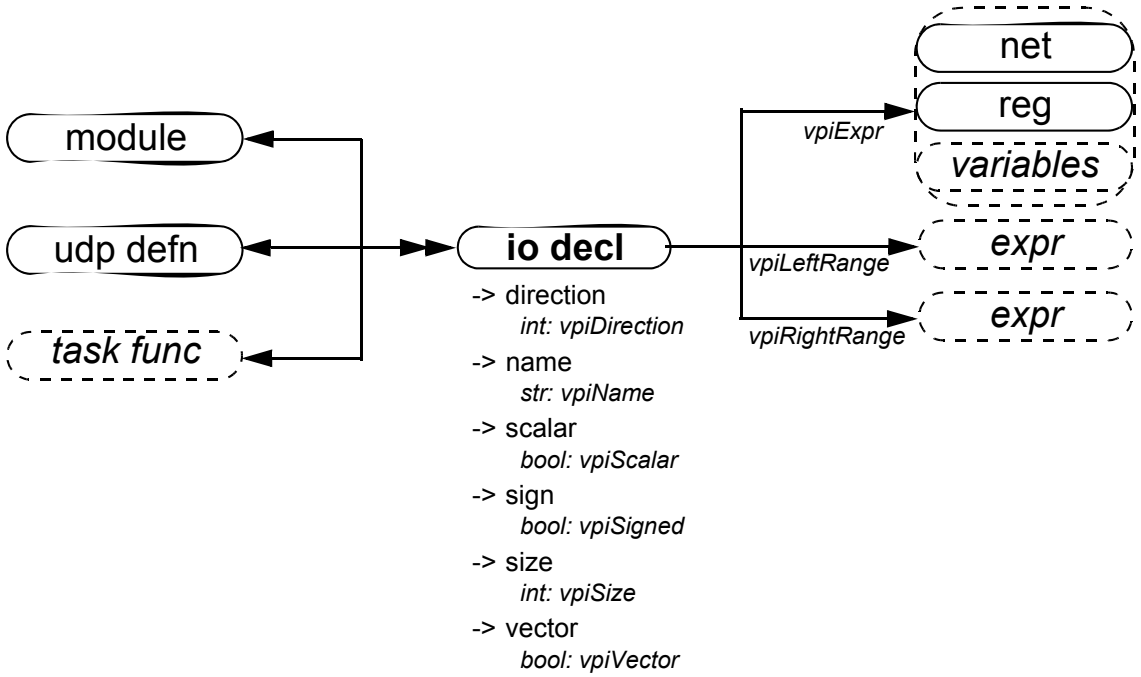
Traversing from the instance array to **expr** shall return a simple expression object of type **vpiOperation** with a **vpiOpType** of **vpiListOp**. This expression may be used to access the actual list of connections to the module or primitive instance array in the Verilog source code.

Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.3 Scope



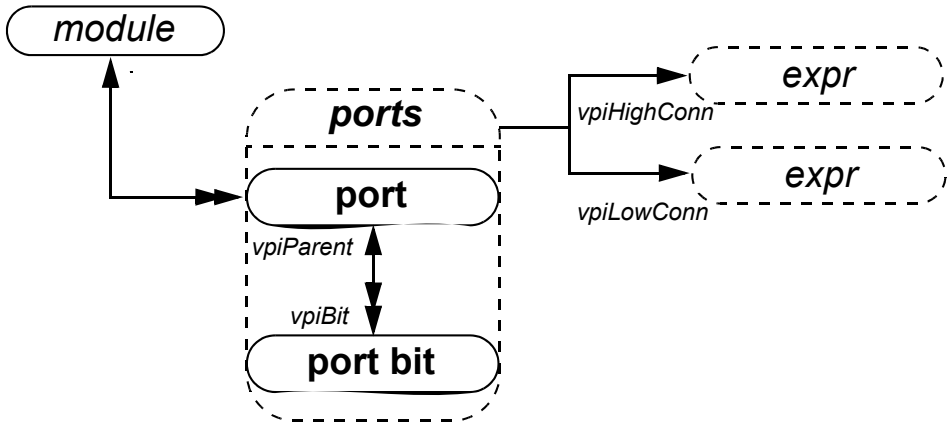
26.6.4 IO declaration



Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.5 Ports



-> access by index
vpi_handle_by_index()
vpi_handle_by_multi_index()
-> connected by name
bool: vpiConnByName
-> delay (mipd)
vpi_get_delays()
vpi_put_delays()
-> direction
int: vpiDirection
-> explicitly named
bool: vpiExplicitName
-> index
int: vpiPortIndex
-> name
str: vpiName
-> scalar
bool: vpiScalar
-> size
int: vpiSize
-> vector
bool: vpiVector

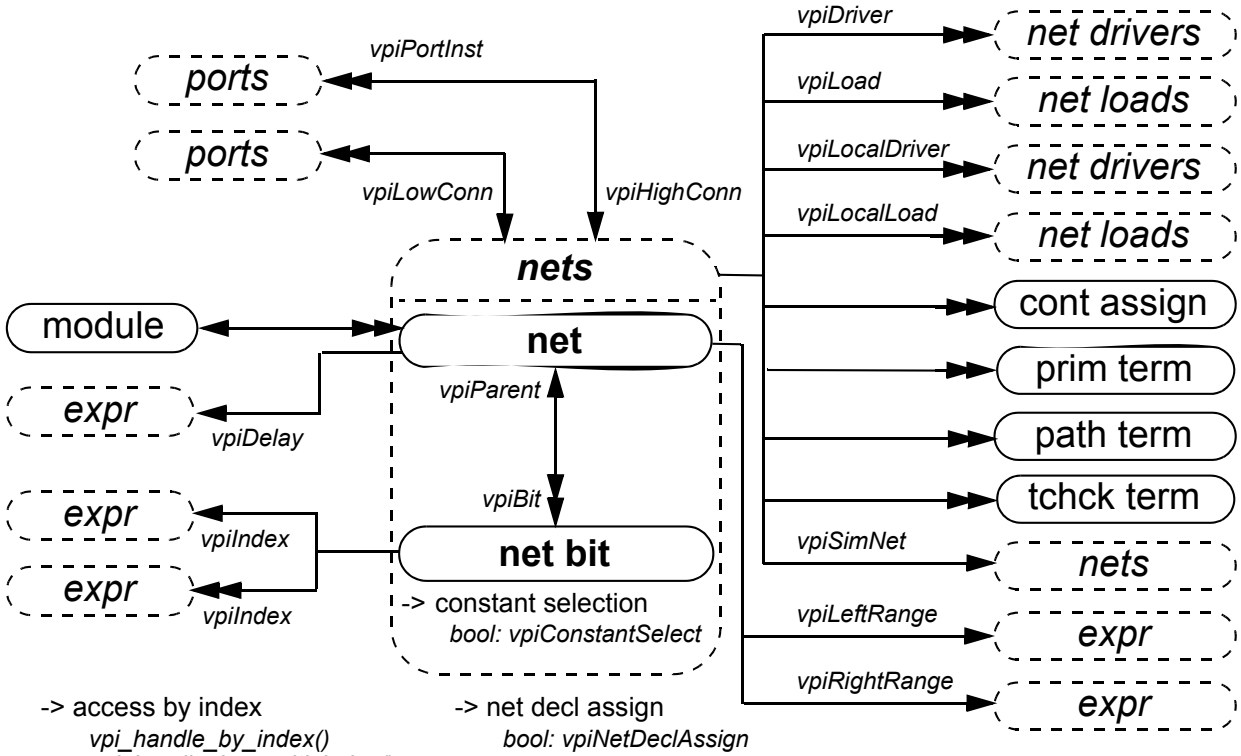
Details:

a) **vpiHighConn** shall indicate the hierarchically higher (closer to the top module) port connection.
b) **vpiLowConn** shall indicate the lower (further from the top module) port connection.
c) Properties **vpiScalar** and **vpiVector** shall indicate if the port is 1 bit or more than 1 bit. They shall not indicate anything about what is connected to the port.
d) Properties **vpiIndex** and **vpiName** shall not apply for port bits.
e) If a port is explicitly named, then the explicit name shall be returned. If not and a name exists, then that name shall be returned. Otherwise, `NULL` shall be returned.
f) **vpiPortIndex** can be used to determine the port order. The first port has a port index of zero.
g) **vpiLowConn** shall return `NULL` if the module port is a null port (e.g., "**module** `M();`"). **vpiHighConn** shall return `NULL` if the instance of the module does not have a connection to the port.
h) **vpiSize** for a null port shall return 0.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

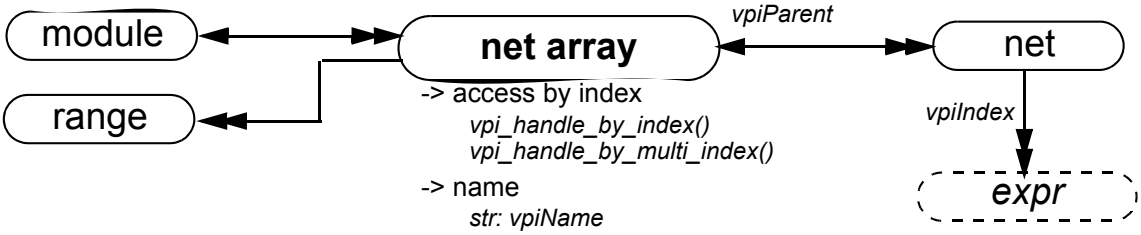
26.6.6 Nets and net arrays



-> access by index
vpi_handle_by_index()
vpi_handle_by_multi_index()
-> array member
bool: vpiArray
-> delay
vpi_get_delays()
-> expanded
bool: vpiExpanded
-> implicitly declared
bool: vpiImplicitDecl
-> name
str: vpiName
str: vpiFullName

-> net decl assign
bool: vpiNetDeclAssign
-> net type
int: vpiNetType
int: vpiResolvedNetType
-> scalar
bool: vpiScalar
-> scalared declaration
bool: vpiExplicitScalared
-> sign
bool: vpiSigned
-> size
int: vpiSize

-> strength
int: vpiStrength0
int: vpiStrength1
int: vpiChargeStrength
-> value
vpi_get_value()
vpi_put_value()
-> vector
bool: vpiVector
-> vectored declaration
bool: vpiExplicitVectored



-> access by index
vpi_handle_by_index()
vpi_handle_by_multi_index()
-> name
str: vpiName
str: vpiFullName
-> size
int: vpiSize
-> scalar
bool: vpiScalar
-> vector
bool: vpiVector

(Details on next page)

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

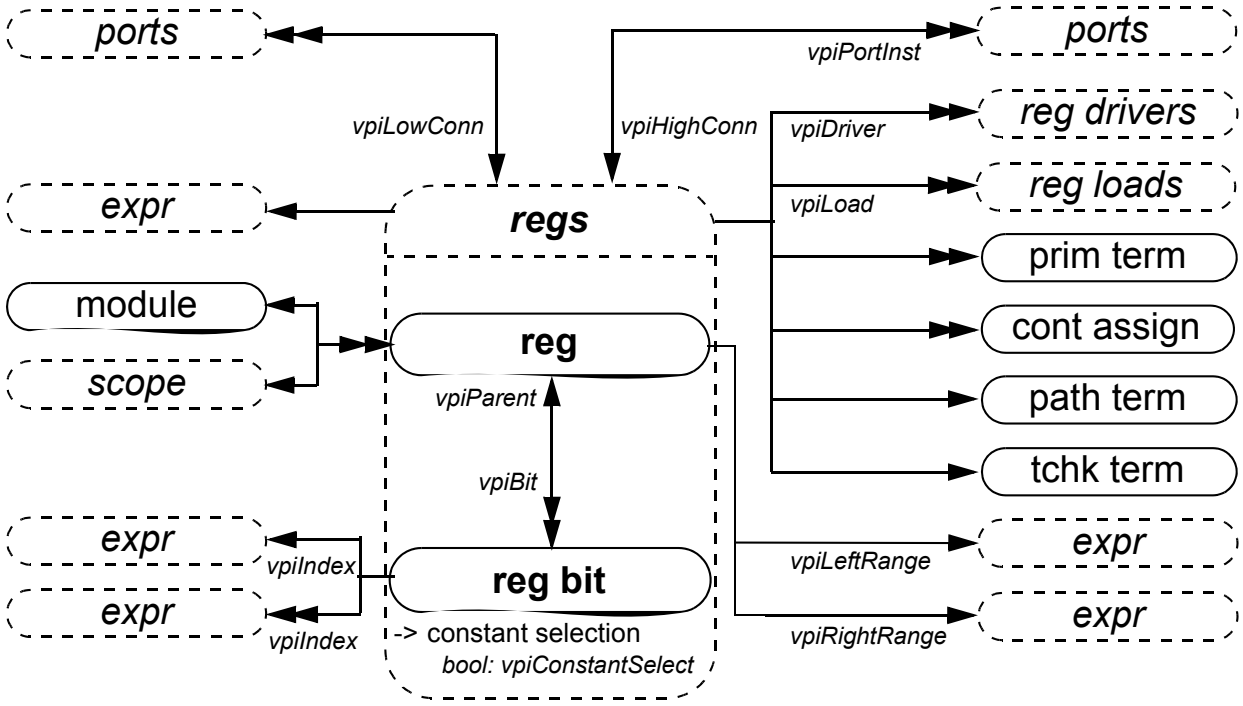
Details:

a) For vectors, net bits shall be available regardless of vector expansion.
b) Continuous assignments and primitive terminals shall be accessed regardless of hierarchical boundaries.
c) Continuous assignments and primitive terminals shall only be accessed from scalar nets or bit-selects.
d) For **vpiPorts**, if the reference handle is a bit, then port bits shall be returned. If it is the entire vector, then a handle to the entire port shall be returned.
e) For **vpiPortInst**, if the reference handle is a bit or scalar, then port bits or scalar ports shall be returned, unless the highconn for the port is a complex expression where the bit index cannot be determined. If this is the case, then the entire port shall be returned. If the reference handle is a vector, then the entire port shall be returned.
f) For **vpiPortInst**, it is possible for the reference handle to be part of the highconn expression, but not connected to any of the bits of the port. This may occur if there is a size mismatch. In this situation, the port shall not qualify as a member for that iteration.
g) For implicit nets, **vpiLineNo** shall return 0, and **vpiFile** shall return the file name where the implicit net is first referenced.
h) **vpi_handle(vpiIndex, net_bit_handle)** shall return the bit index for the net bit. **vpi_iterate(vpiIndex, net_bit_handle)** shall return the set of indices for a multidimensional net array bit-select, starting with the index for the net bit and working outward.
i) Only active forces and assign statements shall be returned for **vpiLoad**.
j) Only active forces shall be returned for **vpiDriver**.
k) **vpiDriver** shall also return ports that are driven by objects other than nets and net bits.
l) **vpiLocalLoad** and **vpiLocalDriver** return only the loads or drivers that are local, i.e., contained by the module instance that contains the net, including any ports connected to the net (output and inout ports are loads, input and inout ports are drivers).
m) For **vpiLoad**, **vpiLocalLoad**, **vpiDriver**, and **vpiLocalDriver** iterators, if the object is **vpiNet** for a vector net, then all loads or drivers are returned exactly once as the loading or driving object. In other words, if a part-select loads or drives only some bits, the load or driver returned is the part-select. If a driver is repeated, it is only returned once. To trace exact bit-by-bit connectivity, pass a **vpiNetBit** object to vpi_iterate.
n) An iteration on loads or drivers for a variable bit-select shall return the set of loads or drivers for whatever bit that the bit-select is referring to at the beginning of the iteration.
o) **vpiSimNet** shall return a unique net if an implementation collapses nets across hierarchy (see 12.3.10 for the definition of simulated net and collapsed net).
p) The property **vpiExpanded** on an object of type **vpiNetBit** shall return the property's value for the parent.
q) The loads and drivers returned from **(vpiLoad, obj_handle)** and **vpi_iterate(vpiDriver, obj_handle)** may not be the same in different implementations due to allowable net collapsing (see 12.3.10). The loads and drivers returned from **vpi_iterate(vpiLocalLoad, obj_handle)** and **vpi_iterate(vpiLocalDriver, obj_handle)** shall be the same for all implementations.
r) The boolean property **vpiConstantSelect** returns `TRUE` if the expression that constitutes the index or indices evaluates to a constant and `FALSE` otherwise.
s) **vpi_get(vpiSize, net_handle)** returns the number of bits in the net. **vpi_get(vpiSize, net_array_handle)** returns the total number of nets in the array.
t) **vpi_iterate(vpiIndex, net_handle)** shall return the set of indices for a net within an array, starting with the index for the net and working outward. If the net is not part of an array, a `NULL` shall be returned.
u) **vpi_iterate(vpiRange, net_array_handle)** shall return the set of array range declarations beginning with the leftmost range of the array declaration and iterate to the rightmost range of the array declaration.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

26.6.7 Regs and reg arrays



-> access by index
vpi_handle_by_index()
vpi_handle_by_multi_index()

-> array member
bool: vpiArray

-> name
str: vpiName
str: vpiFullName

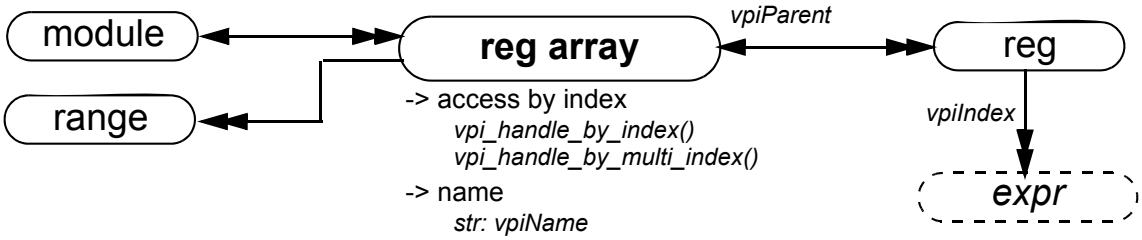
-> scalar
bool: vpiScalar

-> sign
bool: vpiSigned

-> size
int: vpiSize

-> value
vpi_get_value()
vpi_put_value()

-> vector
bool: vpiVector



-> access by index
vpi_handle_by_index()
vpi_handle_by_multi_index()

-> name
str: vpiName
str: vpiFullName

-> size
int: vpiSize

-> scalar
bool: vpiScalar

-> vector
bool: vpiVector

(Details on next page)

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

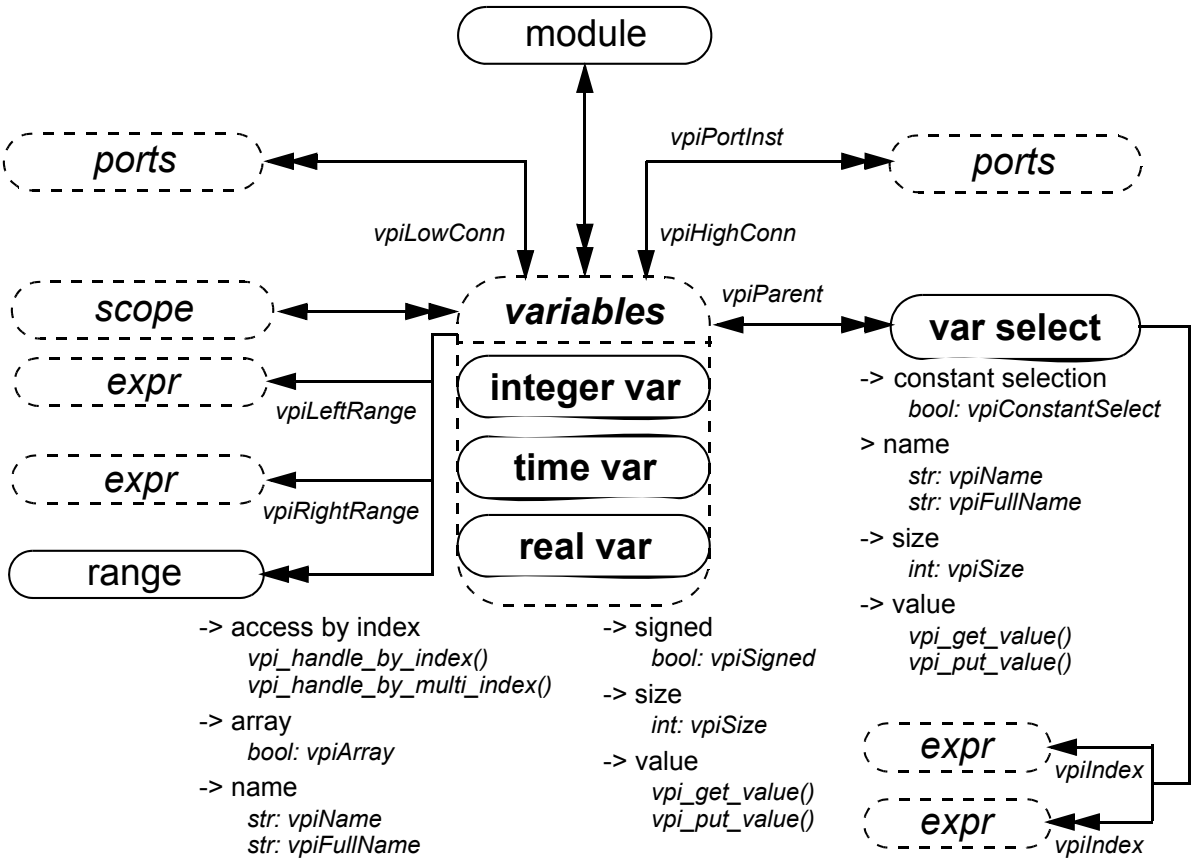
Details:

a) Continuous assignments and primitive terminals shall be accessed regardless of hierarchical boundaries.
b) Continuous assignments and primitive terminals shall only be accessed from scalar regs and bit-selects.
c) For **vpiPorts**, if the reference handle is a bit, then port bits shall be returned. If it is the entire vector, then a handle to the entire port shall be returned.
d) For **vpiPortInst**, if the reference handle is a bit or scalar, then port bits or scalar ports shall be returned, unless the highconn for the port is a complex expression where the bit index cannot be determined. If this is the case, then the entire port shall be returned. If the reference handle is a vector, then the entire port shall be returned.
e) For **vpiPortInst**, it is possible for the reference handle to be part of the highconn expression, but not connected to any of the bits of the port. This may occur if there is a size mismatch. In this case, the port shall not qualify as a member for that iteration.
f) **vpi_handle(vpiIndex, reg_bit_handle)** shall return the bit index for the reg bit. **vpi_iterate(vpiIndex, reg_bit_handle)** shall return the set of indices for a multidimensional reg array bit-select, starting with the index for the reg bit and working outward.
g) Only active forces and assign statements shall be returned for **vpiLoad** and **vpiDriver**.
h) For **vpiLoad** and **vpiDriver** iterators, if the object is **vpiReg** for a vectored reg, then all loads or drivers are returned exactly once as the loading or driving object. In other words, if a part-select loads or drives only some bits, the load or driver returned is the part-select. If a driver is repeated, it is only returned once. To trace exact bit-by-bit connectivity, pass a **vpiRegBit** object to the iterator.
i) The loads and drivers returned from **vpi_iterate(vpiLoad, obj_handle)** and **vpi_iterate(vpiDriver, obj_handle)** may not be the same in different implementations due to allowable net collapsing (see 12.3.10).
j) An iteration on loads or drivers for a variable bit-select shall return the set of loads or drivers for whatever bit that the bit-select is referring to at the beginning of the iteration.
k) If the reg has a default initialization assignment, the expression can be accessed using **vpi_handle(vpiExpr, reg_handle)** or **vpi_handle(vpiExpr, reg_bit_handle)**.
l) **vpi_get(vpiSize, reg_handle)** returns the number of bits in the reg. **vpi_get(vpiSize, reg_array_handle)** returns the total number of regs in the array.
m) **vpi_iterate(vpiIndex, reg_handle)** shall return the set of indices for a reg within an array, starting with the index for the reg and working outward. If the reg is not part of an array, a `NULL` shall be returned.
n) **vpi_iterate(vpiRange, array_handle)** shall return the set of array range declarations beginning with the leftmost range of the array declaration and iterate to the rightmost range of the array declaration.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

26.6.8 Variables



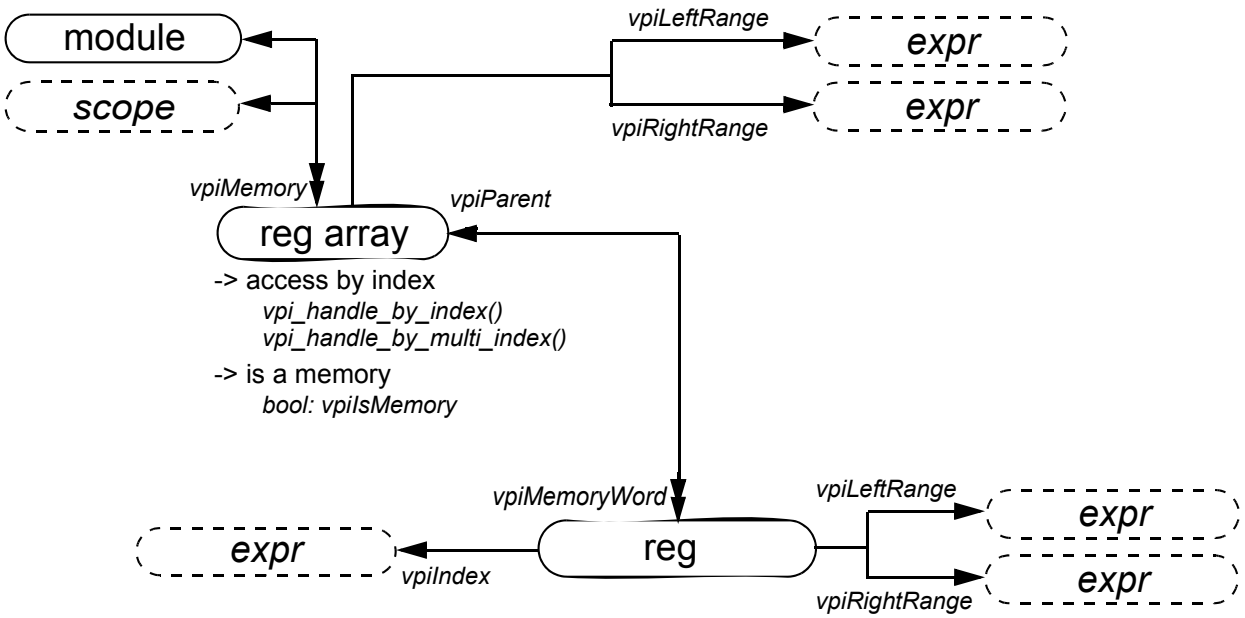
Details:

a) A var select is a word selected from a variable array.
b) The VPI does not provide access to bits of variables. If a handle to a bit-select of a variable is obtained, the object shall be a **vpiBitSelect** in the simple expression class. The variable containing the bit can be accessed using **vpiParent**. See 26.6.25.
c) The boolean property **vpiArray** shall be `TRUE` if the variable handle references an array of variables and `FALSE` otherwise. If the variable is an array, iterate on **vpiVarSelect** to obtain handles to each variable in the array.
d) **vpi_handle(vpiIndex, var_select_handle)** shall return the index of a var select in a one-dimensional array. **vpi_iterate(vpiIndex, var_select_handle)** shall return the set of indices for a var select in a multidimensional array, starting with the index for the var select and working outward.
e) **vpiRange** shall apply to variables when **vpiArray** is `TRUE`. **vpi_iterate(vpiRange, variable_array_handle)** shall return the set of array range declarations beginning with the leftmost range of the array declaration and iterate to the rightmost range of the array declaration.
f) **vpiLeftRange** and **vpiRightRange** shall apply to variables when **vpiArray** is `TRUE` and represent the array range declaration of the rightmost range of an array. These relationships are a shortcut for accessing the range declarations of a one-dimensional variable array. To access the range declarations for all dimensions of a multidimensional array, first iterate on **vpiRange**.
g) **vpiSize** for a variable array shall return the number of variables in the array. For nonarray variables, it shall return the size of the variable in bits.
h) **vpiSize** for a var select shall return the number of bits in the var select.
i) Variables whose boolean property **vpiArray** is `TRUE` do not have a value property.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

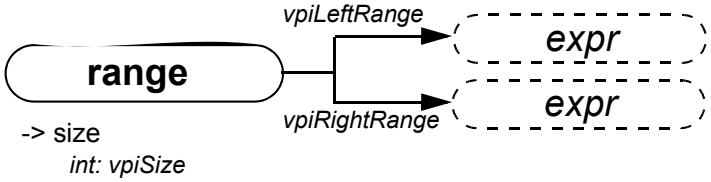
26.6.9 Memory



Details:

The objects **vpiMemory** and **vpiMemoryWord** have been generalized with the addition of arrays of regs. To preserve backward compatibility, they have been converted into methods that will return objects of type **vpiRegArray** and **vpiReg**, respectively. See 26.6.7 for the definitions of regs and reg arrays.

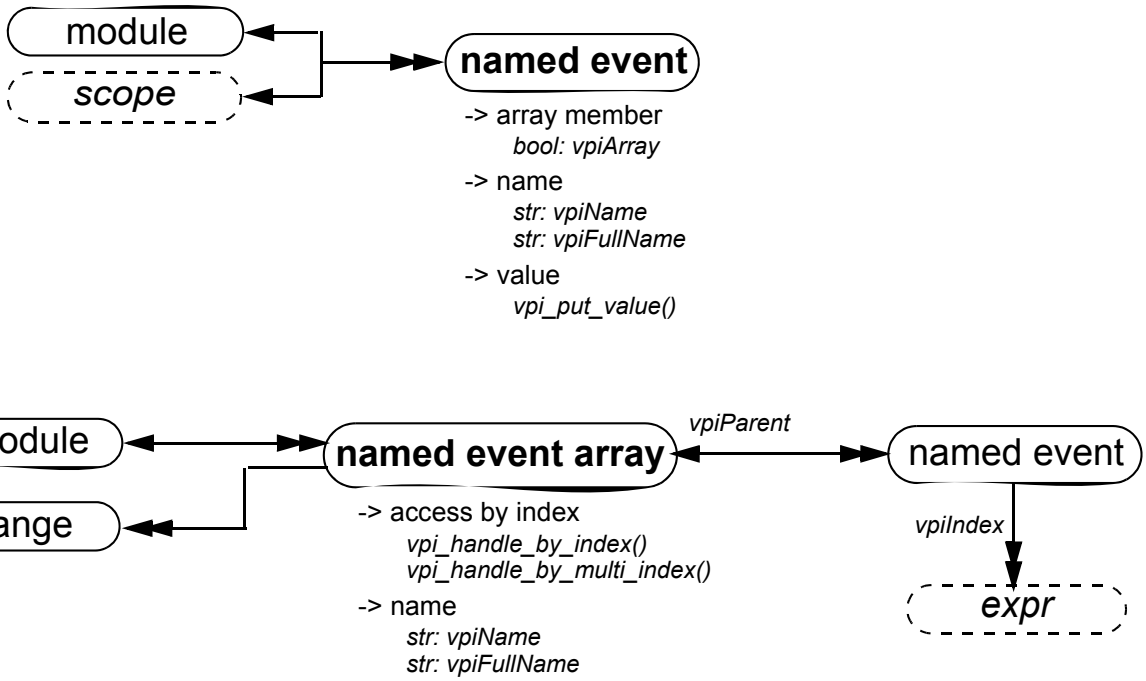
26.6.10 Object range



Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.11 Named event



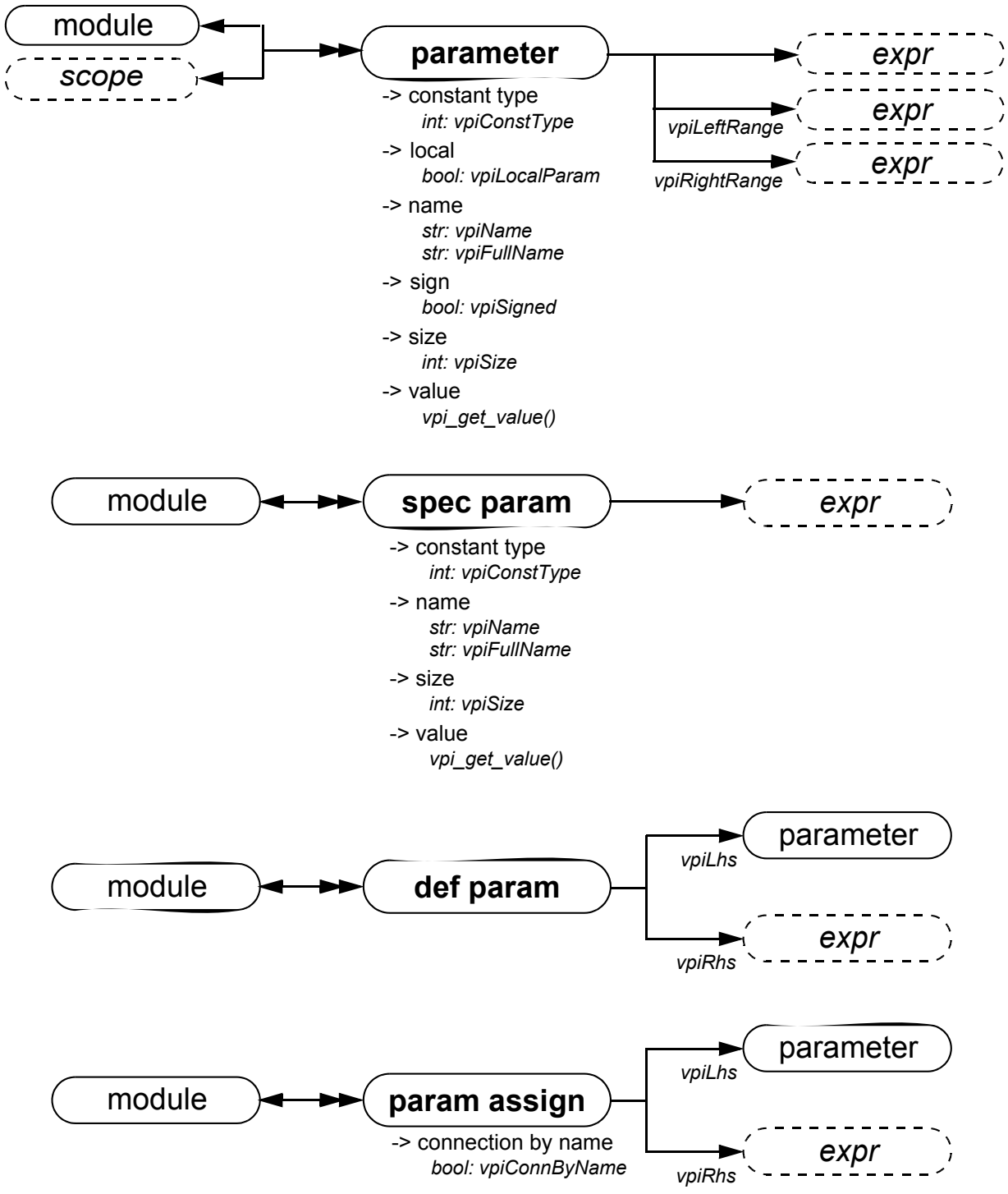
Details:

vpi_iterate(vpiIndex, named_event_handle) shall return the set of indices for a named event within an array, starting with the index for the named event and working outward. If the named event is not part of an array, a `NULL` shall be returned.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.12 Parameter, specparam



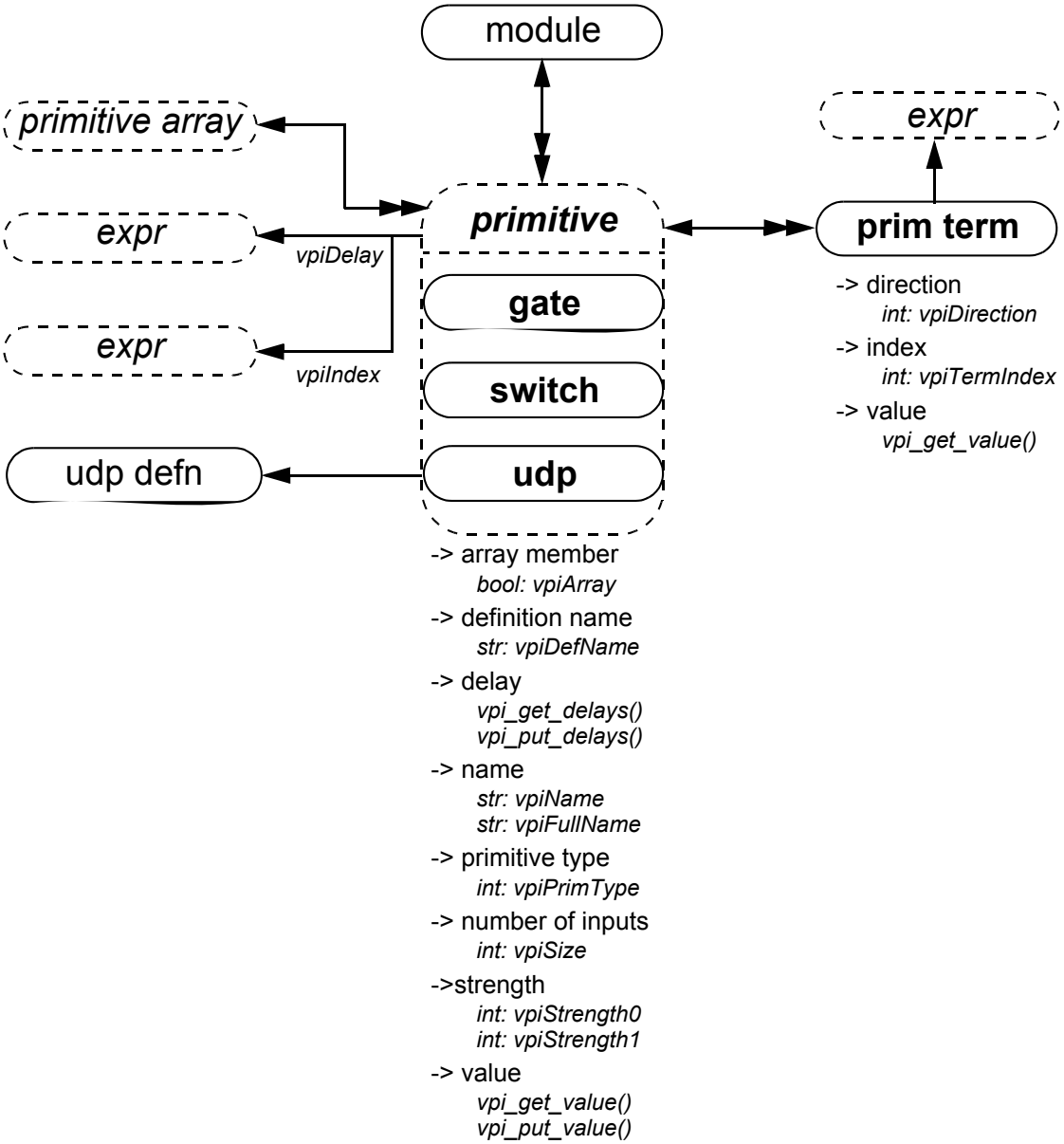
Details:

a) Obtaining the value from the object **parameter** shall return the final value of the parameter after all module instantiation overrides and defparams have been resolved.
b) **vpiLhs** from a param assign object shall return a handle to the overridden parameter.
c) If a parameter does not have an explicitly defined range, **vpiLeftRange** and **vpiRightRange** shall return a NULL handle.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.13 Primitive, prim term



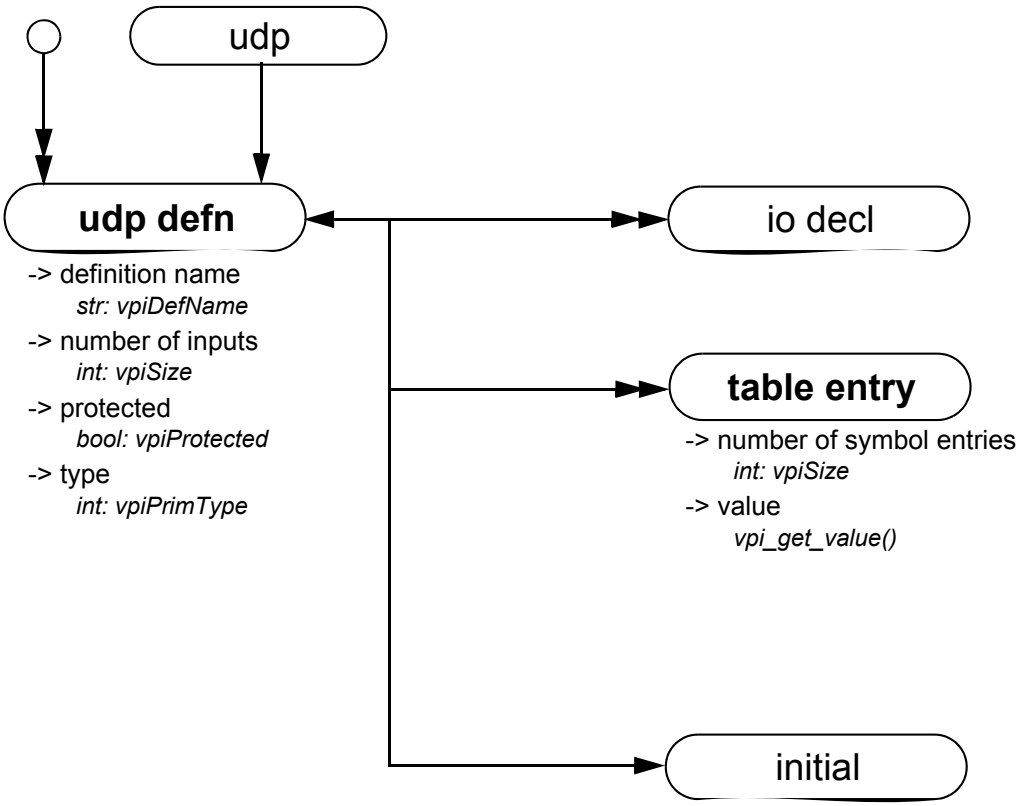
Details:

a) **vpiSize** shall return the number of inputs.
b) For primitives, **vpi_put_value()** shall only be used with sequential UDP primitives.
c) **vpiTermIndex** can be used to determine the terminal order. The first terminal has a term index of zero.
d) If a primitive is an element within a primitive array, the **vpiIndex** transition is used to access the index within the array. If a primitive is not part of a primitive array, this transition shall return `NULL`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.14 UDP



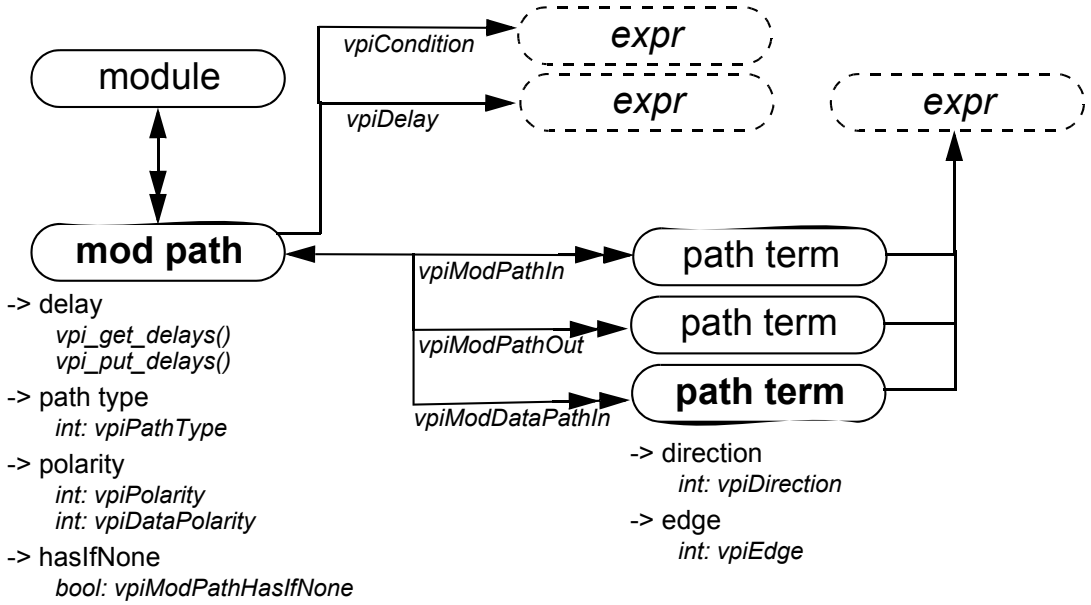
Details:

a) Only string (decompilation) and vector (ASCII values) shall be obtained for table entry objects using **vpi_get_value()**. See the definition of **vpi_get_value()** for additional details.
b) **vpiPrimType** returns **vpiSeqPrim** for sequential UDPs and **vpiCombPrim** for combinatorial UDPs.

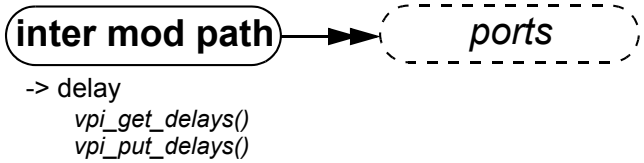
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.15 Module path, path term



26.6.16 Intermodule path



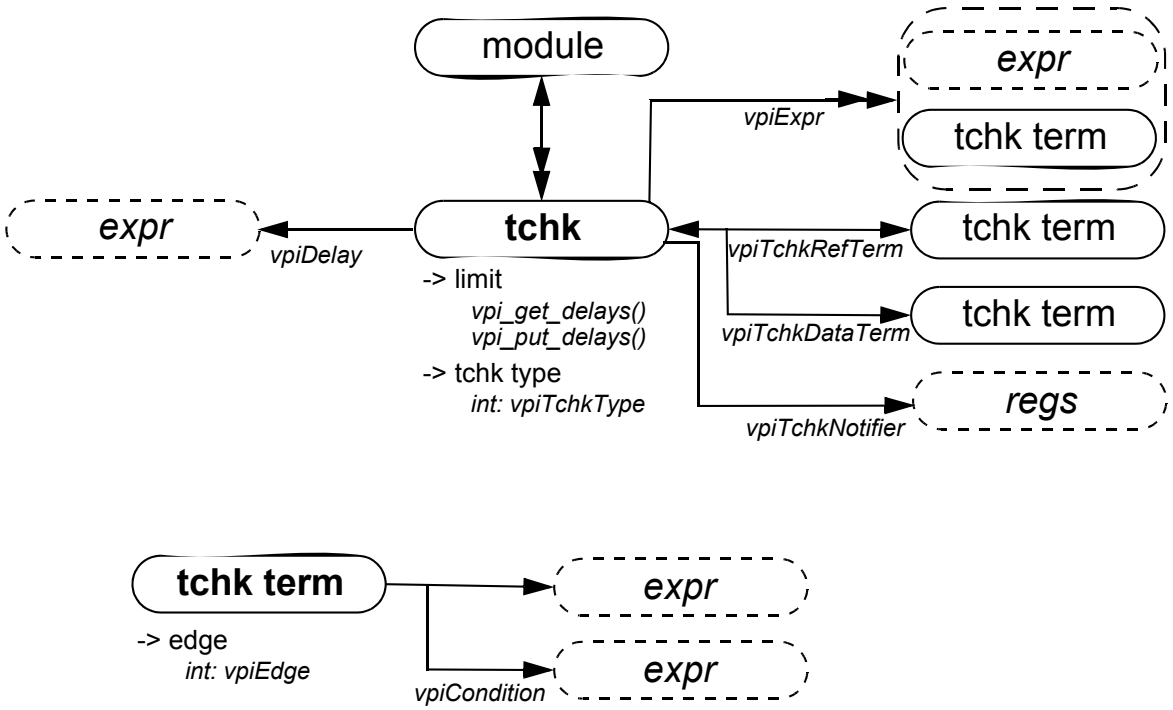
Details:

To get to an intermodule path, **vpi_handle_multi(vpiInterModPath**, port1, port2**)** can be used.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

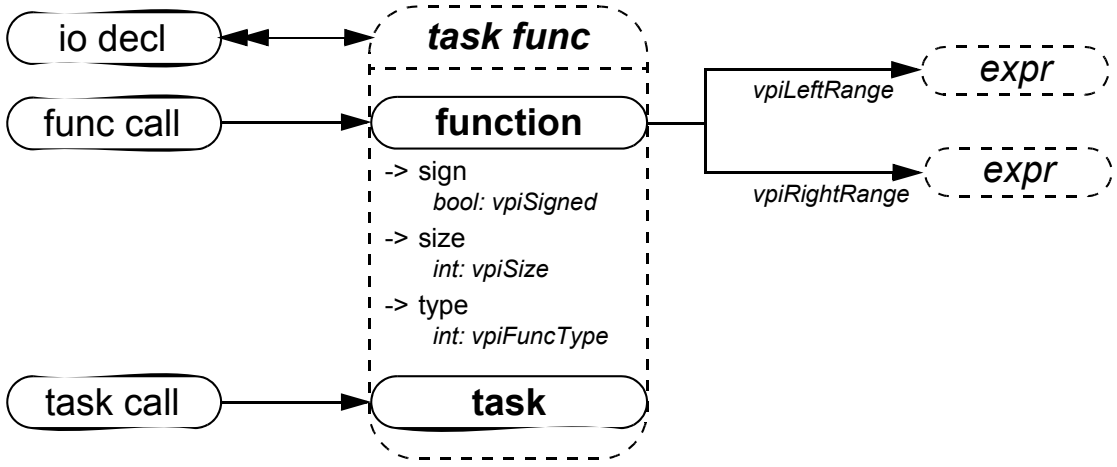
26.6.17 Timing check



Details:

a) For the timing checks in 15.1, the relationship **vpiTchkRefTerm** shall denote the *reference_event* or *controlled_reference_event*, while **vpiTchkDataTerm** shall denote the *data_event*, if any.
b) When iterating over **vpiExpr** from a tchk, the handles returned for a *reference_event*, a *controlled_reference_event*, or a *data_event* shall have the type **vpiTchkTerm**. All other arguments shall have types matching the expression.

26.6.18 Task, function declaration



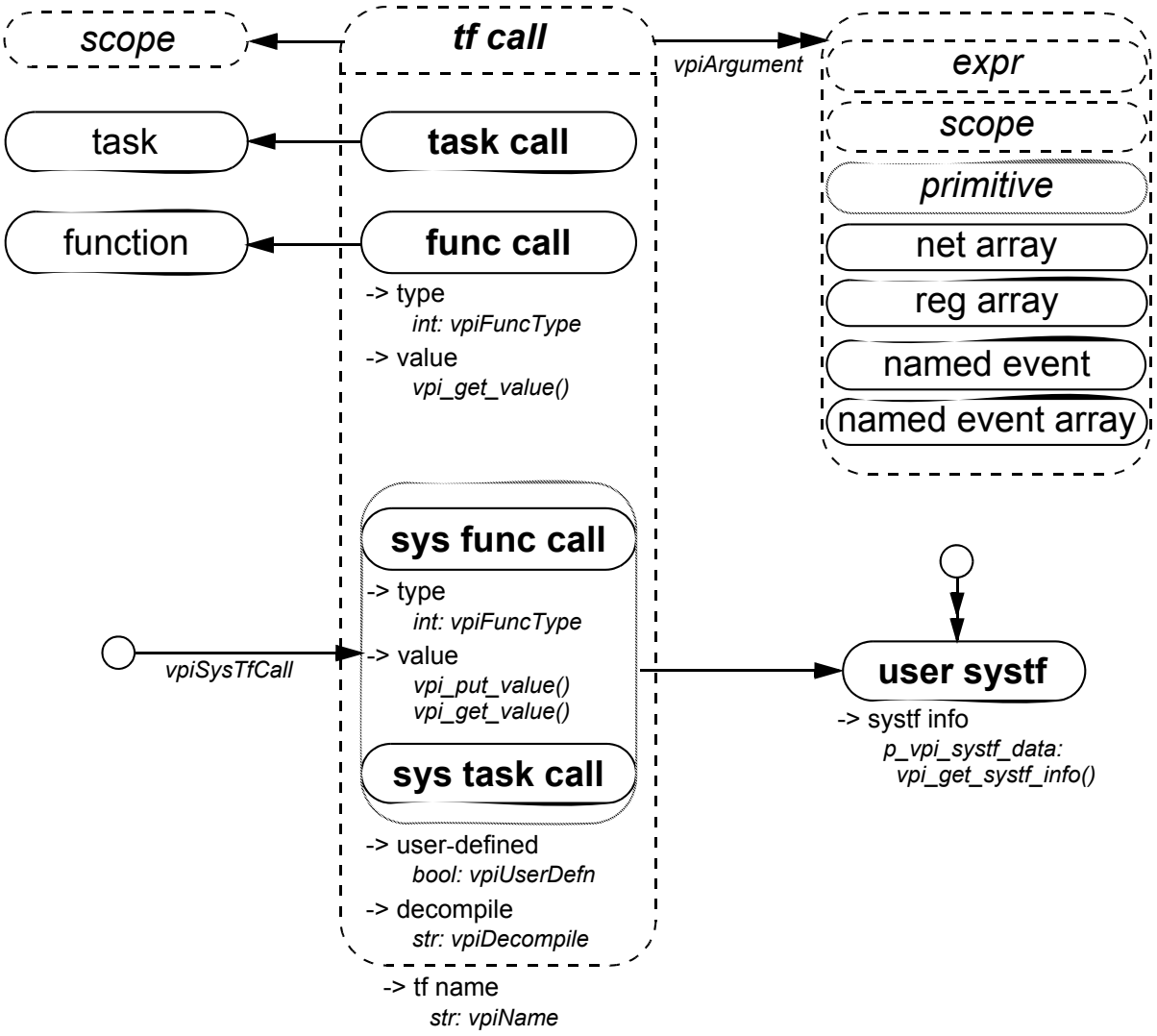
Details:

A Verilog HDL function shall contain an object with the same name, size, and type as the function.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.19 Task/function call



Details:

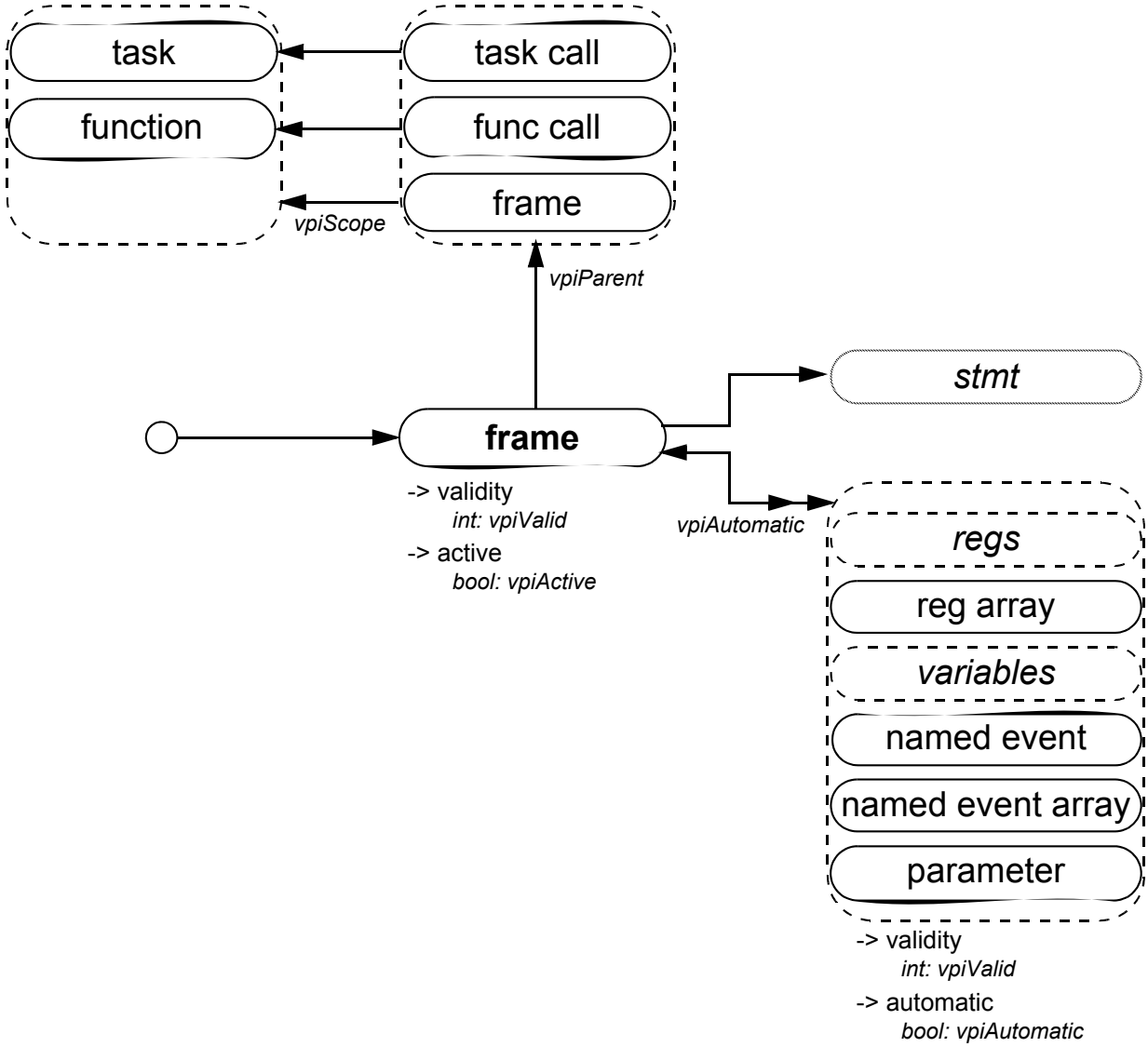
a) The system task/function that invoked an application shall be accessed with **vpi_handle(vpiSysTfCall,** NULL**)**
b) **vpi_get_value()** shall return the current value of the system function.
c) If the **vpiUserDefn** property of a system task/function call is true, then the properties of the corresponding systf object shall be obtained via **vpi_get_systf_info()**.
d) All user-defined system tasks or functions shall be retrieved using **vpi_iterate()** with **vpiUserSystf** as the type argument and with a `NULL` reference argument.
e) Arguments to PLI tasks or functions are not evaluated until an application requests their value. Effectively, the value of any argument is not known until the application asks for it. When an argument is an HDL or system function call, the function cannot be evaluated until the application asks for its value. If the application never asks for the value of the function, it is never evaluated. If the application has a handle to an HDL or system function, it may ask for its value at any time in the simulation. When this happens, the function is called and evaluated at this time.
f) A null argument is an expression with a **vpiType** of **vpiOperation** and a **vpiOpType** of **vpiNullOp**.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

g) The property **vpiDecompile** shall return a string with a functionally equivalent system task/function call to what was in the original HDL. The arguments shall be decompiled using the same manner as any expression is decompiled. See 26.6.26 for a description of expression decompilation.
h) System task/function calls that are protected shall allow iteration over the **vpiArgument** relationship.

26.6.20 Frames

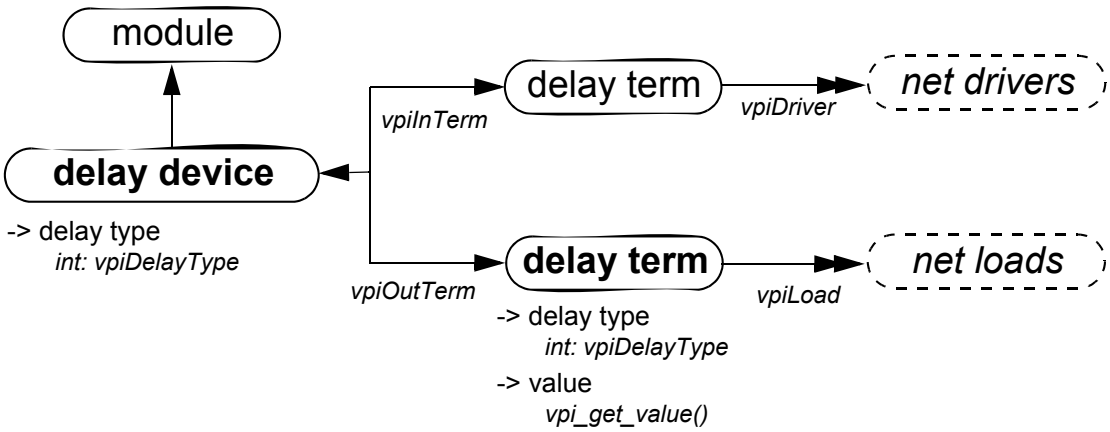


Details:

a) It shall be illegal to place value change callbacks on automatic variables.
b) It shall be illegal to put a value with a delay on automatic variables.
c) There is at most only one active frame at any time. To get a handle to the currently active frame, use **vpi_handle(vpiFrame, `NULL`)**. The **frame**-to-**stmt** transition shall return the currently active statement within the frame.
d) Frame handles must be freed using **vpi_free_object()** once the application no longer needs the handle. If the handle is not freed, it shall continue to exist, even after the frame has completed execution.

Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

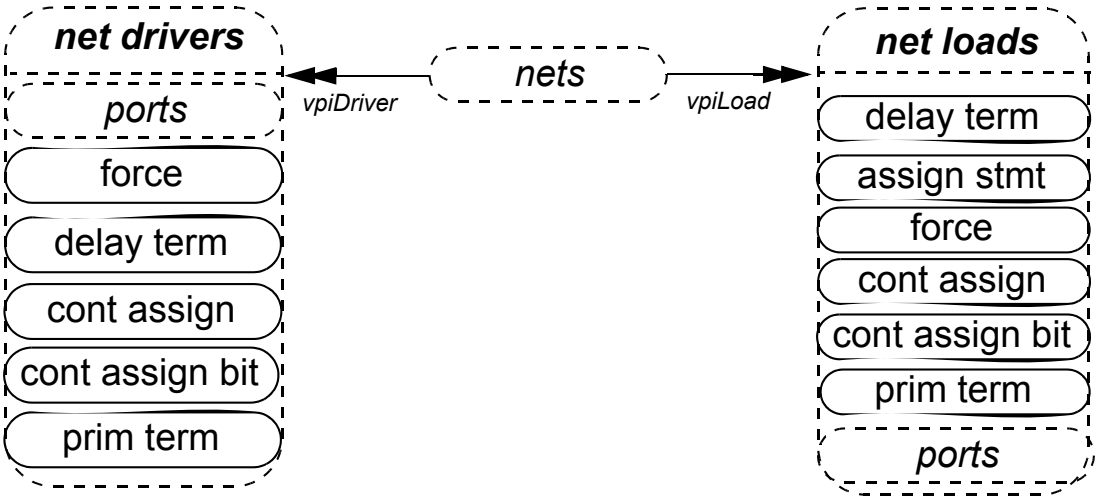
26.6.21 Delay terminals



Details:

a) The value of the input delay term shall change before the delay associated with the delay device.
b) The value of the output delay term shall not change until after the delay has occurred.

26.6.22 Net drivers and loads



Details:

Complex expressions on input ports that are not concatenations shall be considered a load for the net. Iterating on loads for *trinet* in the following example will cause the fourth port of *ram* to be a load:

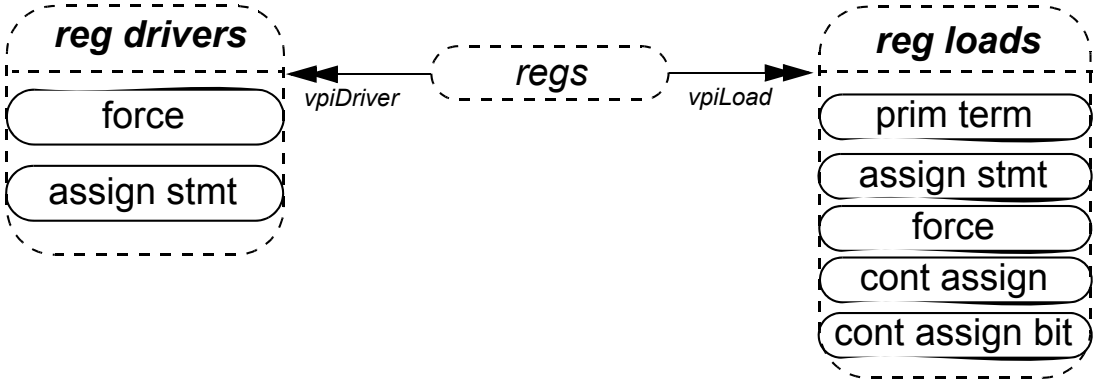
```
module my_module;
   tri trinet;
   ram r0 (a, write, read, !trinet);
endmodule
```

Access to the complex expression shall be available using **vpi_handle(vpiHighConn, portH)** where portH is the handle to the port returned when iterating on loads.

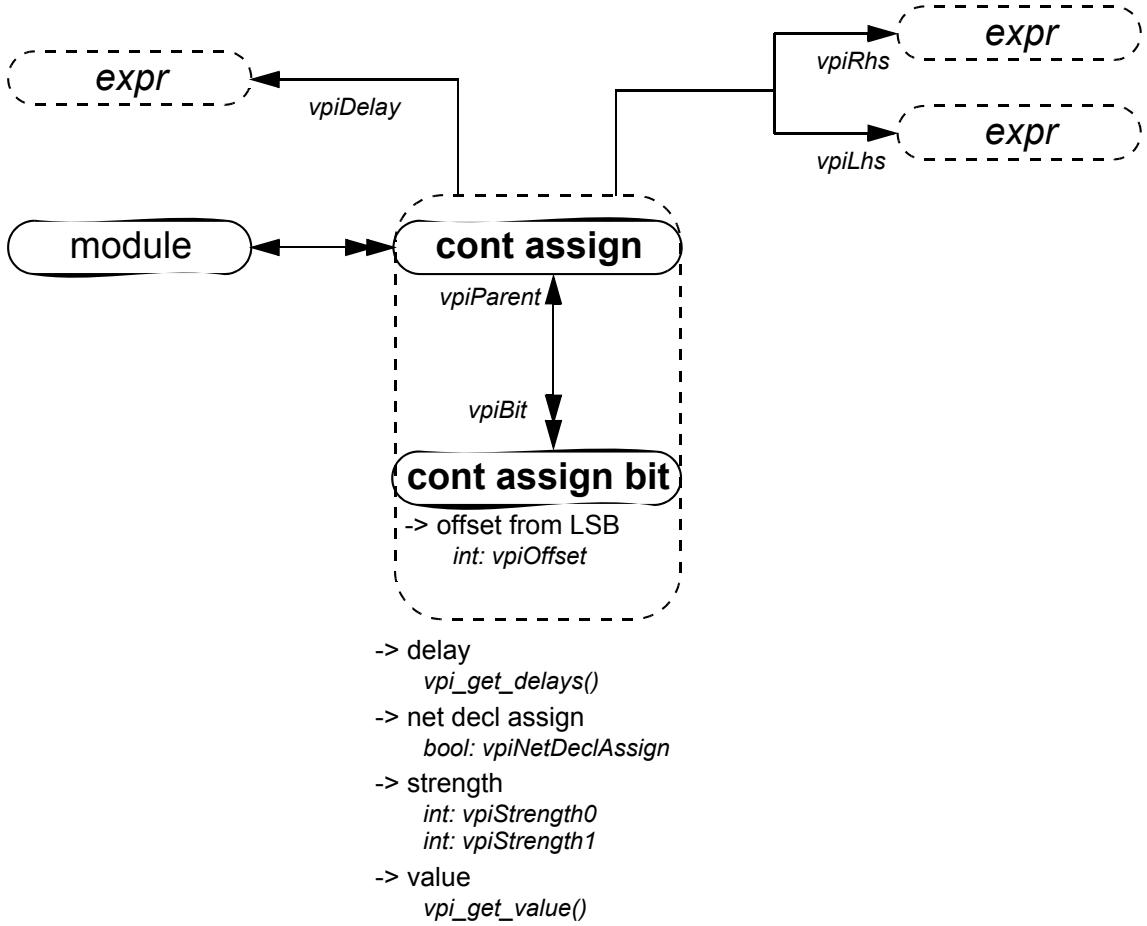
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.23 Reg drivers and loads



26.6.24 Continuous assignment

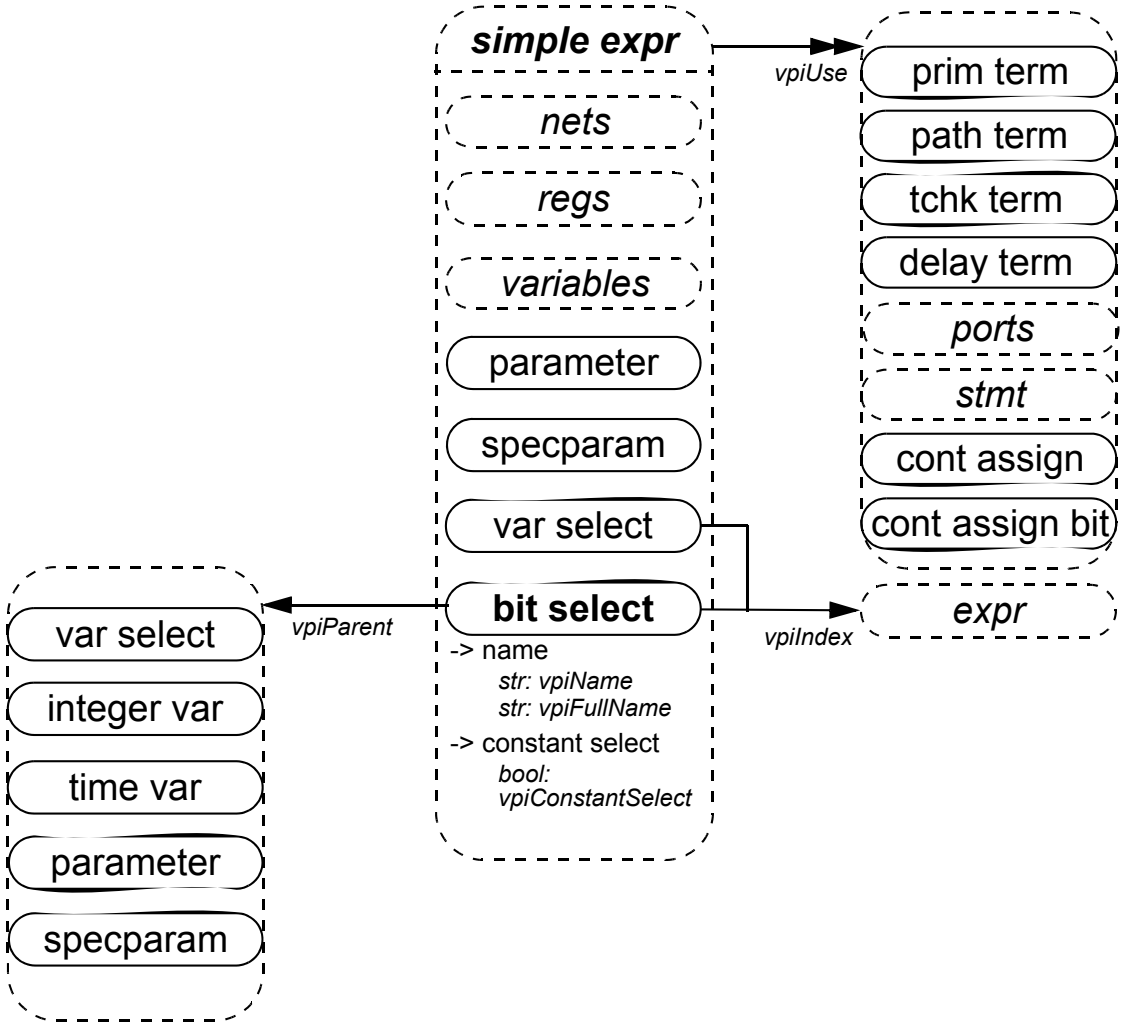


Details:

a) The size of a cont assign bit is always scalar.
b) Callbacks for value changes can be placed onto cont assign or a cont assign bit.
c) **vpiOffset** shall return zero for the least significant bit.

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.25 Simple expressions



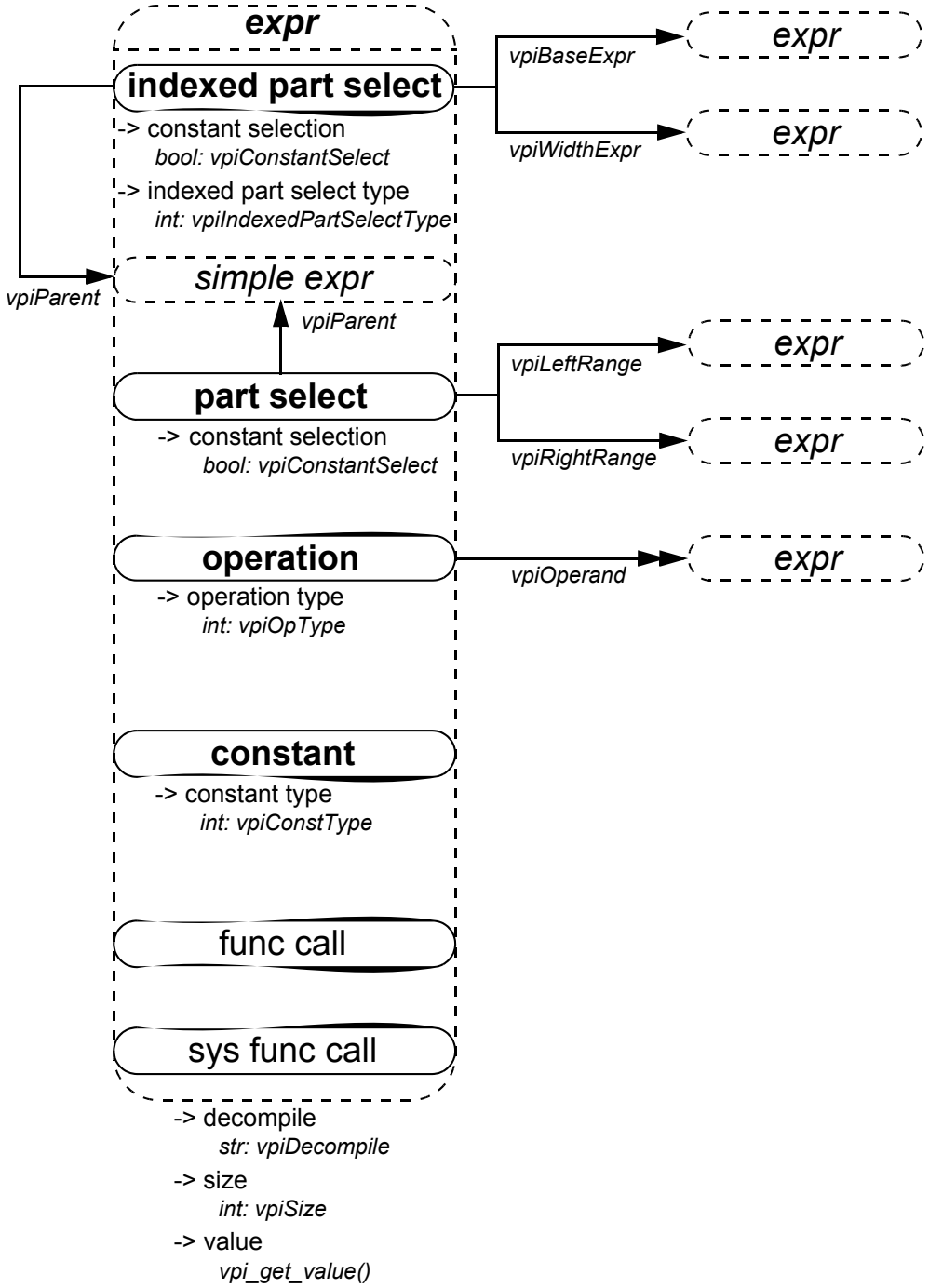
Details:

a) For vectors, the **vpiUse** relationship shall access any use of the vector or part-selects or bit-selects thereof.
b) For bit-selects, the **vpiUse** relationship shall access any specific use of that bit, any use of the parent vector, and any part-select that contains that bit.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.26 Expressions



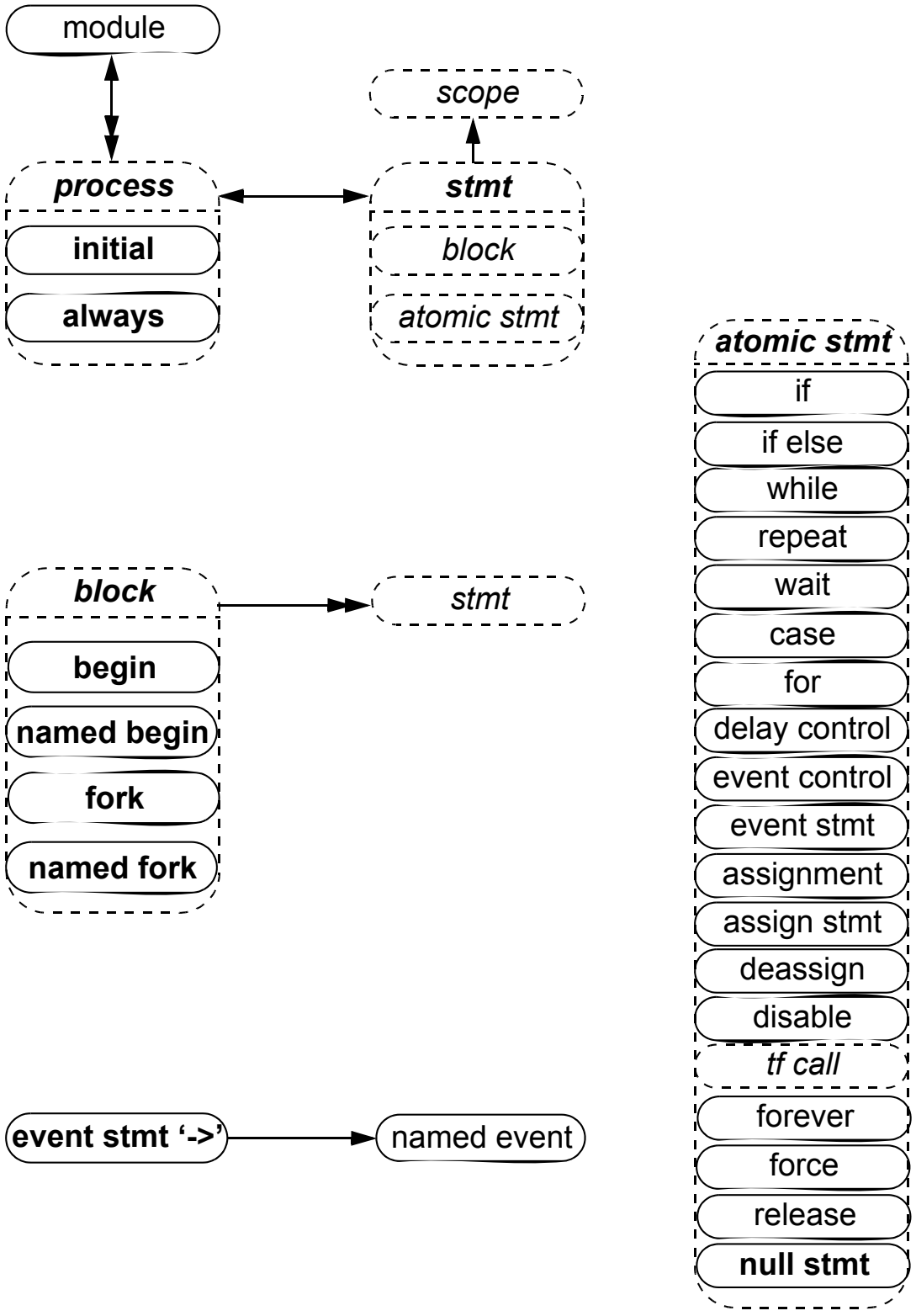
Details:

a) For an operator whose type is **vpiMultiConcatOp**, the first operand shall be the multiplier expression. The remaining operands shall be the expressions within the concatenation.
b) The property **vpiDecompile** shall return a string with a functionally equivalent expression to the original expression within the HDL. Parentheses shall be added only to preserve precedence. Each operand and operator shall be separated by a single space character. No additional white space shall be added due to parentheses.
c) Expressions that are protected shall permit access to the **vpiSize** property.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

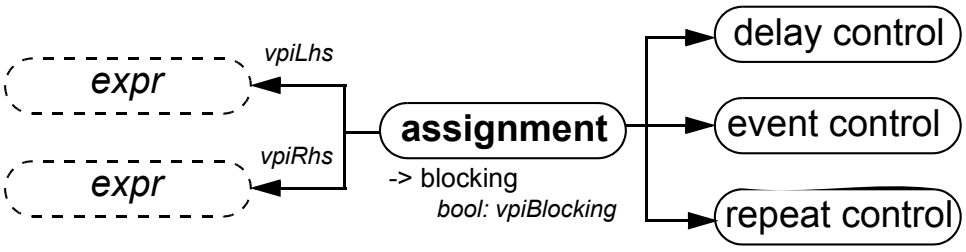
26.6.27 Process, block, statement, event statement



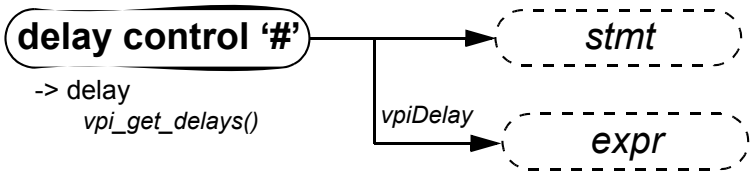
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.28 Assignment



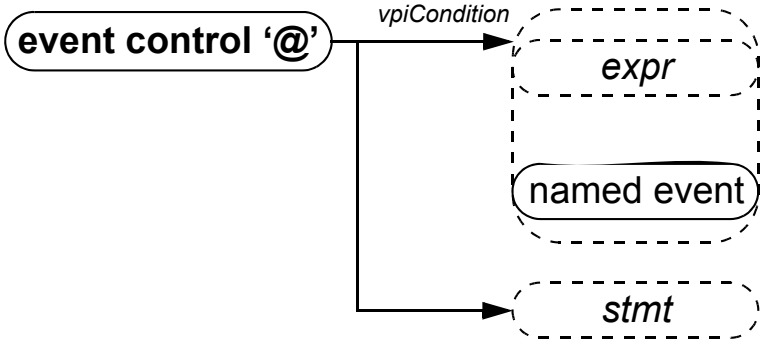
26.6.29 Delay control



Details:

For delay control associated with assignment, the statement shall always be NULL.

26.6.30 Event control



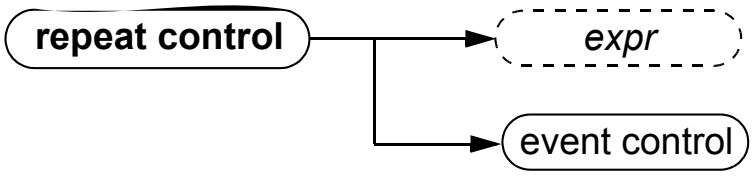
Details:

For event control associated with assignment, the statement shall always be NULL.

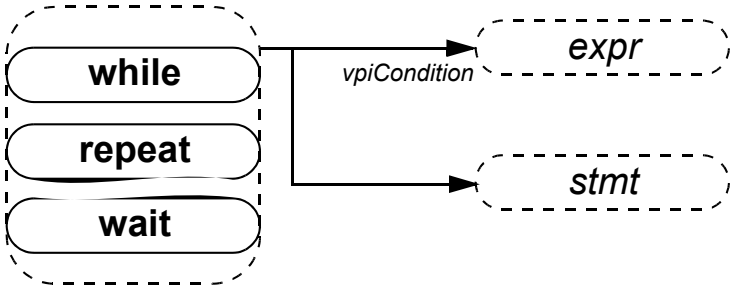
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

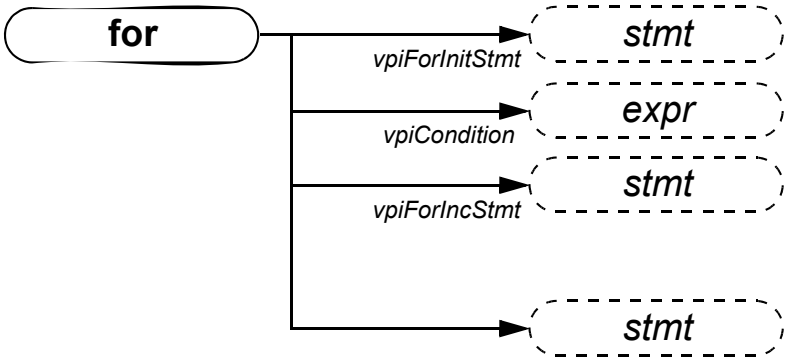
26.6.31 Repeat control



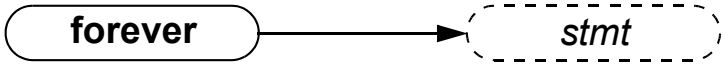
26.6.32 While, repeat, wait



26.6.33 For



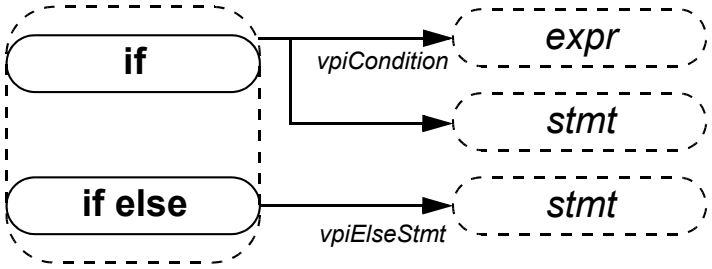
26.6.34 Forever



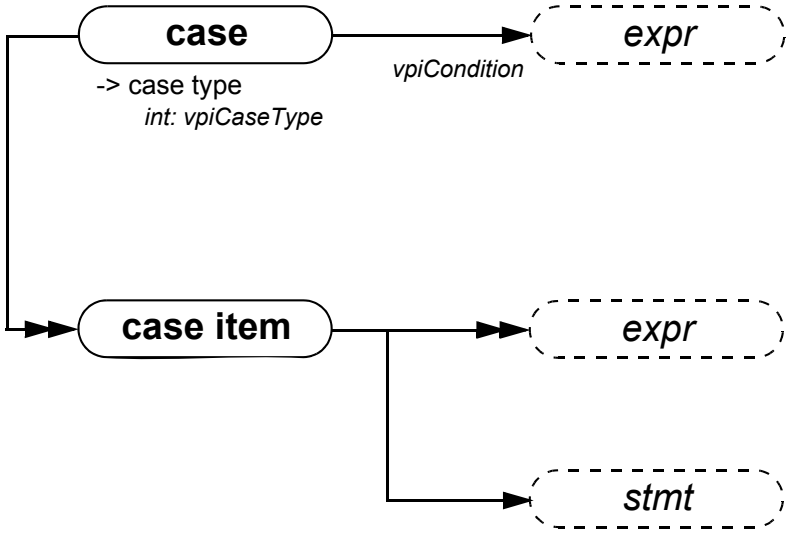
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.35 If, if-else



26.6.36 Case



Details:

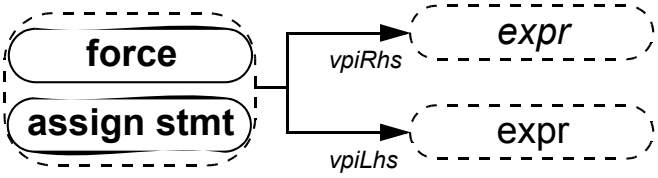
a) The *case item* shall group all case conditions that branch to the same statement.
b) **vpi_iterate()** shall return `NULL` for the default case item because there is no expression with the default case.

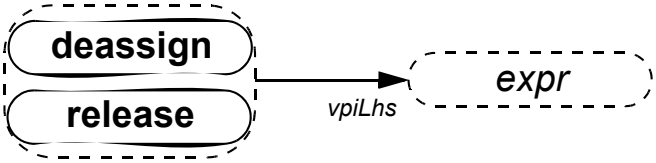
Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---

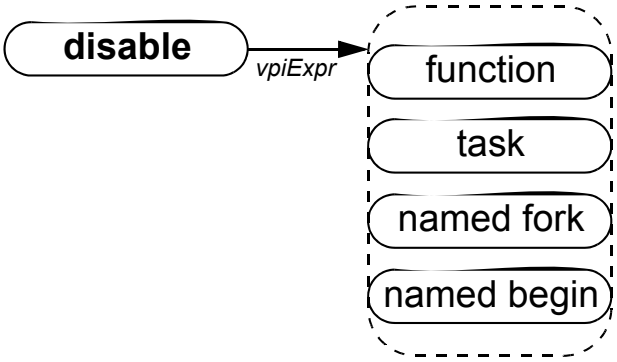
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.37 Assign statement, deassign, force, release





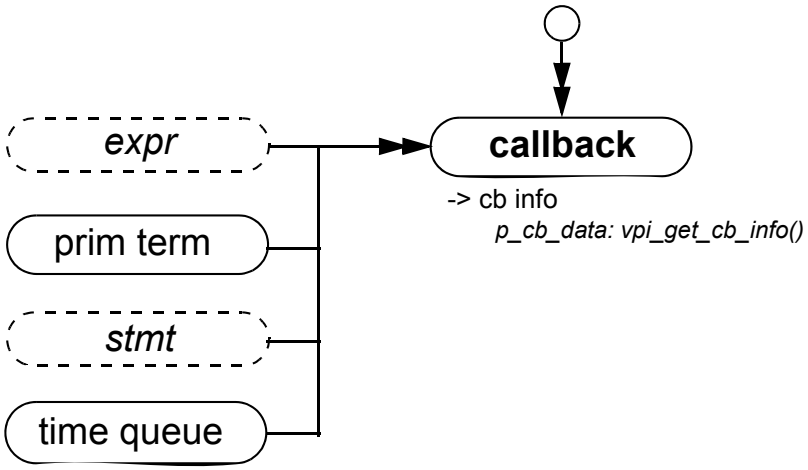
26.6.38 Disable



Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

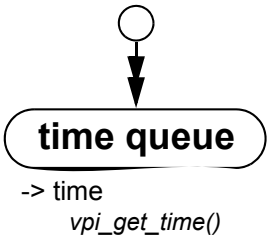
26.6.39 Callback



Details:

a) To get information about the callback object, the routine **vpi_get_cb_info()** can be used.
b) To get callback objects not related to the above objects, the second argument to **vpi_iterate()** shall be `NULL`.

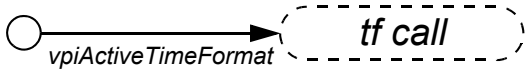
26.6.40 Time queue



Details:

a) The time queue objects shall be returned in increasing order of simulation time.
b) **vpi_iterate()** shall return `NULL` if there is nothing left in the simulation queue.
c) The current time queue shall only be returned as part of the iteration if there are events that precede read only sync.

26.6.41 Active time format



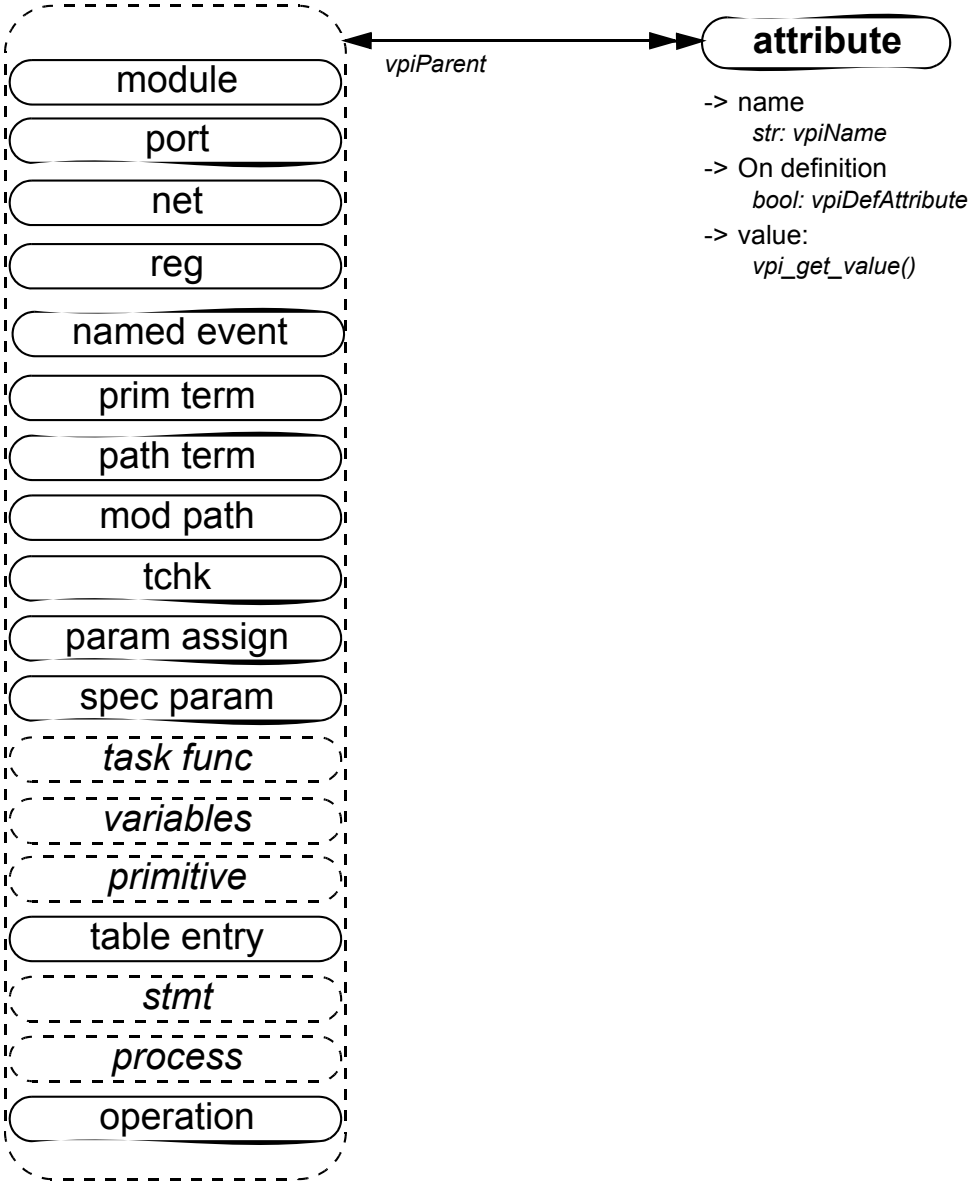
Details:

If **$timeformat()** has not been called, `vpi_handle(vpiActiveTimeFormat,NULL)` shall return a `NULL`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.42 Attributes



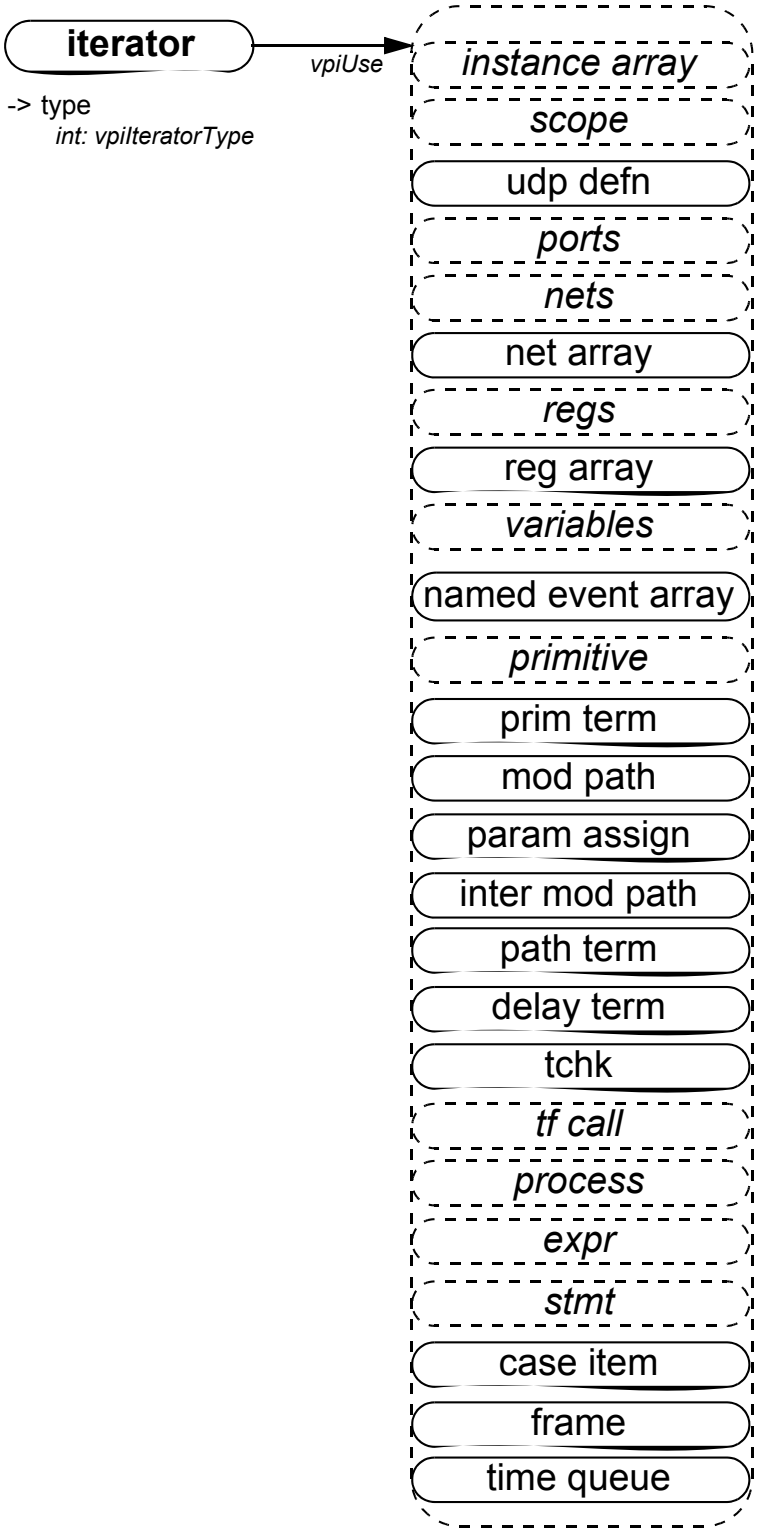
Details:

The property **vpiDefAttribute** shall return true if the attribute was defined on a module as part of the module definition. The property shall return false for attributes defined on a module as part of a module instantiation or for any object other than a module.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.43 Iterator



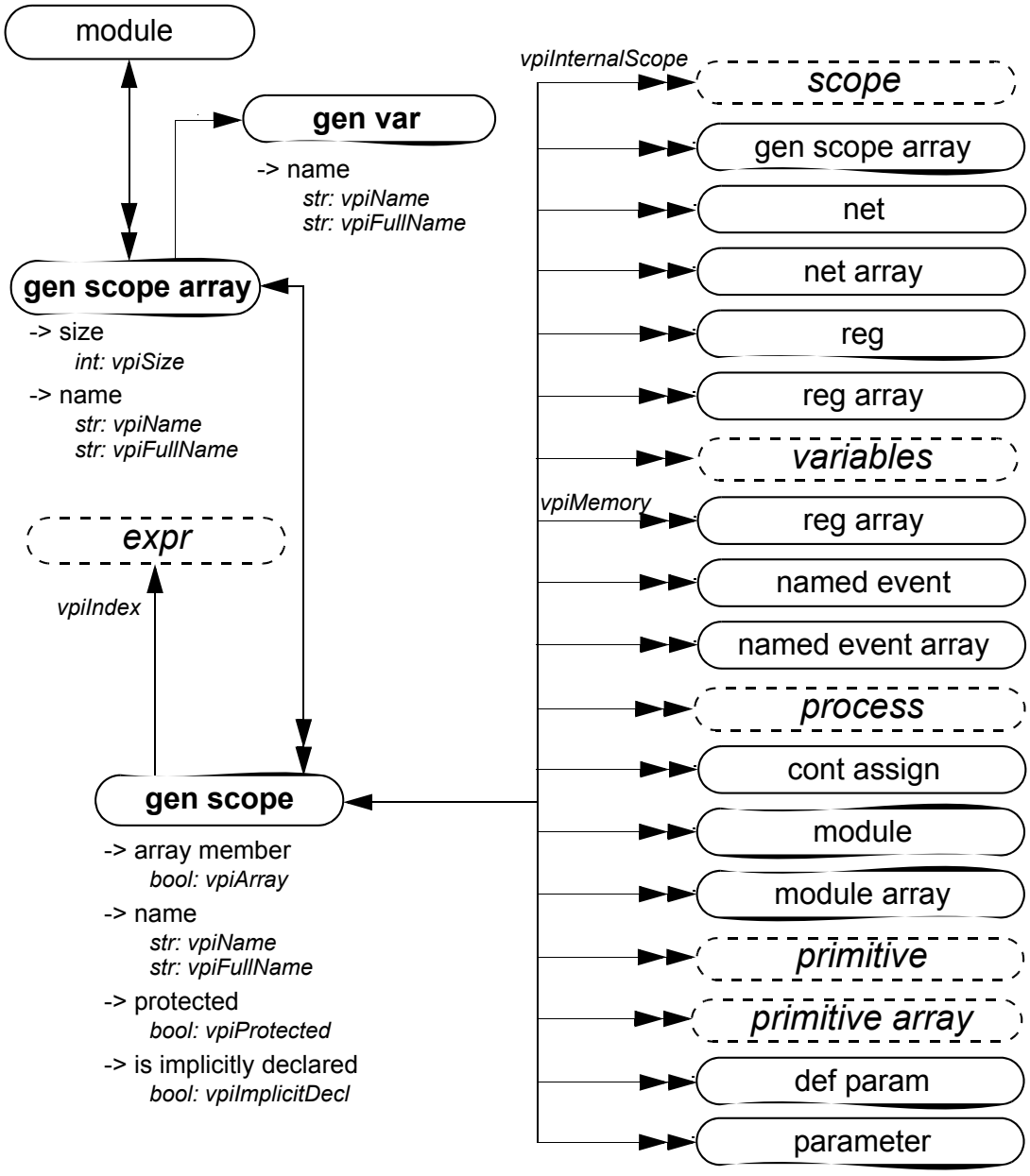
Details:

a) **vpi_handle(vpiUse, iterator_handle)** shall return the reference handle used to create the iterator.
b) It is possible to have a `NULL` reference handle, in which case **vpi_handle(vpiUse, iterator_handle)** shall return `NULL`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

26.6.44 Generates



Details:

a) The size for a genscope array is the number of elements in the array.
b) For unnamed generates, an implicit scope shall be created. Its **vpiImplicitDecl** property shall return TRUE.
c) References to **genvars** within the genscope shall be treated as local parameters.
d) Parameters within the genscope must be local parameters.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

27. VPI routine definitions

This clause describes the VPI routines and explains their function, syntax, and usage. The routines are listed in alphabetical order.

The following conventions are used in the definitions of the PLI routines described in this clause:

— **Synopsis:** A brief description of the PLI routine functionality, intended to be used as a quick reference when searching for PLI routines to perform specific tasks.
— **Syntax:** The exact name of the PLI routine and the order of the arguments passed to the routine.
— **Returns:** The definition of the value returned when the PLI routine is called, along with a brief description of what the value represents. The return definition contains the following fields:
  — **Type:** The data type of the C value that is returned. The data type is either a standard ANSI C type or a special type defined within the PLI.
  — **Description:** A brief description of what the value represents.
— **Arguments:** The definition of the arguments passed with a call to the PLI routine. The argument definition contains the following fields:
  — **Type:** The data type of the C values that are passed as arguments. The data type is either a standard ANSI C type or a special type defined within the PLI.
  — **Name:** The name of the argument used in the syntax definition.
  — **Description:** A brief description of what the value represents.

  All arguments shall be considered mandatory unless specifically noted in the definition of the PLI routine.
— **Related routines:** A list of PLI routines that are typically used with, or provide similar functionality to, the PLI routine being defined. This list is provided as a convenience to facilitate finding information in this standard. It is not intended to be all-inclusive, and it does not imply that the related routines have to be used.

27.1 vpi_chk_error()

vpi_chk_error()			
Synopsis:	Retrieve information about VPI routine errors.		
Syntax:	`vpi_chk_error(error_info_p)`		
Returns:	**Type**	**Description**	
	PLI_INT32	The error severity level if the previous VPI routine call resulted in an error; 0 (false) if no error occurred.	
Arguments:	**Type**	**Name**	**Description**
	p_vpi_error_info	error_info_p	Pointer to a structure containing error information.
Related routines:			

The VPI routine **vpi_chk_error()** shall return an integer constant representing an error severity level if the previous call to a VPI routine resulted in an error. The error constants are shown in Table 27-1. If the previous call to a VPI routine did not result in an error, then **vpi_chk_error()** shall return **0** (false). The error

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

status shall be reset by any VPI routine call except **vpi_chk_error()**. Calling **vpi_chk_error()** shall have no effect on the error status.

Table 27-1—Return error constants for vpi_chk_error()

Error constant	Severity level
vpiNotice	Lowest severity
vpiWarning	
vpiError	
vpiSystem	
vpiInternal	Highest severity

If an error occurred, the `s_vpi_error_info` structure shall contain information about the error. If the error information is not needed, a `NULL` can be passed to the routine. The `s_vpi_error_info` structure used by **vpi_chk_error()** is defined in `vpi_user.h` and is listed in Figure 27-1.

```
typedef struct t_vpi_error_info
{
  PLI_INT32 state;          /* vpi[Compile,PLI,Run] */
  PLI_INT32 level;          /* vpi[Notice,Warning,Error,System,Internal] */
  PLI_BYTE8 *message;
  PLI_BYTE8 *product;
  PLI_BYTE8 *code;
  PLI_BYTE8 *file;
  PLI_INT32 line;
} s_vpi_error_info, *p_vpi_error_info;
```

Figure 27-1—s_vpi_error_info structure definition

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

27.2 vpi_compare_objects()

vpi_compare_objects()			
Synopsis:	Compare two handles to determine whether they reference the same object.		
Syntax:	`vpi_compare_objects(obj1, obj2)`		
	Type	**Description**	
Returns:	PLI_INT32	1 (true) if the two handles refer to the same object; 0 (false) otherwise.	
	Type	**Name**	**Description**
Arguments:	vpiHandle	obj1	Handle to an object.
	vpiHandle	obj2	Handle to an object.
Related routines:			

The VPI routine **vpi_compare_objects()** shall return 1 (true) if the two handles refer to the same object. Otherwise, 0 (false) shall be returned. Handle equivalence cannot be determined with a C '==' comparison.

27.3 vpi_control()

vpi_control()			
Synopsis:	Pass information from the application code to the simulator.		
Syntax:	`vpi_control(operation, varargs)`		
	Type	**Description**	
Returns:	PLI_INT32	1 (true) if successful; 0 (false) on a failure.	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	operation	Select type of operation.
		varargs	Variable number of operation-specific arguments.
Related routines:			

The VPI routine **vpi_control()** shall pass information from a user PLI application to a Verilog software tool, such as a simulator. The following control constants are defined as part of the VPI standard:

vpiStop Causes the **$stop** built-in Verilog system task to be executed upon return of the application routine. This operation shall be passed one additional integer argument, which is the same as the diagnostic message level argument passed to **$stop** (see 17.4.2).

vpiFinish Causes the **$finish** built-in Verilog system task to be executed upon return of the application routine. This operation shall be passed one additional integer argument, which is the same as the diagnostic message level argument passed to **$finish** (see 17.4.1).

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

vpiReset Causes the **$reset** built-in Verilog system task to be executed upon return of the application routine. This operation shall be passed three additional integer arguments: **stop_value**, **reset_value**, and **diagnostic_level**, which are the same values passed to the **$reset** system task (see C.7).

vpiSetInteractiveScope Causes a tool's interactive scope to be immediately changed to a new scope. This operation shall be passed one additional argument, which is a `vpiHandle` object within the **vpiScope** class.

27.4 vpi_flush()

vpi_flush()			
Synopsis:	Flushes the data from the simulator output channel and log file output buffers.		
Syntax:	`vpi_flush()`		
	Type	**Description**	
Returns:	PLI_INT32	0 if successful; nonzero if unsuccessful.	
	Type	**Name**	**Description**
Arguments:	None		
Related routines:	Use vpi_printf() to write a finite number of arguments to the simulator output channel and log file. Use vpi_vprintf() to write a variable number of arguments to the simulator output channel and log file. Use vpi_mcd_printf() to write one or more opened files.		

The routine **vpi_flush()** shall flush the output buffers for the simulator's output channel and current log file.

27.5 vpi_free_object()

vpi_free_object()			
Synopsis:	Free memory allocated by VPI routines.		
Syntax:	`vpi_free_object(obj)`		
	Type	**Description**	
Returns:	PLI_INT32	1 (true) on success; 0 (false) on failure.	
	Type	**Name**	**Description**
Arguments:	vpiHandle	obj	Handle of an object.
Related routines:			

The VPI routine **vpi_free_object()** shall free memory allocated for objects. It shall generally be used to free memory created for iterator objects. The iterator object shall automatically be freed when **vpi_scan()** returns `NULL` because it has either completed an object traversal or encountered an error condition. If neither of these conditions occurs (which can happen if the code breaks out of an iteration loop before it has scanned every object), **vpi_free_object()** should be called to free any memory allocated for the iterator. This routine can also optionally be used for implementations that have to allocate memory for objects. The routine shall return 1 (true) on success and 0 (false) on failure.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

27.6 vpi_get()

vpi_get()			
Synopsis:	Get the value of an integer or boolean property of an object.		
Syntax:	`vpi_get(prop, obj)`		
	Type	**Description**	
Returns:	PLI_INT32	Value of an integer or boolean property.	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	prop	An integer constant representing the property of an object for which to obtain a value.
	vpiHandle	obj	Handle to an object.
Related routines:	Use vpi_get_str() to get string properties.		

The VPI routine **vpi_get()** shall return the value of integer and boolean object properties. These properties shall be of type `PLI_INT32`. Boolean properties shall have a value of 1 for TRUE and 0 for FALSE. For integer object properties such as **vpiSize**, any integer shall be returned. For integer object properties that return a defined value, see Annex G for the value that shall be returned. For object property **vpiTimeUnit** or **vpiTimePrecision**, if the object is `NULL`, then the simulation time unit shall be returned. Unless otherwise specified, calling **vpi_get()** for a protected object shall be an error. Should an error occur, **vpi_get()** shall return **vpiUndefined**.

27.7 vpi_get_cb_info()

vpi_get_cb_info()			
Synopsis:	Retrieve information about a simulation-related callback.		
Syntax:	`vpi_get_cb_info(obj, cb_data_p)`		
	Type	**Description**	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	vpiHandle	obj	Handle to a simulation-related callback.
	p_cb_data	cb_data_p	Pointer to a structure containing callback information.
Related routines:	Use vpi_get_systf_info() to retrieve information about a system task/function callback.		

The VPI routine **vpi_get_cb_info()** shall return information about a simulation-related callback in an `s_cb_data` structure. The memory for this structure shall be allocated by the application.

The `s_cb_data` structure used by **vpi_get_cb_info()** is defined in `vpi_user.h` and is listed in Figure 27-2.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
typedef struct t_cb_data
{
  PLI_INT32    reason;           /* callback reason */
  PLI_INT32    (*cb_rtn)(struct t_cb_data *); /* call routine */
  vpiHandle    obj;              /* trigger object */
  p_vpi_time   time;             /* callback time */
  p_vpi_value  value;            /* trigger object value */
  PLI_INT32    index;            /* index of the memory word or var select
                                    that changed */
  PLI_BYTE8   *user_data;
} s_cb_data, *p_cb_data;
```

Figure 27-2—s_cb_data structure definition

27.8 vpi_get_data()

vpi_get_data()			
Synopsis:	Get data from an implementation's save/restart location.		
Syntax:	`vpi_get_data(id, dataLoc, numOfBytes)`		
	Type	**Description**	
Returns:	PLI_INT32	The number of bytes retrieved.	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	id	A save/restart ID returned from vpi_get(vpiSaveRestartID, NULL) .
	PLI_BYTE8 *	dataLoc	Address of application-allocated storage.
	PLI_INT32	numOfBytes	Number of bytes to be retrieved from save/restart location.
Related routines:	Use vpi_put_data() to write saved data.		

The routine shall place *numOfBytes* of data into the memory location pointed to by *dataLoc* from a simulation's save/restart location. This memory location has to be properly allocated by the application. The first call for a given *id* will retrieve the data starting at what was placed into the save/restart location with the first call to **vpi_put_data()** for a given *id*. The return value shall be the number of bytes retrieved. On a failure, the return value shall be 0. Each subsequent call shall start retrieving data where the last call left off. It shall be a warning for an application to retrieve more data than were placed into the simulation save/restart location for a given *id*. In this case, the *dataLoc* shall be filled with the data that are left for the given *id*, and the remaining bytes shall be filled with "\0". The return value shall be the actual number of bytes retrieved. It shall be acceptable for an application to retrieve less data than were stored for a given *id* with **vpi_put_data()**. This routine can only be called from an application routine that has been called for reason **cbStartOfRestart** or **cbEndOfRestart**. The recommended way to get the "id" for **vpi_get_data()** is to pass it as the value for *user_data* when registering for **cbStartOfRestart** or **cbEndOfRestart** from the **cbStartOfSave** or **cbEndOfSave** application routine. An application can get the path to the simulation's save/restart location by calling **vpi_get_str(vpiSaveRestartLocation, NULL)** from an application routine that has been called for reason **cbStartOfRestart** or **cbEndOfRestart**.

For an example of **vpi_get_data()**, see 27.29.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

27.9 vpi_get_delays()

vpi_get_delays()			
Synopsis:	Retrieve the delays or pulse limits of an object.		
Syntax:	`vpi_get_delays(obj, delay_p)`		
	Type	**Description**	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	vpiHandle	obj	Handle to an object.
	p_vpi_delay	delay_p	Pointer to a structure containing delay information.
Related routines:	Use vpi_put_delays() to set the delays or timing limits of an object.		

The VPI routine **vpi_get_delays()** shall retrieve the delays or pulse limits of an object and place them in an `s_vpi_delay` structure that has been allocated by the application. The format of the delay information shall be controlled by the *time_type* flag in the `s_vpi_delay` structure. This routine shall ignore the value of the *type* flag in the `s_vpi_time` structure.

The `s_vpi_delay` and `s_vpi_time` structures used by both **vpi_get_delays()** and **vpi_put_delays()** are defined in `vpi_user.h` and are listed in Figure 27-3 and Figure 27-4.

```
typedef struct t_vpi_delay
{
  struct t_vpi_time *da;          /* pointer to application-allocated
                                     array of delay values */
  PLI_INT32 no_of_delays;         /* number of delays */
  PLI_INT32 time_type;            /* [vpiScaledRealTime, vpiSimTime,
                                     or vpiSuppressTime] */
  PLI_INT32 mtm_flag;             /* true for mtm values */
  PLI_INT32 append_flag;          /* true for append */
  PLI_INT32 pulsere_flag;         /* true for pulsere values */
} s_vpi_delay, *p_vpi_delay;
```

Figure 27-3—s_vpi_delay structure definition

```
typedef struct t_vpi_time
{
  PLI_INT32  type;          /* [vpiScaledRealTime, vpiSimTime,
                               vpiSuppressTime] */
  PLI_UINT32 high, low;     /* for vpiSimTime */
  double     real;          /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 27-4—s_vpi_time structure definition

The *da* field of the `s_vpi_delay` structure shall be an application-allocated array of `s_vpi_time` structures. This array shall store delay values returned by **vpi_get_delays()**. The number of elements in this array shall be determined by the following:

Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

— The number of delays to be retrieved
— The **mtm_flag** setting
— The **pulsere_flag** setting

The number of delays to be retrieved shall be set in the *no_of_delays* field of the `s_vpi_delay` structure. Legal values for the number of delays shall be determined by the type of object:

— For primitive objects, the *no_of_delays* value shall be 2 or 3.
— For path delay objects, the *no_of_delays* value shall be 1, 2, 3, 6, or 12.
— For timing check objects, the *no_of_delays* value shall match the number of limits existing in the timing check.
— For intermodule path objects, the *no_of_delays* value shall be 2 or 3.

The application-allocated `s_vpi_delay` array shall contain delays in the same order in which they occur in the Verilog HDL description. The number of elements for each delay shall be determined by the flags **mtm_flag** and **pulsere_flag**, as shown in Table 27-2.

Table 27-2—Size of the s_vpi_delay->da array

Flag values	**Number of `s_vpi_time` array elements required for** *s_vpi_delay->da*	**Order in which delay elements shall be filled**

mtm_flag = `FALSE`
pulsere_flag = `FALSE`
	no_of_delays	`1st delay: da[0] -> 1st delay`
		`2nd delay: da[1] -> 2nd delay`
		`...`

mtm_flag = `TRUE`
pulsere_flag = `FALSE`
	3 * *no_of_delays*	`1st delay: da[0] -> min delay`
		`da[1] -> typ delay`
		`da[2] -> max delay`
		`2nd delay: ...`

mtm_flag = `FALSE`
pulsere_flag = `TRUE`
	3 * *no_of_delays*	`1st delay: da[0] -> delay`
		`da[1] -> reject limit`
		`da[2] -> error limit`
		`2nd delay element: ...`

mtm_flag = `TRUE`
pulsere_flag = `TRUE`
	9 * *no_of_delays*	`1st delay: da[0] -> min delay`
		`da[1] -> typ delay`
		`da[2] -> max delay`
		`da[3] -> min reject`
		`da[4] -> typ reject`
		`da[5] -> max reject`
		`da[6] -> min error`
		`da[7] -> typ error`
		`da[8] -> max error`
		`2nd delay: ...`

The delay structure has to be allocated before passing a pointer to **vpi_get_delays()**. In the following example, a static structure, **prim_da**, is allocated for use by each call to the **vpi_get_delays()** function:

```
display_prim_delays(prim)
vpiHandle prim;

{
   static s_vpi_time prim_da[3];
   static s_vpi_delay delay_s = {NULL, 3, vpiScaledRealTime};
   static p_vpi_delay delay_p = &delay_s;
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
    delay_s.da = prim_da;
    vpi_get_delays(prim, delay_p);
    vpi_printf("Delays for primitive %s: %6.2f %6.2f %6.2f\n",
        vpi_get_str(vpiFullName, prim)
        delay_p->da[0].real, delay_p->da[1].real, delay_p->da[2].real);
}
```

27.10 vpi_get_str()

vpi_get_str()			
Synopsis:	Get the value of a string property of an object.		
Syntax:	`vpi_get_str(prop, obj)`		
	Type	**Description**	
Returns:	PLI_BYTE8 *	Pointer to a character string containing the property value.	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	prop	An integer constant representing the property of an object for which to obtain a value.
	vpiHandle	obj	Handle to an object.
Related routines:	Use vpi_get() to get integer and boolean properties.		

The VPI routine **vpi_get_str()** shall return string property values. The string shall be placed in a temporary buffer that shall be used by every call to this routine. If the string is to be used after a subsequent call, the string should be copied to another location. A different string buffer shall be used for string values returned through the `s_vpi_value` structure. Unless otherwise specified, calling **vpi_get_str()** for a protected object shall be an error.

The following example illustrates the usage of **vpi_get_str()**:

```
vpiHandle mod = vpi_handle_by_name("top.mod1",NULL);
vpi_printf ("Module top.mod1 is an instance of %s\n",
      vpi_get_str(vpiDefName, mod));
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

27.11 vpi_get_systf_info()

vpi_get_systf_info()			
Synopsis:	Retrieve information about a user-defined system task/function callback.		
Syntax:	`vpi_get_systf_info(obj, systf_data_p)`		
Returns:	**Type**	**Description**	
	void		
Arguments:	**Type**	**Name**	**Description**
	vpiHandle	obj	Handle to a system task/function callback.
	p_vpi_systf_data	systf_data_p	Pointer to a structure containing callback information.
Related routines:	Use vpi_get_cb_info() to retrieve information about a simulation-related callback.		

The VPI routine **vpi_get_systf_info()** shall return information about a user-defined system task/function callback in an `s_vpi_systf_data` structure. The memory for this structure shall be allocated by the application.

The `s_vpi_systf_data` structure used by **vpi_get_systf_info()** is defined in `vpi_user.h` and is listed in Figure 27-5.

```
typedef struct t_vpi_systf_data
{
         PLI_INT32 type;         /* vpiSysTask, vpiSysFunc */
         PLI_INT32 sysfunctype;  /* vpiSysTask, vpi[Int,Real,Time,Sized,
                                                    SizedSigned]Func */
         PLI_BYTE8 *tfname;      /* first character must be '$' */
         PLI_INT32 (*calltf)(PLI_BYTE8 *);
         PLI_INT32 (*compiletf)(PLI_BYTE8 *);
         PLI_INT32 (*sizetf)(PLI_BYTE8 *);    /* for sized function
                                                 callbacks only */
         PLI_BYTE8 *user_data;
} s_vpi_systf_data, *p_vpi_systf_data;
```

Figure 27-5—s_vpi_systf_data structure definition

Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

27.12 vpi_get_time()

vpi_get_time()			
Synopsis:	Retrieve the current simulation time.		
Syntax:	`vpi_get_time(obj, time_p)`		
Returns:	**Type**	**Description**	
	void		
Arguments:	**Type**	**Name**	**Description**
	vpiHandle	obj	Handle to an object.
	p_vpi_time	time_p	Pointer to a structure containing time information.
Related routines:			

The VPI routine **vpi_get_time()** shall retrieve the current simulation time, using the time scale of the object. If *obj* is `NULL`, the simulation time is retrieved using the simulation time unit. If *obj* is a time queue object, the scheduled time of the future event is retrieved using the simulation time unit. The *time_p->type* field shall be set to indicate if scaled real or simulation time is desired. The memory for the *time_p* structure shall be allocated by the application.

The `s_vpi_time` structure used by **vpi_get_time()** is defined in `vpi_user.h` and is listed in Figure 27-6 [this is the same time structure as used by **vpi_put_value()**].

```
typedef struct t_vpi_time
{
  PLI_INT32  type;          /* [vpiScaledRealTime, vpiSimTime,
                               vpiSuppressTime] */
  PLI_UINT32 high, low;     /* for vpiSimTime */
  double     real;          /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 27-6—s_vpi_time structure definition

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

27.13 vpi_get_userdata()

vpi_get_userdata()			
Synopsis:	Get user-data value from an implementation's system task/function instance storage location.		
Syntax:	`vpi_get_userdata(obj)`		
	Type	**Description**	
Returns:	void *	User-data value associated with a system task instance or system function instance.	
	Type	**Name**	**Description**
Arguments:	vpiHandle	obj	Handle to a system task instance or system function instance.
Related routines:	Use vpi_put_userdata() to write data into the user-data storage area.		

This routine shall return the value of the user data associated with a previous call to **vpi_put_userdata()** for a user-defined system task/function call handle. If no user data had been previously associated with the object or if the routine fails, the return value shall be **NULL**.

After a restart or a reset, subsequent calls to **vpi_get_userdata()** shall return **NULL**. It is the application's responsibility to save the data during a save using **vpi_put_data()** and to then retrieve them using **vpi_get_data()**. The user-data field can be set up again during or after callbacks of type **cbEndOfRestart** or **cbEndOfReset**.

27.14 vpi_get_value()

vpi_get_value()			
Synopsis:	Retrieve the simulation value of an object.		
Syntax:	`vpi_get_value(obj, value_p)`		
	Type	**Description**	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	vpiHandle	obj	Handle to an expression.
	p_vpi_value	value_p	Pointer to a structure containing value information.
Related routines:	Use vpi_put_value() to set the value of an object.		

The VPI routine **vpi_get_value()** shall retrieve the simulation value of VPI objects. The value shall be placed in an `s_vpi_value` structure, which has been allocated by the application. The format of the value shall be set by the *format* field of the structure.

When the *format* field is **vpiObjTypeVal**, the routine shall fill in the value and change the *format* field based on the object type, as follows:

— For an integer, **vpiIntVal**

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

— For a real, **vpiRealVal**
— For a scalar, either **vpiScalar** or **vpiStrength**
— For a time variable, **vpiTimeVal** with **vpiSimTime**
— For a vector, **vpiVectorVal**

The buffer this routine uses for string values shall be different from the buffer that **vpi_get_str()** shall use. The string buffer used by **vpi_get_value()** is overwritten with each call. If the value is needed, it should be saved by the application.

The `s_vpi_value`, `s_vpi_vecval`, and `s_vpi_strengthval` structures used by **vpi_get_value()** are defined in `vpi_user.h` and are listed in Figure 27-7, Figure 27-8, and Figure 27-9.

```
typedef struct t_vpi_value
{
  PLI_INT32 format; /* vpi[[Bin,Oct,Dec,Hex]Str,Scalar,Int,Real,String,
                           Vector,Strength,Suppress,Time,ObjType]Val */
  union
    {
      PLI_BYTE8 *str;                     /* string value */
      PLI_INT32 scalar;                   /* vpi[0,1,X,Z] */
      PLI_INT32 integer;                  /* integer value */
      double    real;                     /* real value */
      struct t_vpi_time *time;            /* time value */
      struct t_vpi_vecval *vector;        /* vector value */
      struct t_vpi_strengthval *strength; /* strength value */
      PLI_BYTE8 *misc;                    /* ...other */
    } value;
} s_vpi_value, *p_vpi_value;
```

Figure 27-7—s_vpi_value structure definition

```
typedef struct t_vpi_vecval
{
  /* following fields are repeated enough times to contain vector */
  PLI_INT32 aval, bval;     /* bit encoding: ab: 00=0, 10=1, 11=X, 01=Z */
} s_vpi_vecval, *p_vpi_vecval;
```

Figure 27-8—s_vpi_vecval structure definition

```
typedef struct t_vpi_strengthval
{
  PLI_INT32 logic;              /* vpi[0,1,X,Z] */
  PLI_INT32 s0, s1;             /* refer to strength coding in the LRM */
} s_vpi_strengthval, *p_vpi_strengthval;
```

Figure 27-9—s_vpi_strengthval structure definition

For vectors, the *p_vpi_vecval* field shall point to an array of `s_vpi_vecval` structures. The size of this array shall be determined by the size of the vector, where *array_size = ((vector_size-1)/32 + 1)*. The lsb of the vector shall be represented by the lsb of the 0-indexed element of `s_vpi_vecval` array. The 33rd bit of the vector shall be represented by the lsb of the 1-indexed element of the array, and so on. The memory for

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

the union members *str*, *time*, *vector*, *strength*, and *misc* of the value union in the `s_vpi_value` structure shall be provided by the routine **vpi_get_value()**. This memory shall only be valid until the next call to **vpi_get_value()**. The application must provide the memory for these members when calling **vpi_put_value()**. When a value change callback occurs for a value type of **vpiVectorVal**, the system shall create the associated memory (an array of `s_vpi_vecval` structures) and free the memory upon the return of the callback.

Table 27-3—Return value field of the s_vpi_value structure union

Format	Union member	Return description
vpiBinStrVal	str	String of binary character(s) [**1, 0, x, z**]
vpiOctStrVal	str	String of octal character(s) [**0–7, x, X, z, Z**] **x** when all the bits are **x** **X** when some of the bits are **x** **z** when all the bits are **z** **Z** when some of the bits are **z**
vpiDecStrVal	str	String of decimal character(s) [0–9]
vpiHexStrVal	str	String of hex character(s) [**0–f, x, X, z, Z**] **x** when all the bits are **x** **X** when some of the bits are **x** **z** when all the bits are **z** **Z** when some of the bits are **z**
vpiScalarVal	scalar	**vpi1, vpi0, vpiX, vpiZ, vpiH, vpiL**
vpiIntVal	integer	Integer value of the handle. Any bits x or z in the value of the object are mapped to a 0
vpiRealVal	real	Value of the handle as a double
vpiStringVal	str	A string where each 8-bit group of the value of the object is assumed to represent an ASCII character
vpiTimeVal	time	Integer value of the handle using two integers
vpiVectorVal	vector	*aval/bval* representation of the value of the object
vpiStrengthVal	strength	Value plus strength information
vpiObjTypeVal	—	Return a value in the closest format of the object

If the format field in the s_vpi_value structure is set to **vpiStrengthVal**, the *value.strength* pointer must point to an array of `s_vpi_strengthval` structures. This array must have at least as many elements as there are bits in the vector. If the object is a reg or variable, the strength will always be returned as strong.

If the logic value retrieved by **vpi_get_value()** needs to be preserved for later use, the application must allocate storage and copy the value. The following example can be used to copy a value that was retrieved into an **s_vpi_value** structure into another structure allocated by the application:

```
/*
 * Copy s_vpi_value structure - must first allocate pointed to fields.
 * nvalp must be previously allocated.
 * Need to first determine size for vector value.
 */
void copy_vpi_value(s_vpi_value *nvalp, s_vpi_value *ovalp,
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
                    PLI_INT32 blen, PLI_INT32 nd_alloc)
{
 int i;
 PLI_INT32 numvals;
 nvalp->format = ovalp->format;
 switch (nvalp->format) {
  /* all string values */
  case vpiBinStrVal: case vpiOctStrVal: case vpiDecStrVal:
  case vpiHexStrVal: case vpiStringVal:
   if (nd_alloc) nvalp->value.str = malloc(strlen(ovalp->value.str)+1);
   strcpy(nvalp->value.str, ovalp->value.str);
   break;
  case vpiScalarVal:
   nvalp->value.scalar = ovalp->value.scalar;
   break;
  case vpiIntVal:
   nvalp->value.integer = ovalp->value.integer;
   break;
  case vpiRealVal:
   nvalp->value.real = ovalp->value.real;
   break;
  case vpiVectorVal:
   numvals = (blen + 31) >> 5;
   if (nd_alloc)
    {
     nvalp->value.vector = (p_vpi_vecval)
      malloc(numvals*sizeof(s_vpi_vecval));
    }
   /* t_vpi_vecval is really array of the 2 integer a/b sections */
   /* memcpy or bcopy better here */
   for (i = 0; i <numvals; i++)
     nvalp->value.vector[i] = ovalp->value.vector[i];
   break;
  case vpiStrengthVal:
   if (nd_alloc)
    {
     nvalp->value.strength = (p_vpi_strengthval)
      malloc(sizeof(s_vpi_strengthval));
    }
   /* assumes C compiler supports struct assign */
   *(nvalp->value.strength) = *(ovalp->value.strength);
   break;
  case vpiTimeVal:
   nvalp->value.time = (p_vpi_time) malloc(sizeof(s_vpi_time));
   /* assumes C compiler supports struct assign */
   *(nvalp->value.time) = *(ovalp->value.time);
   break;
  /* not sure what to do here? */
  case vpiObjTypeVal: case vpiSuppressVal:
   vpi_printf(
    "**ERR: cannot copy vpiObjTypeVal or vpiSuppressVal formats",
    " - not for filled records.\n");
  break;
 }
}
```

To get the ASCII values of UDP table entries (see Table 8-1 in 8.1.6), the *p_vpi_vecval* field shall point to an array of `s_vpi_vecval` structures. The size of this array shall be determined by the size of the table

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

entry (number of symbols per table entry), where *array_size = ((table_entry_size-1)/4 + 1)*. Each symbol shall require two bytes; the ordering of the symbols within `s_vpi_vecval` shall be the most significant byte of *abit* first, then the least significant byte of *abit*, then the most significant byte of *bbit*, and then the least significant byte of *bbit*. Each symbol can be either one or two characters; when it is a single character, the second byte of the pair shall be an ASCII "\0".

Real valued objects shall be converted to an integer using the rounding defined in 4.8.2 before being returned in a format other than **vpiRealVal** and **vpiStringVal**. If the format specified is **vpiStringVal**, then the value shall be returned as a string representation of a floating point number. The format of this string shall be in decimal notation with at most 16 digits of precision.

If a constant object's **vpiConstType** is **vpiStringVal**, the value shall be retrieved using a format of either **vpiStringVal** or **vpiVectorVal**.

The *misc* field in the `s_vpi_value` structure shall provide for alternative value types, which can be implementation-specific. If this field is utilized, one or more corresponding format types shall also be provided.

In the following example, the binary value of each net that is contained in a particular module and whose name begins with a particular string is displayed. [This function makes use of the `strcmp()` facility normally declared in a `string.h` C library.]

```
void display_certain_net_values(mod, target)
vpiHandle mod;
PLI_BYTE8 *target;
{
       static s_vpi_value value_s = {vpiBinStrVal};
       static p_vpi_value value_p = &value_s;
       vpiHandle net, itr;

       itr = vpi_iterate(vpiNet, mod);
       while (net = vpi_scan(itr))
       {
           PLI_BYTE8 *net_name = vpi_get_str(vpiName, net);
           if (strcmp(target, net_name) == 0)
           {
               vpi_get_value(net, value_p);
               vpi_printf("Value of net %s: %s\n",
                  vpi_get_str(vpiFullName, net),value_p->value.str);
           }
       }
}
```

The following example illustrates the use of vpi_get_value() to access UDP table entries. Two sample outputs from this example are provided after the example.

```
/*
 * hUDP must be a handle to a UDP definition
 */
static void dumpUDPTableEntries(vpiHandle hUDP)
{
  vpiHandle hEntry, hEntryIter;
  s_vpi_value value;
  PLI_INT32 numb;
  PLI_INT32 udpType;
  PLI_INT32 item;
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
  PLI_INT32 entryVal;
  PLI_INT32 *abItem;
  PLI_INT32 cnt, cnt2;
  numb = vpi_get(vpiSize, hUDP);
  udpType = vpi_get(vpiPrimType, hUDP);
  if (udpType == vpiSeqPrim)
    numb++; /* There is one more table entry for state */
  numb++;   /* There is a table entry for the output */
  hEntryIter = vpi_iterate(vpiTableEntry, hUDP);
  if (!hEntryIter)
    return;
  value.format = vpiVectorVal;
  while(hEntry = vpi_scan(hEntryIter))
    {
      vpi_printf("\n");
      /* Show the entry as a string */
      value.format = vpiStringVal;
      vpi_get_value(hEntry, &value);
      vpi_printf("%s\n", value.value.str);
      /* Decode the vector value format */
      value.format = vpiVectorVal;
      vpi_get_value(hEntry, &value);
      abItem = (PLI_INT32 *)value.value.vector;
      for(cnt=((numb-1)/2+1);cnt>0;cnt--)
        {
          entryVal = *abItem;
          abItem++;
          /* Rip out 4 characters */
          for (cnt2=0;cnt2<4;cnt2++)
            {
              item = entryVal&0xff;
              if (item)
                vpi_printf("%c", item);
              else
                vpi_printf("_");
              entryVal = entryVal>>8;
            }
        }
    }
  vpi_printf("\n");
}
```

For a UDP table of

```
1    0   :?:1;
0   (01) :?:-;
(10) 0   :0:1;
```

the output from the preceding example would be

```
10:1
_0_1___1
01:0
_1_0___0
00:1
_0_0___1
```

 Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

For a UDP table entry of

```
1    0   :?:1;
0   (01) :?:-;
(10) 0   :0:1;
```

the output from the preceding example would be

```
10:?:1
_0_1_1_?
0(01):?:-
10_0_-_?
(10)0:0:1
_001_1_0
```

27.15 vpi_get_vlog_info()

vpi_get_vlog_info()			
Synopsis:	Retrieve information about Verilog simulation execution.		
Syntax:	`vpi_get_vlog_info(vlog_info_p)`		
	Type	**Description**	
Returns:	PLI_INT32	1 (true) on success; 0 (false) on failure.	
	Type	**Name**	**Description**
Arguments:	p_vpi_vlog_info	vlog_info_p	Pointer to a structure containing simulation information.
Related routines:			

The VPI routine **vpi_get_vlog_info()** shall obtain the following information about Verilog product execution:

— The number of invocation options (*argc*)
— Invocation option values (*argv*)
— Product and version strings

The information shall be contained in an `s_vpi_vlog_info` structure. The routine shall return 1 (true) on success and 0 (false) on failure.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The `s_vpi_vlog_info` structure used by **vpi_get_vlog_info()** is defined in `vpi_user.h` and is listed in Figure 27-10.

```
typedef struct t_vpi_vlog_info
{
  PLI_INT32 argc;
  PLI_BYTE8 **argv;
  PLI_BYTE8 *product;
  PLI_BYTE8 *version;
} s_vpi_vlog_info, *p_vpi_vlog_info;
```

Figure 27-10—s_vpi_vlog_info structure definition

The format of the *argv* array is that each pointer in the array shall point to a `NULL`-terminated character array that contains the string located on the tool's invocation command line. There shall be *argc* entries in the argv array. The value in entry zero shall be the tool's name.

The vendor tool may provide a command-line option to pass a file containing a set of options. In that case, the argument strings returned by **vpi_get_vlog_info()** shall contain the vendor option string name followed by a pointer to a `NULL`-terminated array of pointers to characters. This new array shall contain the parsed contents of the file. The value in entry zero shall contain the name of the file. The remaining entries shall contain pointers to `NULL`-terminated character arrays containing the different options in the file. The last entry in this array shall be `NULL`. If one of the options is the vendor file option, then the next pointer shall behave the same as described above.

27.16 vpi_handle()

<table>
<tr><th colspan="4">vpi_handle()</th></tr>
<tr><td>Synopsis:</td><td colspan="3">Obtain a handle to an object with a one-to-one relationship.</td></tr>
<tr><td>Syntax:</td><td colspan="3"><code>vpi_handle(type, ref)</code></td></tr>
<tr><td rowspan="2">Returns:</td><td>Type</td><td colspan="2">Description</td></tr>
<tr><td>vpiHandle</td><td colspan="2">Handle to an object.</td></tr>
<tr><td rowspan="3">Arguments:</td><td>Type</td><td>Name</td><td>Description</td></tr>
<tr><td>PLI_INT32</td><td>type</td><td>An integer constant representing the type of object for which to obtain a handle.</td></tr>
<tr><td>vpiHandle</td><td>ref</td><td>Handle to a reference object.</td></tr>
<tr><td>Related routines:</td><td colspan="3">Use vpi_iterate() and vpi_scan() to obtain handles to objects with a one-to-many relationship.
Use vpi_handle_multi() to obtain a handle to an object with a many-to-one relationship.</td></tr>
</table>

The VPI routine **vpi_handle()** shall return the object of type *type* associated with object *ref*. Unless otherwise specified, calling **vpi_handle()** for a protected object shall be an error. The one-to-one relationships that are traversed with this routine are indicated as single arrows in the data model diagrams.

The following example application displays each primitive that an input net drives:

```
void display_driven_primitives(net)
vpiHandle net;
{
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
            vpiHandle load, prim, itr;
            vpi_printf("Net %s drives terminals of the primitives: \n",
               vpi_get_str(vpiFullName, net));
            itr = vpi_iterate(vpiLoad, net);
            if (!itr)
               return;
            while (load = vpi_scan(itr))
            {
               switch(vpi_get(vpiType, load))
               {
                  case vpiGate:
                  case vpiSwitch:
                  case vpiUdp:
                     prim = vpi_handle(vpiPrimitive, load);
                     vpi_printf("\t%s\n", vpi_get_str(vpiFullName, prim));
               }
            }
      }
```

27.17 vpi_handle_by_index()

vpi_handle_by_index()			
Synopsis:	Get a handle to an object using its index number within a parent object.		
Syntax:	`vpi_handle_by_index(obj, index)`		
Returns:	**Type**	**Description**	
	vpiHandle	Handle to an object.	
Arguments:	**Type**	**Name**	**Description**
	vpiHandle	obj	Handle to an object.
	PLI_INT32	index	Index number of the object for which to obtain a handle.
Related routines:			

The VPI routine **vpi_handle_by_index()** shall return a handle to an object based on the index number of the object within the reference object, *obj*. The reference object shall be an object that has the **access by index** property. Unless otherwise specified, calling **vpi_handle_by_index()** for a protected object shall be an error. For example, to access a net bit, *obj* would be the associated net; to access an element of a reg array, *obj* would be the array. If the selection represented by the index number does not lead to the construction of a legal Verilog index select expression, the routine shall return a `null` handle.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

27.18 vpi_handle_by_multi_index()

vpi_handle_by_multi_index()			
Synopsis:	Obtain a handle to a subobject using an array of indices and a reference object.		
Syntax:	`vpi_handle_by_multi_index(obj, num_index, index_array)`		
	Type	**Description**	
Returns:	vpiHandle	Handle to an object.	
	Type	**Name**	**Description**
Arguments:	vpiHandle	obj	Handle to an object.
	PLI_INT32	num_index	Number of indices in the index array.
	PLI_INT32 *	index_array	Array of indices. Leftmost index first.
Related routines:			

The VPI routine **vpi_handle_by_multi_index()** shall provide access to an index-selected subobject of the reference handle. The reference object shall be an object that has the **access by index** property. Unless otherwise specified, calling **vpi_handle_by_multi_index()** for a protected object shall be an error. This routine shall return a handle to a valid Verilog object based on the list of indices provided by the argument *index_array* and reference handle denoted by *obj*. The argument *num_index* shall contain the number of indices in the provided array index_array.

The order of the indices provided shall follow the array dimension declaration from the leftmost range to the rightmost range of the reference handle; the array indices may be optionally followed by a bit-select index. If the indices provided do not lead to the construction of a legal Verilog index select expression, the routine shall return a `null` handle.

27.19 vpi_handle_by_name()

vpi_handle_by_name()			
Synopsis:	Get a handle to an object with a specific name.		
Syntax:	`vpi_handle_by_name(name, scope)`		
	Type	**Description**	
Returns:	vpiHandle	Handle to an object.	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	name	A character string or pointer to a string containing the name of an object.
	vpiHandle	scope	Handle to a Verilog HDL scope.
Related routines:			

The VPI routine **vpi_handle_by_name()** shall return a handle to an object with a specific name. This function can be applied to all objects with a *fullname* property. The *name* can be hierarchical or simple. If

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

scope is `NULL`, then *name* shall be searched for from the top level of hierarchy. If a scope object is provided, then search within that scope only. Unless otherwise specified, calling **vpi_handle_by_name()** for a protected scope object shall be an error. If the *name* is hierarchical and includes a protected scope, the call shall be an error.

27.20 vpi_handle_multi()

vpi_handle_multi()			
Synopsis:	Obtain a handle for an object in a many-to-one relationship.		
Syntax:	`vpi_handle_multi(type, ref1, ref2, ...)`		
	Type	**Description**	
Returns:	vpiHandle	Handle to an object.	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	type	An integer constant representing the type of object for which to obtain a handle.
	vpiHandle	ref1, ref2, ...	Handles to two or more reference objects.
Related routines:	Use vpi_iterate() and vpi_scan() to obtain handles to objects with a one-to-many relationship. Use vpi_handle() to obtain handles to objects with a one-to-one relationship.		

The VPI routine **vpi_handle_multi()** can be used to return a handle to an object of type **vpiInterModPath** associated with a list of *output port* and *input port* reference objects. The ports shall be of the same size and can be at different levels of the hierarchy.

27.21 vpi_iterate()

vpi_iterate()			
Synopsis:	Obtain an iterator handle to objects with a one-to-many relationship.		
Syntax:	`vpi_iterate(type, ref)`		
	Type	**Description**	
Returns:	vpiHandle	Handle to an iterator for an object.	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	type	An integer constant representing the type of object for which to obtain iterator handles.
	vpiHandle	ref	Handle to a reference object.
Related routines:	Use vpi_scan() to traverse the HDL hierarchy using the iterator handle returned from vpi_iterate(). Use vpi_handle() to obtain handles to object with a one-to-one relationship. Use vpi_handle_multi() to obtain a handle to an object with a many-to-one relationship.		

The VPI routine **vpi_iterate()** shall be used to traverse one-to-many relationships, which are indicated as double arrows in the data model diagrams. Unless otherwise specified, calling **vpi_iterate()** for a protected object shall be an error. The **vpi_iterate()** routine shall return a handle to an iterator, whose type shall be **vpiIterator**, which can used by **vpi_scan()** to traverse all objects of type *type* associated with object *ref*. To

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

get the reference object from the iterator object, use **vpi_handle(vpiUse, iterator_handle)**. If there are no objects of type *type* associated with the reference handle *ref*, then the **vpi_iterate()** routine shall return NULL.

The following example application uses **vpi_iterate()** and **vpi_scan()** to display each net (including the size for vectors) declared in the module. The example assumes it shall be passed a valid module handle.

```
void display_nets(mod)
vpiHandle mod;
{
       vpiHandle net;
       vpiHandle itr;

       vpi_printf("Nets declared in module %s\n",
       vpi_get_str(vpiFullName, mod));

       itr = vpi_iterate(vpiNet, mod);
       while (net = vpi_scan(itr))
       {
          vpi_printf("\t%s", vpi_get_str(vpiName, net));
          if (vpi_get(vpiVector, net))
          {
             vpi_printf(" of size %d\n", vpi_get(vpiSize, net));
          }
          else vpi_printf("\n");
       }
}
```

27.22 vpi_mcd_close()

vpi_mcd_close()			
Synopsis:	Close one or more files opened by vpi_mcd_open().		
Syntax:	vpi_mcd_close(mcd)		
	Type	**Description**	
Returns:	PLI_UINT32	0 if successful; the *mcd* of unclosed channels if unsuccessful.	
	Type	**Name**	**Description**
Arguments:	PLI_UINT32	mcd	A multichannel descriptor representing the files to close.
Related routines:	Use vpi_mcd_open() to open a file. Use vpi_mcd_printf() to write to an opened file. Use vpi_mcd_vprintf() to write a variable number of arguments to an opened file. Use vpi_mcd_flush() to flush a file output buffer. Use vpi_mcd_name() to get the name of a file represented by a channel descriptor.		

The VPI routine **vpi_mcd_close()** shall close the file(s) specified by a multichannel descriptor, *mcd*. Several channels can be closed simultaneously because channels are represented by discrete bits in the integer *mcd*. On success, this routine shall return a 0; on error, it shall return the *mcd* value of the unclosed channels. This routine can also be used to close file descriptors that were opened using the system function **$fopen()**. See 17.2.1 for the functional description of **$fopen()**.

The following descriptors are predefined and cannot be closed using **vpi_mcd_close()**:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

descriptor 1 is for the output channel of the software product that invoked the PLI application and the current log file

27.23 vpi_mcd_flush()

vpi_mcd_flush()			
Synopsis:	Flushes the data from the given *mcd* output buffers.		
Syntax:	`vpi_mcd_flush(mcd)`		
	Type	**Description**	
Returns:	PLI_INT32	0 if successful; nonzero if unsuccessful.	
	Type	**Name**	**Description**
Arguments:	PLI_UINT32	mcd	A multichannel descriptor representing the files to which to write.
Related routines:	Use vpi_mcd_printf() to write a finite number of arguments to an opened file. Use vpi_mcd_vprintf() to write a variable number of arguments to an opened file. Use vpi_mcd_open() to open a file. Use vpi_mcd_close() to close a file. Use vpi_mcd_name() to get the name of a file represented by a channel descriptor.		

The routine **vpi_mcd_flush()** shall flush the output buffers for the file(s) specified by the multichannel descriptor *mcd*.

27.24 vpi_mcd_name()

vpi_mcd_name()			
Synopsis:	Get the name of a file represented by a channel descriptor.		
Syntax:	`vpi_mcd_name(cd)`		
	Type	**Description**	
Returns:	PLI_BYTE8 *	Pointer to a character string containing the name of a file.	
	Type	**Name**	**Description**
Arguments:	PLI_UINT32	cd	A channel descriptor representing a file.
Related routines:	Use vpi_mcd_open() to open a file. Use vpi_mcd_close() to close files. Use vpi_mcd_printf() to write to an opened file. Use vpi_mcd_flush() to flush a file output buffer. Use vpi_mcd_vprintf() to write a variable number of arguments to an opened file.		

The VPI routine **vpi_mcd_name()** shall return the name of a file represented by a single-channel descriptor, *cd*. On error, the routine shall return `NULL`. This routine shall overwrite the returned value on subsequent calls. If the application needs to retain the string, it should copy it. This routine can be used to get the name of any file opened using the system function **$fopen** or the VPI routine **vpi_mcd_open()**. The channel descriptor *cd* could be an *fd* file descriptor returned from **$fopen** (indicated by the most significant bit being set) or an *mcd* multichannel descriptor returned by either the system function **$fopen** or the VPI routine **vpi_mcd_open()**. See 17.2.1 for the functional description of **$fopen**.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

27.25 vpi_mcd_open()

vpi_mcd_open()			
Synopsis:	Open a file for writing.		
Syntax:	`vpi_mcd_open(file)`		
	Type	**Description**	
Returns:	PLI_UINT32	A multichannel descriptor representing the file that was opened.	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	file	A character string or pointer to a string containing the file name to be opened.
Related routines:	Use vpi_mcd_close() to close a file. Use vpi_mcd_printf() to write to an opened file. Use vpi_mcd_vprintf() to write a variable number of arguments to an opened file. Use vpi_mcd_flush() to flush a file output buffer. Use vpi_mcd_name() to get the name of a file represented by a channel descriptor.		

The VPI routine **vpi_mcd_open()** shall open a file for writing and shall return a corresponding multichannel description number (*mcd*). The channel descriptor 1 (least significant bit) is reserved for representing the output channel of the software product that invoked the PLI application and the log file (if one is currently open). The channel descriptor 32 (most significant bit) is reserved to represent a file descriptor (*fd*) returned from the Verilog HDL **$fopen** system function.

The *mcd* descriptor returned by **vpi_mcd_open()** routine is compatible with the *mcd* descriptors returned from the **$fopen** system function. The *mcd* descriptors returned from **vpi_mcd_open()** and from **$fopen** may be shared between the HDL system tasks that use *mcd* descriptors and the VPI routines that use *mcd* descriptors. If the most significant bit of the return value from **$fopen** is set, then the value is an *fd* file descriptor, which is not compatible with the *mcd* descriptor returned by **vpi_mcd_open()**. See 17.2.1 for the functional description of **$fopen**.

The **vpi_mcd_open()** routine shall return a 0 on error. If the file has already been opened either by a previous call to **vpi_mcd_open()** or using **$fopen** in the Verilog source code, then **vpi_mcd_open()** shall return the descriptor number.

-`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

27.26 vpi_mcd_printf()

vpi_mcd_printf()			
Synopsis:	Write to one or more files opened with vpi_mcd_open() or $fopen.		
Syntax:	`vpi_mcd_printf(mcd, format, ...)`		
	Type	**Description**	
Returns:	PLI_INT32	The number of characters written.	
	Type	**Name**	**Description**
Arguments:	PLI_UINT32	mcd	A multichannel descriptor representing the files to which to write.
	PLI_BYTE8 *	format	A format string using the C fprintf() format.
Related routines:	Use vpi_mcd_vprintf() to write a variable number of arguments to an opened file. Use vpi_mcd_open() to open a file. Use vpi_mcd_close() to close a file. Use vpi_mcd_flush() to flush a file output buffer. Use vpi_mcd_name() to get the name of a file represented by a channel descriptor.		

The VPI routine **vpi_mcd_printf()** shall write to one or more channels (up to 31) determined by the *mcd*. An *mcd* of 1 (bit 0 set) corresponds to the channel 1, an *mcd* of 2 (bit 1 set) corresponds to channel 2, an *mcd* of 4 (bit 2 set) corresponds to channel 3, and so on. Channel 1 is reserved for the output channel of the software product that invoked the PLI application and the current log file. The most significant bit of the descriptor is reserved by the tool to indicate that the descriptor is actually a file descriptor instead of an *mcd*. **vpi_mcd_printf()** shall also write to a file represented by an *mcd* that was returned from the Verilog HDL **$fopen** system function. **vpi_mcd_printf()** shall not write to a file represented by an *fd* file descriptor returned from **$fopen** (indicated by the most significant bit being set). See 17.2.1 for the functional description of **$fopen**.

Several channels can be written to simultaneously because channels are represented by discrete bits in the integer *mcd*.

The text written shall be controlled by one or more format strings. The format strings shall use the same format as the C fprintf() routine. The routine shall return the number of characters printed or return EOF if an error occurred.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

27.27 vpi_mcd_vprintf()

vpi_mcd_vprintf()			
Synopsis:	Write to one or more files opened with vpi_mcd_open() or $fopen using varargs that are already started.		
Syntax:	`vpi_mcd_vprintf(mcd, format, ap)`		
	Type	**Description**	
Returns:	PLI_INT32	The number of characters written.	
	Type	**Name**	**Description**
Arguments:	PLI_UINT32	mcd	A multichannel descriptor representing the files to which to write.
	PLI_BYTE8 *	format	A format string using the C printf() format.
	va_list	ap	An already started varargs list.
Related routines:	Use vpi_mcd_printf() to write a finite number of arguments to an opened file. Use vpi_mcd_open() to open a file. Use vpi_mcd_close() to close a file. Use vpi_mcd_flush() to flush a file output buffer. Use vpi_mcd_name() to get the name of a file represented by a channel descriptor.		

This routine performs the same function as **vpi_mcd_printf()**, except that varargs have already been started.

27.28 vpi_printf()

vpi_printf()			
Synopsis:	Write to the output channel of the software product that invoked the PLI application and the current product log file.		
Syntax:	`vpi_printf(format, ...)`		
	Type	**Description**	
Returns:	PLI_INT32	The number of characters written.	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	format	A format string using the C printf() format.
Related routines:	Use vpi_vprintf() to write a variable number of arguments. Use vpi_mcd_printf() to write to an opened file. Use vpi_mcd_flush() to flush a file output buffer. Use vpi_mcd_vprintf() to write a variable number of arguments to an opened file.		

The VPI routine **vpi_printf()** shall write to both the output channel of the software product that invoked the PLI application and the current product log file. The format string shall use the same format as the C `printf()` routine. The routine shall return the number of characters printed or return EOF if an error occurred.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

27.29 vpi_put_data()

vpi_put_data()			
Synopsis:	Put data into an implementation's save/restart location.		
Syntax:	`vpi_put_data(id, dataLoc, numOfBytes)`		
	Type	**Description**	
Returns:	PLI_INT32	The number of bytes written.	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	id	A save/restart ID returned from vpi_get(vpiSaveRestartID, NULL).
	PLI_BYTE8 *	dataLoc	Address of application-allocated storage.
	PLI_INT32	numOfBytes	Number of bytes to be added to save/restart location.
Related routines:	Use vpi_get_data() to retrieve saved data.		

This routine shall place **numOfBytes**, which must be greater than zero, of data located at **dataLoc** into an implementation's save/restart location. The return value shall be the number of bytes written. A zero shall be returned if an error is detected. There shall be no restrictions on the following:

— How many times the routine can be called for a given ***id***
— The order applications put data using the different ***id***s

The data from multiple calls to **vpi_put_data()** with the same ***id*** shall be stored by the simulator in such a way that the opposing routine **vpi_get_data()** can pull data out of the save/restart location using different sizes of chunks. This routine can only be called from an application routine that has been called for the reason **cbStartOfSave** or **cbEndOfSave**. An application can get the path to the implementation's save/restart location by calling **vpi_get_str(vpiSaveRestartLocation, NULL)** from an application callback routine that has been called for reason **cbStartOfSave** or **cbEndOfSave**.

The following example illustrates using **vpi_put_data()** and **vpi_get_data()**:

```
#include <stdlib.h>
#include <assert.h>
#include "vpi_user.h"

typedef struct myStruct *myStruct_p;
typedef struct myStruct {
  PLI_INT32 d1;
  PLI_INT32 d2;
  myStruct_p next;
} myStruct_s;

static myStruct_p firstWrk = NULL;

PLI_INT32 consumer_restart(p_cb_data data)
{
  struct myStruct *wrk;
  PLI_INT32 status;
  PLI_INT32 cnt, size;
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
  PLI_INT32 id = (PLI_INT32)data->user_data;

  /* Get the number of structures */

  status = vpi_get_data(id,(PLI_BYTE8 *)&cnt,sizeof(PLI_INT32));
  assert(status > 0); /* Check returned status */

  /* allocate memory for the structures */

  size = cnt * sizeof(struct myStruct);
  firstWrk = (myStruct_p)malloc(size);

  /* retrieve the data structures */

  if (cnt != vpi_get_data(id, (PLI_BYTE8 *)firstWrk,cnt))
    return(1); /* error */

  firstWrk = wrk;

  /* Fix the next pointers in the linked list */

  for (wrk = firstWrk; cnt > 0; cnt--)
    {
      wrk->next = wrk + 1;
      wrk = wrk->next;
    }
  wrk->next = NULL;
  return(0); /* SUCCESS */
}

PLI_INT32 consumer_save(p_cb_data data)
{
  myStruct_p wrk;
  s_cb_data cbData;
  vpiHandle cbHdl;
  PLI_INT32 id = 0;
  PLI_INT32 cnt = 0;

  /* Get the number of structures */

  wrk = firstWrk;
  while (wrk)
    {
      cnt++;
      wrk = wrk->next;
    }

  /* now save the data */

  wrk = firstWrk;
  id = vpi_get(vpiSaveRestartID, NULL);

  /* save the number of data structures */

  vpi_put_data(id,(PLI_BYTE8 *)cnt,sizeof(PLI_INT32));

  /* Save the different data structures.  Note that a pointer
   * is being saved.  While this is allowed, an application
   * must change it to something useful on a restart.
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
    */

  while (wrk)
    {
      vpi_put_data(id,(PLI_BYTE8 *)wrk,sizeof(myStruct_s));
      wrk = wrk->next;
    }

  /* register a call for restart */
  /* We need the "id" so that the saved data can be retrieved.
   * Using the user_data field of the callback structure is the
   * easiest way to pass this information to retrieval operation.
   */

  cbData.user_data = (PLI_BYTE8 *)id;
  cbData.reason = cbStartOfRestart;

  /* See 27.8 vpi_get_data() for a description of how
   * the callback routine can be used to retrieve the data.
   */

  cbData.cb_rtn = consumer_restart;

  cbData.value = NULL;
  cbData.time = NULL;
  cbHdl = vpi_register_cb(&cbData);
  vpi_free_object(cbHdl);
  return(0);
}
```

27.30 vpi_put_delays()

vpi_put_delays()			
Synopsis:	Set the delays or timing limits of an object.		
Syntax:	`vpi_put_delays(obj, delay_p)`		
Returns:	**Type**	**Description**	
	void		
Arguments:	**Type**	**Name**	**Description**
	vpiHandle	obj	Handle to an object.
	p_vpi_delay	delay_p	Pointer to a structure containing delay information.
Related routines:	Use vpi_get_delays() to retrieve delays or timing limits of an object.		

The VPI routine **vpi_put_delays()** shall set the delays or timing limits of an object as indicated in the *delay_p* structure. The same ordering of delays shall be used as described in the **vpi_get_delays()** function. If only the delay changes and not the pulse limits, the pulse limits shall retain the values they had before the delays where altered.

The `s_vpi_delay` and `s_vpi_time` structures used by both **vpi_get_delays()** and **vpi_put_delays()** are defined in `vpi_user.h` and are listed in Figure 27-11 and Figure 27-12.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
typedef struct t_vpi_delay
{
  struct t_vpi_time *da;  /* pointer to application-allocated
                             array of delay values*/
  PLI_INT32 no_of_delays; /* number of delays */
  PLI_INT32 time_type;    /* [vpiScaledRealTime,vpiSimTime,
                             vpiSuppressTime]*/
  PLI_INT32 mtm_flag;     /* true for mtm values */
  PLI_INT32 append_flag;  /* true for append */
  PLI_INT32 pulsere_flag; /* true for pulsere values */
} s_vpi_delay, *p_vpi_delay;
```

Figure 27-11—s_vpi_delay structure definition

```
typedef struct t_vpi_time
{
  PLI_INT32  type;   /* [vpiScaledRealTime, vpiSimTime, vpiSuppressTime] */
  PLI_UINT32 high, low;     /* for vpiSimTime */
  double     real;          /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 27-12—s_vpi_time structure definition

The *da* field of the `s_vpi_delay` structure shall be an application-allocated array of `s_vpi_time` structures. This array stores the delay values to be written by **vpi_put_delays()**. The number of elements in this array is determined by the following:

- The number of delays to be written
- The **mtm_flag** setting
- The **pulsere_flag** setting

The number of delays to be set shall be set in the *no_of_delays* field of the `s_vpi_delay` structure. Legal values for the number of delays shall be determined by the type of object:

- For primitive objects, the *no_of_delays* value shall be 2 or 3.
- For path delay objects, the *no_of_delays* value shall be 1, 2, 3, 6, or 12.
- For timing check objects, the *no_of_delays* value shall match the number of limits existing in the timing check.
- For intermodule path objects, the *no_of_delays* value shall be 2 or 3.

The application-allocated `s_vpi_delay` array shall contain delays in the same order in which they occur in the Verilog HDL description. The number of elements for each delay shall be determined by the flags **mtm_flag** and **pulsere_flag**, as shown in Table 27-4.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 27-4—Size of the s_vpi_delay->da array

Flag values	Number of `s_vpi_time` array elements required for *s_vpi_delay->da*	Order in which delay elements shall be filled

mtm_flag = `FALSE`
pulsere_flag = `FALSE`
	no_of_delays	`1st delay: da[0] -> 1st delay`
		`2nd delay: da[1] -> 2nd delay`
		`...`

mtm_flag = `TRUE`
pulsere_flag = `FALSE`
	3 * *no_of_delays*	`1st delay: da[0] -> min delay`
		`da[1] -> typ delay`
		`da[2] -> max delay`
		`2nd delay: ...`

mtm_flag = `FALSE`
pulsere_flag = `TRUE`
	3 * *no_of_delays*	`1st delay: da[0] -> delay`
		`da[1] -> reject limit`
		`da[2] -> error limit`
		`2nd delay element: ...`

mtm_flag = `TRUE`
pulsere_flag = `TRUE`
	9 * *no_of_delays*	`1st delay: da[0] -> min delay`
		`da[1] -> typ delay`
		`da[2] -> max delay`
		`da[3] -> min reject`
		`da[4] -> typ reject`
		`da[5] -> max reject`
		`da[6] -> min error`
		`da[7] -> typ error`
		`da[8] -> max error`
		`2nd delay: ...`

The following example application accepts a module path handle, rise and fall delays, and replaces the delays of the indicated path.

```
void set_path_rise_fall_delays(path, rise, fall)
vpiHandle path;
double rise, fall;
{
       static s_vpi_time path_da[2];
       static s_vpi_delay delay_s = {NULL, 2, vpiScaledRealTime};
       static p_vpi_delay delay_p = &delay_s;

       delay_s.da = path_da;
       path_da[0].real = rise;
       path_da[1].real = fall;
       vpi_put_delays(path, delay_p);
}
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

27.31 vpi_put_userdata()

vpi_put_userdata()			
Synopsis:	Put user-data value into an implementation's system task/function instance storage location.		
Syntax:	`vpi_put_userdata(obj, userdata)`		
	Type	**Description**	
Returns:	PLI_INT32	1 on success; 0 if an error occurs.	
	Type	**Name**	**Description**
Arguments:	vpiHandle	obj	Handle to a system task instance or system function instance.
	void *	userdata	User-data value to be associated with the system task instance or system function instance.
Related routines:	Use vpi_get_userdata() to retrieve the user-data value.		

This routine will associate the value of the input *userdata* with the specified user-defined system task/function call handle. The stored value can later be retrieved with the routine **vpi_get_userdata()**. The routine will return a value of 1 on success or a 0 if it fails.

After a restart or a reset, subsequent calls to **vpi_get_userdata()** shall return **NULL**. It is the application's responsibility to save the data during a save using **vpi_put_data()** and to then retrieve it using **vpi_get_data()**. The user-data field can be set up again during or after callbacks of type **cbEndOfRestart** or **cbEndOfReset**.

27.32 vpi_put_value()

vpi_put_value()			
Synopsis:	Set a value on an object.		
Syntax:	`vpi_put_value(obj, value_p, time_p, flags)`		
	Type	**Description**	
Returns:	vpiHandle	Handle to the scheduled event caused by vpi_put_value().	
	Type	**Name**	**Description**
Arguments:	vpiHandle	obj	Handle to an object.
	p_vpi_value	value_p	Pointer to a structure with value information.
	p_vpi_time	time_p	Pointer to a structure with delay information.
	PLI_INT32	flags	Integer constants that set the delay mode.
Related routines:	Use vpi_get_value() to retrieve the value of an expression.		

The VPI routine **vpi_put_value()** shall set simulation logic values on an object. The value to be set shall be stored in an `s_vpi_value` structure that has been allocated by the calling routine. Any storage referenced

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

by the `s_vpi_value` structure shall also be allocated by the calling routine. The legal values that may be specified for each value format are listed in Table 27-3 in 27.14. The delay time before the value is set shall be stored in an `s_vpi_time` structure that has been allocated by the calling routine. The routine can be applied to nets, regs, variables, variable selects, memory words, named events, system function calls, sequential UDPs, and scheduled events. The *flags* argument shall be used to direct the routine to use one of the following delay modes:

vpiInertialDelay — All scheduled events on the object shall be removed before this event is scheduled.

vpiTransportDelay — All events on the object scheduled for times later than this event shall be removed (modified transport delay).

vpiPureTransportDelay — No events on the object shall be removed (transport delay).

vpiNoDelay — The object shall be set to the passed value with no delay. Argument *time_p* shall be ignored and can be set to `NULL`.

vpiForceFlag — The object shall be forced to the passed value with no delay (same as the Verilog HDL procedural **force**). Argument *time_p* shall be ignored and can be set to `NULL`.

vpiReleaseFlag — The object shall be released from a forced value (same as the Verilog HDL procedural **release**). Argument *time_p* shall be ignored and can be set to `NULL`. The *value_p* shall be updated with the value of the object after its release. If the value is a string, time, vector, strength, or miscellaneous value, the data pointed to by the *value_p* argument shall be owned by the interface.

vpiCancelEvent — A previously scheduled event shall be cancelled. The object passed to **vpi_put_value()** shall be a handle to an object of type **vpiSchedEvent**.

If the *flags* argument also has the bit mask **vpiReturnEvent**, **vpi_put_value()** shall return a handle of type **vpiSchedEvent** to the newly scheduled event, provided there is some form of a delay and an event is scheduled. If the bit mask is not used, or if no delay is used, or if an event is not scheduled, the return value shall be `NULL`.

A scheduled event can be cancelled by calling **vpi_put_value()** with *obj* set to the **vpiSchedEvent** handle and *flags* set to **vpiCancelEvent**. The *value_p* and *time_p* arguments to **vpi_put_value()** are not needed for cancelling an event and can be set to **NULL**. It shall not be an error to cancel an event that has already occurred. The scheduled event can be tested by calling **vpi_get()** with the flag **vpiScheduled**. If an event is cancelled, it shall simply be removed from the event queue. Any effects that were caused by scheduling the event shall remain in effect (e.g., events that where cancelled due to inertial delay). Cancelling an event shall also free the handle to that event.

Calling **vpi_free_object()** on the handle shall free the handle, but shall not affect the event.

When **vpi_put_value()** is called for an object of type **vpiNet** or **vpiNetBit**, and with modes of **vpiInertialDelay**, **vpiTransportDelay**, **vpiPureTransportDelay**, or **vpiNoDelay**, the value supplied overrides the resolved value of the net. This value shall remain in effect until one of the drivers of the net changes value. When this occurs, the net shall be reevaluated using the normal resolution algorithms.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

It shall be illegal to specify the format of the value as **vpiStringVal** when putting a value to a real variable or a system function call of type **vpiRealFunc**. It shall be illegal to specify the format of the value as **vpiStrengthVal** when putting a value to a vector object.

When **vpi_put_value()** with a **vpiForce** flag is used, it shall perform a procedural force of a value onto the same types of objects as supported by a procedural force. A **vpiRelease** flag shall release the forced value. This shall be the same functionality as the procedural **force** and **release** keywords in the Verilog HDL (see 9.3.2).

Sequential UDPs shall be set to the indicated value with no delay regardless of any delay on the primitive instance. Putting values to UDP instances must be done using the **vpiNoDelay** flag. Attempting to use the other delay modes shall result in an error.

Calling **vpi_put_value()** on an object of type **vpiNamedEvent** shall cause the named event to toggle. Objects of type **vpiNamedEvent** shall not require an actual value, and the *value_p* argument may be `NULL`.

The **vpi_put_value()** routine shall also return the value of a system function by passing a handle to the user-defined system function as the object handle. This should only occur during execution of the *calltf* routine for the system function. Attempts to use **vpi_put_value()** with a handle to the system function when the *calltf* routine is not active shall be ignored. Should the *calltf* routine for a user-defined system function fail to put a value during its execution, the default value of 0 will be applied. Putting return values to system functions must be done using the **vpiNoDelay** flag.

The **vpi_put_value()** routine shall only return a system function value in a *calltf* application when the call to the system function is active. The action of **vpi_put_value()** to a system function shall be ignored when the system function is not active. Putting values to system function must be done using the **vpiNoDelay** flag.

The `s_vpi_value` and `s_vpi_time` structures used by **vpi_put_value()** are defined in `vpi_user.h` and are listed in Figure 27-13 and Figure 27-14.

```
typedef struct t_vpi_value
{
  PLI_INT32 format; /* vpi[[Bin,Oct,Dec,Hex]Str,Scalar,Int,Real,String,
                          Vector,Strength,Suppress,Time,ObjType]Val */
  union
    {
      PLI_BYTE8 *str;                     /* string value */
      PLI_INT32 scalar;                   /* vpi[0,1,X,Z] */
      PLI_INT32 integer;                  /* integer value */
      double    real;                     /* real value */
      struct t_vpi_time *time;            /* time value */
      struct t_vpi_vecval *vector;        /* vector value */
      struct t_vpi_strengthval *strength; /* strength value */
      PLI_BYTE8 *misc;                    /* ...other */
    } value;
} s_vpi_value, *p_vpi_value;
```

Figure 27-13—s_vpi_value structure definition

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
typedef struct t_vpi_time
{
  PLI_INT32  type;   /* [vpiScaledRealTime, vpiSimTime, vpiSuppressTime] */
  PLI_UINT32 high, low;     /* for vpiSimTime */
  double     real;          /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 27-14—s_vpi_time structure definition

The `s_vpi_vecval` and `s_vpi_strengthval` structures found in Figure 27-13 are listed in Figure 27-15 and Figure 27-16.

```
typedef struct t_vpi_vecval
{
  /* following fields are repeated enough times to contain vector */
  PLI_INT32 aval, bval;       /* bit encoding: ab: 00=0, 10=1, 11=X, 01=Z */
} s_vpi_vecval, *p_vpi_vecval;
```

Figure 27-15—s_vpi_vecval structure definition

```
typedef struct t_vpi_strengthval
{
  PLI_INT32 logic;              /* vpi[0,1,X,Z] */
  PLI_INT32 s0, s1;             /* refer to strength coding below */
} s_vpi_strengthval, *p_vpi_strengthval;
```

Figure 27-16—s_vpi_strengthval structure definition

For **vpiScaledRealTime**, the indicated time shall be in the timescale associated with the object.

27.33 vpi_register_cb()

vpi_register_cb()			
Synopsis:	Register simulation-related callbacks.		
Syntax:	`vpi_register_cb(cb_data_p)`		
	Type	**Description**	
Returns:	vpiHandle	Handle to the callback object.	
	Type	**Name**	**Description**
Arguments:	p_cb_data	cb_data_p	Pointer to a structure with data about when callbacks should occur and the data to be passed.
Related routines:	Use vpi_register_systf() to register callbacks for user-defined system tasks and functions. Use vpi_remove_cb() to remove callbacks registered with vpi_register_cb().		

The VPI routine **vpi_register_cb()** is used for registration of simulation-related callbacks to a user-provided application for a variety of reasons during a simulation. The reasons for which a callback can occur are divided into three categories:

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

— Simulation event
— Simulation time
— Simulation action or feature

How callbacks are registered for each of these categories is explained in this subclause.

The *cb_data_p* argument shall point to a `s_cb_data` structure, which is defined in `vpi_user.h` and given in Figure 27-17.

```
typedef struct t_cb_data
{
  PLI_INT32    reason;           /* callback reason */
  PLI_INT32    (*cb_rtn)(struct t_cb_data *); /* call routine */
  vpiHandle    obj;              /* trigger object */
  p_vpi_time   time;             /* callback time */
  p_vpi_value  value;            /* trigger object value */
  PLI_INT32    index;            /* index of the memory word or var select
                                    that changed */
  PLI_BYTE8   *user_data;
} s_cb_data, *p_cb_data;
```

Figure 27-17—s_cb_data structure definition

For all callbacks, the *reason* field of the `s_cb_data` structure shall be set to a predefined constant, e.g., **cbValueChange**, **cbAtStartOfSimTime**, **cbEndOfCompile**. The reason constant shall determine when the application shall be called back. See the `vpi_user.h` file listing in Annex G for a list of all callback reason constants.

The *cb_rtn* field of the `s_cb_data` structure shall be set to the application routine, which shall be invoked when the simulator executes the callback. The uses of the remaining fields are detailed in 27.33.1 through 27.33.3.

The callback routine shall be passed a pointer to an `s_cb_data` structure. This structure and all structures to which it points belong to the simulator. If the application needs any of these data, it must copy the data prior to returning from the callback routine.

27.33.1 Simulation event callbacks

The **vpi_register_cb()** callback mechanism can be registered for callbacks to occur for simulation events, such as value changes on an expression or terminal, or the execution of a behavioral statement. When the *cb_data_p->reason* field is set to one of the following, the callback shall occur as described below:

cbValueChange	After value change on an expression or terminal or after execution of an event statement
cbStmt	Before execution of a behavioral statement
cbForce/cbRelease	After a force or release has occurred
cbAssign/cbDeassign	After a procedural assign or deassign statement has been executed
cbDisable	After a named block or task containing a system task/function has been disabled

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The following fields shall need to be initialized before passing the `s_cb_data` structure to **vpi_register_cb()**:

cb_data_p->obj This field shall be assigned a handle to an expression, terminal, or statement for which the callback shall occur. For force and release callbacks, if this is set to `NULL`, every force and release shall generate a callback.

cb_data_p->time->type This field shall be set to either **vpiScaledRealTime** or **vpiSimTime**, depending on what time information the application requires during the callback. If simulation time information is not needed during the callback, this field can be set to **vpiSuppressTime**.

cb_data_p->value->format This field shall be set to one of the value formats indicated in Table 27-5. If value information is not needed during the callback, this field can be set to **vpiSuppressVal**. For **cbStmt** callbacks, value information is not passed to the callback routine; therefore, this field shall be ignored.

Table 27-5—Value format field of cb_data_p->value->format

Format	Registers a callback to return
vpiBinStrVal	String of binary character(s) [**1, 0, x, z**]
vpiOctStrVal	String of octal character(s) [**0–7, x, X, z, Z**]
vpiDecStrVal	String of decimal character(s) [0–9]
vpiHexStrVal	String of hex character(s) [**0–f, x, X, z, Z**]
vpiScalarVal	**vpi1, vpi0, vpiX, vpiZ, vpiH, vpiL**
vpiIntVal	Integer value of the handle
vpiRealVal	Value of the handle as a double
vpiStringVal	An ASCII string
vpiTimeVal	Integer value of the handle using two integers
vpiVectorVal	*aval/bval* representation of the value of the object
vpiStrengthVal	Value plus strength information of a scalar object only
vpiObjectVal	Return a value in the closest format of the object

When a simulation event callback occurs, the application shall be passed a single argument, which is a pointer to an `s_cb_data` structure (this is not a pointer to the same structure that was passed to **vpi_register_cb()**). The *time* and *value* information shall be set as directed by the time *type* and *value* format fields in the call to **vpi_register_cb()**. The *user_data* field shall be equivalent to the *user_data* field passed to **vpi_register_cb()**. The application can use the information in the passed structure and information retrieved from other VPI routines to perform the desired callback processing.

cbValueChange callbacks can be placed onto event statements. When the event statement is executed, the callback routine will be called. Because event statements do not have a value, when the callback routine is called, the value field of the `s_cb_data` structure will be `NULL`.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

For a **cbValueChange** callback, if the *obj* has the **vpiArray** property set to `TRUE`, the *value* in the `s_cb_data` structure shall be the value of the array member that changed value. The *index* field shall contain the index of the rightmost range of the array declaration. Use **vpi_iterate(vpiIndex,obj)** to find all the indices.

If a **cbValueChange** callback is registered and the format is set to **vpiStrengthVal**, then the callback shall occur whenever the object changes strength, including changes that do not result in a value change.

For **cbForce**, **cbRelease**, **cbAssign**, and **cbDeassign** callbacks, the object returned in the *obj* field shall be a handle to the force, release, assign, or deassign statement. The *value* field shall contain the resultant value of the left-hand expression. In the case of a release, the *value* field shall contain the value after the release has occurred.

For a **cbDisable** callback, *obj* shall be a handle to a system task call, system function call, named begin, named fork, task, or function.

It is illegal to attempt to place a callback for reason **cbForce**, **cbRelease**, or **cbDisable** on a variable bit-select.

The following example shows an implementation of a simple monitor functionality for scalar nets, using a simulation event callback:

```
setup_monitor(net)
vpiHandle net;
{
       static s_vpi_time time_s = {vpiSimTime};
       static s_vpi_value value_s = {vpiBinStrVal};
       static s_cb_data cb_data_s =
           {cbValueChange, my_monitor, NULL, &time_s, &value_s};
       PLI_BYTE8 *net_name = vpi_get_str(vpiFullName, net);
       cb_data_s.obj = net;
       cb_data_s.user_data = malloc(strlen(net_name)+1);
       strcpy(cb_data_s.user_data, net_name);
       vpi_register_cb(&cb_data_s);
}

my_monitor(cb_data_p)
p_cb_data cb_data_p; {
       vpi_printf("%d %d: %s = %s\n",
           cb_data_p->time->high, cb_data_p->time->low,
           cb_data_p->user_data,
           cb_data_p->value->value.str);
}
```

27.33.1.1 Callbacks on individual statements

When **cbStmt** is used in the reason field of the `s_cb_data` structure, the other fields in the structure will be defined as follows:

cb_data_p->cb_rtn The function to call before the given statement executes.

cb_data_p->obj A handle to the statement on which to place the callback (the allowable objects are listed in Table 27-6).

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

cb_data_p->time A pointer to an `s_vpi_time` structure, in which only the type is used, to indicate the type of time that will be returned when the callback is made. This type can be **vpiScaledRealTime**, **vpiSimTime**, or **vpiSuppressTime** if no time information is needed by the callback routine.

cb_data_p->value Not used.

cb_data_p->index Not used.

cb_data_p->user_data Data to be passed to the callback function.

Just before the indicated statement executes, the indicated function will be called with a pointer to a new `s_cb_data` structure, which will contain the following information:

cb_data_p->reason **cbStmt**.

cb_data_p->cb_rtn The same value as passed to **vpi_register_cb()**.

cb_data_p->obj A handle to the statement which is about to execute.

cb_data_p->time A pointer to an `s_vpi_time` structure, which will contain the current simulation time, of the type (**vpiScaledRealTime** or **vpiSimTime**) indicated in the call to **vpi_register_cb()**. If the value in the call to **vpi_register_cb()** was **vpiSuppressTime**, then the time pointer in the `s_cb_data` structure will be set to **NULL**.

cb_data_p->value Always **NULL**.

cb_data_p->index Always set to 0.

cb_data_p->user_data The value passed in as *user_data* in the call to **vpi_register_cb()**.

Multiple calls to **vpi_register_cb()** with the same data shall result in multiple callbacks.

Placing callbacks on statements that reside in protected portions of the code shall not be allowed and shall cause **vpi_register_cb()** to return a **NULL** with an appropriate error message printed.

27.33.1.2 Behavior by statement type

Every possible object within the ***stmt*** class qualifies for having a **cbStmt** callback placed on it. Each possible object is listed in Table 27-6, for further clarification.

Table 27-6—cbStmt callbacks

Object	Description
vpiBegin **vpiNamedBegin** **vpiFork** **vpiNamedFork**	One callback will occur prior to any of the statements within the block executing. The handle returned in the obj field will be the handle to the block object.
vpiIf **vpiIfElse**	The callback will occur before the condition expression in the if statement is evaluated.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 27-6—cbStmt callbacks *(continued)*

Object	Description
vpiWhile	A callback will occur prior to the evaluation of the condition expression on every iteration of the loop.
vpiRepeat	A callback will occur when the repeat statement is first encountered and on every subsequent iteration of the repeat loop.
vpiFor	A callback will occur prior to any of the control expressions being evaluated. Then on every iteration of the loop, a callback will occur prior to the evaluation of the incremental statement.
vpiForever	A callback will occur when the forever statement is first encountered and on every subsequent iteration of the forever loop.
vpiWait **vpiCase** **vpiAssignment** **vpiAssignStmt** **vpiDeassign** **vpiDisable** **vpiForce** **vpiRelease** **vpiEventStmt**	The callback will occur before the statement executes.
vpiDelayControl	The callback will occur when the delay control is encountered, before the delay occurs.
vpiEventControl	The callback will occur when the event control is encountered, before the event has occurred.
vpiTaskCall **vpiSysTaskCall**	The callback will occur before the given task is executed.

27.33.1.3 Registering callbacks on module-wide basis

vpi_register_cb() allows a handle to a module instance in the ***obj*** field of the `s_cb_data` structure. When this is done, the effect will be to place a callback on every statement that can have a callback placed on it.

When using **vpi_register_cb()** on a module object, the call will return a handle to a single callback object that can be passed to **vpi_remove_cb()** to remove the callback on every statement in the module instance.

Statements that reside in protected portions of the code shall not have callbacks placed on them.

27.33.2 Simulation time callbacks

The **vpi_register_cb()** can register callbacks to occur for simulation time reasons, including callbacks at the beginning or end of the execution of a particular time queue. The following time-related callback reasons are defined:

cbAtStartOfSimTime Callback shall occur before execution of events in a specified time queue. A callback can be set for any time, even if no event is present.

cbNBASynch Callback shall occur immediately before the nonblocking assignment events are processed.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

cbReadWriteSynch Callback shall occur after execution of events for a specified time. This time may be before or after nonblocking assignment events have been processed.

cbAtEndOfSimTime Callback shall occur after execution of nonblocking events, but before entering the read-only phase of the time slice.

cbReadOnlySynch Callback shall occur the same as for **cbReadWriteSynch**, except that writing values or scheduling events before the next scheduled event is not allowed.

cbNextSimTime Callback shall occur before execution of events in the next event queue.

cbAfterDelay Callback shall occur after a specified amount of time, before execution of events in a specified time queue. A callback can be set for any time, even if no event is present.

For reason **cbNextSimTime**, the time field in the time structure is ignored. The following fields shall need to be set before passing the `s_cb_data` structure to **vpi_register_cb()**:

cb_data_p->time->type This field shall be set to either **vpiScaledRealTime** or **vpiSimTime**, depending on what time information the application requires during the callback. **vpiSuppressTime** (or `NULL` for the `cb_data_p->time` field) will result in an error.

cb_data_p->[time->low,time->high,time->real]
These fields shall contain the requested time of the callback or the delay before the callback.

The following situations will generate an error, and no callback will be created:

— Attempting to place a **cbAtStartOfSimTime** callback with a delay of zero when simulation has progressed into a time slice and the application is not currently within a **cbAtStartOfSimTime** callback.
— Attempting to place a **cbReadWriteSynch** callback with a delay of zero at read-only synch time.

Placing a callback for **cbAtStartOfSimTime** and a delay of zero during a callback for reason **cbAtStartOfSimTime** will result in another **cbAtStartOfSimTime** callback occurring during the same time slice.

The *value* fields are ignored for all reasons with simulation time callbacks.

When the *cb_data_p->time->type* is set to **vpiScaledRealTime**, the *cb_data_p->obj* field shall be used as the object for determining the time scaling.

When a simulation time callback occurs, the application callback routine shall be passed a single argument, which is a pointer to an `s_cb_data` structure [this is not a pointer to the same structure that was passed to **vpi_register_cb()**]. The *time* structure shall contain the current simulation time. The *user_data* field shall be equivalent to the *user_data* field passed to **vpi_register_cb()**.

The callback application can use the information in the passed structure and information retrieved from other interface routines to perform the desired callback processing.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

27.33.3 Simulator action or feature callbacks

The **vpi_register_cb()** routine can register callbacks to occur for simulator action reasons or simulator feature reasons. *Simulator action reasons* are callbacks such as the end of compilation or end of simulation. *Simulator feature reasons* are software-product-specific features, such as restarting from a saved simulation state or entering an interactive mode. Actions are differentiated from features in that actions shall occur in all VPI-compliant products, whereas features might not exist in all VPI-compliant products.

The following action-related callbacks shall be defined:

cbEndOfCompile	End of simulation data structure compilation or build
cbStartOfSimulation	Start of simulation (beginning of time 0 simulation cycle)
cbEndOfSimulation	End of simulation (simulation ended because no more events remain in the event queue or a **$finish** system task executed)
cbError	Simulation run-time error occurred
cbPLIError	Simulation run-time error occurred in a PLI function call
cbTchkViolation	Timing check error occurred
cbSignal	A signal occurred

Examples of possible feature-related callbacks are as follows:

cbStartOfSave	Simulation save state command invoked
cbEndOfSave	Simulation save state command completed
cbStartOfRestart	Simulation restart from saved state command invoked
cbEndOfRestart	Simulation restart command completed
cbEnterInteractive	Simulation entering interactive debug mode (e.g., **$stop** system task executed)
cbExitInteractive	Simulation exiting interactive mode
cbInteractiveScopeChange	Simulation command to change interactive scope executed
cbUnresolvedSystf	Unknown user-defined system task/function encountered

The only fields in the `s_cb_data` structure that shall need to be set up for simulation action or feature callbacks are the *reason*, *cb_rtn*, and *user_data* (if desired) fields.

vpi_register_cb() can be used to set up a signal handler. To do this, set the reason field to cbSignal, and set the index field to one of the legal signals specified by the operating system. When this signal occurs, the simulator will trap the signal, proceed to a safe point (if possible), and then call the callback routine.

When a simulation action or feature callback occurs, the application routine shall be passed a pointer to an `s_cb_data` structure. The *reason* field shall contain the reason for the callback. For **cbTchkViolation** callbacks, the *obj* field shall be a handle to the timing check. For **cbInteractiveScopeChange**, *obj* shall be a

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

handle to the new scope. For **cbUnresolvedSystf**, *user_data* shall point to the name of the unresolved task/function. On a **cbError** callback, the routine **vpi_chk_error()** can be called to retrieve error information.

When an implementation restarts, the only VPI callbacks that shall exist are those for **cbStartOfRestart** and **cbEndOfRestart**.

NOTE—When an application registers for these two callbacks, the *user_data* field should not be a pointer into memory. The reason for this is that the executable used to restart an implementation may not be the exact same one used to save the implementation state. A typical use of the *user_data* field for these two callbacks would be to store the identifier returned from a call to **vpi_put_data()**.

With the exception of **cbStartOfRestart** and **cbEndOfRestart** callbacks, when a restart occurs all registered callbacks shall be removed.

The following example shows a callback application that reports CPU usage at the end of a simulation. If the application routine `setup_report_cpu()` is placed in the **vlog_startup_routines** list, it shall be called just after the simulator is invoked.

```
static PLI_INT32 initial_cputime_g;

void report_cpu()
{
       PLI_INT32 total = get_current_cputime() - initial_cputime_g;
       vpi_printf("Simulation complete. CPU time used: %d\n", total);
}

void setup_report_cpu()
{
       static s_cb_data cb_data_s = {cbEndOfSimulation, report_cpu};
       initial_cputime_g = get_current_cputime();
       vpi_register_cb(&cb_data_s);
}
```

27.34 vpi_register_systf()

vpi_register_systf()			
Synopsis:	Register user-defined system task/function callbacks.		
Syntax:	`vpi_register_systf(systf_data_p)`		
	Type	**Description**	
Returns:	vpiHandle	Handle to the callback object.	
	Type	**Name**	**Description**
Arguments:	p_vpi_systf_data	systf_data_p	Pointer to a structure with data about when callbacks should occur and the data to be passed.
Related routines:	Use vpi_register_cb() to register callbacks for simulation events.		

The VPI routine **vpi_register_systf()** shall register callbacks for user-defined system tasks or functions. Callbacks can be registered to occur when a user-defined system task/function is encountered during compilation or execution of Verilog HDL source code.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The *systf_data_p* argument shall point to a `s_vpi_systf_data` structure, which is defined in `vpi_user.h` and listed in Figure 27-18.

```
typedef struct t_vpi_systf_data
{
         PLI_INT32 type;         /* vpiSysTask, vpiSysFunc */
         PLI_INT32 sysfunctype;  /* vpiSysTask, vpi[Int,Real,Time,Sized,
                                                    SizedSigned]Func */
         PLI_BYTE8 *tfname;      /* first character must be '$' */
         PLI_INT32 (*calltf)(PLI_BYTE8 *);
         PLI_INT32 (*compiletf)(PLI_BYTE8 *);
         PLI_INT32 (*sizetf)(PLI_BYTE8 *);    /* for sized function
                                                 callbacks only */
         PLI_BYTE8 *user_data;
} s_vpi_systf_data, *p_vpi_systf_data;
```

Figure 27-18—s_vpi_systf_data structure definition

27.34.1 System task/function callbacks

User-defined Verilog system tasks and functions that use VPI routines can be registered with **vpi_register_systf()**. The following system task/function callbacks are defined:

The *type* field of the `s_vpi_systf_data` structure shall register the application to be a system task or a system function. The *type* field value shall be an integer constant of **vpiSysTask** or **vpiSysFunc**.

The *sysfunctype* field of the `s_vpi_systf_data` structure shall define the type of value that a system function shall return. The *sysfunctype* field shall be an integer constant of **vpiIntFunc**, **vpiRealFunc**, **vpiTimeFunc**, **vpiSizedFunc**, or **vpiSizedSignedFunc**. This field shall only be used when the *type* field is set to **vpiSysFunc**.

tfname is a character string containing the name of the system task/function as it will be used in Verilog source code. The name shall begin with a dollar sign ($) and shall be followed by one or more ASCII characters that are legal in Verilog HDL simple identifiers. These are the characters A through Z, a through z, 0 through 9, underscore (_), and the dollar sign ($). The maximum name length shall be the same as for Verilog HDL identifiers.

The *compiletf*, *calltf*, and *sizetf* fields of the `s_vpi_systf_data` structure shall be pointers to the user-provided applications that are to be invoked by the system task/function callback mechanism. One or more of the *compiletf*, *calltf*, and *sizetf* fields can be set to NULL if they are not needed. Callbacks to the applications pointed to by the *compiletf* and *sizetf* fields shall occur when the simulation data structure is compiled or built (or for the first invocation if the system task/function is invoked from an interactive mode). Callbacks to the application pointed to by the *calltf* routine shall occur each time the system task/function is invoked during simulation execution.

The *sizetf* application shall only be called if the PLI application type is **vpiSysFunc** and the *sysfunctype* is **vpiSizedFunc** or **vpiSizedSignedFunc**. If no sizetf is provided, a user-defined system function of type **vpiSizedFunc** or **vpiSizedSignedFunc** shall return 32 bits.

The contents of the *user_data* field of the `s_vpi_systf_data` structure shall be the only argument passed to the *compiletf*, *sizetf*, and *calltf* routines when they are called. This argument shall be of the type "`PLI_BYTE8 *`".

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

The following two examples illustrate allocating and filling in the `s_vpi_systf_data` structure and calling the **vpi_register_systf()** function. These examples show two different C programming methods of filling in the structure fields. A third method is shown in 27.34.3.

```
/*
 * VPI registration data for a $list_nets system task
 */
void listnets_register()
{
  s_vpi_systf_data tf_data;
  tf_data.type      = vpiSysTask;
  tf_data.tfname    = "$list_nets";
  tf_data.calltf    = ListCall;
  tf_data.compiletf = ListCheck;
  vpi_register_systf(&tf_data);
}

/*
 * VPI registration data for a $my_random system function
 */
void my_random_init()
{
  s_vpi_systf_data func_data;
  p_vpi_systf_data func_data_p = &func_data;
  PLI_BYTE8 *my_workarea;
  my_workarea = malloc(256);
  func_data_p->type        = vpiSysFunc;
  func_data_p->sysfunctype = vpiSizedFunc;
  func_data_p->tfname      = "$my_random";
  func_data_p->calltf      = my_random;
  func_data_p->compiletf   = my_random_compiletf;
  func_data_p->sizetf      = my_random_sizetf;
  func_data_p->user_data   = my_workarea;
  vpi_register_systf(func_data_p);
}
```

27.34.2 Initializing VPI system task/function callbacks

A means of initializing system task/function callbacks and performing any other desired task just after the simulator is invoked shall be provided by placing routines in a `NULL`-terminated static array, **vlog_startup_routines**. A C function using the array definition shall be provided as follows:

```
void (*vlog_startup_routines[]) ();
```

This C function shall be provided with a VPI-compliant product. Entries in the array shall be added by the user. The location of **vlog_startup_routines** and the procedure for linking **vlog_startup_routines** with a software product shall be defined by the product vendor.

NOTE—Callbacks can also be registered or removed at any time during an application routine, not just at startup time.

This array of C functions shall be for registering system tasks and functions. User-defined system tasks and functions that appear in a compiled description shall generally be registered by a routine in this array.

The following example uses **vlog_startup_routines** to register the system task and system function that were defined in the examples in 27.34.1.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

A tool vendor shall supply a file that contains the **vlog_startup_routines** array. The names of the PLI application register functions shall be added to this vendor-supplied file.

```
extern void listnets_register();
extern void my_random_init();
void (*vlog_startup_routines[]) () =
{
  listnets_register,
  my_random_init,
  0
}
```

27.34.3 Registering multiple system tasks and functions

Multiple system tasks and functions can be registered at least two different ways:

— Allocate and define separate `s_vpi_systf_data` structures for each system task/function, and call **vpi_register_systf()** once for each structure. This is the method that was used by the examples in 27.34.1 and 27.34.2.

— Allocate a static array of `s_vpi_systf_data` structures, and call **vpi_register_systf()** once for each structure in the array. If the final element in the array is set to zero, then the calls to **vpi_register_systf()** can be placed in a loop that terminates when it reaches the 0.

The following example uses a static structure to declare three system tasks and functions and places **vpi_register_systf()** in a loop to register them:

```
/*In a vendor product file which contains vlog_startup_routines ...*/
extern void register_my_systfs();
extern void my_init();
void (*vlog_startup_routines[])() =
{
       setup_report_cpu,       /* user routine example in 27.33.3 */
       register_my_systfs,     /* user routine listed below */
       0                       /* must be last entry in list */
}

/* In a user provided file... */
void register_my_systfs()
{
   static s_vpi_systf_data systfTestList[] = {
       {vpiSysTask, 0, "$my_task", my_task_calltf, my_task_comptf,0,0},
       {vpiSysFunc, vpiIntFunc, "$my_int_func", my_int_func_calltf,
          my_int_func_comptf, 0,0},
       {vpiSysFunc, vpiSizedFunc, "$my_sized_func",
          my_sized_func_calltf, my_sized_func_comptf,
          my_sized_func_sizetf,0},
       0};

   p_vpi_systf_data systf_data_p = &(systfTestList[0]);

   while (systf_data_p->type)
       vpi_register_systf(systf_data_p++);
}
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

27.35 vpi_remove_cb()

vpi_remove_cb()			
Synopsis:	Remove a simulation-related callback registered with vpi_register_cb().		
Syntax:	`vpi_remove_cb(cb_obj)`		
	Type	**Description**	
Returns:	PLI_INT32	1 (true) if successful; 0 (false) on a failure.	
	Type	**Name**	**Description**
Arguments:	vpiHandle	cb_obj	Handle to the callback object.
Related routines:	Use vpi_register_cb() to register callbacks for simulation events.		

The VPI routine **vpi_remove_cb()** shall remove callbacks that were registered with *vpi_register_cb()*. The argument to this routine shall be a handle to the callback object. The routine shall return a 1 (true) if successful and a 0 (false) on a failure. After **vpi_remove_cb()** is called with a handle to the callback, the handle is no longer valid.

27.36 vpi_scan()

vpi_scan()			
Synopsis:	Scan the Verilog HDL hierarchy for objects with a one-to-many relationship.		
Syntax:	`vpi_scan(itr)`		
	Type	**Description**	
Returns:	vpiHandle	Handle to an object.	
	Type	**Name**	**Description**
Arguments:	vpiHandle	itr	Handle to an iterator object returned from vpi_iterate().
Related routines:	Use vpi_iterate() to obtain an iterator handle. Use vpi_handle() to obtain handles to an object with a one-to-one relationship. Use vpi_handle_multi() to obtain a handle to an object with a many-to-one relationship.		

The VPI routine **vpi_scan()** shall traverse the instantiated Verilog HDL hierarchy and return handles to objects as directed by the iterator *itr*. The iterator handle shall be obtained by calling **vpi_iterate()** for a specific object type. Once **vpi_scan()** returns NULL, the iterator handle is no longer valid and cannot be used again.

The following example application uses **vpi_iterate()** and **vpi_scan()** to display each net (including the size for vectors) declared in the module. The example assumes it shall be passed a valid module handle.

```
void display_nets(mod)
vpiHandle mod;
{
       vpiHandle net;
       vpiHandle itr;
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
            vpi_printf("Nets declared in module %s\n",
            vpi_get_str(vpiFullName, mod));

            itr = vpi_iterate(vpiNet, mod);
            while (net = vpi_scan(itr))
            {
                vpi_printf("\t%s", vpi_get_str(vpiName, net));
                if (vpi_get(vpiVector, net))
                {
                    vpi_printf(" of size %d\n", vpi_get(vpiSize, net));
                }
                else vpi_printf("\n");
            }
        }
```

27.37 vpi_vprintf()

vpi_vprintf()			
Synopsis:	Write to the output channel of the software product that invoked the PLI application and the current product log file using varargs that are already started.		
Syntax:	`vpi_vprintf(format, ap)`		
Returns:	**Type**	**Description**	
	PLI_INT32	The number of characters written.	
Arguments:	**Type**	**Name**	**Description**
	PLI_BYTE8 *	format	A format string using the C printf() format.
	va_list	ap	An already started varargs list.
Related routines:	Use vpi_printf() to write a finite number of arguments. Use vpi_mcd_printf() to write to an opened file. Use vpi_mcd_vprintf() to write a variable number of arguments to an opened file.		

This routine performs the same function as **vpi_printf()**, except that varargs have already been started.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

28. Protected envelopes

28.1 General

Protected envelopes specify a region of text that shall be transformed prior to analysis by the source language processor. These regions of text are structured to provide the source language processor with the specification of the cryptographic algorithm, key, envelope attributes, and textual design data.

All information that identifies a protected envelope is introduced by the **protect** pragma (see 19.10). This pragma is reserved by this standard for the description of protected envelopes and is the prefix for specifying the regions and processing specifications for each protected envelope. Additional information is associated with the pragma by appending pragma expressions. The pragma expressions of the **protect** pragma are evaluated in sequence from left to right. Interpretation of protected envelopes shall not be altered based on whether the sequence of pragma expressions occurs in a single **protect** pragma directive or in a sequence of **protect** pragma directives. In this clause, unless otherwise specified, pragma directives, pragma keywords, and pragma expressions shall refer to occurrences of **protect** pragma directives and their associated pragma keywords and pragma expressions.

Envelopes may be defined for either of two modes of processing. *Encryption envelopes* specify the pragma expressions for encrypting source text regions. An encryption envelope begins in the source text when a **begin** pragma expression is encountered. The end of the encryption envelope occurs at the point where an **end** pragma expression is encountered. The **end** pragma expression is said to close the envelope and shall be associated with the most recent **begin** pragma expression.

Decryption envelopes specify the pragma expressions for decrypting encrypted text regions. A decryption envelope begins in the source text when a **begin_protected** pragma expression is encountered. The end of the decryption envelope occurs at the point where an **end_protected** pragma expression is encountered. The **end_protected** pragma expression is said to close the envelope and shall be associated with the most recent **begin_protected** that has not already been closed. Decryption envelopes may contain other envelopes within their enclosed data block. The number of nested decryption envelopes that can be processed is implementation-specified; however, that number shall be no less than 8. Code that is contained within a decryption envelope is said to be protected.

Pragma expressions that precede **begin** or **begin_protected** are designated as *envelope keywords*. Pragma expressions that follow the **begin**/**begin_protected** keywords and precede the associated **end**/**end_protected** keywords are designated as *content keywords*. Content keywords are pragma expressions that are within the region of text that is processed during encryption or decryption of a protected envelope.

28.2 Processing protected envelopes

Two modes of processing are defined for protected envelopes. *Envelope encryption* is the process of recognizing encryption envelopes in the source text and transforming them into decryption envelopes. *Envelope decryption* is the process of recognizing decryption envelopes in the input text and transforming them into the corresponding cleartext for the compilation step that follows.

Tools that process the Verilog HDL shall perform envelope decryption for all decryption envelopes contained in the source text, where the proper key is supplied by the user. Tools that perform envelope encryption shall only be required to process the **protect** pragma directives and shall apply no other interpretation to text that is not part of a **protect** pragma directive.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

28.2.1 Encryption

Verilog tools that provide encryption services shall transform source text containing encryption envelopes by replacing each encryption envelope with a decryption envelope formed by encrypting the source text of the encryption envelope according to the specified pragma expressions.

Source text that is not contained in an encryption envelope shall not be modified by the encrypting language processor, unless otherwise specified.

Decryption envelopes are formed from encryption envelopes by transforming the specified encryption envelope pragma expressions into decryption envelope pragma expressions and decryption content pragma expressions. The body of the encryption envelope is encrypted using the specified key, referred to as the *exchange key*, and is recorded in the decryption envelope as a **data_block**.

Encryption algorithms that use the same key to encrypt cleartext and decrypt the corresponding ciphertext are said to be *symmetric*. Algorithms that require different keys to encrypt and decrypt are said to be *asymmetric*. This description may be applied to both the algorithm and the key.

Tools that provide encryption services may support *session keys* to limit exposure to the exchange key that is specified by the IP author using the encryption envelope pragma expressions. A session key is created in an unspecified manner to encrypt the data from the encryption envelope. A copy of the session key is encrypted using the exchange key and is recorded in a **key_block** in the decryption envelope. Next, the body of the encryption envelope is encrypted using the session key and is recorded in the decryption envelope as a **data_block**.

The following example shows the use of the **protect** pragma to specify encryption of design data. The encryption method is a simple substitution cipher where each alphabetic character is replaced with the 13th character in alphabetic sequence, commonly referred to as “rot13”. Nonalphabetic characters are not substituted. The following design data contain an encryption envelope that specifies the desired protection.

```
module secret (a, b);
    input a;
    output b;

`pragma protect encoding=(enctype="raw")
`pragma protect data_method="x-caesar", data_keyname="rot13", begin
`pragma protect runtime_license=(library="lic.so",feature="runSecret",entry="chk", match=42)
    reg b;

    initial
       begin
          b = 0;
       end

    always
       begin
          #5 b = a;
       end
`pragma protect end

endmodule // secret
```

After encryption processing, the following design data are produced. The decryption envelope is written with a “raw” encoding to make the substitution encryption directly visible.

NOTE—The encoded line beginning "`centzn" is actually one long line, but it wraps over to the following line on the printed page.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
module secret (a, b);
    input a;
    output b;

`pragma protect encoding=(enctype="raw")
`pragma protect data_method="x-caesar", data_keyname="rot13", begin_protected
`pragma protect data_block encoding=(enctype="raw", bytes=190)
'centzn cebgrpg ehagvzr_yvprafr=(yvoenel="yvp.fb",srngher="ehaFrperg",
ragel="pux",zngpu=42)
   ert o;

   vavgvny
     ortva
        o = 0;
     raq

   nyjnlf
     ortva
        #5 o = n;
     raq
`pragma protect end_protected
`pragma reset protect

endmodule // secret
```

NOTE—Products that include cryptographic algorithms may be subject to government regulations in many jurisdictions. Users of this standard are advised to seek the advice of competent counsel to determine their obligations under those regulations.

28.2.2 Decryption

Verilog tools that support decrypting compilation shall transform source text containing decryption envelopes by replacing each decryption envelope with the decrypted source text from the **data_block**, according to the specified pragma expressions. The substituted text may contain usages of text macros, which shall be substituted after replacement of the decryption envelope. The substituted text may also contain decryption envelopes, which shall be decrypted and substituted after replacement of their enclosing decryption envelope.

28.3 Protect pragma directives

Protected envelopes are lexical regions delimited by **protect** pragma directives. The effect of a particular **protect** pragma directive is specified by its pragma expressions. This standard defines the pragma keyword names listed in Table 28-1 for use with the **protect** pragma. These pragma keywords are defined in 28.4 with a specification of how each participates in the encryption and decryption processing modes.

Table 28-1—protect pragma keywords

Pragma keyword	Description
begin	Opens a new encryption envelope
end	Closes an encryption envelope
begin_protected	Opens a new decryption envelope
end_protected	Closes a decryption envelope
author	Identifies the author of an envelope
author_info	Specifies additional author information

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 28-1—protect pragma keywords *(continued)*

Pragma keyword	Description
encrypt_agent	Identifies the encryption service
encrypt_agent_info	Specifies additional encryption service information
encoding	Specifies the coding scheme for encrypted data
data_keyowner	Identifies the owner of the data encryption key
data_method	Identifies the data encryption algorithm
data_keyname	Specifies the name of the data encryption key
data_public_key	Specifies the public key for data encryption
data_decrypt_key	Specifies the data session key
data_block	Begins an encoded block of encrypted data
digest_keyowner	Identifies the owner of the digest encryption key
digest_key_method	Identifies the digest encryption algorithm
digest_keyname	Specifies the name of the digest encryption key
digest_public_key	Specifies the public key for digest encryption
digest_decrypt_key	Specifies the digest session key
digest_method	Specifies the digest computation algorithm
digest_block	Specifies a message digest for data integrity
key_keyowner	Identifies the owner of the key encryption key
key_method	Specifies the key encryption algorithm
key_keyname	Specifies the name of the key encryption key
key_public_key	Specifies the public key for key encryption
key_block	Begins an encoded block of key data
decrypt_license	Specifies licensing constraints on decryption
runtime_license	Specifies licensing constraints on simulation
comment	Uninterpreted documentation string
reset	Resets pragma keyword values to default
viewport	Modifies scope of access into decryption envelope

The scope of **protect** pragma directives is completely lexical and not associated with any declarative region or declaration in the HDL text itself. This lexical scope may cross file boundaries and included files.

In protected envelopes where a specific pragma keyword is absent, the Verilog tool shall use the default value. Verilog tools that perform encryption should explicitly output all relevant pragma keywords for each envelope in order to avoid unintended interpretations during decryption. Further robustness can be achieved by appending a **reset** pragma keyword after each envelope.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

28.4 Protect pragma keywords

28.4.1 begin

28.4.1.1 Syntax

begin

28.4.1.2 Description

ENCRYPTION INPUT: The **begin** pragma expression is used in the input text to indicate to an encrypting tool the point at which encryption shall begin.

Nesting of pragma **begin-end** blocks shall be an error. There may be **begin_protected-end_protected** blocks containing previously encrypted content inside such a block. They are simply treated as a byte stream and encrypted as if they were text.

ENCRYPTION OUTPUT: The **begin** pragma expression is replaced in the encryption output stream by the **begin_protected** pragma expression. Following **begin_protected**, all pragma expressions required as encryption output shall be generated prior to the **end_protected** pragma expression. Protected envelopes should be completely self-contained to avoid any undesired interaction when multiple encrypted models exist in the decryption input stream. The **data_block** and **key_block** pragma expressions introduce the encrypted data or keys and will always be found within a **begin_protected-end_protected** envelope. All text, including comments and other **protect** pragmas, occurring between the **begin** pragma expression and the corresponding **end** pragma expression shall, unless otherwise specified, be encrypted and placed in the encryption output stream using the **data_block** pragma expression. An unspecified length of arbitrary comment text may be added by the encrypting tool to the beginning and end of the input text in order to prevent known text attacks on the encrypted content of the **data_block**.

DECRYPTION INPUT: none

28.4.2 end

28.4.2.1 Syntax

end

28.4.2.2 Description

ENCRYPTION INPUT: The **end** pragma expression is used in the input cleartext to indicate the end of the region that shall be encrypted. The **end** pragma expression is replaced in the encryption output stream by the **end_protected** pragma expression.

ENCRYPTION OUTPUT: none

DECRYPTION INPUT: none

28.4.3 begin_protected

28.4.3.1 Syntax

begin_protected

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

28.4.3.2 Description

ENCRYPTION INPUT: When a **begin_protected-end_protected** block is found in an input file during encryption, its contents are treated as input cleartext. This allows a previously encrypted model to be reencrypted as a portion of a larger model. Any other **protect** pragmas inside the **begin_protected-end_protected** block shall not be interpreted and shall not override pragmas in effect. Nested encryption must not corrupt pragma values in the current encryption in process.

ENCRYPTION OUTPUT: The **begin_protected** pragma expression, and the entire content of the protected envelope up to the corresponding **end_protect** pragma expression, shall be encrypted into the current **data_block** as specified by the current method and keys.

DECRYPTION INPUT: The **begin_protected** pragma expression begins a previously encrypted region. A decrypting tool shall accumulate all the pragma expressions in the block for use in decryption of the block.

28.4.4 end_protected

28.4.4.1 Syntax

end_protected

28.4.4.2 Description

ENCRYPTION INPUT: This pragma expression indicates the end of a previous **begin_protected** block. This indicates that the block is complete, and subsequent pragma expression values will be accumulated for the next envelope.

ENCRYPTION OUTPUT: The **end_protected** pragma expression following the corresponding **begin_protected** pragma expression shall be encrypted into the current **data_block** as specified by the current method and keys.

DECRYPTION INPUT: The **end_protected** pragma expression indicates the end of a set of pragmas that are sufficient to decrypt the current block.

28.4.5 author

28.4.5.1 Syntax

author = `<string>`

28.4.5.2 Description

ENCRYPTION INPUT: The **author** pragma expression specifies a string that identifies the name of the IP author. It is distinct from the comment pragma expression so that this information can be recognized without need for parsing of a comment string value.

ENCRYPTION OUTPUT: If present in the encryption envelope, the **author** pragma expression shall be placed in a pragma directive enclosed within the protected envelope, but shall not be encrypted into the **data_block**. Otherwise, it is copied without change into the output stream.

DECRYPTION INPUT: none

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

28.4.6 author_info

28.4.6.1 Syntax

author_info = <string>

28.4.6.2 Description

ENCRYPTION INPUT: The **author_info** pragma expression specifies a string that contains additional information provided by the IP author. It is distinct from the comment pragma expression so that this information can be recognized without need for parsing of a comment string value.

ENCRYPTION OUTPUT: If present in the encryption envelope, the **author_info** pragma expression shall be placed in a pragma directive enclosed within the protected envelope, but shall not be encrypted into the **data_block**. Otherwise, it is copied without change into the output stream.

DECRYPTION INPUT: none

28.4.7 encrypt_agent

28.4.7.1 Syntax

encrypt_agent = <string>

28.4.7.2 Description

ENCRYPTION INPUT: none

ENCRYPTION OUTPUT: The **encrypt_agent** pragma expression specifies a string that identifies the name of the encrypting tool. The encrypting tool shall generate this pragma expression and place it in a pragma directive enclosed within the protected envelope, but shall not encrypt it into the **data_block**.

DECRYPTION INPUT: none

28.4.8 encrypt_agent_info

28.4.8.1 Syntax

encrypt_agent_info = <string>

28.4.8.2 Description

ENCRYPTION INPUT: none

ENCRYPTION OUTPUT: The **encrypt_agent_info** pragma expression specifies a string that contains additional information provided by the encrypting tool. If provided, the **encrypt_agent_info** pragma expression shall be placed within a pragma directive enclosed within the protected envelope, but shall not be encrypted into the **data_block**.

DECRYPTION INPUT: none

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

28.4.9 encoding

28.4.9.1 Syntax

encoding = **(** **enctype** = <string> **,** **line_length** = <number> **,** **bytes** = <number> **)**

28.4.9.2 Description

ENCRYPTION INPUT: The **encoding** pragma expression specifies how the **data_block**, **digest_block**, and **key_block** content shall be encoded. This encoding ensures that all binary data produced in the encryption process can be treated as text. If an **encoding** pragma expression is present in the input stream, it specifies how the output shall be encoded.

The **encoding** pragma expression shall be a pragma_expression value containing encoding subkeywords separated by white space. The following subkeywords are defined for the value of the **encoding** pragma expression:

enctype=<string> The method for calculating the encoding. This standard specifies the identifiers in Table 28-2 as string values for the enctype subkeyword. These identifiers are associated with their respective encoding algorithms. The required methods are standard in every implementation. Optional identifiers are implementation-specific, but are required to use these identifiers for the corresponding encoding algorithm. Additional identifier values and their corresponding encoding algorithms are implementation-defined.

Table 28-2—Encoding algorithm identifiers

enctype	Required /optional	Encoding algorithm
uuencode	Required	IEEE Std 1003.1 (uuencode historical algorithm)
base64	Required	IETF RFC 2045 [also IEEE Std 1003.1 (uuencode -m)]
quoted-printable	Optional	IETF RFC 2045
raw	Optional	Identity transformation; No encoding shall be performed, and the data may contain nonprintable characters.

line_length=<number> The maximum number of characters (after any encoding) in a single line of the **data_block**. Insertion of line breaks in the **data_block** after encryption and encoding allows the generated text files to be usable by commonly available text tools.

bytes=<number> The number of bytes in the original block of data before any encoding or the addition of line breaks. This encoding keyword shall be ignored in the encryption input.

ENCRYPTION OUTPUT: The **encoding** directive shall be output in each **begin_protected-end_protected** block to explicitly specify the encoding used by the **encrypt_agent**. A tool may choose to encode the data even if no **encoding** pragma expression was found in the input stream and shall output the corresponding

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

encoding pragma expression. The tool shall generate an encoding descriptor that specifies in the bytes keyword the number of bytes in the original block of data.

The **data_block**, **data_public_key**, **data_decrypt_key**, **digest_block**, **key_block**, and **key_public_key** are all encoded using this encoding. If separate encoding is desired for each of these fields, then multiple **encoding** pragma expressions can be given in the input stream prior to each of the above pragma expressions. The **bytes** value is added by the encrypting tool for each block that it encrypts.

DECRYPTION INPUT: During decryption, the **encoding** directive is used to find the encoding algorithm used and the size of actual data.

28.4.10 data_keyowner

28.4.10.1 Syntax

data_keyowner = `<string>`

28.4.10.2 Description

ENCRYPTION INPUT: The **data_keyowner** specifies the legal entity or tool that provided the keys used for encryption and decryption of the data. This pragma keyword permits use of a third-party key, distinct from one associated with either **author** or **encrypt_agent**. The **data_keyowner** value is used by the encrypting tool to select the key used to encrypt the **data_block**. The values for **data_keyname**, **data_decrypt_key**, and **data_public_key** must be unique for the specified **data_keyowner**.

ENCRYPTION OUTPUT: The **data_keyowner** shall be unchanged in the output file, except where a digital signature is used, in which case it is encrypted with the **key_method** and placed in a **key_block**.

DECRYPTION INPUT: During decryption, the **data_keyowner** is combined with the **data_keyname** or **data_public_key** to determine the appropriate secret/private key to use during decryption of the **data_block**.

28.4.11 data_method

28.4.11.1 Syntax

data_method = `<string>`

28.4.11.2 Description

ENCRYPTION INPUT: The **data_method** pragma expression specifies the encryption algorithm that shall be used to encrypt subsequent **begin-end** blocks. The encryption method is an identifier that is commonly associated with a specific encryption algorithm.

This standard specifies the identifiers in Table 28-3 as string values for the data_method pragma expression. These identifiers are associated with their respective encryption types. The required methods are standard in every implementation. Optional identifiers are implementation-specific, but are required to use these identifiers for the corresponding cipher. Additional identifier values and their corresponding ciphers are implementation-defined.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 28-3—Encryption algorithm identifiers

Identifier	Required /optional	Encryption algorithm
des-cbc	Required	Data Encryption Standard (DES) in CBC mode, see FIPS 46-3[a].
3des-cbc	Optional	Triple DES in CBC mode, see FIPS 46-3; ANSI X9.52-1998.
aes128-cbc	Optional	Advanced Encryption Standard (AES) with 128-bit key, see FIPS 197.
aes256-cbc	Optional	AES in CBC mode, with 256-bit key.
aes192-cbc	Optional	AES with 192-bit key.
blowfish-cbc	Optional	Blowfish in CBC mode, see Schneier (Blowfish).
twofish256-cbc	Optional	Twofish in CBC mode, with 256-bit key, see Schneier (Twofish).
twofish192-cbc	Optional	Twofish with 192-bit key.
twofish128-cbc	Optional	Twofish with 128-bit key.
serpent256-cbc	Optional	Serpent in CBC mode, with 256-bit key, see Anderson, et al.
serpent192-cbc	Optional	Serpent with 192-bit key.
serpent128-cbc	Optional	Serpent with 128-bit key.
cast128-cbc	Optional	CAST-128 in CBC mode, see IETF RFC 2144.
rsa	Optional	RSA, see IETF RFC 2437.
elgamal	Optional	ElGamal, see ElGamal.
pgp-rsa	Optional	OpenPGP RSA key, see IETF RFC 2440.

[a]For information on references, see Clause 2.

ENCRYPTION OUTPUT: The **data_method** shall be unchanged in the output file, except where a digital signature is used, in which case it is encrypted with the **key_method** and placed in a **key_block**.

DECRYPTION INPUT: The **data_method** specifies the algorithm that should be used to decrypt the data_block.

28.4.12 data_keyname

28.4.12.1 Syntax

data_keyname = `<string>`

28.4.12.2 Description

ENCRYPTION INPUT: The **data_keyname** pragma expression specifies the name of the key, or key pair for an asymmetric encryption algorithm, that should be used to decrypt the **data_block**. It shall be an error to specify a **data_keyname** that is not a member of the list of keys known for the given **data_keyowner.**

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

ENCRYPTION OUTPUT: When a **data_keyname** is provided in the input, it indicates the key that should be used for encrypting the data. The encrypting tool shall combine this pragma expression with the **data_keyowner** and determine the key to use. The **data_keyname** itself shall be output as cleartext in the output file except where a digital envelope is used. For a digital envelope mechanism, the **data_keyname** is encrypted using **key_method** and **key_keyname**/**key_public_key** and encoded in the **key_block**.

DECRYPTION INPUT: The **data_keyname** value is combined with the **data_keyowner** to select a single key that shall be used to decrypt the **data_block** from the protected envelope**.**

28.4.13 data_public_key

28.4.13.1 Syntax

data_public_key

28.4.13.2 Description

ENCRYPTION INPUT: The **data_public_key** pragma expression specifies that the next line of the file contains the encoded value of the public key to be used to encrypt the data. The encoding is specified by the **encoding** pragma expression that is currently in effect. If both **data_public_key** and **data_keyname** are present, then they must refer to the same key.

ENCRYPTION OUTPUT: The **data_public_key** pragma expression shall be output in each protected block for which it is used, followed by the encoded value. The **data_method** and **data_public_key** can be combined to fully specify the required encryption.

DECRYPTION INPUT: The **data_keyowner** and **data_method** can be combined with the **data_public_key** to determine whether the decrypting tool knows the corresponding private key to decrypt a given **data_block**. If the decrypting tool can compute the required key, the model can be decrypted (if licensing allows it).

28.4.14 data_decrypt_key

28.4.14.1 Syntax

data_decrypt_key

28.4.14.2 Description

ENCRYPTION INPUT: The **data_decrypt_key** indicates that the next line contains the encoded value of the key that will decrypt the **data_block**. This pragma expression should only be used when digital signatures are used. An IP author can generate a key and use it to encrypt the cleartext. This encrypted text is then stored in the output file as the **data_block**. Then the **data_method** and **data_decrypt_key** are encrypted using the **key_method** and stored in the output file as the contents of the **key_block**. The **data_block** itself is not reencrypted; only the information about the data key is.

ENCRYPTION OUTPUT: The **data_decrypt_key** is output as part of the encrypted content of the **key_block**. The value is encoded as specified by the **encoding** pragma expression.

DECRYPTION INPUT: Upon determining that a digital signature was in use for a given protected region, the decrypting tool must decrypt the **key_block** to find the **data_decrypt_key** and **data_method** that in turn can be used to decrypt the **data_block**.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

28.4.15 data_block

28.4.15.1 Syntax

data_block

28.4.15.2 Description

ENCRYPTION INPUT: It shall be an error if a **data_block** is found in an input file unless it is contained within a previously generated **begin_protected-end_protected** block, in which case it is ignored.

ENCRYPTION OUTPUT: The **data_block** pragma expression indicates that a data block begins on the next line in the file. The encrypting tool shall take each **begin-end** block, encrypt the contents as specified by the **data_method** pragma expression, and then encode the block as specified by the **encoding** pragma expression. The resultant text shall be output.

DECRYPTION INPUT: The **data_block** is first read in the encoded form. The encoding shall be reversed, and then the block shall be internally decrypted.

28.4.16 digest_keyowner

28.4.16.1 Syntax

digest_keyowner = `<string>`

28.4.16.2 Description

ENCRYPTION INPUT: The **data_keyowner** specifies the legal entity or tool that provided the keys used for encryption and decryption of the data. This pragma keyword permits use of a third-party key, distinct from one associated with either **author** or **encrypt_agent**. The **digest_keyowner** value is used by the encrypting tool to select the key used to encrypt the **digest_block**. The values for **digest_keyname**, **digest_decrypt_key**, and **digest_public_key** must be unique for the specified **digest_keyowner**. If no **digest_keyowner** is specified in the input, then the default value of **digest_keyowner** shall be the current value of **data_keyowner**.

ENCRYPTION OUTPUT: The **digest_keyowner** shall be unchanged in the output file, except where a digital signature is used, in which case it is encrypted with the **digest_key_method** and placed in a **digest_key_block**.

DECRYPTION INPUT: During decryption, the **digest_keyowner** is combined with the **digest_keyname** or **digest_public_key** to determine the appropriate secret/private key to use during decryption of the **digest_block**.

28.4.17 digest_key_method

28.4.17.1 Syntax

digest_key_method = `<string>`

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

28.4.17.2 Description

ENCRYPTION INPUT: The **digest_key_method** pragma expression indicates the encryption algorithm that shall be used to encrypt subsequent **digest_block** contents. The values specified for **digest_key_method** to identify encryption algorithms are the same as those specified for **data_method**. If no **digest_key_method** is specified in the input, then the default value of **digest_key_method** shall be the current value of **data_method**.

ENCRYPTION OUTPUT: The **digest_key_method** shall be unchanged in the output file, except where a digital signature is used, in which case it is encrypted with the **key_method** algorithm and uses the key found in the **key_block**.

DECRYPTION INPUT: The **digest_key_method** indicates the algorithm that shall be used to decrypt the **digest_block**.

28.4.18 digest_keyname

28.4.18.1 Syntax

digest_keyname = `<string>`

28.4.18.2 Description

ENCRYPTION INPUT: The **digest_keyname** pragma expression provides the name of the key, or key pair for an asymmetric encryption algorithm, that shall be used to decrypt the **digest_block**. It shall be an error to specify a **digest_keyname** that is not a member of the list of keys known for the given **digest_keyowner.** If no **digest_keyname** is specified in the input, then the default value of **digest_keyname** shall be the current value of **data_keyname**.

ENCRYPTION OUTPUT: When a **digest_keyname** is provided in the input, it indicates the key that shall be used for encrypting the data. The encrypting tool must be able to combine this pragma expression with the **digest_keyowner** and determine the key to use. The **digest_keyname** itself shall be output as cleartext in the output file except where a digital envelope is used. For a digital envelope mechanism, the **digest_keyname** is encrypted using **key_method** and **key_keyname**/**key_public_key** and encoded in the **key_block**.

DECRYPTION INPUT: The **digest_keyname** value is combined with the **digest_keyowner** to select a single key that shall be used to decrypt the **digest_block** from the protected envelope**.**

28.4.19 digest_public_key

28.4.19.1 Syntax

digest_public_key

28.4.19.2 Description

ENCRYPTION INPUT: The **digest_public_key** pragma expression indicates that the next line of the file contains the encoded value of the public key used to encrypt the digest. The encoding is specified by the **encoding** pragma expression that is currently in effect. If both **digest_public_key** and **digest_keyname** are present, then they must refer to the same key. If no **digest_public_key** is specified in the input, then the default value of **digest_public_key** shall be the current value of **data_public_key**.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

ENCRYPTION OUTPUT: The **digest_public_key** pragma expression shall be output in each protected block for which it is used, followed by the encoded value. The **digest_key_method** and **digest_public_key** can be combined to fully specify the required encryption.

DECRYPTION INPUT: The **digest_keyowner** and **digest_key_method** can be combined with the **digest_public_key** to determine whether the decrypting tool knows the corresponding private key to decrypt a given **digest_block**. If the decrypting tool can compute the required key, the model can be decrypted (if licensing allows it).

28.4.20 digest_decrypt_key

28.4.20.1 Syntax

digest_decrypt_key

28.4.20.2 Description

ENCRYPTION INPUT: The **digest_decrypt_key** indicates that the next line contains the encoded value of the key that will decrypt the **digest_block**. This pragma expression should only be used when digital signatures are used. An IP author can generate a key and use it to encrypt the digest. This encrypted text is then stored in the output file as the **digest_block**. Then the **digest_key_method** and **digest_decrypt_key** are encrypted using the key_method and stored in the output file as the contents of the **key_block**. The **digest_block** itself is not reencrypted; only the information about the digest key is. If no **digest_decrypt_key** is specified in the input, then the default value of **digest_decrypt_key** shall be the current value of **data_decrypt_key**.

ENCRYPTION OUTPUT: The **digest_decrypt_key** is output as part of the encrypted content of the **key_block**. The value is encoded as specified by the **encoding** pragma expression.

DECRYPTION INPUT: Upon determining that a digital signature was in use for a given protected region, the decrypting tool must decrypt the **key_block** to find the **digest_decrypt_key** and **digest_key_method** that in turn can be used to decrypt the digest block.

28.4.21 digest_method

28.4.21.1 Syntax

digest_method = `<string>`

28.4.21.2 Description

ENCRYPTION INPUT: The **digest_method** pragma expression specifies the message digest algorithm that shall be used to generate message digests for subsequent **data_block** and **key_block** output. The string value is an identifier commonly associated with a specific message digest algorithm.

This standard specifies the values Table 28-4 for the **digest_method** pragma expression. Additional identifier values are implementation-defined.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table 28-4—Message digest algorithm identifiers

Identifier	Required /optional	Message digest algorithm
sha1	Required	Secure Hash Algorithm 1 (SHA-1), see FIPS 180-2.
md5	Required	Message Digest Algorithm 5, see IETF RFC 1321.
md2	Optional	Message Digest Algorithm 2, see IETF RFC 1319.
ripemd-160	Optional	RIPEMD-160, see ISO/IEC 10118-3:2004.

ENCRYPTION OUTPUT: The **digest_method** shall be unchanged in the output file, except where a digital signature is used, in which case it is encrypted with the **key_method** and placed in a **key_block**.

DECRYPTION INPUT: The **digest_method** indicates the algorithm that shall be used to generate the digest from the **data_block**.

28.4.22 digest_block

28.4.22.1 Syntax

digest_block

28.4.22.2 Description

ENCRYPTION INPUT: If a **digest_block** pragma expression is found in an input file (other than in a **begin_protected-end_protected** block), it shall be treated by the encrypting tool as a request to generate a message digest in the output file.

ENCRYPTION OUTPUT: A message digest is used to ensure that the encrypted data have not been modified. The encrypting tool generates the message digest (a fixed-length, computationally unique identifier corresponding to a set of data) using the algorithm specified by the **digest_method** pragma expression and encrypts the message digest as specified by the **digest_key_method** pragma keyword using the key specified by **digest_keyname, digest_key_keyowner**, **digest_public_key**, and **digest_decrypt_key**. If **digest_key_method** is not specified for the encryption envelope, then the current **data_method** encryption key shall be used.

This digest shall then be encoded using the current **encoding** pragma expression and output on the next line of the output file following the **digest_block** pragma expression. A **digest_block** shall be generated for each **key_block** and **data_block** that are generated in the encryption process and shall immediately follow the **key_block** or **data_block** to which it refers.

DECRYPTION INPUT: In order to authenticate the message, the consuming tool will decrypt the encrypted data, generate a message digest from the decrypted data, decrypt the message digest in the **digest_block** with the specified key, and compare the two message digests. If the two digests do not match, then either the **digest_block** or the encrypted data has been altered since the input data was encrypted. The message digest for a **key_block** or **data_block** shall be contained in a **digest_block** immediately following the **key_block** or **data_block**.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

28.4.23 key_keyowner

28.4.23.1 Syntax

key_keyowner = <string>

28.4.23.2 Description

ENCRYPTION INPUT: The **key_keyowner** specifies the legal entity or tool that provided the keys used for encryption and decryption of the key information. The value of the **key_keyowner** also has the same constraints specified for the **data_keyowner** values.

ENCRYPTION OUTPUT: The **key_keyowner** shall be unchanged in the output file.

DECRYPTION INPUT: During decryption, the **key_keyowner** can be combined with the **key_keyname** or **key_public_key** to determine the appropriate secret/private key to use during decryption of the **key_block**.

28.4.24 key_method

28.4.24.1 Syntax

key_method = <string>

28.4.24.2 Description

ENCRYPTION INPUT: The **key_method** pragma expression indicates the encryption algorithm that shall be used to encrypt the keys used to encrypt the **data_block**. The values specified for **key_method** to identify encryption algorithms are the same as those specified for **data_method**.

ENCRYPTION OUTPUT: The **key_method** shall be unchanged in the output file.

DECRYPTION INPUT: The **key_method** indicates the algorithm that shall be used to decrypt the **key_block**.

28.4.25 key_keyname

28.4.25.1 Syntax

key_keyname = <string>

28.4.25.2 Description

ENCRYPTION INPUT: The **key_keyname** pragma expression provides the name of the key, or key pair for an asymmetric encryption algorithm, that shall be used to decrypt the **key_block**. It shall be an error to specify a **key_keyname** that is not a member of the list of keys known for the given **key_keyowner.**

ENCRYPTION OUTPUT: When a **key_keyname** is provided in the input, it indicates the key that shall be used for encrypting the data encryption keys. The encrypting tool must be able to combine this pragma expression with the **key_keyowner** and determine the key to use. The **key_keyname** itself shall be output as cleartext in the output file.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

DECRYPTION INPUT: The **key_keyname** value is combined with the **key_keyowner** to select a single key that shall be used to decrypt the **data_block** from the protected envelope**.**

28.4.26 key_public_key

28.4.26.1 Syntax

key_public_key

28.4.26.2 Description

ENCRYPTION INPUT: The **key_public_key** pragma expression indicates that the next line of the file contains the encoded value of the public key to be used to encrypt the key data. The encoding is specified by the **encoding** pragma expression that is currently in effect. If both a **key_public_key** and **key_keyname** are present, then they must refer to the same key.

ENCRYPTION OUTPUT: The **key_public_key** pragma expression shall be output in each protected block for which it is used, followed by the encoded value. The **key_method** and **key_public_key** can be combined to fully specify the required encryption of data keys.

DECRYPTION INPUT: The **key_keyowner** and **key_method** can be combined with the **key_public_key** to determine whether the decryption tool knows the corresponding private key to decrypt a given **key_block**. If the decrypting tool can compute the required key, the data keys can be decrypted.

28.4.27 key_block

28.4.27.1 Syntax

key_block

28.4.27.2 Description

ENCRYPTION INPUT: It shall be an error if a **key_block** is found in an input file unless it is contained within a previously generated **begin_protected-end_protected** block, in which case it is ignored.

ENCRYPTION OUTPUT: The **key_block** pragma expression indicates that a key block begins on the next line in the file. When requested to use a digital signature, the encrypting tool shall take any of the **data_method**, **data_public_key**, **data_keyname**, **data_decrypt_key**, and **digest_block** to form a text buffer. This buffer shall then be encrypted with the appropriate **key_public_key**, and then the encrypted region shall be encoded using the **encoding** pragma expression in effect. The output of this encoding shall be generated as the contents of the **key_block**.

Where more than one **key_block** pragma expression occurs within a single **begin-end** block, the generated key blocks shall all encode the same data decryption key data. It shall be an error if the data decryption pragma expressions change value between **key_block** pragma expressions of a single encryption envelope. Multiple key blocks are specified for the purpose of providing alternative decryption keys for a single decryption envelope.

DECRYPTION INPUT: The **key_block** is first read in the encoded form, the encoding is reversed, and then the block is internally decrypted. The resulting text is then parsed to determine the keys required to decrypt the **data_block**.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

28.4.28 decrypt_license

28.4.28.1 Syntax

decrypt_license = (library = `<string>` **, entry =** `<string>` **, feature =** `<string>` **, [exit =** `<string>` **,] [match =** `<number>` **])**

28.4.28.2 Description

ENCRYPTION INPUT: The **decrypt_license** pragma expression will typically be found inside a **begin**/**end** pair in the original cleartext. This is necessary so that it is encrypted in the output IP shipped to the end user.

ENCRYPTION OUTPUT: The **decrypt_license** is output unchanged in the output description except for encryption and encoding of the pragma exactly as other cleartext in the **begin/end** pair. Typically, it will be output in the **data_block**.

DECRYPTION INPUT: After encountering a **decrypt_license** pragma expression in an encrypted model, prior to processing the decrypted text, the application shall load the specified library and call the **entry** function, passing it the **feature** specified string. The return value of the **entry** function shall be compared to the **match** value. If the application is licensed to decrypt the model, the returned value shall compare equal to the **match** value and shall compare nonequal otherwise. If the application is not licensed to decrypt the model, no decryption shall be performed, and the application shall produce an error message that includes the return value of the **entry** function. If an **exit** function is specified, then it shall be called prior to exiting the decrypting application to allow for releasing the license.

NOTE—This mechanism only provides limited security because the end users of the model have the shared library and could use readily available debuggers to debug the calling sequence of the licensing mechanism. They could then produce an equivalent library that returns a 0, but avoids the license check.

28.4.29 runtime_license

28.4.29.1 Syntax

runtime_license = (library = `<string>` **, entry =** `<string>` **, feature =** `<string>` **[, exit =** `<string>` **] [, match =** `<number>` **])**

28.4.29.2 Description

ENCRYPTION INPUT: The **runtime_license** pragma expression will typically be found inside a **begin**/**end** pair in the original cleartext. This is necessary so that it is encrypted in the output IP shipped to the end user.

ENCRYPTION OUTPUT: The **runtime_license** is output unchanged in the output description except for encryption and encoding of the pragma exactly as other cleartext in the **begin**/**end** pair.

DECRYPTION INPUT: After encountering a **runtime_license** pragma expression in an encrypted model, prior to executing, the application shall load the specified library and call the entry function, passing it the **feature** specified string. The return value of the entry function shall be compared to the match value. If the application is licensed to execute the model, the returned value shall compare equal to the match value and shall compare nonequal otherwise. If the application is not licensed to execute the model, execution shall not begin, and the application shall produce an error message that includes the return value of the entry function. If an **exit** is specified, then it shall be called prior to exiting the executing application to allow for releasing the license.

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

NOTE 1—Execution could mean any evaluation of the model, including simulation, layout, or synthesis.

NOTE 2—This mechanism only provides limited security because the end users of the model have the shared library and could use readily available debuggers to debug the calling sequence of the licensing mechanism. They could then produce an equivalent library that returns a 0, but avoids the license check. IP authors may wish to implement their own licensing scheme embedded within the behavior of the model, possibly using PLI and/or system tasks.

28.4.30 comment

28.4.30.1 Syntax

comment = `<string>`

28.4.30.2 Description

ENCRYPTION INPUT: The **comment** pragma expression can be found anywhere in an input file and indicates that even if this is found inside a **begin-end** block, the value shall be output as a comment in cleartext in the output immediately prior to the **data_block**.

This is provided so that comments that may end up being included in other files inside a **begin-end** block can protect themselves from being encrypted. This is important so that critical information such as copyright notices can be explicitly excluded from encryption.

Because this constitutes known cleartext that would be found inside the **data_block**, the pragma itself and the value should not be included in the encrypted text.

ENCRYPTION OUTPUT: The entire comment including the beginning pragma shall be output in cleartext immediately prior to the **data_block** corresponding to the **begin-end** in which the comment was found.

DECRYPTION INPUT: none

28.4.31 reset

28.4.31.1 Syntax

reset

28.4.31.2 Description

ENCRYPTION INPUT: The **reset** pragma expression is a synonym for a reset pragma directive that contains **protect** in the pragma keyword list. Following the reset, all **protect** pragma keywords are restored to their default values.

Because the scope of pragma definitions is lexical and extends from the point of the directive until the end of the compilation input, if an IP author chooses to put common pragmas such as **author** and **author_info** at the beginning of a list of files, they should include a **reset** pragma at the end of the list of files to ensure that this information is not unintentionally visible in other files.

ENCRYPTION OUTPUT: none

DECRYPTION INPUT: none

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

28.4.32 viewport

28.4.32.1 Syntax

viewport = (object = `<string>` **, access =** `<string>` **)**

28.4.32.2 Description

The **viewport** pragma expression describes objects within the current protected envelope for which access shall be permitted by the Verilog tool. The specified object name shall be contained within the current envelope. The access value is an implementation-specified relaxation of protection.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

Annex A

(normative)

Formal syntax definition

The formal syntax of Verilog HDL is described using Backus-Naur Form (BNF). The syntax of Verilog HDL source is derived from the starting symbol source_text. The syntax of a library map file is derived from the starting symbol library_text.

A.1 Source text

A.1.1 Library source text

```
library_text ::= { library_description }
library_description ::=
        library_declaration
      | include_statement
      | config_declaration
library_declaration ::=
        library library_identifier file_path_spec [ { , file_path_spec } ]
        [ -incdir file_path_spec { , file_path_spec } ] ;
include_statement ::= include file_path_spec ;
```

A.1.2 Verilog source text

```
source_text ::= { description }
description ::=
        module_declaration
      | udp_declaration
      | config_declaration
module_declaration ::=
        { attribute_instance } module_keyword module_identifier [ module_parameter_port_list ]
              list_of_ports ; { module_item }
              endmodule
      | { attribute_instance } module_keyword module_identifier [ module_parameter_port_list ]
              [ list_of_port_declarations ] ; { non_port_module_item }
              endmodule
module_keyword ::= module | macromodule
```

A.1.3 Module parameters and ports

```
module_parameter_port_list ::= # ( parameter_declaration { , parameter_declaration } )
list_of_ports ::= ( port { , port } )
list_of_port_declarations ::=
        ( port_declaration { , port_declaration } )
      | ( )
port ::=
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
        [ port_expression ]
      | . port_identifier ( [ port_expression ] )
port_expression ::=
        port_reference
      | { port_reference { , port_reference } }
port_reference ::=
      port_identifier [ [ constant_range_expression ] ]
port_declaration ::=
        {attribute_instance} inout_declaration
      | {attribute_instance} input_declaration
      | {attribute_instance} output_declaration
```

A.1.4 Module items

```
module_item ::=
        port_declaration ;
      | non_port_module_item
module_or_generate_item ::=
        { attribute_instance } module_or_generate_item_declaration
      | { attribute_instance } local_parameter_declaration ;
      | { attribute_instance } parameter_override
      | { attribute_instance } continuous_assign
      | { attribute_instance } gate_instantiation
      | { attribute_instance } udp_instantiation
      | { attribute_instance } module_instantiation
      | { attribute_instance } initial_construct
      | { attribute_instance } always_construct
      | { attribute_instance } loop_generate_construct
      | { attribute_instance } conditional_generate_construct
module_or_generate_item_declaration ::=
        net_declaration
      | reg_declaration
      | integer_declaration
      | real_declaration
      | time_declaration
      | realtime_declaration
      | event_declaration
      | genvar_declaration
      | task_declaration
      | function_declaration
non_port_module_item ::=
        module_or_generate_item
      | generate_region
      | specify_block
      | { attribute_instance } parameter_declaration ;
      | { attribute_instance } specparam_declaration
parameter_override ::= defparam list_of_defparam_assignments ;
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

A.1.5 Configuration source text

config_declaration ::=
config config_identifier **;**
design_statement
{config_rule_statement}
endconfig
design_statement ::= **design** { [library_identifier**.**]cell_identifier } **;**
config_rule_statement ::=
default_clause liblist_clause **;**
| inst_clause liblist_clause **;**
| inst_clause use_clause **;**
| cell_clause liblist_clause **;**
| cell_clause use_clause **;**
default_clause ::= **default**
inst_clause ::= **instance** inst_name
inst_name ::= topmodule_identifier{**.**instance_identifier}
cell_clause ::= **cell** [library_identifier**.**]cell_identifier
liblist_clause ::= **liblist** { library_identifier }
use_clause ::= **use** [library_identifier**.**]cell_identifier[**:config**]

A.2 Declarations

A.2.1 Declaration types

A.2.1.1 Module parameter declarations

local_parameter_declaration ::=
localparam [**signed**] [range] list_of_param_assignments
| **localparam** parameter_type list_of_param_assignments
parameter_declaration ::=
parameter [**signed**] [range] list_of_param_assignments
| **parameter** parameter_type list_of_param_assignments
specparam_declaration ::= **specparam** [range] list_of_specparam_assignments **;**
parameter_type ::=
integer | **real** | **realtime** | **time**

A.2.1.2 Port declarations

inout_declaration ::= **inout** [net_type] [**signed**] [range]
list_of_port_identifiers
input_declaration ::= **input** [net_type] [**signed**] [range]
list_of_port_identifiers
output_declaration ::=
output [net_type] [**signed**] [range]
list_of_port_identifiers
| **output reg** [**signed**] [range]
list_of_variable_port_identifiers
| **output** output_variable_type
list_of_variable_port_identifiers

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

A.2.1.3 Type declarations

event_declaration ::= **event** list_of_event_identifiers **;**
integer_declaration ::= **integer** list_of_variable_identifiers **;**
net_declaration ::=
net_type [**signed**]
[delay3] list_of_net_identifiers **;**
| net_type [drive_strength] [**signed**]
[delay3] list_of_net_decl_assignments **;**
| net_type [**vectored** | **scalared**] [**signed**]
range [delay3] list_of_net_identifiers **;**
| net_type [drive_strength] [**vectored** | **scalared**] [**signed**]
range [delay3] list_of_net_decl_assignments **;**
| **trireg** [charge_strength] [**signed**]
[delay3] list_of_net_identifiers **;**
| **trireg** [drive_strength] [**signed**]
[delay3] list_of_net_decl_assignments **;**
| **trireg** [charge_strength] [**vectored** | **scalared**] [**signed**]
range [delay3] list_of_net_identifiers **;**
| **trireg** [drive_strength] [**vectored** | **scalared**] [**signed**]
range [delay3] list_of_net_decl_assignments **;**
real_declaration ::= **real** list_of_real_identifiers **;**
realtime_declaration ::= **realtime** list_of_real_identifiers **;**
reg_declaration ::= **reg** [**signed**] [range]
list_of_variable_identifiers **;**
time_declaration ::= **time** list_of_variable_identifiers **;**

A.2.2 Declaration data types

A.2.2.1 Net and variable types

net_type ::=
supply0 | **supply1**
| **tri** | **triand** | **trior** | **tri0** | **tri1**
| **uwire** | **wire** | **wand** | **wor**
output_variable_type ::= **integer** | **time**
real_type ::=
real_identifier { dimension }
| real_identifier **=** constant_expression
variable_type ::=
variable_identifier { dimension }
| variable_identifier **=** constant_expression

A.2.2.2 Strengths

drive_strength ::=
(strength0 **,** strength1 **)**
| **(** strength1 **,** strength0 **)**
| **(** strength0 **, highz1)**
| **(** strength1 **, highz0)**

..`,,`.`,,,,`.`.`.`-`-`,,`,,`,`,,`.`---
Copyright © 2006 IEEE. All rights reserved.
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

| **(highz0 ,** strength1 **)**
| **(highz1 ,** strength0 **)**
strength0 ::= **supply0** | **strong0** | **pull0** | **weak0**
strength1 ::= **supply1** | **strong1** | **pull1** | **weak1**
charge_strength ::= **(small)** | **(medium)** | **(large)**

A.2.2.3 Delays

delay3 ::=
delay_value
| **#(** mintypmax_expression [**,** mintypmax_expression [**,** mintypmax_expression]] **)**
delay2 ::=
delay_value
| **#(** mintypmax_expression [**,** mintypmax_expression] **)**
delay_value ::=
unsigned_number
| real_number
| identifier

A.2.3 Declaration lists

list_of_defparam_assignments ::= defparam_assignment { **,** defparam_assignment }
list_of_event_identifiers ::= event_identifier { dimension }
{ **,** event_identifier { dimension } }
list_of_net_decl_assignments ::= net_decl_assignment { **,** net_decl_assignment }
list_of_net_identifiers ::= net_identifier { dimension }
{ **,** net_identifier { dimension } }
list_of_param_assignments ::= param_assignment { **,** param_assignment }
list_of_port_identifiers ::= port_identifier { **,** port_identifier }
list_of_real_identifiers ::= real_type { **,** real_type }
list_of_specparam_assignments ::= specparam_assignment { **,** specparam_assignment }
list_of_variable_identifiers ::= variable_type { **,** variable_type }
list_of_variable_port_identifiers ::= port_identifier [**=** constant_expression]
{ **,** port_identifier [**=** constant_expression] }

A.2.4 Declaration assignments

defparam_assignment ::= hierarchical_parameter_identifier **=** constant_mintypmax_expression
net_decl_assignment ::= net_identifier **=** expression
param_assignment ::= parameter_identifier **=** constant_mintypmax_expression
specparam_assignment ::=
specparam_identifier **=** constant_mintypmax_expression
| pulse_control_specparam
pulse_control_specparam ::=
PATHPULSE$ = (reject_limit_value [**,** error_limit_value] **)**
| **PATHPULSE$**specify_input_terminal_descriptor**$**specify_output_terminal_descriptor
= (reject_limit_value [**,** error_limit_value] **)**
error_limit_value ::= limit_value
reject_limit_value ::= limit_value
limit_value ::= constant_mintypmax_expression

Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

A.2.5 Declaration ranges

dimension ::= **[** dimension_constant_expression **:** dimension_constant_expression **]**
range ::= **[** msb_constant_expression **:** lsb_constant_expression **]**

A.2.6 Function declarations

function_declaration ::=
function [**automatic**] [function_range_or_type] function_identifier **;**
function_item_declaration { function_item_declaration }
function_statement
endfunction
| **function** [**automatic**] [function_range_or_type] function_identifier **(** function_port_list **) ;**
{ block_item_declaration }
function_statement
endfunction
function_item_declaration ::=
block_item_declaration
| { attribute_instance } tf_input_declaration **;**
function_port_list ::= { attribute_instance } tf_input_declaration { **,** { attribute_instance } tf_input_declaration }
function_range_or_type ::=
[**signed**] [range]
| **integer**
| **real**
| **realtime**
| **time**

A.2.7 Task declarations

task_declaration ::=
task [**automatic**] task_identifier **;**
{ task_item_declaration }
statement_or_null
endtask
| **task** [**automatic**] task_identifier **(** [task_port_list] **) ;**
{ block_item_declaration }
statement_or_null
endtask
task_item_declaration ::=
block_item_declaration
| { attribute_instance } tf_input_declaration **;**
| { attribute_instance } tf_output_declaration **;**
| { attribute_instance } tf_inout_declaration **;**
task_port_list ::= task_port_item { **,** task_port_item }
task_port_item ::=
{ attribute_instance } tf_input_declaration
| { attribute_instance } tf_output_declaration
| { attribute_instance } tf_inout_declaration

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

tf_input_declaration ::=
input [**reg**] [**signed**] [range] list_of_port_identifiers
| **input** task_port_type list_of_port_identifiers
tf_output_declaration ::=
output [**reg**] [**signed**] [range] list_of_port_identifiers
| **output** task_port_type list_of_port_identifiers
tf_inout_declaration ::=
inout [**reg**] [**signed**] [range] list_of_port_identifiers
| **inout** task_port_type list_of_port_identifiers
task_port_type ::=
integer | **real** | **realtime** | **time**

A.2.8 Block item declarations

block_item_declaration ::=
{ attribute_instance } **reg** [**signed**] [range] list_of_block_variable_identifiers **;**
| { attribute_instance } **integer** list_of_block_variable_identifiers **;**
| { attribute_instance } **time** list_of_block_variable_identifiers **;**
| { attribute_instance } **real** list_of_block_real_identifiers **;**
| { attribute_instance } **realtime** list_of_block_real_identifiers **;**
| { attribute_instance } event_declaration
| { attribute_instance } local_parameter_declaration **;**
| { attribute_instance } parameter_declaration **;**
list_of_block_variable_identifiers ::= block_variable_type { **,** block_variable_type }
list_of_block_real_identifiers ::= block_real_type { **,** block_real_type }
block_variable_type ::= variable_identifier { dimension }
block_real_type ::= real_identifier { dimension }

A.3 Primitive instances

A.3.1 Primitive instantiation and instances

gate_instantiation ::=
cmos_switchtype [delay3]
cmos_switch_instance { **,** cmos_switch_instance } **;**
| enable_gatetype [drive_strength] [delay3]
enable_gate_instance { **,** enable_gate_instance } **;**
| mos_switchtype [delay3]
mos_switch_instance { **,** mos_switch_instance } **;**
| n_input_gatetype [drive_strength] [delay2]
n_input_gate_instance { **,** n_input_gate_instance } **;**
| n_output_gatetype [drive_strength] [delay2]
n_output_gate_instance { **,** n_output_gate_instance } **;**
| pass_en_switchtype [delay2]
pass_enable_switch_instance { **,** pass_enable_switch_instance } **;**
| pass_switchtype
pass_switch_instance { **,** pass_switch_instance } **;**
| **pulldown** [pulldown_strength]

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

pull_gate_instance { **,** pull_gate_instance } **;**
| **pullup** [pullup_strength]
pull_gate_instance { **,** pull_gate_instance } **;**
cmos_switch_instance ::= [name_of_gate_instance] **(** output_terminal **,** input_terminal **,**
ncontrol_terminal **,** pcontrol_terminal **)**
enable_gate_instance ::= [name_of_gate_instance] **(** output_terminal **,** input_terminal **,** enable_terminal **)**
mos_switch_instance ::= [name_of_gate_instance] **(** output_terminal **,** input_terminal **,** enable_terminal **)**
n_input_gate_instance ::= [name_of_gate_instance] **(** output_terminal **,** input_terminal { **,** input_terminal } **)**
n_output_gate_instance ::= [name_of_gate_instance] **(** output_terminal { **,** output_terminal } **,**
input_terminal **)**
pass_switch_instance ::= [name_of_gate_instance] **(** inout_terminal **,** inout_terminal **)**
pass_enable_switch_instance ::= [name_of_gate_instance] **(** inout_terminal **,** inout_terminal **,**
enable_terminal **)**
pull_gate_instance ::= [name_of_gate_instance] **(** output_terminal **)**
name_of_gate_instance ::= gate_instance_identifier [range]

A.3.2 Primitive strengths

pulldown_strength ::=
(strength0 **,** strength1 **)**
| **(** strength1 **,** strength0 **)**
| **(** strength0 **)**
pullup_strength ::=
(strength0 **,** strength1 **)**
| **(** strength1 **,** strength0 **)**
| **(** strength1 **)**

A.3.3 Primitive terminals

enable_terminal ::= expression
inout_terminal ::= net_lvalue
input_terminal ::= expression
ncontrol_terminal ::= expression
output_terminal ::= net_lvalue
pcontrol_terminal ::= expression

A.3.4 Primitive gate and switch types

cmos_switchtype ::= **cmos** | **rcmos**
enable_gatetype ::= **bufif0** | **bufif1** | **notif0** | **notif1**
mos_switchtype ::= **nmos** | **pmos** | **rnmos** | **rpmos**
n_input_gatetype ::= **and** | **nand** | **or** | **nor** | **xor** | **xnor**
n_output_gatetype ::= **buf** | **not**
pass_en_switchtype ::= **tranif0** | **tranif1** | **rtranif1** | **rtranif0**
pass_switchtype ::= **tran** | **rtran**

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

A.4 Module instantiation and generate construct

A.4.1 Module instantiation

```
module_instantiation ::=
        module_identifier [ parameter_value_assignment ]
          module_instance { , module_instance } ;
parameter_value_assignment ::= # ( list_of_parameter_assignments )
list_of_parameter_assignments ::=
      ordered_parameter_assignment { , ordered_parameter_assignment } |
      named_parameter_assignment   { , named_parameter_assignment }
ordered_parameter_assignment ::= expression
named_parameter_assignment ::= . parameter_identifier ( [ mintypmax_expression ] )
module_instance ::= name_of_module_instance ( [ list_of_port_connections ] )
name_of_module_instance ::= module_instance_identifier [ range ]
list_of_port_connections ::=
        ordered_port_connection { , ordered_port_connection }
      | named_port_connection { , named_port_connection }
ordered_port_connection ::= { attribute_instance } [ expression ]
named_port_connection ::= { attribute_instance } . port_identifier ( [ expression ] )
```

A.4.2 Generate construct

```
generate_region ::=
      generate { module_or_generate_item } endgenerate
genvar_declaration ::=
      genvar list_of_genvar_identifiers ;
list_of_genvar_identifiers ::=
      genvar_identifier { , genvar_identifier }
loop_generate_construct ::=
      for ( genvar_initialization ; genvar_expression ; genvar_iteration )
        generate_block
genvar_initialization ::=
      genvar_identifier = constant_expression
genvar_expression ::=
        genvar_primary
      | unary_operator { attribute_instance } genvar_primary
      | genvar_expression binary_operator { attribute_instance } genvar_expression
      | genvar_expression ? { attribute_instance } genvar_expression : genvar_expression
genvar_iteration ::=
      genvar_identifier = genvar_expression
genvar_primary ::=
        constant_primary
      | genvar_identifier
conditional_generate_construct ::=
        if_generate_construct
      | case_generate_construct
if_generate_construct ::=
      if ( constant_expression ) generate_block_or_null
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
        [ else generate_block_or_null ]
case_generate_construct ::=
      case ( constant_expression )
        case_generate_item { case_generate_item } endcase
case_generate_item ::=
        constant_expression { , constant_expression } : generate_block_or_null
      | default [ : ] generate_block_or_null
generate_block ::=
        module_or_generate_item
      | begin [ : generate_block_identifier ] { module_or_generate_item } end
generate_block_or_null ::=
        generate_block
      | ;
```

A.5 UDP declaration and instantiation

A.5.1 UDP declaration

```
udp_declaration ::=
        { attribute_instance }  primitive udp_identifier ( udp_port_list ) ;
        udp_port_declaration { udp_port_declaration }
        udp_body
        endprimitive
      | { attribute_instance } primitive udp_identifier ( udp_declaration_port_list ) ;
        udp_body
        endprimitive
```

A.5.2 UDP ports

```
udp_port_list ::= output_port_identifier , input_port_identifier { , input_port_identifier }
udp_declaration_port_list ::=
      udp_output_declaration , udp_input_declaration { , udp_input_declaration }
udp_port_declaration ::=
        udp_output_declaration ;
      | udp_input_declaration ;
      | udp_reg_declaration ;
udp_output_declaration ::=
        { attribute_instance } output port_identifier
      | { attribute_instance } output reg port_identifier [ = constant_expression ]
udp_input_declaration ::= { attribute_instance } input list_of_port_identifiers
udp_reg_declaration ::= { attribute_instance } reg variable_identifier
```

A.5.3 UDP body

```
udp_body ::= combinational_body | sequential_body
combinational_body ::= table combinational_entry { combinational_entry } endtable
combinational_entry ::= level_input_list : output_symbol ;
sequential_body ::= [ udp_initial_statement ] table sequential_entry { sequential_entry } endtable
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

udp_initial_statement ::= **initial** output_port_identifier **=** init_val **;**
init_val ::= **1'b0** | **1'b1** | **1'bx** | **1'bX** | **1'B0** | **1'B1** | **1'Bx** | **1'BX** | **1** | **0**
sequential_entry ::= seq_input_list **:** current_state **:** next_state **;**
seq_input_list ::= level_input_list | edge_input_list
level_input_list ::= level_symbol { level_symbol }
edge_input_list ::= { level_symbol } edge_indicator { level_symbol }
edge_indicator ::= **(** level_symbol level_symbol **)** | edge_symbol
current_state ::= level_symbol
next_state ::= output_symbol | **-**
output_symbol ::= **0** | **1** | **x** | **X**
level_symbol ::= **0** | **1** | **x** | **X** | **?** | **b** | **B**
edge_symbol ::= **r** | **R** | **f** | **F** | **p** | **P** | **n** | **N** | *****

A.5.4 UDP instantiation

udp_instantiation ::= udp_identifier [drive_strength] [delay2]
udp_instance { **,** udp_instance } **;**
udp_instance ::= [name_of_udp_instance] **(** output_terminal **,** input_terminal
{ **,** input_terminal } **)**
name_of_udp_instance ::= udp_instance_identifier [range]

A.6 Behavioral statements

A.6.1 Continuous assignment statements

continuous_assign ::= **assign** [drive_strength] [delay3] list_of_net_assignments **;**
list_of_net_assignments ::= net_assignment { **,** net_assignment }
net_assignment ::= net_lvalue **=** expression

A.6.2 Procedural blocks and assignments

initial_construct ::= **initial** statement
always_construct ::= **always** statement
blocking_assignment ::= variable_lvalue **=** [delay_or_event_control] expression
nonblocking_assignment ::= variable_lvalue **<=** [delay_or_event_control] expression
procedural_continuous_assignments ::=
assign variable_assignment
| **deassign** variable_lvalue
| **force** variable_assignment
| **force** net_assignment
| **release** variable_lvalue
| **release** net_lvalue
variable_assignment ::= variable_lvalue **=** expression

A.6.3 Parallel and sequential blocks

par_block ::= **fork** [**:** block_identifier
{ block_item_declaration }] { statement } **join**

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

seq_block ::= **begin** [**:** block_identifier
{ block_item_declaration }] { statement } **end**

A.6.4 Statements

statement ::=
{ attribute_instance } blocking_assignment **;**
| { attribute_instance } case_statement
| { attribute_instance } conditional_statement
| { attribute_instance } disable_statement
| { attribute_instance } event_trigger
| { attribute_instance } loop_statement
| { attribute_instance } nonblocking_assignment **;**
| { attribute_instance } par_block
| { attribute_instance } procedural_continuous_assignments **;**
| { attribute_instance } procedural_timing_control_statement
| { attribute_instance } seq_block
| { attribute_instance } system_task_enable
| { attribute_instance } task_enable
| { attribute_instance } wait_statement
statement_or_null ::=
statement
| { attribute_instance } **;**
function_statement[1] ::= statement

A.6.5 Timing control statements

delay_control ::=
delay_value
| **#** **(** mintypmax_expression **)**
delay_or_event_control ::=
delay_control
| event_control
| **repeat** **(** expression **)** event_control
disable_statement ::=
disable hierarchical_task_identifier **;**
| **disable** hierarchical_block_identifier **;**
event_control ::=
@ hierarchical_event_identifier
| **@** **(** event_expression **)**
| **@***
| **@** **(*)**
event_trigger ::=
-> hierarchical_event_identifier { **[** expression **]** } **;**
event_expression ::=
expression
| **posedge** expression
| **negedge** expression
| event_expression **or** event_expression

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
      | event_expression , event_expression
procedural_timing_control ::=
        delay_control
      | event_control
procedural_timing_control_statement ::=
        procedural_timing_control statement_or_null
wait_statement ::=
        wait ( expression ) statement_or_null
```

A.6.6 Conditional statements

```
conditional_statement ::=
        if ( expression )
              statement_or_null [ else statement_or_null ]
      | if_else_if_statement
if_else_if_statement ::=
        if ( expression ) statement_or_null
        { else if ( expression ) statement_or_null }
        [ else statement_or_null ]
```

A.6.7 Case statements

```
case_statement ::=
        case ( expression )
              case_item { case_item } endcase
      | casez ( expression )
              case_item { case_item } endcase
      | casex ( expression )
              case_item { case_item } endcase
case_item ::=
        expression { , expression } : statement_or_null
      | default [ : ] statement_or_null
```

A.6.8 Looping statements

```
loop_statement ::=
        forever statement
      | repeat ( expression ) statement
      | while ( expression ) statement
      | for ( variable_assignment ;  expression ; variable_assignment )
              statement
```

A.6.9 Task enable statements

```
system_task_enable ::= system_task_identifier [ ( [ expression ] { , [ expression ] } ) ] ;
task_enable ::= hierarchical_task_identifier [ ( expression { , expression } ) ] ;
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

A.7 Specify section

A.7.1 Specify block declaration

specify_block ::= **specify** { specify_item } **endspecify**
specify_item ::=
  specparam_declaration
| pulsestyle_declaration
| showcancelled_declaration
| path_declaration
| system_timing_check
pulsestyle_declaration ::=
  pulsestyle_onevent list_of_path_outputs **;**
| **pulsestyle_ondetect** list_of_path_outputs **;**
showcancelled_declaration ::=
  showcancelled list_of_path_outputs **;**
| **noshowcancelled** list_of_path_outputs **;**

A.7.2 Specify path declarations

path_declaration ::=
  simple_path_declaration **;**
| edge_sensitive_path_declaration **;**
| state_dependent_path_declaration **;**
simple_path_declaration ::=
  parallel_path_description **=** path_delay_value
| full_path_description **=** path_delay_value
parallel_path_description ::=
  (specify_input_terminal_descriptor [polarity_operator] **=>** specify_output_terminal_descriptor **)**
full_path_description ::=
  (list_of_path_inputs [polarity_operator] ***>** list_of_path_outputs **)**
list_of_path_inputs ::=
  specify_input_terminal_descriptor { **,** specify_input_terminal_descriptor }
list_of_path_outputs ::=
  specify_output_terminal_descriptor { **,** specify_output_terminal_descriptor }

A.7.3 Specify block terminals

specify_input_terminal_descriptor ::=
  input_identifier [**[** constant_range_expression **]**]
specify_output_terminal_descriptor ::=
  output_identifier [**[** constant_range_expression **]**]
input_identifier ::= input_port_identifier | inout_port_identifier
output_identifier ::= output_port_identifier | inout_port_identifier

A.7.4 Specify path delays

path_delay_value ::=
  list_of_path_delay_expressions

 Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
      | ( list_of_path_delay_expressions )
list_of_path_delay_expressions ::=
        t_path_delay_expression
      | trise_path_delay_expression , tfall_path_delay_expression
      | trise_path_delay_expression , tfall_path_delay_expression , tz_path_delay_expression
      | t01_path_delay_expression , t10_path_delay_expression , t0z_path_delay_expression ,
        tz1_path_delay_expression , t1z_path_delay_expression , tz0_path_delay_expression
      | t01_path_delay_expression , t10_path_delay_expression , t0z_path_delay_expression ,
        tz1_path_delay_expression , t1z_path_delay_expression , tz0_path_delay_expression ,
        t0x_path_delay_expression , tx1_path_delay_expression , t1x_path_delay_expression ,
        tx0_path_delay_expression , txz_path_delay_expression , tzx_path_delay_expression
t_path_delay_expression ::= path_delay_expression
trise_path_delay_expression ::= path_delay_expression
tfall_path_delay_expression ::= path_delay_expression
tz_path_delay_expression ::= path_delay_expression
t01_path_delay_expression ::= path_delay_expression
t10_path_delay_expression ::= path_delay_expression
t0z_path_delay_expression ::= path_delay_expression
tz1_path_delay_expression ::= path_delay_expression
t1z_path_delay_expression ::= path_delay_expression
tz0_path_delay_expression ::= path_delay_expression
t0x_path_delay_expression ::= path_delay_expression
tx1_path_delay_expression ::= path_delay_expression
t1x_path_delay_expression ::= path_delay_expression
tx0_path_delay_expression ::= path_delay_expression
txz_path_delay_expression ::= path_delay_expression
tzx_path_delay_expression ::= path_delay_expression
path_delay_expression ::= constant_mintypmax_expression
edge_sensitive_path_declaration ::=
        parallel_edge_sensitive_path_description = path_delay_value
      | full_edge_sensitive_path_description = path_delay_value
parallel_edge_sensitive_path_description ::=
        ( [ edge_identifier ] specify_input_terminal_descriptor =>
            ( specify_output_terminal_descriptor [ polarity_operator ] : data_source_expression ) )
full_edge_sensitive_path_description ::=
        ( [ edge_identifier ] list_of_path_inputs *>
            ( list_of_path_outputs [ polarity_operator ] : data_source_expression ) )
data_source_expression ::= expression
edge_identifier ::= posedge | negedge
state_dependent_path_declaration ::=
        if ( module_path_expression ) simple_path_declaration
      | if ( module_path_expression ) edge_sensitive_path_declaration
      | ifnone simple_path_declaration
polarity_operator ::= + | -
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

A.7.5 System timing checks

A.7.5.1 System timing check commands

```
system_timing_check ::=
        $setup_timing_check
      | $hold_timing_check
      | $setuphold_timing_check
      | $recovery_timing_check
      | $removal_timing_check
      | $recrem_timing_check
      | $skew_timing_check
      | $timeskew_timing_check
      | $fullskew_timing_check
      | $period_timing_check
      | $width_timing_check
      | $nochange_timing_check
$setup_timing_check ::=
        $setup ( data_event , reference_event , timing_check_limit [ , [ notifier ] ] ) ;
$hold_timing_check ::=
        $hold ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
$setuphold_timing_check ::=
        $setuphold ( reference_event , data_event , timing_check_limit ,  timing_check_limit
                        [ , [ notifier ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
                        [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ] ) ;
$recovery_timing_check ::=
        $recovery ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
$removal_timing_check ::=
        $removal ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
$recrem_timing_check ::=
        $recrem ( reference_event , data_event , timing_check_limit , timing_check_limit
                        [ , [ notifier ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
                        [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ] ) ;
$skew_timing_check ::=
        $skew ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
$timeskew_timing_check ::=
        $timeskew ( reference_event , data_event , timing_check_limit
                        [ , [ notifier ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] ) ;
$fullskew_timing_check ::=
        $fullskew ( reference_event , data_event , timing_check_limit , timing_check_limit
                        [ , [ notifier ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] ) ;
$period_timing_check ::=
        $period ( controlled_reference_event , timing_check_limit [ , [ notifier ] ] ) ;
$width_timing_check ::=
        $width ( controlled_reference_event , timing_check_limit
                        [ , threshold [ , notifier ] ] ) ;
$nochange_timing_check ::=
        $nochange ( reference_event , data_event , start_edge_offset ,
                        end_edge_offset [ , [ notifier ] ] ) ;
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

A.7.5.2 System timing check command arguments

checktime_condition ::= mintypmax_expression
controlled_reference_event ::= controlled_timing_check_event
data_event ::= timing_check_event
delayed_data ::=
    terminal_identifier
  | terminal_identifier **[** constant_mintypmax_expression **]**
delayed_reference ::=
    terminal_identifier
  | terminal_identifier **[** constant_mintypmax_expression **]**
end_edge_offset ::= mintypmax_expression
event_based_flag ::= constant_expression
notifier ::= variable_identifier
reference_event ::= timing_check_event
remain_active_flag ::= constant_expression
stamptime_condition ::= mintypmax_expression
start_edge_offset ::= mintypmax_expression
threshold ::= constant_expression
timing_check_limit ::= expression

A.7.5.3 System timing check event definitions

timing_check_event ::=
    [timing_check_event_control] specify_terminal_descriptor [**&&&** timing_check_condition]
controlled_timing_check_event ::=
    timing_check_event_control specify_terminal_descriptor [**&&&** timing_check_condition]
timing_check_event_control ::=
    posedge
  | **negedge**
  | edge_control_specifier
specify_terminal_descriptor ::=
    specify_input_terminal_descriptor
  | specify_output_terminal_descriptor
edge_control_specifier ::= **edge [** edge_descriptor { **,** edge_descriptor } **]**
edge_descriptor[2] ::=
    01
  | **10**
  | z_or_x zero_or_one
  | zero_or_one z_or_x
zero_or_one ::= **0** | **1**
z_or_x ::= **x** | **X** | **z** | **Z**
timing_check_condition ::=
    scalar_timing_check_condition
  | **(** scalar_timing_check_condition **)**

Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

scalar_timing_check_condition ::=
  expression
| ~ expression
| expression **==** scalar_constant
| expression **===** scalar_constant
| expression **!=** scalar_constant
| expression **!==** scalar_constant
scalar_constant ::=
  1'b0 | **1'b1** | **1'B0** | **1'B1** | **'b0** | **'b1** | **'B0** | **'B1** | **1** | **0**

A.8 Expressions

A.8.1 Concatenations

concatenation ::= **{** expression { **,** expression } **}**
constant_concatenation ::= **{** constant_expression { **,** constant_expression } **}**
constant_multiple_concatenation ::= **{** constant_expression constant_concatenation **}**
module_path_concatenation ::= **{** module_path_expression { **,** module_path_expression } **}**
module_path_multiple_concatenation ::= **{** constant_expression module_path_concatenation **}**
multiple_concatenation ::= **{** constant_expression concatenation **}**

A.8.2 Function calls

constant_function_call ::= function_identifier { attribute_instance }
  (constant_expression { **,** constant_expression } **)**
constant_system_function_call ::= system_function_identifier
  (constant_expression { **,** constant_expression } **)**
function_call ::= hierarchical_function_identifier{ attribute_instance }
  (expression { **,** expression } **)**
system_function_call ::= system_function_identifier
  [**(** expression { **,** expression } **)**]

A.8.3 Expressions

base_expression ::= expression
conditional_expression ::= expression1 **?** { attribute_instance } expression2 **:** expression3
constant_base_expression ::= constant_expression
constant_expression ::=
  constant_primary
| unary_operator { attribute_instance } constant_primary
| constant_expression binary_operator { attribute_instance } constant_expression
| constant_expression **?** { attribute_instance } constant_expression **:** constant_expression
constant_mintypmax_expression ::=
  constant_expression
| constant_expression **:** constant_expression **:** constant_expression
constant_range_expression ::=
  constant_expression
| msb_constant_expression **:** lsb_constant_expression

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
      | constant_base_expression +: width_constant_expression
      | constant_base_expression -: width_constant_expression
dimension_constant_expression ::= constant_expression
expression ::=
        primary
      | unary_operator { attribute_instance } primary
      | expression binary_operator { attribute_instance } expression
      | conditional_expression
expression1 ::= expression
expression2 ::= expression
expression3 ::= expression
lsb_constant_expression ::= constant_expression
mintypmax_expression ::=
        expression
      | expression : expression : expression
module_path_conditional_expression ::= module_path_expression ? { attribute_instance }
            module_path_expression : module_path_expression
module_path_expression ::=
        module_path_primary
      | unary_module_path_operator { attribute_instance } module_path_primary
      | module_path_expression binary_module_path_operator { attribute_instance }
            module_path_expression
      | module_path_conditional_expression
module_path_mintypmax_expression ::=
        module_path_expression
      | module_path_expression : module_path_expression : module_path_expression
msb_constant_expression ::= constant_expression
range_expression ::=
        expression
      | msb_constant_expression : lsb_constant_expression
      | base_expression +: width_constant_expression
      | base_expression -: width_constant_expression
width_constant_expression ::= constant_expression
```

A.8.4 Primaries

```
constant_primary ::=
        number
      | parameter_identifier [ [ constant_range_expression ] ]
      | specparam_identifier [ [ constant_range_expression ] ]
      | constant_concatenation
      | constant_multiple_concatenation
      | constant_function_call
      | constant_system_function_call
      | ( constant_mintypmax_expression )
      | string
```

Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

module_path_primary ::=
    number
  | identifier
  | module_path_concatenation
  | module_path_multiple_concatenation
  | function_call
  | system_function_call
  | **(** module_path_mintypmax_expression **)**
primary ::=
    number
  | hierarchical_identifier [{ **[** expression **]** } **[** range_expression **]**]
  | concatenation
  | multiple_concatenation
  | function_call
  | system_function_call
  | **(** mintypmax_expression **)**
  | string

A.8.5 Expression left-side values

net_lvalue ::=
    hierarchical_net_identifier [{ **[** constant_expression **]** } **[** constant_range_expression **]**]
  | **{** net_lvalue { **,** net_lvalue } **}**
variable_lvalue ::=
    hierarchical_variable_identifier [{ **[** expression **]** } **[** range_expression **]**]
  | **{** variable_lvalue { **,** variable_lvalue } **}**

A.8.6 Operators

unary_operator ::=
    + | **-** | **!** | **~** | **&** | **~&** | **|** | **~|** | **^** | **~^** | **^~**
binary_operator ::=
    + | **-** | ***** | **/** | **%** | **==** | **!=** | **===** | **!==** | **&&** | **||** | ******
  | **<** | **<=** | **>** | **>=** | **&** | **|** | **^** | **^~** | **~^** | **>>** | **<<** | **>>>** | **<<<**
unary_module_path_operator ::=
    ! | **~** | **&** | **~&** | **|** | **~|** | **^** | **~^** | **^~**
binary_module_path_operator ::=
    == | **!=** | **&&** | **||** | **&** | **|** | **^** | **^~** | **~^**

A.8.7 Numbers

number ::=
    decimal_number
  | octal_number
  | binary_number
  | hex_number
  | real_number
real_number[2] ::=
    unsigned_number **.** unsigned_number

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

| unsigned_number [**.** unsigned_number] exp [sign] unsigned_number
exp ::= **e** | **E**
decimal_number ::=
unsigned_number
| [size] decimal_base unsigned_number
| [size] decimal_base x_digit { _ }
| [size] decimal_base z_digit { _ }
binary_number ::= [size] binary_base binary_value
octal_number ::= [size] octal_base octal_value
hex_number ::= [size] hex_base hex_value
sign ::= **+** | **-**
size ::= non_zero_unsigned_number
non_zero_unsigned_number[2] ::= non_zero_decimal_digit { _ | decimal_digit}
unsigned_number[2] ::= decimal_digit { _ | decimal_digit }
binary_value[2] ::= binary_digit { _ | binary_digit }
octal_value[2] ::= octal_digit { _ | octal_digit }
hex_value[2] ::= hex_digit { _ | hex_digit }
decimal_base[2] ::= **'[s|S]d** | **'[s|S]D**
binary_base[2] ::= **'[s|S]b** | **'[s|S]B**
octal_base[2] ::= **'[s|S]o** | **'[s|S]O**
hex_base[2] ::= **'[s|S]h** | **'[s|S]H**
non_zero_decimal_digit ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
decimal_digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
binary_digit ::= x_digit | z_digit | **0** | **1**
octal_digit ::= x_digit | z_digit | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**
hex_digit ::=
x_digit | z_digit | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
| **a** | **b** | **c** | **d** | **e** | **f** | **A** | **B** | **C** | **D** | **E** | **F**
x_digit ::= **x** | **X**
z_digit ::= **z** | **Z** | **?**

A.8.8 Strings

string ::= **"** { Any_ASCII_Characters_except_new_line } **"**

A.9 General

A.9.1 Attributes

attribute_instance ::= **(*** attr_spec { **,** attr_spec } ***)**
attr_spec ::=
attr_name [= constant_expression]
attr_name ::= identifier

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

A.9.2 Comments

comment ::=
one_line_comment
| block_comment
one_line_comment ::= // comment_text \n
block_comment ::= /* comment_text */
comment_text ::= { Any_ASCII_character }

A.9.3 Identifiers

block_identifier ::= identifier
cell_identifier ::= identifier
config_identifier ::= identifier
escaped_identifier ::= \ {Any_ASCII_character_except_white_space} white_space
event_identifier ::= identifier
function_identifier ::= identifier
gate_instance_identifier ::= identifier
generate_block_identifier ::= identifier
genvar_identifier ::= identifier
hierarchical_block_identifier ::= hierarchical_identifier
hierarchical_event_identifier ::= hierarchical_identifier
hierarchical_function_identifier ::= hierarchical_identifier
hierarchical_identifier ::= { identifier [**[** constant_expression **]**] **.** } identifier
hierarchical_net_identifier ::= hierarchical_identifier
hierarchical_parameter_identifier ::= hierarchical_identifier
hierarchical_variable_identifier ::= hierarchical_identifier
hierarchical_task_identifier ::= hierarchical_identifier
identifier ::=
simple_identifier
| escaped_identifier
inout_port_identifier ::= identifier
input_port_identifier ::= identifier
instance_identifier ::= identifier
library_identifier ::= identifier
module_identifier ::= identifier
module_instance_identifier ::= identifier
net_identifier ::= identifier
output_port_identifier ::= identifier
parameter_identifier ::= identifier
port_identifier ::= identifier
real_identifier ::= identifier
simple_identifier[3] ::= [**a-zA-Z_**] { [**a-zA-Z0-9_$**] }
specparam_identifier ::= identifier
system_function_identifier[4] ::= **$**[**a-zA-Z0-9_$**]{ [**a-zA-Z0-9_$**] }
system_task_identifier[4] ::= **$**[**a-zA-Z0-9_$**]{ [**a-zA-Z0-9_$**] }
task_identifier ::= identifier
terminal_identifier ::= identifier
text_macro_identifier ::= identifier

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

topmodule_identifier ::= identifier
udp_identifier ::= identifier
udp_instance_identifier ::= identifier
variable_identifier ::= identifier

A.9.4 White space

white_space ::= space | tab | newline | eof[5]

Details:

1) Function statements are limited by the rules of 10.4.4.
2) Embedded spaces are illegal.
3) A simple_identifier shall start with an alpha or underscore (_) character, shall have at least one character, and shall not have any spaces.
4) The dollar sign ($) in a system_function_identifier or system_task_identifier shall not be followed by white_space. A system_function_identifier or system_task_identifier shall not be escaped.
5) End of file.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Annex B

(normative)

List of keywords

Keywords are predefined nonescaped identifiers that define Verilog language constructs. An escaped identifier shall not be treated as a keyword.

always
and
assign
automatic
begin
buf
bufif0
bufif1
case
casex
casez
cell
cmos
config
deassign
default
defparam
design
disable
edge
else
end
endcase
endconfig
endfunction
endgenerate
endmodule
endprimitive
endspecify
endtable
endtask
event
for
force
forever
fork
function
generate
genvar
highz0
highz1
if
ifnone
incdir
include
initial
inout
input
instance
integer
join
large
liblist
library
localparam
macromodule
medium
module
nand
negedge
nmos
nor
noshowcancelled
not
notif0
notif1
or
output
parameter
pmos
posedge
primitive
pull0
pull1
pulldown
pullup
pulsestyle_onevent
pulsestyle_ondetect
rcmos
real
realtime
reg
release
repeat
rnmos
rpmos
rtran
rtranif0
rtranif1
scalared
showcancelled
signed
small
specify
specparam
strong0
strong1
supply0
supply1
table
task
time
tran
tranif0
tranif1
tri
tri0
tri1
triand
trior
trireg
unsigned[1]
use
uwire
vectored
wait
wand
weak0
weak1
while
wire
wor
xnor
xor

[1] **unsigned** is reserved for possible future usage.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Annex C

(informative)

System tasks and functions

The system tasks and functions described in this annex are for informative purposes only and are not part of this standard.

This annex describes system tasks and functions as companions to the system tasks and functions described in Clause 17 The system tasks and functions described in this annex may not be available in all implementations of the Verilog HDL. The following system tasks and functions are described in this annex:

$countdrivers	[C.1]	**$reset_count**	[C.7]
$getpattern	[C.2]	**$reset_value**	[C.7]
$incsave	[C.8]	**$restart**	[C.8]
$input	[C.3]	**$save**	[C.8]
$key	[C.4]	**$scale**	[C.9]
$list	[C.5]	**$scope**	[C.10]
$log	[C.6]	**$showscopes**	[C.11]
$nokey	[C.4]	**$showvars**	[C.12]
$nolog	[C.6]	**$sreadmemb**	[C.13]
$reset	[C.7]	**$sreadmemh**	[C.13]

The word *tool* in this annex refers to an implementation of Verilog HDL, typically a logic simulator.

C.1 $countdrivers

Syntax:

$countdrivers (net, [net_is_forced, number_of_01x_drivers, number_of_0_drivers, number_of_1_drivers, number_of_x_drivers])**;**

The **$countdrivers** system function is provided to count the number of drivers on a specified net so that bus contention can be identified.

This system function returns a `0` if there is no more than one driver on the net and returns a `1` otherwise (indicating contention). The specified net shall be a scalar or a bit-select of a vector net. The number of arguments to the system function may vary according to how much information is desired.

If additional arguments are supplied to the **$countdrivers** function, each argument returns the information described in Table C.1.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Table C.1—Argument return value for $countdriver function

Argument	Return value
net_is_forced	1 if net is forced. 0 otherwise.
number_of_01x_drivers	An integer representing the number of drivers on the net that are in 0, 1, or x state. This represents the total number of drivers that are not forced.
number_of_0_drivers	An integer representing the number of drivers on the net that are in `0` state.
number_of_1_drivers	An integer representing the number of drivers on the net that are in `1` state.
number_of_x_drivers	An integer representing the number of drivers on the net that are in `x` state.

C.2 $getpattern

Syntax:

$getpattern `( mem_element )`**;**

The system function **$getpattern** provides for fast processing of stimulus patterns that have to be propagated to a large number of scalar inputs. The function reads stimulus patterns that have been loaded into a memory using the **$readmemb** or **$readmemh** system tasks.

Use of this function is limited, however: it may only be used in a continuous assignment statement where the left-hand side is a concatenation of scalar nets and the argument to the system function is a memory element reference.

For example:

The following example shows how stimuli stored in a file can be read into a memory using **$readmemb** and applied to the circuit one pattern at a time using **$getpattern**.

The memory `in_mem` is initialized with the stimulus patterns by the **$readmemb** task. The integer variable `index` selects which pattern is being applied to the circuit. The `for` loop increments the integer variable `index` periodically to sequence the patterns.

```
module top;
parameter in_width=10,
          patterns=200,
          step=20;
reg [1:in_width] in_mem[1:patterns];
integer index;

// declare scalar inputs
wire i1,i2,i3,i4,i5,i6,i7,i8,i9,i10;

// assign patterns to circuit scalar inputs (a new pattern
// is applied to the circuit each time index changes value)
assign {i1,i2,i3,i4,i5,i6,i7,i8,i9,i10} = $getpattern(in_mem[index]);
initial begin
    // read stimulus patterns into memory
    $readmemb("patt.mem", in_mem);
```


Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
    // step through patterns (each assignment
    // to index will drive a new pattern onto the circuit
    // inputs from the $getpattern system task specified above
    for (index = 1; index <= patterns; index = index + 1)
      #step;
end

    // instantiate the circuit module - e.g.,
mod1 cct (o1,o2,o3,o4,o5, i1,i2,i3,i4,i5,i6,i7,i8,i9,i10);

endmodule
```

C.3 $input

Syntax:

$input ("*filename*");

The **$input** system task allows command input text to come from a named file instead of from the terminal. At the end of the command file, the input is switched back to the terminal.

C.4 $key and $nokey

Syntax:

$key [("*filename*")] ;
$nokey ;

A key file is created whenever interactive mode is entered for the first time during simulation. The key file contains all of the text that has been typed in from the standard input. The file also contains information about asynchronous interrupts.

The **$nokey** and **$key** system tasks are used to disable and reenable output to the key file. An optional file name argument for **$key** causes the old key file to be closed, a new file to be created, and output to be directed to the new file.

C.5 $list

Syntax:

$list [(hierarchical_name)] ;

When invoked without an argument, **$list** produces a listing of the module, task, function, or named block that is defined as the current scope setting. If an optional argument is supplied, it shall refer to a specific module, task, function, or named block, in which case the specified object is listed.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

C.6 $log and $nolog

Syntax:

$log [(**"***filename***"**)] **;**
$nolog ;

A log file contains a copy of all the text that is printed to the standard output. The log file may also contain, at the beginning of the file, the host command that was used to run the tool.

The **$nolog** and **$log** system tasks are used to disable and reenable output to the log file. The **$nolog** task disables output to the log file, while the **$log** task reenables the output. An optional file name argument for **$log** causes the old file to be closed, a new log file to be created, and output to be directed to the new log file.

C.7 $reset, $reset_count, and $reset_value

Syntax:

$reset [(stop_value [, reset_value , [diagnostics_value]])] **;**
$reset_count ;
$reset_value ;

The **$reset** system task enables a tool to be reset to its "time 0" state so that processing (e.g., simulation) can begin again.

The **$reset_count** system function keeps track of the number of times the tool is reset. The **$reset_value** system function returns the value specified by the `reset_value` argument to the **$reset** system task. The **$reset_value** system function is used to communicate information from before a reset of a tool to the time 0 state to after the reset.

The following are some of the simulation methods that can be employed with this system task and these system functions:

— Determine the **force** statements a design needs to operate correctly, reset the simulation time to 0, enter these **force** statements, and start to simulate again.
— Reset the simulation time to 0 and apply new stimuli.
— Determine that debug system tasks, such as **$monitor** and **$strobe**, are keeping track of the correct nets or regs, reset the simulation time to 0, and begin simulation again.

The **$reset** system task tells a tool to return the processing of the design to its logical state at time 0. When a tool executes the **$reset** system task, it takes the following actions to stop the process:

a) Disables all concurrent activity, initiated in either **initial** or **always** procedural blocks in the source description or through interactive mode (disables, for example, all **force** and **assign** statements, the current **$monitor** system task, and any other active tasks).
b) Cancels all scheduled simulation events.

After a simulation tool executes the **$reset** system task, the simulation is in the following state:

— The simulation time is `0`.
— All regs and nets contain their initial values.
— The tool begins to execute the first procedural statements in all **initial** and **always** blocks.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

The `stop_value` argument indicates if interactive mode or processing is entered immediately after resetting of the tool. A value of `0` or no argument causes interactive mode to be entered after resetting the tool. A nonzero value passed to **$reset** causes the tool to begin processing immediately.

The `reset_value` argument is an integer that specifies the value that shall be returned by the **$reset_value** system function after the tool is reset. All declared integers return to their initial value after reset, but entering an integer as this argument allows access to what its value was before the reset with the **$reset_value** system function. This argument provides a means of communicating information from before the reset of a tool to after the reset of the tool.

The `diagnostic_value` specifies the kind of diagnostic messages a tool displays before it resets the time to `0`. Increasing integer values results in increased information. A value of zero results in no diagnostic message.

C.8 $save, $restart, and $incsave

Three system tasks **$save**, **$restart**, and **$incsave** work in conjunction with one another to save the complete state of simulation into a permanent file so that the simulation state can be reloaded at a later time and processing can continue where it left off.

Syntax:

$save("*file_name*");
$restart("*file_name*");
$incsave("*incremental_file_name*");

All three system tasks take a file name as an argument. The file name has to be supplied as a string enclosed in quotation marks.

The **$save** system task saves the complete state into the file specified as an argument.

The **$incsave** system task saves only what has changed since the last invocation of **$save**. It is not possible to do an incremental save on any file other than the one produced by the last **$save**.

The **$restart** system task restores a previously saved state from a specified file.

Restarting from an incremental save is similar to restarting from a full save, except that the name of the incremental save file is specified in the restart command. The full save file on which the incremental save file was based shall still be present, as it is required for a successful restart. If the full save file has been changed in any way since the incremental save was performed, errors will result.

The incremental restart is useful for going back in time. If a full save is performed near the beginning of processing and an incremental save is done at regular intervals, then going back in time is as simple as restarting from the appropriate file.

For example:

```
module checkpoint;

initial
   #500 $save("save.dat"); // full save

always begin          // incremental save every 10000 units,
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
                    // files are recycled every 40000 units
    #100000 $incsave("inc1.dat");
    #100000 $incsave("inc2.dat");
    #100000 $incsave("inc3.dat");
    #100000 $incsave("inc4.dat");
end
endmodule
```

C.9 $scale

Syntax:

$scale (hierarchical_name) **;**

The **$scale** function takes a time value from a module with one time unit to be used in a module with a different time unit. The time value is converted from the time unit of one module to the time unit of the module that invokes **$scale**.

C.10 $scope

Syntax:

$scope (hierarchical_name) **;**

The **$scope** system task allows a particular level of hierarchy to be specified as the scope for identifying objects. This task accepts a single argument that shall be the complete hierarchical name of a module, task, function, or named block. The initial setting of the interactive scope is the first top-level module.

C.11 $showscopes

Syntax:

$showscopes [(n)]**;**

The **$showscopes** system task produces a complete list of modules, tasks, functions, and named blocks that are defined at the current scope level. An optional integer argument can be given to **$showscopes**. A nonzero argument value causes all the modules, tasks, functions, and named blocks in or below the current hierarchical scope to be listed. No argument or a zero value results in only objects at the current scope level being listed.

C.12 $showvars

Syntax:

$showvars [(list_of_variables)] **;**

The **$showvars** system task produces status information for reg and net variables, both scalar and vector. When invoked without arguments, **$showvars** displays the status of all variables in the current scope. When invoked with a list of variables, **$showvars** shows only the status of the specified variables. If the list of

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

variables includes a bit-select or part-select of a reg or net, then the status information for all the bits of that reg or net are displayed.

C.13 $sreadmemb and $sreadmemh

Syntax:

$sreadmemb (mem_name , start_address , finish_address , string { , string }) **;**
$sreadmemh (mem_name , start_address , finish_address , string { , string }) **;**

The system tasks **$sreadmemb** and **$sreadmemh** load data into memory `mem_name` from a character string.

The **$sreadmemh** and **$sreadmemb** system tasks take memory data values and addresses as string arguments. The start and finish addresses indicate the bounds for where the data from strings will be stored in the memory. These strings take the same format as the strings that appear in the input files passed as arguments to **$readmemb** and **$readmemh**.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Annex D

(informative)

Compiler directives

The compiler directives described in this annex are for informative purposes only and are not part of this standard.

This annex describes additional compiler directives as companions to the compiler directives described in Clause 19. The compiler directives described in this annex may not be available in all implementations of the Verilog HDL. The following compiler directives are described in this annex:

`default_decay_time[D.1]
`default_trireg_strength[D.2]
`delay_mode_distributed[D.3]
`delay_mode_path[D.4]
`delay_mode_unit[D.5]
`delay_mode_zero[D.6]

The word *tool* in this annex refers to an implementation of Verilog HDL, typically a logic simulator.

D.1 `default_decay_time

The **`default_decay_time** compiler directive specifies the decay time for the trireg nets that do not have any decay time specified in the declaration. This compiler directive applies to all of the trireg nets in all the modules that follow it in the source description. An argument specifying the charge decay time shall be used with this compiler directive.

Syntax:

`default_decay_time *integer*_constant | *real*_constant | **infinite**

For example:

Example 1—The following example shows how the default decay time for all trireg nets can be set to 100 time units:

```
`default_decay_time 100
```

Example 2—The following example shows how to avoid charge decay on trireg nets:

```
`default_decay_time infinite
```

The keyword **infinite** specifies no charge decay for all the trireg nets that do not have decay time specification.

D.2 `default_trireg_strength

The **`default_trireg_strength** compiler directive specifies the charge strength of **trireg** nets.

Syntax:

`default_trireg_strength *integer*_constant

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

The integer constant shall be between 0 and 250. It indicates the relative strength of the capacitance on the trireg net.

D.3 `delay_mode_distributed

The **`delay_mode_distributed** compiler directive specifies the distributed delay mode for all modules that follow this directive in the source description.

Syntax:

`delay_mode_distributed

This compiler directive shall be used before the declaration of the module whose delay mode is being controlled.

D.4 `delay_mode_path

The **`delay_mode_path** compiler directive specifies the path delay mode for all modules that follow this directive in the source description.

Syntax:

`delay_mode_path

This compiler directive shall be used before the declaration of the module whose delay mode is being controlled.

D.5 `delay_mode_unit

The **`delay_mode_unit** compiler directive specifies the unit delay mode for all modules that follow this directive in the source description.

Syntax:

`delay_mode_unit

This compiler directive shall be used before the declaration of the module whose delay mode is being controlled.

D.6 `delay_mode_zero

The **`delay_mode_zero** compiler directive specifies the zero delay mode for all modules that follow this directive in the source description.

Syntax:

`delay_mode_zero

This compiler directive shall be used before the declaration of the module whose delay mode is being controlled.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Annex E

(normative)

acc_user.h (deprecated)

This annex has been deprecated (see 1.6).

Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS

Not for Resale

Annex F

(normative)

veriuser.h (deprecated)

This annex has been deprecated (see 1.6).

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Annex G

(normative)

vpi_user.h

```
/*************************************************************************
 * vpi_user.h
 *
 * IEEE 1364-2005 Verilog HDL Programming Language Interface (PLI)
 *
 * This file contains the constant definitions, structure definitions, and
 * routine declarations used by the Verilog PLI procedural interface VPI
 * access routines.
 *
 *************************************************************************/

/*************************************************************************
 * NOTE: the constant values 1 through 299 are reserved for use in this
 * vpi_user.h file.
 *************************************************************************/

#ifndef VPI_USER_H
#define VPI_USER_H

#include <stdarg.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*-------------------------------------------------------------------------*/
/*----------------------------- Portability Help --------------------------*/
/*-------------------------------------------------------------------------*/

/* Sized variables */

#ifndef PLI_TYPES
#define PLI_TYPES
typedef int             PLI_INT32;
typedef unsigned int    PLI_UINT32;
typedef short           PLI_INT16;
typedef unsigned short  PLI_UINT16;
typedef char            PLI_BYTE8;
typedef unsigned char   PLI_UBYTE8;
#endif

/* Use to export a symbol */

#if WIN32
#ifndef PLI_DLLISPEC
#define PLI_DLLISPEC __declspec(dllimport)
#define VPI_USER_DEFINED_DLLISPEC 1
#endif
#else
#ifndef PLI_DLLISPEC
#define PLI_DLLISPEC
#endif
#endif
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
/* Use to import a symbol */

#if WIN32
#ifndef PLI_DLLESPEC
#define PLI_DLLESPEC __declspec(dllexport)
#define VPI_USER_DEFINED_DLLESPEC 1
#endif
#else
#ifndef PLI_DLLESPEC
#define PLI_DLLESPEC
#endif
#endif

/* Use to mark a function as external */

#ifndef PLI_EXTERN
#define PLI_EXTERN
#endif

/* Use to mark a variable as external */

#ifndef PLI_VEXTERN
#define PLI_VEXTERN extern
#endif

#ifndef PLI_PROTOTYPES
#define PLI_PROTOTYPES
#define PROTO_PARAMS(params) params

/* object is defined imported by the application */

#define XXTERN PLI_EXTERN PLI_DLLISPEC

/* object is exported by the application */

#define EETERN PLI_EXTERN PLI_DLLESPEC
#endif

/********************************* TYPEDEFS *********************************/

typedef PLI_UINT32 *vpiHandle;

/***************************** OBJECT TYPES *****************************/

#define vpiAlways              1   /* always construct */
#define vpiAssignStmt          2   /* quasi-continuous assignment */
#define vpiAssignment          3   /* procedural assignment */
#define vpiBegin               4   /* block statement */
#define vpiCase                5   /* case statement */
#define vpiCaseItem            6   /* case statement item */
#define vpiConstant            7   /* numerical constant or literal string */
#define vpiContAssign          8   /* continuous assignment */
#define vpiDeassign            9   /* deassignment statement */
#define vpiDefParam           10   /* defparam */
#define vpiDelayControl       11   /* delay statement (e.g., #10) */
#define vpiDisable            12   /* named block disable statement */
#define vpiEventControl       13   /* wait on event, e.g., @e */
#define vpiEventStmt          14   /* event trigger, e.g., ->e */
#define vpiFor                15   /* for statement */
#define vpiForce              16   /* force statement */
#define vpiForever            17   /* forever statement */
#define vpiFork               18   /* fork-join block */
#define vpiFuncCall           19   /* HDL function call */
#define vpiFunction           20   /* HDL function */
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
#define vpiGate               21   /* primitive gate */
#define vpiIf                 22   /* if statement */
#define vpiIfElse             23   /* if-else statement */
#define vpiInitial            24   /* initial construct */
#define vpiIntegerVar         25   /* integer variable */
#define vpiInterModPath       26   /* intermodule wire delay */
#define vpiIterator           27   /* iterator */
#define vpiIODecl             28   /* input/output declaration */
#define vpiMemory             29   /* behavioral memory */
#define vpiMemoryWord         30   /* single word of memory */
#define vpiModPath            31   /* module path for path delays */
#define vpiModule             32   /* module instance */
#define vpiNamedBegin         33   /* named block statement */
#define vpiNamedEvent         34   /* event variable */
#define vpiNamedFork          35   /* named fork-join block */
#define vpiNet                36   /* scalar or vector net */
#define vpiNetBit             37   /* bit of vector net */
#define vpiNullStmt           38   /* a semicolon. Ie. #10 ; */
#define vpiOperation          39   /* behavioral operation */
#define vpiParamAssign        40   /* module parameter assignment */
#define vpiParameter          41   /* module parameter */
#define vpiPartSelect         42   /* part-select */
#define vpiPathTerm           43   /* terminal of module path */
#define vpiPort               44   /* module port */
#define vpiPortBit            45   /* bit of vector module port */
#define vpiPrimTerm           46   /* primitive terminal */
#define vpiRealVar            47   /* real variable */
#define vpiReg                48   /* scalar or vector reg */
#define vpiRegBit             49   /* bit of vector reg */
#define vpiRelease            50   /* release statement */
#define vpiRepeat             51   /* repeat statement */
#define vpiRepeatControl      52   /* repeat control in an assign stmt */
#define vpiSchedEvent         53   /* vpi_put_value() event */
#define vpiSpecParam          54   /* specparam */
#define vpiSwitch             55   /* transistor switch */
#define vpiSysFuncCall        56   /* system function call */
#define vpiSysTaskCall        57   /* system task call */
#define vpiTableEntry         58   /* UDP state table entry */
#define vpiTask               59   /* HDL task */
#define vpiTaskCall           60   /* HDL task call */
#define vpiTchk               61   /* timing check */
#define vpiTchkTerm           62   /* terminal of timing check */
#define vpiTimeVar            63   /* time variable */
#define vpiTimeQueue          64   /* simulation event queue */
#define vpiUdp                65   /* user-defined primitive */
#define vpiUdpDefn            66   /* UDP definition */
#define vpiUserSystf          67   /* user-defined system task/function */
#define vpiVarSelect          68   /* variable array selection */
#define vpiWait               69   /* wait statement */
#define vpiWhile              70   /* while statement */

/********************** object types added with 1364-2001 *********************/

#define vpiAttribute         105   /* attribute of an object */
#define vpiBitSelect         106   /* Bit-select of parameter, var select */
#define vpiCallback          107   /* callback object */
#define vpiDelayTerm         108   /* Delay term which is a load or driver */
#define vpiDelayDevice       109   /* Delay object within a net */
#define vpiFrame             110   /* reentrant task/func frame */
#define vpiGateArray         111   /* gate instance array */
#define vpiModuleArray       112   /* module instance array */
#define vpiPrimitiveArray    113   /* vpiprimitiveArray type */
#define vpiNetArray          114   /* multidimensional net */
#define vpiRange             115   /* range declaration */
```


Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
#define vpiRegArray          116   /* multidimensional reg */
#define vpiSwitchArray       117   /* switch instance array */
#define vpiUdpArray          118   /* UDP instance array */
#define vpiContAssignBit     128   /* Bit of a vector continuous assignment */
#define vpiNamedEventArray   129   /* multidimensional named event */

/********************** object types added with 1364-2005 ********************/

#define vpiIndexedPartSelect 130   /* Indexed part-select object */
#define vpiGenScopeArray     133   /* array of generated scopes */
#define vpiGenScope          134   /* A generated scope */
#define vpiGenVar            135   /* Object used to instantiate gen scopes */

/********************************** METHODS **********************************/
/***************** methods used to traverse 1 to 1 relationships ***************/

#define vpiCondition          71   /* condition expression */
#define vpiDelay              72   /* net or gate delay */
#define vpiElseStmt           73   /* else statement */
#define vpiForIncStmt         74   /* increment statement in for loop */
#define vpiForInitStmt        75   /* initialization statement in for loop */
#define vpiHighConn           76   /* higher connection to port */
#define vpiLhs                77   /* left-hand side of assignment */
#define vpiIndex              78   /* index of var select, bit-select, etc. */
#define vpiLeftRange          79   /* left range of vector or part-select */
#define vpiLowConn            80   /* lower connection to port */
#define vpiParent             81   /* parent object */
#define vpiRhs                82   /* right-hand side of assignment */
#define vpiRightRange         83   /* right range of vector or part-select */
#define vpiScope              84   /* containing scope object */
#define vpiSysTfCall          85   /* task function call */
#define vpiTchkDataTerm       86   /* timing check data term */
#define vpiTchkNotifier       87   /* timing check notifier */
#define vpiTchkRefTerm        88   /* timing check reference term */

/************* methods used to traverse 1 to many relationships ***************/

#define vpiArgument           89   /* argument to (system) task/function */
#define vpiBit                90   /* bit of vector net or port */
#define vpiDriver             91   /* driver for a net */
#define vpiInternalScope      92   /* internal scope in module */
#define vpiLoad               93   /* load on net or reg */
#define vpiModDataPathIn      94   /* data terminal of a module path */
#define vpiModPathIn          95   /* Input terminal of a module path */
#define vpiModPathOut         96   /* output terminal of a module path */
#define vpiOperand            97   /* operand of expression */
#define vpiPortInst           98   /* connected port instance */
#define vpiProcess            99   /* process in module */
#define vpiVariables         100   /* variables in module */
#define vpiUse               101   /* usage */

/******** methods which can traverse 1 to 1, or 1 to many relationships *******/

#define vpiExpr              102   /* connected expression */
#define vpiPrimitive         103   /* primitive (gate, switch, UDP) */
#define vpiStmt              104   /* statement in process or task */

/************************ methods added with 1364-2001 ***********************/

#define vpiActiveTimeFormat  119   /* active $timeformat() system task */
#define vpiInTerm            120   /* To get to a delay device's drivers. */
#define vpiInstanceArray     121   /* vpiInstance arrays */
#define vpiLocalDriver       122   /* local drivers (within a module */
#define vpiLocalLoad         123   /* local loads (within a module */
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
#define vpiOutTerm           124   /* To get to a delay device's loads. */
#define vpiPorts             125   /* Module port */
#define vpiSimNet            126   /* simulated net after collapsing */
#define vpiTaskFunc          127   /* HDL task/function */

/************************ methods added with 1364-2005 ************************/

#define vpiBaseExpr          131   /* Indexed part-select's base expression  */
#define vpiWidthExpr         132   /* Indexed part-select's width expression */

/******************************** PROPERTIES ********************************/
/************************** generic object properties **************************/

#define vpiUndefined            -1   /* undefined property */
#define vpiType                  1   /* type of object */
#define vpiName                  2   /* local name of object */
#define vpiFullName              3   /* full hierarchical name */
#define vpiSize                  4   /* size of gate, net, port, etc. */
#define vpiFile                  5    /* File name in which the object is used*/
#define vpiLineNo                6    /* line number where the object is used */

/***************************** module properties *****************************/

#define vpiTopModule             7   /* top-level module (boolean) */
#define vpiCellInstance          8   /* cell (boolean) */
#define vpiDefName               9   /* module definition name */
#define vpiProtected            10   /* source protected module (boolean) */
#define vpiTimeUnit             11   /* module time unit */
#define vpiTimePrecision        12   /* module time precision */
#define vpiDefNetType           13   /* default net type */
#define vpiUnconnDrive          14   /* unconnected port drive strength */
#define vpiHighZ                 1    /* No default drive given */
#define vpiPull1                 2    /* default pull1 drive */
#define vpiPull0                 3    /* default pull0 drive */
#define vpiDefFile              15   /* File name where the module is defined*/
#define vpiDefLineNo            16   /* line number for module definition */
#define vpiDefDelayMode         47   /* Default delay mode for a module */
#define vpiDelayModeNone         1    /* no delay mode specified */
#define vpiDelayModePath         2    /* path delay mode */
#define vpiDelayModeDistrib      3    /* distributed delay mode */
#define vpiDelayModeUnit         4    /* unit delay mode */
#define vpiDelayModeZero         5    /* zero delay mode */
#define vpiDelayModeMTM          6    /* min:typ:max delay mode */
#define vpiDefDecayTime         48   /* Default decay time for a module */

/************************** port and net properties **************************/

#define vpiScalar               17   /* scalar (boolean) */
#define vpiVector               18   /* vector (boolean) */
#define vpiExplicitName         19   /* port is explicitly named */
#define vpiDirection            20   /* direction of port: */
#define vpiInput                 1    /* input */
#define vpiOutput                2    /* output */
#define vpiInout                 3    /* inout */
#define vpiMixedIO               4    /* mixed input-output */
#define vpiNoDirection           5    /* no direction */
#define vpiConnByName           21   /* connected by name (boolean) */

#define vpiNetType              22   /* net subtypes: */
#define vpiWire                  1    /* wire net */
#define vpiWand                  2    /* wire-and net */
#define vpiWor                   3    /* wire-or net */
#define vpiTri                   4    /* three-state net */
#define vpiTri0                  5    /* pull-down net */
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
#define vpiTri1                    6   /* pull-up net */
#define vpiTriReg                  7   /* tri state reg net */
#define vpiTriAnd                  8   /* three-state wire-and net */
#define vpiTriOr                   9   /* three-state wire-or net */
#define vpiSupply1                10   /* supply 1 net */
#define vpiSupply0                11   /* supply zero net */
#define vpiNone                   12   /* no default net type (1364-2001) */
#define vpiUwire                  13   /* unresolved wire net (1364-2005) */

#define vpiExplicitScalared      23   /* explicitly scalared (boolean) */
#define vpiExplicitVectored      24   /* explicitly vectored (boolean) */
#define vpiExpanded              25   /* expanded vector net (boolean) */
#define vpiImplicitDecl          26   /* implicitly declared net (boolean) */
#define vpiChargeStrength        27   /* charge decay strength of net */

/*  Defined as part of strengths section.
#define vpiLargeCharge             0x10
#define vpiMediumCharge            0x04
#define vpiSmallCharge             0x02
*/

#define vpiArray                 28   /* variable array (boolean) */
#define vpiPortIndex             29   /* Port index */

/************************ gate and terminal properties ************************/

#define vpiTermIndex             30   /* Index of a primitive terminal */
#define vpiStrength0             31   /* 0-strength of net or gate */
#define vpiStrength1             32   /* 1-strength of net or gate */
#define vpiPrimType              33   /* prmitive subtypes: */
#define vpiAndPrim                1   /* and gate */
#define vpiNandPrim               2   /* nand gate */
#define vpiNorPrim                3   /* nor gate */
#define vpiOrPrim                 4   /* or gate */
#define vpiXorPrim                5   /* xor gate */
#define vpiXnorPrim               6   /* xnor gate */
#define vpiBufPrim                7   /* buffer */
#define vpiNotPrim                8   /* not gate */
#define vpiBufif0Prim             9   /* zero-enabled buffer */
#define vpiBufif1Prim            10   /* one-enabled buffer */
#define vpiNotif0Prim            11   /* zero-enabled not gate */
#define vpiNotif1Prim            12   /* one-enabled not gate */
#define vpiNmosPrim              13   /* nmos switch */
#define vpiPmosPrim              14   /* pmos switch */
#define vpiCmosPrim              15   /* cmos switch */
#define vpiRnmosPrim             16   /* resistive nmos switch */
#define vpiRpmosPrim             17   /* resistive pmos switch */
#define vpiRcmosPrim             18   /* resistive cmos switch */
#define vpiRtranPrim             19   /* resistive bidirectional */
#define vpiRtranif0Prim           20   /* zero-enable resistive bidirectional */
#define vpiRtranif1Prim           21   /* one-enable resistive bidirectional */
#define vpiTranPrim              22   /* bidirectional */
#define vpiTranif0Prim           23   /* zero-enabled bidirectional */
#define vpiTranif1Prim           24   /* one-enabled bidirectional */
#define vpiPullupPrim            25   /* pullup */
#define vpiPulldownPrim          26   /* pulldown */
#define vpiSeqPrim               27   /* sequential UDP */
#define vpiCombPrim              28   /* combinational UDP */

/*************** path, path terminal, timing check properties ***************/

#define vpiPolarity              34   /* polarity of module path... */
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
#define vpiDataPolarity          35   /* ...or data path: */
#define vpiPositive               1    /* positive */
#define vpiNegative               2    /* negative */
#define vpiUnknown                3    /* unknown (unspecified) */

#define vpiEdge                  36   /* edge type of module path: */
#define vpiNoEdge                 0x00     /* no edge */
#define vpiEdge01                 0x01     /* 0 -> 1 */
#define vpiEdge10                 0x02     /* 1 -> 0 */
#define vpiEdge0x                 0x04     /* 0 -> x */
#define vpiEdgex1                 0x08     /* x -> 1 */
#define vpiEdge1x                 0x10     /* 1 -> x */
#define vpiEdgex0                 0x20     /* x -> 0 */
#define vpiPosedge                (vpiEdgex1 | vpiEdge01 | vpiEdge0x)
#define vpiNegedge                (vpiEdgex0 | vpiEdge10 | vpiEdge1x)
#define vpiAnyEdge                (vpiPosedge | vpiNegedge)

#define vpiPathType              37   /* path delay connection subtypes: */
#define vpiPathFull               1    /* ( a *> b ) */
#define vpiPathParallel           2    /* ( a => b ) */

#define vpiTchkType              38   /* timing check subtypes: */
#define vpiSetup                  1    /* $setup */
#define vpiHold                   2    /* $hold */
#define vpiPeriod                 3    /* $period */
#define vpiWidth                  4    /* $width */
#define vpiSkew                   5    /* $skew */
#define vpiRecovery               6    /* $recovery */
#define vpiNoChange               7    /* $nochange */
#define vpiSetupHold              8    /* $setuphold */
#define vpiFullskew               9    /* $fullskew -- added for 1364-2001 */
#define vpiRecrem                10    /* $recrem   -- added for 1364-2001 */
#define vpiRemoval               11    /* $removal  -- added for 1364-2001 */
#define vpiTimeskew              12    /* $timeskew -- added for 1364-2001 */

/*************************** expression properties ***************************/

#define vpiOpType                39   /* operation subtypes: */
#define vpiMinusOp                1    /* unary minus */
#define vpiPlusOp                 2    /* unary plus */
#define vpiNotOp                  3    /* unary not */
#define vpiBitNegOp               4    /* bitwise negation */
#define vpiUnaryAndOp             5    /* bitwise reduction and */
#define vpiUnaryNandOp            6    /* bitwise reduction nand */
#define vpiUnaryOrOp              7    /* bitwise reduction or */
#define vpiUnaryNorOp             8    /* bitwise reduction nor */
#define vpiUnaryXorOp             9    /* bitwise reduction xor */
#define vpiUnaryXNorOp           10    /* bitwise reduction xnor */
#define vpiSubOp                 11    /* binary subtraction */
#define vpiDivOp                 12    /* binary division */
#define vpiModOp                 13    /* binary modulus */
#define vpiEqOp                  14    /* binary equality */
#define vpiNeqOp                 15    /* binary inequality */
#define vpiCaseEqOp              16    /* case (x and z) equality */
#define vpiCaseNeqOp             17    /* case inequality */
#define vpiGtOp                  18    /* binary greater than */
#define vpiGeOp                  19    /* binary greater than or equal */
#define vpiLtOp                  20    /* binary less than */
#define vpiLeOp                  21    /* binary less than or equal */
#define vpiLShiftOp              22    /* binary left shift */
#define vpiRShiftOp              23    /* binary right shift */
#define vpiAddOp                 24    /* binary addition */
#define vpiMultOp                25    /* binary multiplication */
#define vpiLogAndOp              26    /* binary logical and */
```


Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
#define vpiLogOrOp               27   /* binary logical or */
#define vpiBitAndOp              28   /* binary bitwise and */
#define vpiBitOrOp               29   /* binary bitwise or */
#define vpiBitXorOp              30   /* binary bitwise xor */
#define vpiBitXNorOp             31   /* binary bitwise xnor */
#define vpiBitXnorOp             vpiBitXNorOp /* added with 1364-2001 */
#define vpiConditionOp           32   /* ternary conditional */
#define vpiConcatOp              33   /* n-ary concatenation */
#define vpiMultiConcatOp         34   /* repeated concatenation */
#define vpiEventOrOp             35   /* event or */
#define vpiNullOp                36   /* null operation */
#define vpiListOp                37   /* list of expressions */
#define vpiMinTypMaxOp           38   /* min:typ:max: delay expression */
#define vpiPosedgeOp             39   /* posedge */
#define vpiNegedgeOp             40   /* negedge */
#define vpiArithLShiftOp         41   /* arithmetic left shift  (1364-2001) */
#define vpiArithRShiftOp         42   /* arithmetic right shift (1364-2001) */
#define vpiPowerOp               43   /* arithmetic power op    (1364-2001) */

#define vpiConstType             40   /* constant subtypes: */
#define vpiDecConst               1   /* decimal integer */
#define vpiRealConst              2   /* real */
#define vpiBinaryConst            3   /* binary integer */
#define vpiOctConst               4   /* octal integer */
#define vpiHexConst               5   /* hexadecimal integer */
#define vpiStringConst            6   /* string literal */
#define vpiIntConst               7   /* HDL integer constant (1364-2001) */
#define vpiTimeConst              8   /* HDL time constant */

#define vpiBlocking              41   /* blocking assignment (boolean) */
#define vpiCaseType              42   /* case statement subtypes: */
#define vpiCaseExact              1   /* exact match */
#define vpiCaseX                  2   /* ignore X's */
#define vpiCaseZ                  3   /* ignore Z's */
#define vpiNetDeclAssign         43   /* assign part of decl (boolean) */

/************************** task/function properties *************************/

#define vpiFuncType              44   /* HDL function & system function type */
#define vpiIntFunc                1   /* returns integer */
#define vpiRealFunc               2   /* returns real */
#define vpiTimeFunc               3   /* returns time */
#define vpiSizedFunc              4   /* returns an arbitrary size */
#define vpiSizedSignedFunc        5   /* returns sized signed value */

/** alias 1364-1995 system function subtypes to 1364-2001 function subtypes ***/

#define vpiSysFuncType             vpiFuncType
#define vpiSysFuncInt              vpiIntFunc
#define vpiSysFuncReal             vpiRealFunc
#define vpiSysFuncTime             vpiTimeFunc
#define vpiSysFuncSized            vpiSizedFunc

#define vpiUserDefn               45   /*user-defined system task/func(boolean)*/
#define vpiScheduled              46   /* object still scheduled (boolean) */

/*********************** properties added with 1364-2001 *********************/

#define vpiActive                 49  /* reentrant task/func frame is active */
#define vpiAutomatic              50  /* task/func obj is automatic */
#define vpiCell                   51  /* configuration cell */
#define vpiConfig                 52  /* configuration config file */
#define vpiConstantSelect         53  /* (boolean) bit-select or part-select
                                         indices are constant expressions */
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
#define vpiDecompile             54  /* decompile the object */
#define vpiDefAttribute          55  /* Attribute defined for the obj */
#define vpiDelayType             56  /* delay subtype */
#define vpiModPathDelay           1  /* module path delay */
#define vpiInterModPathDelay      2  /* intermodule path delay */
#define vpiMIPDelay               3  /* module input port delay */
#define vpiIteratorType          57  /* object type of an iterator */
#define vpiLibrary               58  /* configuration library */
#define vpiMultiArray            59  /* Object is a multidimensional array */
#define vpiOffset                60  /* offset from LSB */
#define vpiResolvedNetType       61  /* net subtype after resolution, returns
                                        same subtypes as vpiNetType */
#define vpiSaveRestartID         62  /* unique ID for save/restart data */
#define vpiSaveRestartLocation   63  /* name of save/restart data file */
#define vpiValid                 64  /* reentrant task/func frame or automatic
                                        variable is valid */
#define vpiValidFalse             0
#define vpiValidTrue              1
#define vpiSigned                65  /* TRUE for vpiIODecl and any object in
                                        the expression class if the object
                                        has the signed attribute */
#define vpiLocalParam            70  /* TRUE when a param is declared as a
                                        localparam */
#define vpiModPathHasIfNone      71  /* Mod path has an ifnone statement */

/************************ properties added with 1364-2005 ***********************/

#define vpiIndexedPartSelectType 72  /* Indexed part-select type */
#define vpiPosIndexed             1  /* +: */
#define vpiNegIndexed             2  /* -: */
#define vpiIsMemory              73  /* TRUE for a one-dimensional reg array */

/*************** vpi_control() constants (added with 1364-2001) ***************/

#define vpiStop                  66  /* execute simulator's $stop */
#define vpiFinish                67  /* execute simulator's $finish */
#define vpiReset                 68  /* execute simulator's $reset */
#define vpiSetInteractiveScope   69  /* set simulator's interactive scope */

/*************************** I/O related defines ****************************/

#define VPI_MCD_STDOUT  0x00000001

/************************** STRUCTURE DEFINITIONS ***************************/

/******************************* time structure *****************************/

typedef struct t_vpi_time
{
  PLI_INT32  type;              /* [vpiScaledRealTime, vpiSimTime,
                                    vpiSuppressTime] */
  PLI_UINT32 high, low;         /* for vpiSimTime */
  double     real;              /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;

/* time types */

#define vpiScaledRealTime 1
#define vpiSimTime        2
#define vpiSuppressTime   3

/***************************** delay structures *****************************/

typedef struct t_vpi_delay
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
{
  struct t_vpi_time *da;         /* pointer to application-allocated
                                    array of delay values */
  PLI_INT32 no_of_delays;        /* number of delays */
  PLI_INT32 time_type;           /* [vpiScaledRealTime, vpiSimTime,
                                     vpiSuppressTime] */
  PLI_INT32 mtm_flag;            /* true for mtm values */
  PLI_INT32 append_flag;         /* true for append */
  PLI_INT32 pulsere_flag;        /* true for pulsere values */
} s_vpi_delay, *p_vpi_delay;

/*************************** value structures ******************************/

/* vector value */

#ifndef VPI_VECVAL  /* added in 1364-2005 */
#define VPI_VECVAL

typedef struct t_vpi_vecval
{
  /* following fields are repeated enough times to contain vector */
  PLI_INT32 aval, bval;          /* bit encoding: ab: 00=0, 10=1, 11=X, 01=Z */
} s_vpi_vecval, *p_vpi_vecval;

#endif

/* strength (scalar) value */

typedef struct t_vpi_strengthval
{
  PLI_INT32 logic;               /* vpi[0,1,X,Z] */
  PLI_INT32 s0, s1;              /* refer to strength coding below */
} s_vpi_strengthval, *p_vpi_strengthval;

/* strength values */

#define vpiSupplyDrive     0x80
#define vpiStrongDrive     0x40
#define vpiPullDrive       0x20
#define vpiWeakDrive       0x08
#define vpiLargeCharge     0x10
#define vpiMediumCharge    0x04
#define vpiSmallCharge     0x02
#define vpiHiZ             0x01

/* generic value */

typedef struct t_vpi_value
{
  PLI_INT32 format; /* vpi[[Bin,Oct,Dec,Hex]Str,Scalar,Int,Real,String,
                       Vector,Strength,Suppress,Time,ObjType]Val */
  union
    {
      PLI_BYTE8                *str;       /* string value */
      PLI_INT32                 scalar;    /* vpi[0,1,X,Z] */
      PLI_INT32                 integer;   /* integer value */
      double                    real;      /* real value */
      struct t_vpi_time        *time;      /* time value */
      struct t_vpi_vecval      *vector;    /* vector value */
      struct t_vpi_strengthval *strength;  /* strength value */
      PLI_BYTE8                *misc;      /* ...other */
    } value;
} s_vpi_value, *p_vpi_value;
```

Copyright © 2006 IEEE. All rights reserved.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
/* value formats */

#define vpiBinStrVal          1
#define vpiOctStrVal          2
#define vpiDecStrVal          3
#define vpiHexStrVal          4
#define vpiScalarVal          5
#define vpiIntVal             6
#define vpiRealVal            7
#define vpiStringVal          8
#define vpiVectorVal          9
#define vpiStrengthVal       10
#define vpiTimeVal           11
#define vpiObjTypeVal        12
#define vpiSuppressVal       13

/* delay modes */

#define vpiNoDelay            1
#define vpiInertialDelay      2
#define vpiTransportDelay     3
#define vpiPureTransportDelay 4

/* force and release flags */

#define vpiForceFlag          5
#define vpiReleaseFlag        6

/* scheduled event cancel flag */

#define vpiCancelEvent        7

/* bit mask for the flags argument to vpi_put_value() */

#define vpiReturnEvent        0x1000

/* scalar values */

#define vpi0                  0
#define vpi1                  1
#define vpiZ                  2
#define vpiX                  3
#define vpiH                  4
#define vpiL                  5
#define vpiDontCare           6
/*
#define vpiNoChange           7   Defined under vpiTchkType, but
                                  can be used here.
*/

/********************** system task/function structure **********************/

typedef struct t_vpi_systf_data
{
  PLI_INT32 type;                     /* vpiSysTask, vpiSysFunc */
  PLI_INT32 sysfunctype;               /* vpiSysTask, vpi[Int,Real,Time,Sized,
                                                      SizedSigned]Func */
  PLI_BYTE8 *tfname;                  /* first character must be '$' */
  PLI_INT32 (*calltf)(PLI_BYTE8 *);
  PLI_INT32 (*compiletf)(PLI_BYTE8 *);
  PLI_INT32 (*sizetf)(PLI_BYTE8 *);     /* for sized function callbacks only */
  PLI_BYTE8 *user_data;
} s_vpi_systf_data, *p_vpi_systf_data;
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

```
#define vpiSysTask            1
#define vpiSysFunc            2

/* the subtypes are defined under the vpiFuncType property */

/****************** Verilog execution information structure ******************/

typedef struct t_vpi_vlog_info
{
  PLI_INT32 argc;
  PLI_BYTE8 **argv;
  PLI_BYTE8 *product;
  PLI_BYTE8 *version;
} s_vpi_vlog_info, *p_vpi_vlog_info;

/********************** PLI error information structure **********************/

typedef struct t_vpi_error_info
{
  PLI_INT32 state;          /* vpi[Compile,PLI,Run] */
  PLI_INT32 level;          /* vpi[Notice,Warning,Error,System,Internal] */
  PLI_BYTE8 *message;
  PLI_BYTE8 *product;
  PLI_BYTE8 *code;
  PLI_BYTE8 *file;
  PLI_INT32 line;
} s_vpi_error_info, *p_vpi_error_info;

/* state when error occurred */

#define vpiCompile             1
#define vpiPLI                 2
#define vpiRun                 3

/* error severity levels */

#define vpiNotice              1
#define vpiWarning             2
#define vpiError               3
#define vpiSystem              4
#define vpiInternal            5

/***************************** callback structures ****************************/

/* normal callback structure */

typedef struct t_cb_data
{
  PLI_INT32    reason;                       /* callback reason */
  PLI_INT32    (*cb_rtn)(struct t_cb_data *); /* call routine */
  vpiHandle    obj;                          /* trigger object */
  p_vpi_time   time;                         /* callback time */
  p_vpi_value  value;                        /* trigger object value */
  PLI_INT32    index;                        /* index of the memory word or
                                                var select that changed */
  PLI_BYTE8   *user_data;
} s_cb_data, *p_cb_data;

/***************************** CALLBACK REASONS *******************************/
/***************************** Simulation related *****************************/

#define cbValueChange            1
#define cbStmt                   2
#define cbForce                  3
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
#define cbRelease                 4

/******************************** Time related ********************************/

#define cbAtStartOfSimTime        5
#define cbReadWriteSynch          6
#define cbReadOnlySynch           7
#define cbNextSimTime             8
#define cbAfterDelay              9

/******************************** Action related ******************************/

#define cbEndOfCompile           10
#define cbStartOfSimulation      11
#define cbEndOfSimulation        12
#define cbError                  13
#define cbTchkViolation          14
#define cbStartOfSave            15
#define cbEndOfSave              16
#define cbStartOfRestart         17
#define cbEndOfRestart           18
#define cbStartOfReset           19
#define cbEndOfReset             20
#define cbEnterInteractive       21
#define cbExitInteractive        22
#define cbInteractiveScopeChange 23
#define cbUnresolvedSystf        24

/*************************** Added with 1364-2001 ***************************/

#define cbAssign                 25
#define cbDeassign               26
#define cbDisable                27
#define cbPLIError               28
#define cbSignal                 29

/*************************** FUNCTION DECLARATIONS **************************/

/* callback related */

XXTERN vpiHandle  vpi_register_cb     PROTO_PARAMS((p_cb_data cb_data_p));
XXTERN PLI_INT32  vpi_remove_cb       PROTO_PARAMS((vpiHandle cb_obj));
XXTERN void       vpi_get_cb_info     PROTO_PARAMS((vpiHandle object,
                                                    p_cb_data cb_data_p));
XXTERN vpiHandle  vpi_register_systf  PROTO_PARAMS((p_vpi_systf_data
                                                      systf_data_p));
XXTERN void       vpi_get_systf_info  PROTO_PARAMS((vpiHandle object,
                                                    p_vpi_systf_data
                                                      systf_data_p));

/* for obtaining handles */

XXTERN vpiHandle  vpi_handle_by_name  PROTO_PARAMS((PLI_BYTE8 *name,
                                                    vpiHandle scope));
XXTERN vpiHandle  vpi_handle_by_index PROTO_PARAMS((vpiHandle object,
                                                    PLI_INT32 indx));

/* for traversing relationships */

XXTERN vpiHandle  vpi_handle          PROTO_PARAMS((PLI_INT32 type,
                                                    vpiHandle refHandle));
XXTERN vpiHandle  vpi_handle_multi    PROTO_PARAMS((PLI_INT32 type,
                                                    vpiHandle refHandle1,
                                                    vpiHandle refHandle2,
```


Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
                                                    ... ));
XXTERN vpiHandle  vpi_iterate         PROTO_PARAMS((PLI_INT32 type,
                                                    vpiHandle refHandle));
XXTERN vpiHandle  vpi_scan            PROTO_PARAMS((vpiHandle iterator));

/* for processing properties */

XXTERN PLI_INT32  vpi_get             PROTO_PARAMS((PLI_INT32 property,
                                                    vpiHandle object));
XXTERN PLI_BYTE8 *vpi_get_str         PROTO_PARAMS((PLI_INT32 property,
                                                    vpiHandle object));

/* delay processing */

XXTERN void       vpi_get_delays      PROTO_PARAMS((vpiHandle object,
                                                    p_vpi_delay delay_p));
XXTERN void       vpi_put_delays      PROTO_PARAMS((vpiHandle object,
                                                    p_vpi_delay delay_p));

/* value processing */

XXTERN void       vpi_get_value       PROTO_PARAMS((vpiHandle expr,
                                                    p_vpi_value value_p));
XXTERN vpiHandle  vpi_put_value       PROTO_PARAMS((vpiHandle object,
                                                    p_vpi_value value_p,
                                                    p_vpi_time time_p,
                                                    PLI_INT32 flags));

/* time processing */

XXTERN void       vpi_get_time        PROTO_PARAMS((vpiHandle object,
                                                    p_vpi_time time_p));

/* I/O routines */

XXTERN PLI_UINT32 vpi_mcd_open        PROTO_PARAMS((PLI_BYTE8 *fileName));
XXTERN PLI_UINT32 vpi_mcd_close       PROTO_PARAMS((PLI_UINT32 mcd));
XXTERN PLI_BYTE8 *vpi_mcd_name        PROTO_PARAMS((PLI_UINT32 cd));
XXTERN PLI_INT32  vpi_mcd_printf      PROTO_PARAMS((PLI_UINT32 mcd,
                                                    PLI_BYTE8 *format,
                                                    ...));
XXTERN PLI_INT32  vpi_printf          PROTO_PARAMS((PLI_BYTE8 *format,
                                                    ...));

/* utility routines */

XXTERN PLI_INT32  vpi_compare_objects PROTO_PARAMS((vpiHandle object1,
                                                    vpiHandle object2));
XXTERN PLI_INT32  vpi_chk_error       PROTO_PARAMS((p_vpi_error_info
                                                      error_info_p));
XXTERN PLI_INT32  vpi_free_object     PROTO_PARAMS((vpiHandle object));
XXTERN PLI_INT32  vpi_get_vlog_info   PROTO_PARAMS((p_vpi_vlog_info
                                                      vlog_info_p));

/* routines added with 1364-2001 */

XXTERN PLI_INT32  vpi_get_data        PROTO_PARAMS((PLI_INT32 id,
                                                    PLI_BYTE8 *dataLoc,
                                                    PLI_INT32 numOfBytes));
XXTERN PLI_INT32  vpi_put_data        PROTO_PARAMS((PLI_INT32 id,
                                                    PLI_BYTE8 *dataLoc,
                                                    PLI_INT32 numOfBytes));
XXTERN void      *vpi_get_userdata    PROTO_PARAMS((vpiHandle obj));
XXTERN PLI_INT32  vpi_put_userdata    PROTO_PARAMS((vpiHandle obj,
```

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

```
                                                    void *userdata));
XXTERN PLI_INT32  vpi_vprintf         PROTO_PARAMS((PLI_BYTE8 *format,
                                                    va_list ap));
XXTERN PLI_INT32  vpi_mcd_vprintf     PROTO_PARAMS((PLI_UINT32 mcd,
                                                    PLI_BYTE8 *format,
                                                    va_list ap));
XXTERN PLI_INT32  vpi_flush           PROTO_PARAMS((void));
XXTERN PLI_INT32  vpi_mcd_flush       PROTO_PARAMS((PLI_UINT32 mcd));
XXTERN PLI_INT32  vpi_control         PROTO_PARAMS((PLI_INT32 operation,
                                                    ...));
XXTERN vpiHandle  vpi_handle_by_multi_index PROTO_PARAMS((vpiHandle obj,
                                                    PLI_INT32 num_index,
                                                    PLI_INT32 *index_array));

/***************************** GLOBAL VARIABLES ******************************/

PLI_VEXTERN PLI_DLLESPEC void (*vlog_startup_routines[])();

  /* array of function pointers, last pointer should be null */

#undef PLI_EXTERN
#undef PLI_VEXTERN

#ifdef VPI_USER_DEFINED_DLLISPEC
#undef VPI_USER_DEFINED_DLLISPEC
#undef PLI_DLLISPEC
#endif
#ifdef VPI_USER_DEFINED_DLLESPEC
#undef VPI_USER_DEFINED_DLLESPEC
#undef PLI_DLLESPEC
#endif

#ifdef PLI_PROTOTYPES
#undef PLI_PROTOTYPES
#undef PROTO_PARAMS
#undef XXTERN
#undef EETERN
#endif

#ifdef  __cplusplus
}
#endif

#endif /* VPI_USER_H */
```


--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Annex H

(informative)

Encryption/decryption flow

This annex describes a number of scenarios that can be used for IP protection, and it also shows how the relevant pragmas are used to achieve the desired effect of securely protecting, distributing, and decrypting the model.

The data to be protected from inappropriate access or from unauthorized modification is placed within a protect **begin-end** block. Information in the **begin-end** block, once encrypted, is also protected.

H.1 Tool vendor secret key encryption system

In the secret key encryption system, the key is tool vendor proprietary and is embedded within the tool itself. The same key is used for both encryption and decryption. (In the electronic design automation domain, this is the simplest scenario and is roughly equivalent to the historical `protect technique.) It has the drawback of being completely tool vendor-specific. Using this technique, the IP author can encrypt the IP, and any IP consumer with appropriate licenses and the same tool vendor can utilize the IP.

H.1.1 Encryption input

The following pragmas are expected when using the tool vendor secret key encryption system. The pragmas required in the encryption input for use of the secret key encryption system are as follows:

data_keyname= <key name>	Where <key name> is a valid name of an tool's embedded key.
begin/**end**	Surrounding the region(s) to be encrypted.

Additional optional pragmas that may be included are as follows:

author=<string>	To embed author name.
author_info=<string>	To embed arbitrary author information.
data_keyowner= <owner identity>	This must be the key owner of the provided name.
data_method= <method-specifier>	A method appropriate for the given key name. This may be necessary if something other than the default number of rounds, initialization vector, or key width is used.
encoding=<encoding-specifier>	To specify a different encoding.
digest_block	If a message authorization code is desired to validate that the message has not been modified.
decrypt_license	If the IP author desires a decryption license.
runtime_license	If the IP author desires a run-time license.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

H.1.2 Encryption output

The encrypting tool should take the input file and copy all cleartext to the corresponding output sections. For each protect begin-end block, it should generate the following:

begin_protected	To start the protected region.
data_keyowner= <owner identity>	
data_keyname=<key name>	
data_method=<method-specifier>	
encoding=<encoding-specifier>	
author=<string>	If provided in the input.
author_info=<string>	If provided in the input.
digest_block	Followed on the next line(s) by the encoded encrypted digest.
data_block	Followed on the next line(s) by the encoded encrypted data composed of the following: **decrypt_license** **encrypt_license** <text found between begin/end>
end_protected	

H.2 IP author secret key encryption system

In this mechanism, the IP is encrypted with the public key (of a public/private key pair) of the IP author, and the decrypting tool will have the IP author's private key in its secure key database. The IP authors will have to provide their private keys to the tools' database so that the tool will be able to decrypt the design.

H.2.1 Encryption input

The following pragmas are expected when using the IP author secret key encryption system:

data_keyname= <providers key name>	
begin/**end**	Surrounding the region(s) to be encrypted.

Additional optional pragmas that may be included are as follows:

author=<string>	To embed author name.
author_info=<string>	To embed arbitrary author information.
data_keyowner=<owner identity>	This must be the key owner of the provided name.

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

data_method= some_publ_priv_encryption_scheme_name <method-specifier>
A method appropriate for the given key name. This may be necessary if something other than the default number of rounds, initialization vector, or key width is used.

encoding=<encoding-specifier> To specify a different encoding.

digest_block If a message authorization code is desired to validate that the message has not been modified.

decrypt_license If the IP author desires a decryption license.

runtime_license If the IP author desires a run-time license.

H.2.2 Encryption output

The encrypting tool should take the input file and copy all cleartext to the corresponding output sections. For each protect **begin-end** block, it should generate the following:

begin_protected To start the protected region.

data_keyowner= <owner identity>

data_keyname=<providers key name>

data_method=some_publ_priv_encryption_scheme_name

encoding=<encoding-specifier>

author=<string> If provided in the input.

author_info=<string> If provided in the input.

digest_block Followed on the next line(s) by the encoded encrypted digest.

data_block Followed on the next line(s) by the encoded encrypted data composed of the following:
decrypt_license
encrypt_license
<text found between begin/end>

end_protected

H.3 Digital envelopes

In this mechanism, each recipient has a public and private key for an asymmetric encryption algorithm. The sender encrypts the design using a symmetric key encryption algorithm and then encrypts the symmetric key using the recipient's public key. The encrypted symmetric key is recorded in a **key_block** in the protected envelope. The recipient is able to recover the symmetric key using the appropriate private key and then decrypts the design with the symmetric key. This technique permits efficient encryption methods for the design data, yet secret information is never transmitted without encryption. Digital envelopes can be created

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

using either tool secret key or IP author secret key protection schemes. The keys for the recipient user or tool protect the transmission of the symmetric key that encrypts the design data. By using more than one **key_block**, a single protected envelope can be decrypted by tools from different vendors and/or different users.

Instead of using the public key of a public/private key pair, a tool-specific embedded key can also be used to encrypt the **key_block**. In this case also as only the tool know its embedded key, only it can internally decrypt the design; hence the same effect can be achieved. The only disadvantage is that the tool's embedded key will have to be provided to the IP author in some form.

In the following example, the data_method and **data_keyowner**/**data_keyname** are used to encrypt the **data_block**. The key to encrypt the **data_block** can be specified either by a **data_keyowner**/ **data_keyname** pair or by a **data_decrypt_key** pragma expression. In the first case, the encrypting tool encrypts the **data_keyowner** and **data_keyname** pragmas with the **key_keymethod**/**key_keyname** and puts them in the **key_block** along with **data_method**. Alternatively, with the **data_decrypt_key** pragma, the actual key is provided, which is then encrypted with **key_method**/**key_keyname** and stored in the **key_block**.

In the first approach, the **data_keyowner**/**data_keyname** should also be present with the decrypting tool. No such dependency exists with the second approach as the key is present in the file itself.

For better security in the first approach, the encrypting tool can actually read the **data_keyowner**/ **data_keyname** key and put it in the **key_block** as **data_decrypt_key**. This step not only will remove the dependency mentioned above, but will also protect against the hit-and-trial breaking of the **data_block** with the existing keys at the IP user's end.

H.3.1 Encryption input

The following pragmas are expected when using the digital envelopes:

key_keyowner = <owner identity>

key_method = some_encryption_scheme_name

key_keyname= <providers key name>

data_keyname= <providers key name>

begin/**end**	Surrounding the region(s) to be encrypted.

Additional optional pragmas that may be included are as follows:

author=<string>	To embed author name.
author_info=<string>	To embed arbitrary author information.
data_keyowner= <owner identity>	This must be the key owner of the provided name.
data_method= <method-specifier>	A method appropriate for the given key name. This may be necessary if something other than the default number of rounds, initialization vector, or key width is used
encoding=<encoding-specifier>	To specify a different encoding.

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS Not for Resale

digest_block	If a message authorization code is desired to validate that the message has not been modified.
decrypt_license	If the IP author desires a decryption license.
runtime_license	If the IP author desires a run-time license.

H.3.2 Encryption output

The encrypting tool should take the input file and copy all cleartext to the corresponding output sections. For each protect **begin-end** block, it should generate the following:

begin_protected	To start the protected region.
key_keyowner= <owner identity>	
key_method = some_encryption_scheme_name	
key_keyname= <providers key name>	
key_block = <encrypted encoded data>	This contains the data_key_owner, data_method, and the symmetric data_key itself in encrypted form.
encoding=<encoding-specifier>	
author=<string>	If provided in the input.
author_info=<string>	If provided in the input.
digest_block	Followed on the next line(s) by the encoded encrypted digest.
data_block	Followed on the next line(s) by the encoded encrypted data composed of the following: **decrypt_license** **encrypt_license** <text found between begin/end>
end_protected	

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Annex I

(informative)

Bibliography

[B1] IEEE Std 1497-2001, IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process.[11]

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

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

Index

Symbols

!
  compared to ‘==0’, 50
  logical negation operator, 42, 49
!=
  logical inequality operator, 42, 49
!==
  case inequality operator, 42, 49
""
  null string, 60
$, 367, 374
$async$and$array, 303
$async$and$plane, 303
$async$nand$array, 303
$async$nand$plane, 303
$async$nor$array, 303
$async$nor$plane, 303
$async$or$array, 303
$async$or$plane, 303
$bitstoreal, 178, 310–311
$countdrivers, 511–512
$display, 278–285
  compared to $monitor, 285–286
  compared to $write, 278
  escape sequences, 278
  format specifications, 279–281
  size of displayed data, 281–282
$displayb, 278
$displayh, 278
$displayo, 278
$dist_chi_square, 312
$dist_erlang, 312
$dist_exponential, 312
$dist_normal, 312
$dist_poisson, 312
$dist_t, 312
$dist_uniform, 312
$dumpall, 328, 335
$dumpfile, 325
$dumpflush, 328
$dumplimit, 328
$dumpoff, 327, 336, 341
$dumpon, 327
$dumpports, 338
  rules to use, 339
$dumpportsall, 340
$dumpportsflush, 341
$dumpportslimit, 340
$dumpportsoff, 339
$dumpportson, 339
$dumpvars, 326
$fclose, 287–289
$fdisplay, 288–289
$fdisplayb, 288
$fdisplayh, 288
$fdisplayo, 288
$ferror, 290, 295
$fflush, 295
$fgetc, 290
$finish, 302
$fmonitor, 288–289
$fmonitorb, 288
$fmonitorh, 288
$fmonitoro, 288
$fopen, 287–289
$fscanf, 291
$fseek, 290, 294
$fstrobe, 288–289
$fstrobeb, 288
$fstrobeh, 288
$fstrobeo, 288
$ftell, 294
$fullskew, 252
$fwrite, 288–289
$fwriteb, 288
$fwriteh, 288
$fwriteo, 288
$getpattern, 512
$hold, 242
$incsave, 515
$input, 513
$itor, 310
$key, 513
$list, 513
$log, 514
$monitor, 286
  compared to $display, 286
$monitorb, 286
$monitorh, 286
$monitoro, 286
$monitoroff, 286
$monitoron, 286
$nochange, 257
$nokey, 513
$nolog, 514
$period, 256
$printtimescale, 299
$q_add, 307
$q_exam, 308
$q_full, 308
$q_initialize, 307
$q_remove, 307
$random, 311
$readmemb, 296–297
  and loading logic array personality, 304

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

$readmemh, 296–297
  and loading logic array personality, 304
$realtime, 310
$realtobits, 178, 310–311
$recovery, 246
$recrem, 247
$removal, 245
$reset, 514
$reset_count, 514
$reset_value, 514
$restart, 515
$rewind, 294
$rtoi, 310
$save, 515
$scale, 516
$scope, 516
$sdf_annotate system task, 297
$setup, 241
$setuphold, 243
$sformat, 289
$showscopes, 516
$showvars, 516
$signed, 65
$skew, 249
$sreadmemb, 517
$sreadmemh, 517
$sscanf, 291
$stime, 309
$stop, 302
$strobe, 285
  compared to $display, 285
$strobeb, 285
$strobeh, 285
$strobeo, 285
$swrite, 289
$swriteb, 289
$swriteh, 289
$swriteo, 289
$sync$and$array, 303
$sync$and$plane, 303
$sync$nand$array, 303
$sync$nand$plane, 303
$sync$nor$array, 303
$sync$nor$plane, 303
$sync$or$array, 303
$sync$or$plane, 303
$test$plusargs, 320
$time, 33, 309
$timeformat, 300–302
$timeskew, 250
$ungetc, 290
$unsigned, 65
$vcdclose, 341
$width, 255–256
$write, 278–285
  compared to $display, 278
  escape sequences, 278
  format specifications, 279–281
  size of displayed data, 281–282
$writeb, 278
$writeh, 278
$writeo, 278
%
  in format specifications, 278, 282
  modulus operator, 42
&
  bitwise AND operator, 42
  reduction AND operator, 42
&&
  logical AND operator, 42, 49
(??)
  in state table, 108
(01)
  in state table, 108
(0x)
  in state table, 108
(1x)
  in state table, 108
(vw)
  in state table, 108
(x1)
  in state table, 108
*
  arithmetic multiplication operator, 42
  in state table, 108
**, 45
,,
  in null expressions, 278
/
  arithmetic division operator, 42
<
  relational less-than operator, 42, 48
<<
  left shift operator, 53
  logical left shift operator, 42
<<<
  arithmetic left shift operator, 42
<=
  relational less-than-or-equal operator, 42, 48
=
  in assignment statement, 68
==
  logical equality operator, 42, 49
===
  case equality operator, 42, 49
>
  relational greater-than operator, 42, 48
>=

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

relational greater-than-or-equal operator, 42, 48
>>
logical right shift operator, 42
right shift operator, 53
>>>
arithmetic right shift operator, 42
?
equivalent to z in literal number values, 11, 129
in state table, 108, 111
?:
conditional operator, 42
@
for addressing memory, 296
\
backslash character, 14
for escape sequences in strings, 278
\"
as " character, 14
\ddd
specify character as octal digits, 14
\t
tab character, 14
^
bitwise exclusive OR operator, 42
reduction XOR operator, 42
^~
bitwise equivalence operator, 42
reduction XNOR operator, 42
`
in compiler directives, 349
`celldefine, 349
`default_decay_time, 518
`default_nettype, 349
`default_trireg_strength, 518
`define, 350
and text macro substitutions, 352
`delay_mode_distributed, 519
`delay_mode_path, 519
`delay_mode_unit, 519
`delay_mode_zero, 519
`else, 352
`elsif, 353
`endcelldefine, 349
`endif, 353
`ifdef, 352
`ifndef, 352
`include, 356
`nounconnected_drive, 360
`resetall, 356
`timescale, 358
`unconnected_drive, 360
`undef, 352
{{}}
replication operator, 42
{}
concatenation operator, 42, 54
|
bitwise inclusive OR operator, 42
reduction OR operator, 42
||
logical OR operator, 42, 49
~
bitwise negation operator, 42
~&
reduction NAND operator, 42
~^
bitwise equivalence operator, 42
reduction XNOR operator, 42
~|
reduction NOR operator, 42

Numerics

0
for minimizing bit lengths of expressions, 282
in state table, 108
logic zero, 21, 283
01 transition, 111
1
in state table, 108
logic one, 21, 283

A

access routines
history, 366
accurate simulation
requirements, 261
addressing memory, 296–297
always
and activity flow, 116
ambiguous strength, 89–99
and gate, 80–81
arguments
system task/function, 368
arithmetic operators, 42, 45–46
–, 45
%, 45
*, 45
**, 45
+, 45
/, 45
and unknown logic values, 46
arrays, 34
element, 35
format, 304
index, 35
word, 35
assign, 161
assign procedural continuous assignment state-

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

ment, 123
assignment, 68–72
continuous, 68–72, 117
left-hand side, 68
of delays to module paths, 222–224
procedural, 117–125
procedural versus continuous, 117
right-hand side, 68
steps for evaluating, 66
variable declaration, 72
assignments
scheduling implications, 161
asynchronous arrays, 303–307
attributes, 16

B

b
binary number format, 10
in state table, 108
backannotation, 269
backslash character, 14
base format
binary, 10
decimal, 10
hexadecimal, 10
octal, 10
basic configuration elements, 200
begin-end block statement, 125, 140
behavioral modeling, 116–144
bidirectional pass gate, 84
binary display format, 10
and high-impedance state, 283
and unknown logic value, 283
Binary operators, 8
binary operators
{}, 54
binding instances, 199
bit-select
of vector net or register, 56
out of bounds, 56, 58
references of real numbers, 33
bitwise operators, 50
AND, 42
equivalence, 42
exclusive OR, 42
inclusive OR, 42
negation, 42
blank port connection, 166
block comment, 8
block statement, 139–142
fork-join, 139
naming of, 141–142
parallel, 140
sequential, 139
start and finish times, 142
timing for embedded blocks, 142
blocking assignment statement, 161
process, 161
blocking assignments, 117
blocking procedural assignment, 117
buf gate, 81–82
bufif gate, 82–83

C

calltf routines, 462
capacitive networks, 28–30
capacitive state, 28
case
item expressions, 127
case equality operator, 42
case inequality operator, 42
case statement, 127–129
compared to if-else-if statement, 128
constant expression, 129
with do-not-care, 128–129
casex, 128
casez, 128
cbAfterDelay, 459
cbAssign, 454
cbAtEndOfSimTime, 459
cbAtStartOfSimTime, 458
cbDeassign, 454
cbDisable, 454
cbEndOfCompile, 460
cbEndOfRestart, 460
cbEndOfSave, 460
cbEndOfSimulation, 460
cbEnterInteractive, 460
cbError, 460
cbExitInteractive, 460
cbForce, 454
cbInteractiveScopeChange, 460
cbNBASynch, 458
cbNextSimTime, 459
cbPLIError, 460
cbReadOnlySynch, 459
cbReadWriteSynch, 459
cbRelease, 454
cbSignal, 460
cbStartOfRestart, 460
cbStartOfSave, 460
cbStartOfSimulation, 460
cbStmt, 454
cbTchkViolation, 460
cbUnresolvedSystf, 460
cbValueChange, 454
cell, 199
multiple, 202

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

CELL declaration, 269
DELAY, 269
LABEL, 269
TIMINGCHECK, 269
characters
specified as octal digits, 14
charge decay, 30, 103
charge decay process, 103
charge decay time, 103
delay specification, 103
charge storage
strength, 25
charge storage strength, 88
classes of PLI routines
calltf, 462
compiletf, 462
clause
cell, 204
using, 207
default, 203
using, 207
instance, 203
using, 208
liblist, 204
use, 204
cmos, 83, 85
cmos gate, 85–86
combinational UDPs, 105, 109
compared to level-sensitive sequential, 110
input and output fields in state table, 107
combined signal strengths, 88–99
combined signal values, 88–99
command line considerations, 206
comments, 8
compare
string operation, 59
Compiler directives, 15
compiletf routines, 462
concatenation
and unsized numbers, 54
of names, 191
operator, 42, 54
string operation, 59
concurrency
of activity flow, 116
condition
deterministic, 265
nondeterministic, 265
conditional compilation, 352
conditional expression, 215
conditional operator, 42, 53–54
modeling three-state output busses, 54
conditional statement, 125–126
conditioned event, 265–266
versus unconditioned event, 265
config, 199
configurations, 199, 202
hierarchical, 205
conflicts, 26–27
connecting ports
by name, 177–178
by position with ordered list, 176
rules, 179–180
connection
difference between full and parallel, 220
full, 219
parallel, 219
constant expression, 41
constant function, 156
constant numbers, 9
context-determined expression, 62
continuous assignment, 68–72, 161
and connecting ports, 179
and driving strength, 88, 283
and net variables, 117
and wire nets, 26
driving strength of, 71
explicit declaration, 69
implicit declaration, 69
versus procedural assignment, 72
control string, 291
conversion, 12, 33
copy
string operation, 59
counting number of drivers, 512

D

d (decimal number format), 10
data types, 21–40
deassign, 161
deassign procedural statement, 123
decimal display format, 10
and high-impedance state, 282
and unknown logic value, 282
compatibility with $monitor, 282
decimal notation, 12
declaring
events, 133
multiple module paths in a single statement, 220
parameters in specify blocks, 38–39
default
in case statement, 127
in if-else-if statements, 126
default statement, 200
defparam, 168–170
delay
calculating for high-impedance (z) transitions, 101

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

calculating for unknown logic value (x) transitions, 101
control, 131–132
default, 101
distributed, 211–226
fall, 101
falling, 102
for continuous assignment, 71
gate, 101–103
minimum:typical:maximum values, 102
module path, 211–226
net, 101–103
propagation, 77, 101
rise, 101–102
rules for delays controling the assignment, 71
specify one value, 101
specify three values, 101
specify two values, 101
trireg charge decay, 103
turn-off, 102
delay selection, 225
delay specification, 77
delays
inertial, 451
pure transport, 451
transport, 451
describing simple module paths, 213
design, 200
design statement, 202
determinism in simulation execution, 160
diagnostic messages
from $stop and $finish, 302
disable
named blocks, 150
tasks, 150
use of, 150
displaying information, 278–285
displaying library binding information, 208
do-not-care bits
in case statements, 129
double quote character, 14
drive strength specification, 76
driven state, 28
driving strength, 88
compared to charge storage strength, 283
keywords, 72

E

edge transitions, 259
edge-control specifiers, 258–259
edge-sensitive paths, 214–218
edge-sensitive state-dependent paths, 217
edge-sensitive UDPs, 110
compared to level-sensitive UDPs, 110
element (reg in array), 35
else, 126
embedding modules, 163, 165
enable, 136
enabling tasks, 145
end
sequential block, 139
endconfig, 199
endspecify, 39, 211
equality operators, 49
!=, 49
!==, 49
==, 49
===, 49
and ambiguous results, 49
and operands of different sizes, 49
precedence, 49
escape sequences, 278
escaped identifiers, 14
espresso format, 305
event
active, 159
control, 131–132
evaluation, 158
explicit, 131
expression, 131
future, 159
implicit, 131
inactive, 159
level sensitive control, 136
monitor, 159
named, 133–134, 158
nonblocking assign update, 159
OR construct, 134
queue, 158
update, 158
event control
repeat, 137–139
event queue, 158
scheduling an event, 158
event simulation, 158
exit simulator, 302
expanded object, 24
expansion
of vector nets, 24
explicit event, 131
explicit zero delay, 159
expressions, 41–64
bit lengths, 62–64
constant, 41
context-determined, 62
self-determined, 62
steps for evaluating, 65

Copyright © 2006 IEEE. All rights reserved.
--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

F
f (in state table), 108
fall delay, 101–102
file descriptor, 288
file inclusion, 356
file path resolution, 201
file positioning, 294
finish time
  in parallel block statements, 142
  in sequential block statements, 142
flushing output, 295
for loop, 130
force, 161
forever loop, 130
fork-join block statement, 139
fork-join construct, 138
format specifications, 279–281
  ASCII character, 279
  b or B, 279
  binary, 279
  c or C, 279
  d or D, 279
  decimal, 279
  h or H, 279
  hexadecimal, 279
  hierarchical name, 280
  library binding, 280
  m or M, 280
  net signal strength, 280, 283–285
  o or O, 279
  octal, 279
  s or S, 280
  string, 280, 285
  t or T, 280–281
  time format, 280
  timescales, 281
  u or U, 280
  v or V, 280
  z or Z, 280
formats
  array, 304
  of logic array personality, 304–307
  plane, 305
formatting data to a string, 289
frames, 404
full connection, 219–220
fullname, 438
function
  call, 155
  constant
    calls, 156
functions, 152–156, 162
  and scope, 195
  as structured procedures, 143
  definition, 143
  purpose, 145
  returning a value, 154
  rules, 155

G
gate type specification, 76
gate-level modeling, 74–104
gates
  and, 80–81
  bidirectional pass, 84
    delay specifications, 85
  buf, 81–82
  bufif, 82–83
  cmos, 85–86
    delay specification, 85
  compared to continuous assignments, 74
  connection list, 78
  delay, 101–103
  MOS, 83–84
  nand, 80–81
  nor, 80–81
  not, 81–82
  notif, 82–83
  notif0, 82–83
  notif1, 82–83
  or, 80–81
  pulldown, 86
  pullup, 86
  rules for instance connections, 78
  terminal list, 78
  xnor, 80–81
  xor, 80–81

H
h (hexadecimal number format), 10
H (logic 1 or high-impedance state in strength format), 283
handles
  vpiHandle data type, 378
hexadecimal display format, 10
  and high-impedance state, 282
  and unknown logic value, 282
Hi (high-impedance in strength format), 284
hierarchical config
  using, 208
hierarchical configurations, 205
hierarchical path name, 191
hierarchy
  level, 191
  name referencing, 191, 280
  of modules, 163
  scope, 191
  scope rules for naming, 195–196

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

structures, 163–197
high-impedance state
and numbers, 10
and trireg nets, 28
and UDPs, 113
display formats, 282–284
effect in different bases, 10
strength display format, 284
symbolic representation, 21
highz0, 77
highz1, 77

I

I/O error status, 295
identifiers, 14
escaped, 14
keywords, 15
if-else statement
omitting else from nested if, 125
purpose, 125
If-else-if, 126
if-else-if statement
compared to case statement, 128
ifnone condition, 218
implicit
declarations, 25, 349
event, 131
implicit bidirectional connections, 162
implicit continuous assignment statements, 162
implicit conversion, 12, 33
implicit event, 132
include command, 202
incremental restart, 515
incremental save, 515
index
of array, 35
of memory, 35
inertial delays, 451
initial, 143
and activity flow, 116
for specifying waveforms, 144
initial statements
in UDPs, 111–112
instance statement, 200
instantiation
of modules, 163–167
integer constants, 10
integers, 32
division, 45
intra-assignment timing controls, 136–139

K

keywords, 15

L

L (logic 0 or high impedance state in strength format), 283
La (large capacitor in strength format), 284
large, 25, 28
left-hand index, 77
level-sensitive
event control, 136
paths, 215–219
sequential UDPs, 110
versus combinational UDP, 110
level-sensitive UDPs
compared to edge-sensitive UDPs, 110
lexical conventions, 8–16
lexical token
comment, 8
definition of, 8
number, 9
operator, 8
types, 8
white space, 8
liblist clause, 200
libraries, 200
library map
library declaration, 200
library notation, 199
loading memory data from a file, 296
loading timing data from an SDF file, 297
logic array
personality declaration and loading, 304
logic array personality, 304–307
declaration, 304
formats, 304–307
loading, 304
logic gates
and, 80–81
bidirectional pass, 84
buf, 81–82
bufif, 82–83
cmos, 85–86
compared to continuous assignments, 74
delay, 101–103
MOS, 83–84
nand, 80–81
nor, 80–81
not, 81–82
notif, 82–83
or, 80–81
pulldown, 86
pullup, 86
xnor, 80–81
xor, 80–81
logic one, 21
logic planes, 304

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

logic strength modeling, 86–101
logic zero, 21
logical operators, 49
  !, 49
  &&, 49
  ||, 49
  AND, 42
  and ambiguous results, 49
  and unknown logic value, 49
  equality, 42
  inequality, 42
  negation, 42
  OR, 42
  precedence, 49
looping statement, 130–131
  for loop, 130
  forever loop, 130
  repeat loop, 130
  while loop, 130
lsb (least significant bit), 24

M

mapping source files to libraries, 202
Me (medium capacitor in strength format), 284
medium, 25, 28
memory, 34–35
  addressing, 57
  assigning values to, 35
  index, 35
minimum:typical:maximum values
  delay, 102
  for module path delays, 222–223
  format, 61–62
minus sign(-)
  arithmetic subtraction operator, 42
  in state table, 108
mixing path and distributed delays, 225
modeling
  asynchronous clear/preset on an edge-triggered D flip-flop, 123
  logic strength, 86–101
module, 163–166
  and user-defined primitives (UDPs), 105
  definition, 163–164
  hierarchy, 163
  instance parameter value assignment, 170
  instance parameter value assignment by ordered list, 170
  instantiation, 165–167
  overriding parameter values, 167–173
  parameter assignment by name, 171
  parameter dependencies, 173
  port, 166
  terminal, 166
  top-level, 165
module parameter, 36
  dependencies, 173
  overriding values, 167–173
  passing to tasks, 147–148
module path
  definition, 212
  delay, 222–226
  destination, 211, 213, 220
  polarity, 220–221
  simple, 213
  source, 211, 213, 220
module path restrictions, 212
modulus operator, 42
  definition, 45
monitor flag, 286
monitoring
  continuous, 286
  strobed, 285
MOS gate, 83–84
  nmos, 84
  pmos, 84
  rnmos, 84
  rpmos, 84
MOS strength handling, 100
msb (most significant bit), 24
mtm_flag, 425, 448
multichannel descriptor, 287–288
multiple drivers
  at same strength level, 98
  driving the same net, 27
  inside a module, 226
  outside a module, 227
multiple library map files, 202
multiple module path delays
  assigning in one statement, 220
multi-way decisions
  if-else-if statement, 126
multiway decisions
  case statement, 127

N

n (in state table), 108
name, 374, 438
name space, 39
  block name space, 39
  definitions, 39
  module name space, 40
  port name space, 40
  specify block name space, 40
name spaces, 39
named blocks
  and hierarchical names, 191
  and scope, 195

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

purpose, 141–142
named events, 133–134, 158
used with event expressions, 133
names
of hierarchical paths, 191
nand gate, 80–81
negative numbers, 10
negedge, 133, 214, 259
net and register bit addressing, 57
net arrays, 34
net delay, 71
net type resolution rule, 180
net type table, 180
net types, 26
nets, 21–32
delay, 101–103
trireg strength, 88
types of, 26–32
wired logic, 98
new line character, 14, 279
newline character, 14
nmos, 83–84
node
in hierarchical name tree, 191
nonblocking assignment statement, 161
nonblocking procedural assignment, 118–122
evaluating assignments, 119
multiple assignments, 121
nondeterminism in simulation execution, 160
nor gate, 80–81
not gate, 81–82
notif gate, 82–83
notif0, 83
notif1, 83
notifier, 259–261
in edge-sensitive UDP, 260–261
notifiers
user-defined responses to timing violations, 259
null (expression), 278
numbers, 9
base format, 10
size specification, 10

O

o (octal number format), 10
octal display format, 10
on/off control
of monitoring tasks, 286
one-line comment, 8
opening and closing files, 287
operands, 55–60
definition, 41
strings, 58–60
operators, 41
–, 42
!, 42, 49
!=, 42, 49
!==, 42, 49
%, 42
&, 42
&&, 42, 49
*, 42
**, 42
*>, 213–220
+, 42
/, 42
<, 42, 48
<<, 42, 53
<<<, 42, 53
<=, 42, 48
=, 68
==, 42, 49
===, 42, 49
=>, 213–220
>, 42, 48
>=, 42, 48
>>, 42, 53
>>>, 42, 53
?:, 42
^, 42
^~, 42
{{}}, 42
{}, 42, 54
|, 42
||, 42, 49
~, 42
~&, 42
~^, 42
~|, 42
and real numbers, 33
arithmetic, 42, 45–46
binary, 8
bitwise, 50
bitwise AND, 42
bitwise equivalence, 42
bit-wise exclusive OR, 42
bitwise inclusive OR, 42
bit-wise negation, 42
case equality, 42
case inequality, 42
concatenation, 42, 54
conditional, 8, 42, 53–54
definition, 8
equality, 49
left shift
arithmetic, 42
logical, 42
logical, 49

 Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

logical AND, 42
logical equality, 42
logical inequality, 42
logical negation, 42
logical OR, 42
modulus, 42
reduction, 51–52
reduction AND, 42, 51
reduction NAND, 42, 51
reduction NOR, 42, 51
reduction OR, 42, 51
reduction XNOR, 42, 51
reduction XOR, 42, 51
relational, 42, 48
replication, 42
right shift
arithmetic, 42
logical, 42
shift, 53
unary, 8
unary reduction, 51
or gate, 80–81
output
to files, 286–289
overloading system task/function names, 367
overriding module parameter values, 167–173
assigning values in-line within module instances, 170
defparam, 168

P

p (in state table), 108
parallel block, 141
parallel block statement
finish time, 142
start time, 142
parallel connection, 219–220
parameters, 35
parentheses
and changing operator precedence, 43
part-select
of vector net or register, 56
references of real numbers, 33
PATHPULSE$ specparam, 228
personality
memory, 303
of logic array, 304–307
PLA devices
array logic types, 304
array types, 303–307
list of system tasks, 303
logic array personality declaration, 304
logic array personality formats, 304–307
logic array personality loading, 304
plane
format, 305
in programmable logic arrays, 304
PLI history, 366
PLI mechanism, 368
plus sign(+)
arithmetic addition operator, 42
pmos, 83–84
polarity, 220–221
negative, 221
positive, 221
unknown, 221
port, 173–191
connecting
by name, 177–178
by position with ordered list, 176
rules for, 179–180
connecting module instance ports by name, 177
connecting module instance ports by ordered list, 176
declaration, 174
definition, 173
module, 166
port connections, 162
port expression, 177
posedge, 133, 214, 259
power supplies
modeled by supply nets, 32
precedence
equality operators, 49
logical operators, 49
relational operators, 48
precompiling using a separate compilation tool, 206
primitive instance identifier, 77
probabilistic distribution functions, 311–312
$dist_chi_square, 312
$dist_erlang, 312
$dist_exponential, 312
$dist_normal, 312
$dist_poisson, 312
$dist_t, 312
$dist_uniform, 312
procedural assignment, 117–125
and integers, 33
and time variables, 33
blocking, 117
nonblocking, 118–122
versus continuous assignment, 72
procedural assignments
blocking assignment, 117
procedural continuous assignment, 161
procedural continuous assignments, 122–125
assign, 123–124

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

deassign, 123–124
force, 124
release, 124
procedural statements
in behavioral models, 116
procedural timing controls, 131–139
delay control, 132
event control, 131
fork-join block, 141
intra-assignment timing controls, 136–139
procedure
always construct, 143
function, 143
initial construct, 143
task, 143
process, 158
programmable logic arrays
list of system tasks, 303
logic types, 304
personality
declaration, 304
formats, 304–307
loading, 304
types, 303–307
propagation delay
for gates and nets, 101
Pu (pull drive in strength format), 284
pull, 28
pull0, 77, 360
pull1, 76, 360
pulldown, 76
pulldown source, 86
pullup, 76
pullup source, 86
pulse
negative
detection, 232
pulse control, 425, 448
detailed capabilities, 230
pulse filtering
on-event versus on-detect, 230
pulse limit value, 228
global control of, 230
SDF annotation, 230
specify block control, 229
pulsere_flag, 425, 448
pure transport delays, 451

Q

qualified paths, 214–218
edge-sensitive, 214–218
level-sensitive, 215–220
queue management, 307–308
$q_add, 307
$q_exam, 307–308
$q_full, 307–308
$q_initialize, 307
$q_remove, 307
status parameters, 308
queueing models, 307

R

r (in state table), 108
race condition, 138
race conditions, 160
random access memory(RAM)
modeled by register arrays, 35
random number generators
probabilistic distribution functions, 311
range specification, 77
rcmos, 83, 85
reading a character at a time, 290
reading a line at a time, 290
reading binary data, 293
reading formatted data, 291
read-only memory(ROM)
modeled by register arrays, 35
real constant numbers, 12
real declarations, 33
real number constants, 33
real numbers, 32
and operators, 33
conversion to integers, 12, 33
format specifications used with, 281
in port connections, 178
operators with real number operands, 42–43
real variable data types, 33
realtime
variables, 32
realtime declarations, 33
recursive, see frames, 404
reducing pessimism, 128
reduction operators, 51–52
&, 42
~&, 42
inclusive OR, 42
unary AND, 42
unary NAND, 42
unary NOR, 42
XNOR, 42
XOR, 42
reentrant, see frames, 404
reg arrays, 34
reg declaration, 23
registers
and level-sensitive sequential UDPs, 110
notifier, 259
used in procedural assignments, 72

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

regs, 32
relational operators, 42, 48
<, 48
<=, 48
>, 48
>=, 48
precedence, 48
release, 161
repeat event control, 137–139
repeat loop, 130
replication
operator, 42
restrictions on data types
in continuous assignments, 68, 179
in procedural assignments, 68, 72, 117
when connecting ports, 179
right-hand index, 77
rise delay, 101–102
rnmos, 83–84
rpmos, 83–84
rtran, 84
rtranif0, 84
rtranif1, 84
rules
for delays controling the assignment, 71
for describing module paths, 220
for expression bit lengths, 62
for expression types, 65
for instance connections, 78
net type resolution, 180
to use the $dumpports, 339

S

s (in string display format), 285
s_vpi_delay structure, 424
s_vpi_time structure, 424
scalared, 24
scalars
compared to vectors, 24
scalar nets and driving strength of continuous assignment, 71
scheduling semantics, 158
scientific notation, 12
scope
and hierarchical names, 191
rules, 195–196
SDF
INTERCONNECT construct, 273
interconnect delay annotation, 273
multiple annotations, 274
pulse limit annotation, 275
to Verilog delay value mapping, 276
SDF annotation
down-hierarchy annotation, 274
hierarchically overlapping annotations, 274
NETDELAY construct, 273
of interconnect delays, 273
of specparams, 272
PATHPULSE, 275
PATHPULSEPERCENT, 275
PORT construct, 273
up-hierarchy annotations, 274
SDF annotator, 269
SDF constructs
mapping to Verilog, 269
SDF delay constructs
mapping to Verilog declarations, 269
SDF files
backannotation, 269
SDF timing check constructs
mapping to Verilog, 271
seed, 312
self-determined expression, 62
sequential block, 116
sequential block statement, 140
finish time, 142
start time, 142
sequential UDP initialization, 111–112
sequential UDPs
input and output fields in state table, 108
set of values (0, 1, x, z), 21
shift operators, 42, 53
<<, 53
<<<, 53
>>, 53
>>>, 53
short-circuiting, 45
showcancelled behavior, 232
signed expressions, 64
handling ’X’ and ’Z’, 66
signed integers, 10
simple decimal number, 10
simple state-dependent paths, 216
simulating module path delays
when driving wired logic, 226–227
simulation
going back with incremental restart, 515
simulation cycle, 159
simulation reference model, 159
simulation time, 158
single-pass use model
elaboration-time compiling, 206
precompiling, 205
size constant, 10
size of displayed data, 281–282
sized numbers, 10
Sm (small capacitor in strength format), 284
small, 25, 28

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

source
pulldown, 86
pullup, 86
specify, 39, 211
specify block, 211–236
specify block system tasks
$hold, 242
$period, 256
$recovery, 246
$setuphold, 243
$skew, 249
$timeskew, 250
$width, 255–256
specify parameters, 38–39
as run time constant in specify block, 212
specifying the time unit of delays entered interactively, 300
specifying transition delays on module paths, 222–224
x transitions, 224
specparam, 38–39, 272
St (strong drive in strength format), 284
standard output, 287
start time
in parallel block statements, 142
in sequential block statements, 142
state dependent path delays, 215–220
stochastic analysis, 311–312
probabilistic distribution functions, 311–312
queue management, 307–308
stop, 302
strength, 76–77
ambiguous, 89–99
classifications, 89
and MOS gates, 100
and scalar net variables, 21
charge storage, 88
driving, 88
gates that accept specifications, 76
of combined signals, 88–99
on trireg nets, 28
range of possible values, 90
reduction by nonresistive devices, 100
reduction by resistive devices, 100
reduction table, 100
scale of strengths, 88
specification, 87
supply net, 101
tri0, 100
tri1, 100
trireg, 100
strength display format, 283–285
high impedance, 284
large capacitor, 284
logic value 0,1,H,L,X,Z, 283
medium capacitor, 284
pull drive, 284
small capacitor, 284
strong drive, 284
supply drive, 284
weak drive, 284
strengths, 25
of net types, 100
strings, 12–14, 58–279
definition, 12
display format, 280, 285
in vector variables, 59
manipulation, 13
operations, 59
padding, 13
special characters, 13
value padding, 59–60
variable declaration, 13
strobed monitoring, 285
strong, 28
strong0, 77
strong1, 76
structured procedure, 143–144
always construct, 143
function, 143
initial construct, 143
task, 143
Su (supply drive in strength format), 284
supply, 28
supply net strength, 101
supply nets, 32
supply0, 77
supply1, 76
switch processing, 161
switches
MOS, 83–84
synchronous arrays, 303–307
system functions, 277–312
system task/function arguments, 368
system task/function name, 374
system tasks, 277–312
for continuous monitoring, 286
for displaying information, 278–285
for interrupting the simulator, 302
for processing stimulus patterns faster, 512
for showing number of drivers, 512
for writing formatted output to files, 286–289
generating a checkpoint in the value change dump file, 328
limiting the size of the value change dump file, 328
reading the value change dump file during a simulation, 328

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

resuming the dump into the value change dump file, 327
showing the timescale of a module, 299
specifying how %t reports time information, 300–302
specifying the name of the value change dump file, 325
specifying the variables to be dumped in the value change dump file, 326
stopping the dump into the value change dump file, 327
System tasks and functions, 15
system tasks and functions, 277–312

T

t (timescale format), 281
tab character, 14
task/function arguments, 368
task/function name, 374
task/function routines
history, 366
task-enabling statement, 147
tasks, 145–156, 162
and hierarchical names, 191
and scope, 195
as structured procedures, 143
definition, 143
disabling within a nested chain, 150
passing parameters, 147–148
purpose, 145
text macro substitutions, 350–352
and `define, 350
definition, 350
redefinition, 352
with arguments, 350
text output
vpi_mcd_close(), 440
vpi_mcd_name(), 441
vpi_mcd_open(), 442
vpi_mcd_printf(), 443
vpi_printf(), 444
tf_synchrnize(), 159
tfargs, 368
time
arithmetic operations performed on time variables, 33
variables, 32
time precision, 358
time unit, 358
timing checks, 237–268
$hold, 242
$period, 256
$recovery, 246
$recrem, 247
$removal, 245
$setup, 241
$setuphold, 243
$skew, 249
$timeskew, 250
$width, 255–256
negative, 266
conditions, 263
notifiers, 264
using a stability window, 240
vector signals, 266
timing checks for clock and control signals, 248
top-level module, 165
tran, 84
tranif0, 84
tranif1, 84
transistors, 84
transitions
01, 111
unspecified, 110
transport delays, 451
tree structure
of hierarchical names, 191
tri nets, 26
tri0 (net type), 100
tri1 (net type), 100
triand, 27
trior, 27
trireg
and charge storage strength, 88
turn-off delay, 102
types of nets
supply nets, 32
tri nets, 26
tri0, 31, 100
tri1, 31, 100
triand, 27
trior, 27
trireg, 28, 100, 283
wire, 26
wired AND, 27
wired logic, 98
wired nets, 27
wired OR, 27

U

UDP port declarations, 107
UDPs, 105–115
- in state table, 108
(??) in state table, 108
(01) in state table, 108
(0x) in state table, 108
(1x) in state table, 108
(vw) in state table, 108

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

(x1) in state table, 108
* in state table, 108
? in state table, 108
0 in state table, 108
1 in state table, 108
b in state table, 108
combinational UDPs, 109
definition, 105–107
edge-sensitive UDPs, 110
f in state table, 108
instances, 113–114
level-sensitive dominance, 115
level-sensitive sequential UDPs, 110
mixing level- and edge-sensitive descriptions, 114–115
n in state table, 108
p in state table, 108
ports, 107
r in state table, 108
state table, 107
summary of symbols in state table, 108
x in state table, 108
unary arithmetic operators, 46
unary operators, 8
!, 49
<<, 53
>>, 53
unconnected port, 166
undescore character, 11
unexpanded object, 24
unknown logic value
and numbers, 10
display formats, 282–284
effect in different bases, 10
in state table, 108, 111
symbolic representation, 21
unsigned integers, 10
unsigned number, 10
unspecified transitions, 110
upwards name referencing, 193
User-defined primitives, 105
user-defined primitives (UDPs), 105
user-defined system tasks and functions, 374
name overloading, 367
names, 367, 374
types, 367

V

value change dump file, 325–348
creating, 325–329
creating the extended file, 338
extended VCD node information, 344
format, 329–337
formats of variable values, 331–332
general rules for extended VCD system tasks, 341
generating a checkpoint, 328, 340
keyword commands
$comment, 332
$date, 332
$dumpall, 335
$dumpoff, 336, 341
$enddefinitions, 333
$scope, 333
$timescale, 334
$upscope, 334
$var, 334
$version, 335
limiting the size, 328
limiting the size of the dump file, 340
reading the dump file during simulation, 341
reading the value change dump file during a simulation, 328
resuming the dump, 327
rules to conflicts, 347
specifying the dump file name and the ports to be dumped, 338
specifying the name, 325
specifying the variables to be dumped, 326
stopping and resuming the dump, 339
stopping the dump, 327
value changes, 346
value set (0, 1, x, z), 21
values
of combined signals, 88–99
variables, 23–24
VCD file
extended, 342
vectored, 24
vectors, 24
and vector net expansion, 24
vlog_startup_routines array, 463
VPI data model diagrams
active time format, 414
assignments, 410
attributes, 415
case statement, 412
continuous assignments, 406
delay controls, 410
delay terminals, 405
event controls, 410
expressions, 408
expressions, simple, 407
for loops, 411
forever loops, 411
frames, 404
function calls, 403
functions, 402

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

if statement, 412
instance arrays, 388
inter-module paths, 401
IO declarations, 389
iterator, 416
memories, 396
module paths, 401
modules, 387
named events, 397, 409
net drivers and loads, 405
nets, 391
object range, 396
parameters, 398
path term, 401
ports, 390
primitives, 399
procedural assign statement, 413
procedural blocks, 409
procedural deassign statement, 413
procedural disable statement, 413
procedural force statement, 413
procedural release statement, 413
process, 409
reg drivers and loads, 406
regs, 393
repeat controls, 411
repeat loops, 411
scopes, 389
simple expressions, 407
specparams, 398
statements, 409
task calls, 403
tasks, 402
timing check, 402
UDPs, 400
variables, 395
wait control, 411
while loops, 411
VPI mechanism, 375
VPI routines
callback overview, 375
error handling, 376
history, 366
key to data model diagrams, 383
lsited by functional groups, 381
object access overview, 376
object classifications, 377
object types, 379
traversing expressions, 377
vpi_chk_error(), 418
vpi_compare_objects(), 420
vpi_control(), 420
vpi_flush(), 421
vpi_free_object(), 421
vpi_get(), 422
vpi_get_cb_info(), 422
vpi_get_data(), 423
vpi_get_delays(), 424
vpi_get_str(), 426
vpi_get_systf_info(), 427
vpi_get_time(), 428
vpi_get_userdata(), 429
vpi_get_value(), 429
vpi_get_vlog_info(), 435
vpi_handle(), 436
vpi_handle_by_index(), 437
vpi_handle_by_multi_index(), 438
vpi_handle_by_name(), 438
vpi_handle_multi(), 439
vpi_iterate(), 439
vpi_mcd_close(), 440
vpi_mcd_flush(), 441
vpi_mcd_name(), 441
vpi_mcd_open(), 442
vpi_mcd_printf(), 443
vpi_mcd_vprintf(), 444
vpi_printf(), 444
vpi_put_data(), 445
vpi_put_delays(), 447
vpi_put_userdata(), 450
vpi_put_value(), 450
vpi_register_cb(), 159, 453
vpi_register_systf(), 461
vpi_remove_cb(), 465
vpi_scan(), 465
vpi_vprintf(), 466
vpiCancelEvent, 451
vpiFile, 380
vpiForceFlag, 451
vpiHandle, 378
vpiInertialDelay, 451
vpiInterModPath, 439
vpiIntFunc, 462
vpiIterator, 439
vpiLineNo, 380
vpiNoDelay, 451
vpiPureTransportDelay, 451
vpiRealFunc, 462
vpiReleaseFlag, 451
vpiReturnEvent, 451
vpiScaledRealTime, 453
vpiSchedEvent, 451
vpiScheduled, 451
vpiSizedFunc, 462
vpiSizedSignedFunc, 462
vpiSysFunction, 462
vpiSysTask, 462
vpiTimeFunc, 462

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale

vpiTimeUnit, 422
vpiTransportDelay, 451
vpiType, 379

W

wait statement
  as level-sensitive event control, 136
  to advance simulation time, 132
wand, 27
We (weak drive in strength format), 284
weak, 28
weak0, 77
weak1, 76
while loop, 130
white space, 8
wired AND configurations, 27
wired logic nets
  wand, 98
  wired AND configurations, 27
  wired OR configurations, 27
  wor, 98
wired OR configurations, 27
wires, 26
wor, 27
word (reg in array), 35
writing formatted output to files, 286–289

X

X
  as display format for unknown logic value, 282
  unknown logic value in strength format, 283
x
  as display format for unknown logic value, 282
  in state table, 108
  unknown logic value, 21
xnor gate, 80–81
xor gate, 80–81

Z

Z
  as display format for high impedance state, 282
  high-impedance state in strength format, 283
z
  as display format for high impedance state, 282
  high-impedance state, 21

Copyright © 2006 IEEE. All rights reserved.

Copyright The Institute of Electrical and Electronics Engineers, Inc.
Provided by IHS under license with IEEE
No reproduction or networking permitted without license from IHS
Not for Resale