

IEEE Std 1364-2001

(Revision of
IEEE Std 1364-1995)

IEEE Standard Verilog[®] Hardware Description Language

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee



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

28 September 2001

Print: SH94921
PDF: SS94921

IEEE Standard Verilog[®] Hardware Description Language

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 17 March 2001

IEEE-SA Standards Board

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 HDL, Verilog PLI, Verilog[®]

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

Copyright © 2001 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 28 September 2001. Printed in the United States of America.

Print: ISBN 0-7381-2826-0 SH94921
PDF: ISBN 0-7381-2827-9 SS94921

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

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

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

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

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

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

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

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

The IEEE and its designees are the sole entities that may authorize the use of IEEE-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein.

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; (978) 750-8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

About IEEE Std 1364-2001 Version C and the Errata

During the past two years, the IEEE 1364 Working Group's Errata Task Force has thoroughly reviewed the Standard and has identified and corrected a number of production and editorial errors that crept in between balloting and printing of the Standard. IEEE Std 1364-2001 Version C incorporates all of these corrections.

In addition, during the IEEE 1364 Working Group's review of the Standard, the Working Group and its Errata Task Force identified other areas where the standard had logical inconsistencies which were not the result of production problems. The Working Group developed an Errata document that identifies these and specifies the Working Group's statement as to the correct interpretation of the Standard.

Participants—Version C and Errata

At the time IEEE Std 1364-2001 Version C and the Errata were completed, the IEEE 1364 Working Group had the following membership:

Michael T. Y. (Mac) McNamara, *Chair*
Shalom Bresticker, *Editor*
Stefen Boyd, *Web Master*

Kurt Baty
Dennis Brophy
Clifford E. Cummings
Charles Dawson
Tom Fitzpatrick
Krishna Garlapati
Keith Gover
Ennis Hawk

Richard Ho
Atsushi Kasuya
Jay Lawrence
Andrew Lynch
James A. Markevitch
Dennis Marsa
Francoise Martinolle
Mehdi Mohtashemi

Anders Nordstrom
Karen Pieper
Brad Pierce
Steven Sharp
Alec Stanculescu
Stuart Sutherland
Chong Guan Tan
Gordon Vreugdenhil

The Errata Task Force had the following membership:

Karen Pieper, *Chair*
Stefen Boyd, *Vice Chair*

Kurt Baty
Shalom Bresticker
Dennis Brophy
Clifford E. Cummings
Charles Dawson
Ted Elkind
Tom Fitzpatrick
Jay Lawrence

Andrew Lynch
James A. Markevitch
Dennis Marsa
Francoise Martinolle
Michael T. Y. (Mac) McNamara
Elliot Mednick
Don Mills

Mehdi Mohtashemi
Anders Nordstrom
Brad Pierce
David Roberts
Steven Sharp
David Smith
Stuart Sutherland
Gordon Vreugdenhil

The Behavioral Task Force had the following membership:

Steven Sharp, *Chair*

Kurt Baty
Stefen Boyd
Dennis Brophy
Clifford E. Cummings
Tom Fitzpatrick
Ennis Hawk

Atsushi Kasuya
Jay Lawrence
Francoise Martinolle
Michael T. Y. (Mac) McNamara
Don Mills

Mehdi Mohtashemi
Karen Pieper
Brad Pierce
Alec Stanculescu
Stuart Sutherland
Gordon Vreugdenhil

The PLI Task Force had the following membership:

Charles Dawson, *Co-Chair*
Stuart Sutherland, *Co-Chair*

Steven Dovich
Dennis Marsa

Francoise Martinolle

Nisa Parikh
David Roberts

Introduction

(This introduction is not part of IEEE Std 1364-2001, IEEE Standard Verilog[®] Hardware Description Language.)

The Verilog[®] Hardware Description Language (Verilog 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 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 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 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 1364 Working Group started looking for feedback from 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.

Objective of the IEEE Std 1364-2001 effort

The starting point for the IEEE 1364 Working Group for this standard was the feedback received from the IEEE Std 1364-1995 users worldwide. It was clear from the feedback that users wanted improvements in all aspects of the language. Users at the higher levels wanted to expand and improve the language at the RTL and behavioral levels, while users at the lower levels wanted improved capability for ASIC designs and signoff. It was for this reason that the 1364 Working Group was organized into three task forces: Behavioral, ASIC, and PLI.

The clear directive from the users for these three task forces was to start by solving some of the following problems:

- Consolidate existing IEEE Std 1364-1995
- Verilog Generate statement
- Multi-dimensional arrays
- Enhanced Verilog file I/O
- Re-entrant tasks
- Standardize Verilog configurations
- Enhance timing representation
- Enhance the VPI routines

Achievements

Over a period of four years the 1364 Verilog Standards Group (VSG) has produced five drafts of the LRM. The three task forces went through the IEEE Std 1364-1995 LRM very thoroughly and in the process of consolidating the existing LRM have been able to provide nearly three hundred clarifications and errata for the Behavioral, ASIC, and PLI sections. In addition, the VSG has also been able to agree on all the enhancements that were requested (including the ones stated above).

Three new sections have been added. Clause 13, “Configuring the contents of a design,” deals with configuration management and has been added 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 session. Clause 15, “Timing checks,” has been broken out of Clause 17, “System tasks and functions,” and details more fully how timing checks are used in specify blocks. Clause 16, “Backannotation using the Standard Delay Format (SDF),” addresses using back annotation (IEEE Std 1497-1999) within IEEE Std 1364-2001.

Extreme care has been taken to enhance the VPI routines to handle all the enhancements in the Behavioral and other areas of the LRM. Minimum work has been done on the PLI routines and most of the work has been concentrated on the VPI routines. Some of the enhancements in the VPI are the save and restart, simulation control, work area access, error handling, assign/deassign and support for array of instances, generate, and file I/O.

Work on this standard would not have been possible without funding from the CAS society of the IEEE and Open Verilog International.

The IEEE Std 1364-2001 Verilog Standards Group organization

Many individuals from many different organizations participated directly or indirectly in the standardization process. The main body of the IEEE Std 1364-2001 working group is located in the United States, with a subgroup in Japan (EIAJ/1364HDL).

The members of the IEEE Std 1364-2001 working group had voting privileges and all motions had to be approved by this group to be implemented. The three task forces focused on their specific areas and their recommendations were eventually voted on by the IEEE Std 1364-2001 working group.

At the time this document was approved, the IEEE Std 1364-2001 working group had the following membership:

Maqsoodul (Maq) Mannan, *Chair*
Kasumi Hamaguchi, *Vice Chair (Japan)*
Alec G. Stanculescu, *Vice Chair (USA)*
Lynn A. Horobin, *Secretary*
Yatin Trivedi, *Technical Editor*

The Behavioral Task Force consisted of the following members:

Clifford E. Cummings, *Leader*

Kurt Baty
Stefen Boyd
Shalom Bresticker
Tom Fitzpatrick

Adam Krolnik
James A. Markevitch
Michael McNamara
Anders Nordstrom

Karen Pieper
Steven Sharp
Chris Spear
Stuart Sutherland

The ASIC Task Force consisted of the following members:

Steve Wadsworth, *Leader*

Leigh Brady
Paul Colwill
Tom Dewey

Ted Elkind
Naveen Gupta
Prabhakaran Krishnamurthy

Marek Ryniejski
Lukasz Senator

The PLI Task Force consisted of the following members:

Andrew T. Lynch, *Leader*

Stuart Sutherland, *Co-Leader and Editor*

Deborah J. Dalio
Charles Dawson

Steve Meyer

Girish S. Rao
David Roberts

The IEEE 1364 Japan subgroup (EIAJ/1364HDL) consisted of the following members:

Kasumi Hamaguchi, *Vice Chair (Japan)*

Yokozeeki Atsushi
Yasuaki Hatta

Makoto Makino
Takashima Mitsuya
Tatsuro Nakamura

Hiroaki Nishi
Tsutomu Someya

The following members of the balloting committee voted on this standard:

Guy Adam	Masato Ikeda	Ryosuke Okuda
Shigehiro Asano	Mitsuaki Ishikawa	Yoichi Onishi
Peter J. Ashenden	Neil G. Jacobson	Uma P. Parvathy
Victor Berman	Richard O. Jones	William R. Paulsen
J Bhasker	Osamu Karatsu	Karen L. Pieper
Stefan Boyd	Jake Karrfalt	Girish S. Rao
Dennis B. Brophy	Masayuki Katakura	Jaideep Roy
Keith Chow	Kaoru Kawamura	Francesco Sforza
Clifford E. Cummings	Masamichi Kawarabayashi	Charles F. Shelor
Brian A. Dalio	Satoshi Kojima	Chris Spear
Timothy R. Davis	Masuyoshi Kurokawa	Alec G. Stanculescu
Charles Dawson	Gunther Lehmann	Steve Start
Douglas D. Dunlop	Andrew T. Lynch	Stuart Sutherland
Ted Elkind	Serge Maginot	Masahiko Toyonaga
Joerg-Oliver Fischer-Binder	Maqsoodul Mannan	Yatin K. Trivedi
Peter Flake	James A. Markevitch	Cary Ussery
Robert A. Flatt	Francoise Martinolle	Steven D. Wadsworth
Masahiro Fukui	Yoshio Masubuchi	Sui-Ki Wan
Kenji Goto	Paul J. Menchini	Ronald Waxman
Naveen Gupta	Hiroshi Mizuno	John M. Williams
Andrew Guyler	Egbert Molenkamp	John Willis
Yoshiaki Hagiwara	John T. Montague	Takashi Yamada
Anne C. Harris	Akira Motohara	Lun Ye
Lynn A. Horobin	Hiroaki Nishi	Hirokazu Yonezawa
ChiLai Huang	Anders Nordstrom	Tetsuo Yutani
Takahiro Ichinomiya		Mark Zwolinski

When the IEEE-SA Standards Board approved this standard on 17 March 2001, it had the following membership:

Donald N. Heirman, *Chair*

James T. Carlo, *Vice Chair*

Judith Gorman, *Secretary*

Satish K. Aggarwal	James H. Gurney	James W. Moore
Mark D. Bowman	Richard J. Holleman	Robert F. Munzner
Gary R. Engmann	Lowell G. Johnson	Ronald C. Petersen
Harold E. Epstein	Robert J. Kennelly	Gerald H. Peterson
H. Landis Floyd	Joseph L. Koepfinger*	John B. Posey
Jay Forster*	Peter H. Lips	Gary S. Robinson
Howard M. Frazier	L. Bruce McClung	Akio Tojo
Ruben D. Garzon	Daleep C. Mohla	Donald W. Zipse

*Member Emeritus

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

Alan Cookson, *NIST Representative*
Donald R. Volzka, *TAB Representative*

Andrew Ickowicz
IEEE Standards Project Editor

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

Contents

1.	Overview.....	1
1.1	Objectives of this standard.....	1
1.2	Conventions used in this standard.....	1
1.3	Syntactic description.....	2
1.4	Contents of this standard.....	2
1.5	Header file listings.....	4
1.6	Examples.....	5
1.7	Prerequisites.....	5
2.	Lexical conventions.....	6
2.1	Lexical tokens.....	6
2.2	White space.....	6
2.3	Comments.....	6
2.4	Operators.....	6
2.5	Numbers.....	6
2.5.1	Integer constants.....	7
2.5.2	Real constants.....	10
2.5.3	Conversion.....	10
2.6	Strings.....	10
2.6.1	String variable declaration.....	11
2.6.2	String manipulation.....	11
2.6.3	Special characters in strings.....	11
2.7	Identifiers, keywords, and system names.....	12
2.7.1	Escaped identifiers.....	12
2.7.2	Generated identifiers.....	13
2.7.3	Keywords.....	13
2.7.4	System tasks and functions.....	13
2.7.5	Compiler directives.....	14
2.8	Attributes.....	14
2.8.1	Examples.....	15
2.8.2	Syntax.....	16
3.	Data types.....	20
3.1	Value set.....	20
3.2	Nets and variables.....	20
3.2.1	Net declarations.....	20
3.2.2	Variable declarations.....	22
3.3	Vectors.....	23
3.3.1	Specifying vectors.....	23
3.3.2	Vector net accessibility.....	24
3.4	Strengths.....	24
3.4.1	Charge strength.....	24
3.4.2	Drive strength.....	24
3.5	Implicit declarations.....	25
3.6	Net initialization.....	25
3.7	Net types.....	25
3.7.1	Wire and tri nets.....	25
3.7.2	Wired nets.....	26

3.7.3	Trireg net.....	26
3.7.4	Tri0 and tri1 nets.....	30
3.7.5	Supply nets.....	31
3.8	Regs.....	31
3.9	Integers, reals, times, and realsimes.....	31
3.9.1	Operators and real numbers.....	32
3.9.2	Conversion.....	32
3.10	Arrays.....	33
3.10.1	Net arrays.....	33
3.10.2	reg and variable arrays.....	33
3.10.3	Memories.....	33
3.11	Parameters.....	34
3.11.1	Module parameters.....	35
3.11.2	Local parameters - localparam.....	36
3.11.3	Specify parameters.....	37
3.12	Name spaces.....	38
4.	Expressions.....	40
4.1	Operators.....	40
4.1.1	Operators with real operands.....	41
4.1.2	Operator precedence.....	42
4.1.3	Using integer numbers in expressions.....	43
4.1.4	Expression evaluation order.....	43
4.1.5	Arithmetic operators.....	44
4.1.6	Arithmetic expressions with regs and integers.....	45
4.1.7	Relational operators.....	46
4.1.8	Equality operators.....	46
4.1.9	Logical operators.....	47
4.1.10	Bit-wise operators.....	47
4.1.11	Reduction operators.....	48
4.1.12	Shift operators.....	49
4.1.13	Conditional operator.....	50
4.1.14	Concatenations.....	51
4.1.15	Event or.....	52
4.2	Operands.....	52
4.2.1	Vector bit-select and part-select addressing.....	52
4.2.2	Array and memory addressing.....	54
4.2.3	Strings.....	55
4.3	Minimum, typical, and maximum delay expressions.....	57
4.4	Expression bit lengths.....	59
4.4.1	Rules for expression bit lengths.....	59
4.4.2	An example of an expression bit-length problem.....	60
4.4.3	Example of self-determined expressions.....	61
4.5	Signed expressions.....	62
4.5.1	Rules for expression types.....	62
4.5.2	Steps for evaluating an expression.....	62
4.5.3	Steps for evaluating an assignment.....	63
4.5.4	Handling X and Z in signed expressions.....	63
5.	Scheduling semantics.....	64
5.1	Execution of a model.....	64
5.2	Event simulation.....	64

5.3	The stratified event queue.....	64
5.4	The Verilog simulation reference model	65
5.4.1	Determinism.....	66
5.4.2	Nondeterminism.....	66
5.5	Race conditions	66
5.6	Scheduling implication of assignments	66
5.6.1	Continuous assignment	67
5.6.2	Procedural continuous assignment.....	67
5.6.3	Blocking assignment.....	67
5.6.4	Nonblocking assignment.....	67
5.6.5	Switch (transistor) processing.....	67
5.6.6	Port connections.....	68
5.6.7	Functions and tasks	68
6.	Assignments.....	69
6.1	Continuous assignments	69
6.1.1	The net declaration assignment.....	70
6.1.2	The continuous assignment statement	70
6.1.3	Delays	72
6.1.4	Strength.....	72
6.2	Procedural assignments.....	73
6.2.1	Variable declaration assignment	73
6.2.2	Variable declaration syntax.....	74
7.	Gate and switch level modeling.....	75
7.1	Gate and switch declaration syntax.....	75
7.1.1	The gate type specification	77
7.1.2	The drive strength specification.....	77
7.1.3	The delay specification	78
7.1.4	The primitive instance identifier.....	78
7.1.5	The range specification	78
7.1.6	Primitive instance connection list	79
7.2	and, nand, nor, or, xor, and xnor gates.....	81
7.3	buf and not gates	82
7.4	bufif1, bufif0, notif1, and notif0 gates.....	83
7.5	MOS switches	84
7.6	Bidirectional pass switches	86
7.7	CMOS switches	86
7.8	pullup and pulldown sources	87
7.9	Logic strength modeling	88
7.10	Strengths and values of combined signals	89
7.10.1	Combined signals of unambiguous strength.....	89
7.10.2	Ambiguous strengths: sources and combinations	90
7.10.3	Ambiguous strength signals and unambiguous signals	95
7.10.4	Wired logic net types	99
7.11	Strength reduction by nonresistive devices.....	102
7.12	Strength reduction by resistive devices.....	102
7.13	Strengths of net types.....	102
7.13.1	tri0 and tri1 net strengths	102
7.13.2	triereg strength	102
7.13.3	supply0 and supply1 net strengths	102

7.14	Gate and net delays	103
7.14.1	min:typ:max delays	104
7.14.2	trireg net charge decay	105
8.	User-defined primitives (UDPs)	107
8.1	UDP definition	107
8.1.1	UDP header	109
8.1.2	UDP port declarations	109
8.1.3	Sequential UDP initial statement	109
8.1.4	UDP state table	109
8.1.5	Z values in UDP	110
8.1.6	Summary of symbols	110
8.2	Combinational UDPs	111
8.3	Level-sensitive sequential UDPs	112
8.4	Edge-sensitive sequential UDPs	112
8.5	Sequential UDP initialization	113
8.6	UDP instances	115
8.7	Mixing level-sensitive and edge-sensitive descriptions	116
8.8	Level-sensitive dominance	117
9.	Behavioral modeling	118
9.1	Behavioral model overview	118
9.2	Procedural assignments	119
9.2.1	Blocking procedural assignments	119
9.2.2	The nonblocking procedural assignment	121
9.3	Procedural continuous assignments	124
9.3.1	The assign and deassign procedural statements	125
9.3.2	The force and release procedural statements	126
9.4	Conditional statement	127
9.4.1	If-else-if construct	128
9.5	Case statement	130
9.5.1	Case statement with don't-cares	133
9.5.2	Constant expression in case statement	133
9.6	Looping statements	134
9.7	Procedural timing controls	136
9.7.1	Delay control	137
9.7.2	Event control	138
9.7.3	Named events	138
9.7.4	Event or operator	139
9.7.5	Implicit event_expression list	140
9.7.6	Level-sensitive event control	141
9.7.7	Intra-assignment timing controls	142
9.8	Block statements	146
9.8.1	Sequential blocks	146
9.8.2	Parallel blocks	147
9.8.3	Block names	148
9.8.4	Start and finish times	148
9.9	Structured procedures	149
9.9.1	Initial construct	150
9.9.2	Always construct	150

10.	Tasks and functions.....	152
10.1	Distinctions between tasks and functions	152
10.2	Tasks and task enabling	152
10.2.1	Task declarations	153
10.2.2	Task enabling and argument passing	154
10.2.3	Task memory usage and concurrent activation.....	156
10.3	Functions and function calling.....	157
10.3.1	Function declarations	158
10.3.2	Returning a value from a function	159
10.3.3	Calling a function.....	160
10.3.4	Function rules.....	160
10.3.5	Use of constant functions.....	161
11.	Disabling of named blocks and tasks.....	163
12.	Hierarchical structures	166
12.1	Modules.....	166
12.1.1	Top-level modules	168
12.1.2	Module instantiation	168
12.1.3	Generated instantiation	170
12.2	Overriding module parameter values.....	180
12.2.1	defparam statement	182
12.2.2	Module instance parameter value assignment	183
12.2.3	Parameter dependence	185
12.3	Ports	185
12.3.1	Port definition	185
12.3.2	List of ports	185
12.3.3	Port declarations.....	186
12.3.4	List of ports declarations.....	188
12.3.5	Connecting module instance ports by ordered list.....	188
12.3.6	Connecting module instance ports by name	189
12.3.7	Real numbers in port connections.....	190
12.3.8	Connecting dissimilar ports	191
12.3.9	Port connection rules.....	191
12.3.10	Net types resulting from dissimilar port connections	192
12.3.11	Connecting signed values via ports.....	193
12.4	Hierarchical names	193
12.5	Upwards name referencing	196
12.6	Scope rules	198
13.	Configuring the contents of a design	200
13.1	Introduction.....	200
13.1.1	Library notation	200
13.1.2	Basic configuration elements.....	201
13.2	Libraries	201
13.2.1	Specifying libraries - the library map file	201
13.2.2	Using multiple library mapping files	203
13.2.3	Mapping source files to libraries.....	203
13.3	Configurations.....	203
13.3.1	Basic configuration syntax.....	203
13.3.2	Hierarchical configurations.....	206

13.4	Using libraries and configs	207
13.4.1	Precompiling in a single-pass use-model.....	207
13.4.2	Elaboration-time compiling in a single-pass use-model	207
13.4.3	Precompiling using a separate compilation tool	207
13.4.4	Command line considerations.....	207
13.5	Configuration examples	208
13.5.1	Default configuration from library map file	208
13.5.2	Using the default clause	208
13.5.3	Using the cell clause	209
13.5.4	Using the instance clause.....	209
13.5.5	Using a hierarchical config	209
13.6	Displaying library binding information	210
13.7	Library mapping examples	210
13.7.1	Using the command line to control library searching.....	210
13.7.2	File path specification examples.....	210
13.7.3	Resolving multiple path specifications	211
14.	Specify blocks.....	212
14.1	Specify block declaration.....	212
14.2	Module path declarations.....	213
14.2.1	Module path restrictions	214
14.2.2	Simple module paths.....	214
14.2.3	Edge-sensitive paths.....	215
14.2.4	State-dependent paths	216
14.2.5	Full connection and parallel connection paths.....	220
14.2.6	Declaring multiple module paths in a single statement	221
14.2.7	Module path polarity.....	222
14.3	Assigning delays to module paths.....	223
14.3.1	Specifying transition delays on module paths	224
14.3.2	Specifying x transition delays.....	225
14.3.3	Delay selection.....	226
14.4	Mixing module path delays and distributed delays.....	227
14.5	Driving wired logic.....	228
14.6	Detailed control of pulse filtering behavior	229
14.6.1	Specify block control of pulse limit values.....	230
14.6.2	Global control of pulse limit values.....	231
14.6.3	SDF annotation of pulse limit values.....	231
14.6.4	Detailed pulse control capabilities	232
15.	Timing checks.....	238
15.1	Overview.....	238
15.2	Timing checks using a stability window.....	241
15.2.1	\$setup	242
15.2.2	\$hold	242
15.2.3	\$setuphold	243
15.2.4	\$removal	245
15.2.5	\$recovery.....	246
15.2.6	\$crem	247
15.3	Timing checks for clock and control signals	249
15.3.1	\$skew	250
15.3.2	\$timeskew	251
15.3.3	\$fullskew.....	253

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	260
15.5.1	Requirements for accurate simulation	262
15.5.2	Conditions in negative timing checks	264
15.5.3	Notifiers in negative timing checks	266
15.5.4	Option behavior	266
15.6	Enabling timing checks with conditioned events.....	266
15.7	Vector signals in timing checks	267
15.8	Negative timing checks.....	268
16.	Backannotation using the Standard Delay Format (SDF).....	270
16.1	The SDF annotator	270
16.2	Mapping of SDF constructs to Verilog	270
16.2.1	Mapping of SDF delay constructs to Verilog declarations	270
16.2.2	Mapping of SDF timing check constructs to Verilog	272
16.2.3	SDF annotation of specparams	273
16.2.4	SDF annotation of interconnect delays	274
16.3	Multiple annotations	275
16.4	Multiple SDF files	276
16.5	Pulse limit annotation	276
16.6	SDF to Verilog delay value mapping.....	277
17.	System tasks and functions	278
17.1	Display system tasks	278
17.1.1	The display and write tasks	279
17.1.2	Strobed monitoring	286
17.1.3	Continuous monitoring	287
17.2	File input-output system tasks and functions.....	287
17.2.1	Opening and closing files.....	287
17.2.2	File output system tasks	289
17.2.3	Formatting data to a string	290
17.2.4	Reading data from a file.....	291
17.2.5	File positioning	295
17.2.6	Flushing output	295
17.2.7	I/O error status	295
17.2.8	Loading memory data from a file	296
17.2.9	Loading timing data from an SDF file	297
17.3	Timescale system tasks	298
17.3.1	\$prnttimescale	298
17.3.2	\$timeformat.....	299
17.4	Simulation control system tasks	302
17.4.1	\$finish	302
17.4.2	\$stop	302
17.5	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.....	308
17.6.3	\$q_remove.....	308
17.6.4	\$q_full.....	308
17.6.5	\$q_exam.....	308
17.6.6	Status codes.....	309
17.7	Simulation time system functions.....	309
17.7.1	\$time	309
17.7.2	\$stime	310
17.7.3	\$realtime	310
17.8	Conversion functions	311
17.9	Probabilistic distribution functions.....	312
17.9.1	\$random function.....	312
17.9.2	\$dist_ functions.....	313
17.9.3	Algorithm for probabilistic distribution functions.....	314
17.10	Command line input.....	321
17.10.1	\$test\$plusargs (string).....	322
17.10.2	\$value\$plusargs (user_string, variable)	322
18.	Value change dump (VCD) files.....	325
18.1	Creating the four state value change dump file	325
18.1.1	Specifying the name of the dump file (\$dumpfile).....	325
18.1.2	Specifying the 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 the size of the dump file (\$dumplimit)	328
18.1.6	Reading the dump file during simulation (\$dumpflush).....	329
18.2	Format of the four state VCD file.....	330
18.2.1	Syntax of the four state VCD file	330
18.2.2	Formats of variable values.....	332
18.2.3	Description of keyword commands	333
18.2.4	Four state VCD file format example.....	339
18.3	Creating the extended value change dump file	340
18.3.1	Specifying the dumpfile name and the ports to be dumped (\$dumpports).....	340
18.3.2	Stopping and resuming the dump (\$dumpportsoff/\$dumpportson).....	341
18.3.3	Generating a checkpoint (\$dumpportsall).....	342
18.3.4	Limiting the size of the dump file (\$dumpportslimit)	342
18.3.5	Reading the dump file during simulation (\$dumpportsflush).....	343
18.3.6	Description of keyword commands	343
18.3.7	General rules for extended VCD system tasks	344
18.4	Format of the extended VCD file.....	344
18.4.1	Syntax of the extended VCD file.....	344
18.4.2	Extended VCD node information	346
18.4.3	Value changes	348
18.4.4	Extended VCD file format example	349
19.	Compiler directives.....	351
19.1	`celldefine and `endcelldefine.....	351
19.2	`default_nettype	351

19.3	`define and `undef	352
19.3.1	`define	352
19.3.2	`undef	354
19.4	`ifdef, `else, `elsif, `endif, `ifndef	354
19.5	`include	358
19.6	`resetall.....	358
19.7	`line	358
19.8	`timescale	359
19.9	`unconnected_drive and `nounconnected_drive	361
20.	PLI overview.....	362
20.1	PLI purpose and history (informative).....	362
20.2	User-defined system task or function names	362
20.3	User-defined system task or function types	363
20.4	Overriding built-in system task and function names	363
20.5	User-supplied PLI applications.....	363
20.6	PLI interface mechanism	363
20.7	User-defined system task and function arguments	364
20.8	PLI include files.....	364
20.9	PLI Memory Restrictions.....	364
21.	PLI TF and ACC interface mechanism.....	365
21.1	User-supplied PLI applications.....	365
21.1.1	The sizetf class of PLI applications	365
21.1.2	The checktf class of PLI applications	365
21.1.3	The calltf class of PLI applications.....	366
21.1.4	The misctf class of PLI applications.....	366
21.1.5	The consumer class of PLI applications	366
21.2	Associating PLI applications to a class and system task/function name	366
21.3	PLI application arguments	367
21.3.1	The data C argument.....	367
21.3.2	The reason C argument	367
21.3.3	The paramvc C argument.....	368
22.	Using ACC routines.....	369
22.1	ACC routine definition	369
22.2	The handle data type	369
22.3	Using ACC routines.....	370
22.3.1	Header files	370
22.3.2	Initializing ACC routines.....	370
22.3.3	Exiting ACC routines.....	370
22.4	List of ACC routines by major category	370
22.4.1	Fetch routines.....	371
22.4.2	Handle routines	372
22.4.3	Next routines.....	373
22.4.4	Modify routines.....	375
22.4.5	Miscellaneous routines.....	375
22.4.6	VCL routines.....	376
22.5	Accessible objects.....	376
22.5.1	ACC routines that operate on module instances	378
22.5.2	ACC routines that operate on module ports	378

22.5.3	ACC routines that operate on bits of a port	379
22.5.4	ACC routines that operate on module paths or data paths	379
22.5.5	ACC routines that operate on intermodule paths	380
22.5.6	ACC routines that operate on top-level modules.....	380
22.5.7	ACC routines that operate on primitive instances	380
22.5.8	ACC routines that operate on primitive terminals	381
22.5.9	ACC routines that operate on nets	381
22.5.10	ACC routines that operate on reg types	382
22.5.11	ACC routines that operate on integer, real, and time variables	382
22.5.12	ACC routines that operate on named events.....	382
22.5.13	ACC routines that operate on parameters and specparams.....	383
22.5.14	ACC routines that operate on timing checks	383
22.5.15	ACC routines that operate on timing check terminals	383
22.5.16	ACC routines that operate on user-defined system task/function arguments	384
22.6	ACC routine types and fulltypes.....	384
22.7	Error handling	387
22.7.1	Suppressing error messages	388
22.7.2	Enabling warnings	388
22.7.3	Testing for errors.....	388
22.7.4	Example	388
22.7.5	Exception values	389
22.8	Reading and writing delay values.....	389
22.8.1	Number of delays for Verilog HDL objects	390
22.8.2	ACC routine configuration	390
22.8.3	Determining the number of arguments for ACC delay routines.....	391
22.9	String handling.....	395
22.9.1	ACC routines share an internal string buffer	395
22.9.2	String buffer reset	396
22.9.3	Preserving string values	397
22.9.4	Example of preserving string values.....	397
22.10	Using VCL ACC routines.....	397
22.10.1	VCL objects	398
22.10.2	The VCL record definition.....	398
22.10.3	Effects of acc_initialize() and acc_close() on VCL consumer routines	401
22.10.4	An example of using VCL ACC routines	401
23.	ACC routine definitions.....	404
24.	Using TF routines	579
24.1	TF routine definition	579
24.2	TF routine system task/function arguments.....	579
24.3	Reading and writing system task/function argument values.....	579
24.3.1	Reading and writing 2-state parameter argument values.....	579
24.3.2	Reading and writing 4-state values	579
24.3.3	Reading and writing strength values.....	580
24.3.4	Reading and writing to memories	580
24.3.5	Reading and writing string values.....	580
24.3.6	Writing return values of user-defined functions	580
24.3.7	Writing the correct C data types	580
24.4	Value change detection	581
24.5	Simulation time	581
24.6	Simulation synchronization	581
24.7	Instances of user-defined tasks or functions	582

24.8	Module and scope instance names	582
24.9	Saving information from one system TF call to the next.....	582
24.10	Displaying output messages.....	582
24.11	Stopping and finishing.....	582
25.	TF routine definitions	583
26.	Using VPI routines.....	659
26.1	VPI system tasks and functions	659
26.2	The VPI interface	659
26.2.1	VPI callbacks	659
26.2.2	VPI access to Verilog HDL objects and simulation objects.....	660
26.2.3	Error handling.....	660
26.2.4	Function availability	660
26.2.5	Traversing expressions.....	660
26.3	VPI object classifications.....	661
26.3.1	Accessing object relationships and properties	662
26.3.2	Object type properties.....	663
26.3.3	Object file and line properties.....	663
26.3.4	Delays and values	664
26.4	List of VPI routines by functional category.....	664
26.5	Key to data model diagrams	666
26.5.1	Diagram key for objects and classes	667
26.5.2	Diagram key for accessing properties.....	667
26.5.3	Diagram key for traversing relationships	668
26.6	Object data model diagrams.....	669
27.	VPI routine definitions.....	700
Annex A	(normative) Formal syntax definition.....	761
A.1	Source text	761
A.1.1	Library source text	761
A.1.2	Configuration source text.....	761
A.1.3	Module and primitive source text	762
A.1.4	Module parameters and ports.....	762
A.1.5	Module items.....	762
A.2	Declarations	763
A.2.1	Declaration types.....	763
A.2.2	Declaration data types	765
A.2.3	Declaration lists.....	765
A.2.4	Declaration assignments	766
A.2.5	Declaration ranges.....	766
A.2.6	Function declarations	766
A.2.7	Task declarations.....	766
A.2.8	Block item declarations.....	767
A.3	Primitive instances	768
A.3.1	Primitive instantiation and instances.....	768
A.3.2	Primitive strengths	768
A.3.3	Primitive terminals.....	769
A.3.4	Primitive gate and switch types	769

A.4	Module and generated instantiation	769
A.4.1	Module instantiation	769
A.4.2	Generated instantiation	769
A.5	UDP declaration and instantiation	770
A.5.1	UDP declaration	770
A.5.2	UDP ports	770
A.5.3	UDP body	771
A.5.4	UDP instantiation	771
A.6	Behavioral statements	771
A.6.1	Continuous assignment statements	771
A.6.2	Procedural blocks and assignments	771
A.6.3	Parallel and sequential blocks	772
A.6.4	Statements	772
A.6.5	Timing control statements	772
A.6.6	Conditional statements	773
A.6.7	Case statements	774
A.6.8	Looping statements	774
A.6.9	Task enable statements	774
A.7	Specify section	774
A.7.1	Specify block declaration	774
A.7.2	Specify path declarations	775
A.7.3	Specify block terminals	775
A.7.4	Specify path delays	775
A.7.5	System timing checks	777
A.8	Expressions	779
A.8.1	Concatenations	779
A.8.2	Function calls	779
A.8.3	Expressions	780
A.8.4	Primaries	781
A.8.5	Expression left-side values	781
A.8.6	Operators	782
A.8.7	Numbers	782
A.8.8	Strings	783
A.9	General	783
A.9.1	Attributes	783
A.9.2	Comments	783
A.9.3	Identifiers	783
A.9.4	Identifier branches	784
A.9.5	White space	785
	Annex B (normative) List of keywords	786
	Annex C (informative) System tasks and functions	788
C.1	\$countdrivers	788
C.2	\$getpattern	789
C.3	\$input	790
C.4	\$key and \$nokey	790
C.5	\$list	791
C.6	\$log and \$nolog	791
C.7	\$reset, \$reset_count, and \$reset_value	791
C.8	\$save, \$restart, and \$incsave	792
C.9	\$scale	793
C.10	\$scope	793

C.11 \$showscopes	793
C.12 \$showvars	794
C.13 \$sreadmemb and \$sreadmemh.....	794
Annex D (informative) Compiler directives	795
D.1 `default_decay_time.....	795
D.2 `default_trireg_strength.....	795
D.3 `delay_mode_distributed.....	796
D.4 `delay_mode_path.....	796
D.5 `delay_mode_unit	796
D.6 `delay_mode_zero.....	796
Annex E (normative) acc_user.h.....	797
Annex F (normative) veriusers.h.....	806
Annex G (normative) vpi_user.h	814
Annex H (informative) Bibliography.....	828

IEEE Standard Verilog[®] Hardware Description Language

1. Overview

1.1 Objectives of this standard

The intent of this standard is to serve as a complete specification of the Verilog[®] Hardware Description Language (HDL). This document contains

- 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 access routines, task/function routines, and Verilog procedural interface routines
- Informative usage examples
- Informative delay model for SDF
- Listings of header files for PLI

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 *can* is used to indicate optional features. These terms denote different meanings to different readers of this standard:

- 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 *can* denotes optional features that the model

developer can exercise at discretion. If 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:

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

`module_declaration`

- b) Boldface words are used to denote reserved keywords, operators, and punctuation marks as a required part of the syntax. These words appear in a larger font for distinction. For example:

module **=>** ;

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

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

- d) Square brackets enclose optional items. For example:

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

- e) 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`

- f) 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_constant_expression* and *lsb_constant_expression* are equivalent to `constant_expression`.

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

1.4 Contents of this standard

A synopsis of the clauses and annexes is presented as a quick reference. There are 27 clauses and 8 annexes. All clauses, as well as Annex A, Annex B, Annex E, Annex F, and Annex G, are normative parts of this standard. Annex C, Annex D, and Annex H are included for informative purposes only.

Clause 1—Overview: This clause discusses the conventions used in this standard and its contents.

Clause 2—Lexical conventions: This clause describes the lexical tokens used in Verilog HDL source text and their conventions. It describes how to specify and interpret the lexical tokens.

Clause 3—Data types: This clause 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 4—Expressions: This clause describes the operators and operands that can be used in expressions.

Clause 5—Scheduling semantics: This clause describes the scheduling semantics of the Verilog HDL.

Clause 6—Assignments: This clause 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—Gate and switch level modeling: This clause describes the gate and switch level primitives and logic strength modeling.

Clause 8—User-defined primitives (UDPs): This clause describes how a primitive can be defined in the Verilog HDL and how these primitives are included in Verilog HDL models.

Clause 9—Behavioral modeling: This clause describes procedural assignments, procedural continuous assignments, and behavioral language statements.

Clause 10—Tasks and functions: This clause 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.

Clause 11—Disabling of named blocks and tasks: This clause describes how to disable the execution of a task and a block of statements that has a specified name.

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

Clause 13—Configuring the contents of a design: This clause describes how to configure the contents of a design.

Clause 14—Specify blocks: This clause describes how to specify timing relationships between input and output ports of a module.

Clause 15—Timing checks: This clause describes how timing checks are used in specify blocks to determine if signals obey the timing constraints.

Clause 16—Backannotation using the Standard Delay Format (SDF): This clause describes syntax and semantics of Standard Delay Format (SDF) constructs.

Clause 17—System tasks and functions: This clause describes the system tasks and functions.

Clause 18—Value change dump (VCD) files: This clause describes the system tasks associated with Value Change Dump (VCD) file, and the format of the file.

Clause 19—Compiler directives: This clause describes the compiler directives.

Clause 20—PLI overview: This clause previews the C language procedural interface standard (Programming Language Interface or PLI) and interface mechanisms that are part of the Verilog HDL.

Clause 21—PLI TF and ACC interface mechanism: This clause describes the interface mechanism that provides a means for users to link PLI task/function (TF) routine and access (ACC) routine applications to Verilog software tools.

Clause 22—Using ACC routines: This clause describes the ACC routines in general, including how and why to use them.

Clause 23—ACC routine definitions: This clause describes the specific ACC routines, explaining their function, syntax, and usage.

Clause 24—Using TF routines: This clause provides an overview of the types of operations that are done with the TF routines.

Clause 25—TF routine definitions: This clause describes the specific TF routines, explaining their function, syntax, and usage.

Clause 26—Using VPI routines: This clause provides an overview of the types of operations that are done with the Verilog Programming Interface (VPI) routines.

Clause 27—VPI routine definitions: This clause describes the VPI routines.

Annex A—Formal syntax definition: This normative annex describes, using BNF, the syntax of the Verilog HDL.

Annex B—List of keywords: This normative annex lists the Verilog HDL keywords.

Annex C—System tasks and functions: This informative annex describes system tasks and functions that are frequently used, but that are not part of the standard.

Annex D—Compiler directives: This informative annex describes compiler directives that are frequently used, but that are not part of the standard.

Annex E—acc_user.h: This normative annex provides a listing of the contents of the `acc_user.h` file.

Annex F—veriusers.h: This normative annex provides a listing of the contents of the `veriusers.h` file.

Annex G—vpi_user.h: This normative annex provides a listing of the contents of the `vpi_user.h` file.

Annex H—Bibliography: This informative annex contains bibliographic entries pertaining to this standard.

1.5 Header file listings

The header file listings included in the annexes E, F, and G for `acc_user.h`, `veriusers.h`, and `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.

1.6 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.

1.7 Prerequisites

Clauses 20 through 27 and Annexes E through G presuppose a working knowledge of the C programming language.

2. Lexical conventions

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

2.1 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 2.7.1).

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

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

2.2 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 2.6).

2.3 Comments

The Verilog HDL has two forms to introduce comments. A *one-line comment* shall start with the two characters `//` and end with a new line. 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.

2.4 Operators

Operators are single-, double-, or triple-character sequences and are used in expressions. Clause 4 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.

2.5 Numbers

Constant numbers can be specified as integer constants (defined in 2.5.1) or real constants.

```

number ::= (From Annex A - A.8.7)
    decimal_number
    | octal_number
    | binary_number
    | hex_number
    | real_number
real_numbera ::=
    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_numbera ::= non_zero_decimal_digit { _ | decimal_digit }
unsigned_numbera ::= decimal_digit { _ | decimal_digit }
binary_valuea ::= binary_digit { _ | binary_digit }
octal_valuea ::= octal_digit { _ | octal_digit }
hex_valuea ::= hex_digit { _ | hex_digit }
decimal_basea ::= '[s]S)d | '[s]S]D
binary_basea ::= '[s]S]b | '[s]S]B
octal_basea ::= '[s]S]o | '[s]S]O
hex_basea ::= '[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 | ?

```

^aEmbedded spaces are illegal.

Syntax 2-1—Syntax for integer and real numbers

2.5.1 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 *size constant*, which shall be composed of up to three tokens—an optional size constant, a single quote followed by a base format character, and the digits representing the value of the number.

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 non-zero unsigned decimal number. For example, the size specification for two hexadecimal digits is 8, because one hexadecimal digit requires 4 bits. Unsized unsigned constants where the high order bit is unknown (X or x) or three-state (Z or z) are extended to the size of the expression containing the constant.

NOTE—In IEEE Std 1364-1995, unsized constants where the high order bit is unknown or three-state, the x or z was only extended to 32 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 single quote character ('). Legal base specifications are d, D, h, H, o, O, b, or B, for the bases decimal, hexadecimal, octal, and binary respectively.

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

The single quote 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 2's complement form.

An X represents the *unknown value* in hexadecimal, octal, and binary constants. A Z represents the *high-impedance value*. See 3.1 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.

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 don't-care condition. See the discussion of **casez** and **caseX** in 9.5.1. The question-mark character is also used in user-defined primitive state tables. See 8.1.6, Table 40.

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.

*Examples:**Example 1—Unsize 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'.

```

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}}

```

Example 5—Using underscore character in numbers

```

27_195_000
16'b0011_0101_0001_1111
32 'h 12ab_f001

```

NOTES:

- 1) 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 or not.
- 2) Each of the three tokens for specifying a number may be macro substituted.
- 3) 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.

2.5.2 Real constants

The *real constant numbers* shall be represented as described by IEEE Std 754-1985 [B1],¹ 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.

Examples:

```
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
```

2.5.3 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.

2.6 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.

¹The numbers in brackets correspond to those of the bibliography in Annex H.

2.6.1 String variable declaration

String variables are variables of `reg` type (see 3.2) with width equal to the number of characters in the string multiplied by 8.

Example:

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

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

2.6.2 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.

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:

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

NOTE—When a variable is larger than required to hold a value being assigned, the contents on the left are padded with zeros after the assignment. This is consistent with the padding that occurs during assignment of nonstring values. If a string is larger than the destination string variable, the string is truncated to the left, and the leftmost characters will be lost.

2.6.3 Special characters in strings

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

Table 1—Specifying special characters in string

Escape string	Character produced by escape string
\n	New line character
\t	Tab character
\\	\ character
\"	" character
\ddd	A character specified in 1–3 octal digits (0 ≤ d ≤ 7)

2.7 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 2.7.1). 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.

Example:

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

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

2.7.1 Escaped identifiers

Escaped identifiers shall start with the backslash character (\) and end with white space (space, tab, new-line). 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.

Example:

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

2.7.2 Generated identifiers

Generated identifiers are created by generate loops (see 12.1.3.2); and are a special case of identifiers in that they can be used in hierarchical names (see 12.4). A generated identifier is the named generate block identifier terminated with a ([digit(s)]) string. This identifier is used as a node name in hierarchical names (see 12.4).

2.7.3 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.

2.7.4 System tasks and functions

The \$ character introduces a language construct that enables development of user-defined 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 or function is given in Syntax 2-2.

```

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

^aThe \$ character 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 2-2—Syntax for system tasks and functions

The \$identifier system task or function can be defined in three places

- A standard set of \$identifier system tasks and functions, as defined in Clauses 17 and 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 or 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.

Example:

```

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

2.7.5 Compiler directives

The ``` character (the ASCII value 60, called open quote or accent grave) 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.

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.

Example:

```
`define wordsize 8
```

2.8 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 may 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 2-3.

```
attribute_instance ::= (From Annex A - A.9.1)
    (* attr_spec { , attr_spec } *)
attr_spec ::=
    attr_name = constant_expression
    | attr_name
attr_name ::=
    identifier
```

Syntax 2-3—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.

2.8.1 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

```
(* full_case=1, parallel_case=1 *)
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 c l a.

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;
```

2.8.2 Syntax

The syntax for legal statements with attributes is shown in Syntax 2-4 — Syntax 2-9.

The syntax for module declaration attributes is given in Syntax 2-4.

```
module_declaration ::= (From Annex A - A.1.3)
    { 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 2-4—Syntax for module declaration attributes

The syntax for port declaration attributes is given in Syntax 2-5.

```
port_declaration ::= (From Annex A - A.1.4)
    { attribute_instance } inout_declaration
    | { attribute_instance } input_declaration
    | { attribute_instance } output_declaration
```

Syntax 2-5—Syntax for port declaration attributes

The syntax for module item attributes is given in Syntax 2-6.

```

module_item ::= (From Annex A - A.1.5)
    module_or_generate_item
    | port_declaration ;
    | { attribute_instance } generated_instantiation
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration
    | { attribute_instance } specify_block
    | { attribute_instance } specparam_declaration
module_or_generate_item ::=
    { attribute_instance } module_or_generate_item_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
non_port_module_item ::=
    { attribute_instance } generated_instantiation
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } module_or_generate_item
    | { attribute_instance } parameter_declaration
    | { attribute_instance } specify_block
    | { attribute_instance } specparam_declaration

```

Syntax 2-6—Syntax for module item attributes

The syntax for function port, task, and block attributes is given in Syntax 2-7.

```
function_port_list ::= (From Annex A - A.2.6)
    { attribute_instance } input_declaration { , { attribute_instance } input_declaration }
task_item_declaration ::= (From Annex A - 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 Annex A - A.2.8)
    { attribute_instance } block_reg_declaration
    | { attribute_instance } event_declaration
    | { attribute_instance } integer_declaration
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration
    | { attribute_instance } real_declaration
    | { attribute_instance } realtime_declaration
    | { attribute_instance } time_declaration
```

Syntax 2-7—Syntax for function port, task, and block attributes

The syntax for port connection attributes is given in Syntax 2-8.

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

Syntax 2-8—Syntax for port connection attributes

The syntax for udp attributes is given in Syntax 2-9.

```
udp_declaration ::= (From Annex A - 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 Annex A - 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 2-9—Syntax for udp attributes

3. Data types

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

3.1 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. The exception is the *event* type (see 9.7.3), which has no storage. 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.

3.2 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.

3.2.1 Net declarations

The *net* data types shall 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 3.12).

The syntax for net declarations is given in Syntax 3-1.

```

net_declaration ::= (From Annex A - 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 ;
| triereg [ charge_strength ] [ signed ]
    [ delay3 ] list_of_net_identifiers ;
| triereg [ drive_strength ] [ signed ]
    [ delay3 ] list_of_net_decl_assignments ;
| triereg [ charge_strength ] [ vectored | scalared ] [ signed ]
    range [ delay3 ] list_of_net_identifiers ;
| triereg [ drive_strength ] [ vectored | scalared ] [ signed ]
    range [ delay3 ] list_of_net_decl_assignments ;

net_type ::= (From Annex A - A.2.2.1)
    supply0 | supply1
    | tri | triand | trior | tri0 | tri1
    | wire | wand | wor

drive_strength ::= (From Annex A - 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 Annex A - A.2.2.3)
    # delay_value | # ( delay_value [ , delay_value [ , delay_value ] ] )
delay2 ::=
    # delay_value | # ( delay_value [ , delay_value ] )
delay_value ::=
    unsigned_number
    | parameter_identifier
    | specparam_identifier
    | mintypmax_expression

list_of_net_decl_assignments ::= (From Annex A - A.2.3)
    net_decl_assignment { , net_decl_assignment }

list_of_net_identifiers ::=
    net_identifier [ dimension { dimension } ]
    { , net_identifier [ dimension { dimension } ] }

net_decl_assignment ::= (From Annex A - A.2.4)
    net_identifier = expression

dimension ::= (From Annex A - A.2.5)
    [ dimension_constant_expression : dimension_constant_expression ]

range ::=
    [ msb_constant_expression : lsb_constant_expression ]

```

Syntax 3-1—Syntax for net declaration

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

3.2.2 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 datatypes 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 the Verilog standard, the term *register* was used to encompass both the **reg**, **integer**, **time**, **real** and **realtime** types, but that term is no longer used as a Verilog data type.

The syntax for variable declarations is given in Syntax 3-2.

```
integer_declaration ::= (From Annex A - 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 Annex A - A.2.2.1)
    real_identifier [ = constant_expression ]
    | real_identifier dimension { dimension }
variable_type ::=
    variable_identifier [ = constant_expression ]
    | variable_identifier dimension { dimension }
list_of_real_identifiers ::= (From Annex A - A.2.3)
    real_type { , real_type }
list_of_variable_identifiers ::=
    variable_type { , variable_type }
dimension ::= (From Annex A - A.2.5)
    [ dimension_constant_expression : dimension_constant_expression ]
range ::=
    [ msb_constant_expression : lsb_constant_expression ]
```

Syntax 3-2—Syntax for variable declaration

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

CAUTION

Variables can be assigned negative values, but only signed regs, integer, real, and realtime variables shall retain the significance of the sign. The unsigned reg and time variables shall treat the value assigned to them as an unsigned value. Refer to 4.1.6 for a description of how signed and unsigned variables are treated by certain Verilog operators.

3.3 Vectors

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

3.3.1 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 *msb* constant expression and *lsb* constant expression shall be constant expressions. The *msb* and *lsb* constant expressions can be any value—positive, negative, or zero. The *lsb* constant expression can be a greater, equal, or lesser value than *msb* constant expression.

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).

Examples:

```
wand w;           // a scalar net of type “wand”
tri [15:0] busa;   // a three-state 16-bit bus
triereg (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
```

NOTES:

- 1) Implementations may set a limit on the maximum length of a vector, but they will at least be 65536 (2^{16}) bits.
- 2) Implementations do not have to detect overflow of integer operations.

3.3.2 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 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 and part selects of the object shall be permitted, and the PLI shall consider the object *expanded*.

Examples:

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

3.4 Strengths

There are two types of *strengths* that can be specified in a net declaration. They are 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.

3.4.1 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).

Examples:

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

3.4.2 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.

3.5 Implicit declarations

The syntax shown in 3.2 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:

- If an identifier is used in a port expression declaration, then an implicit net of type **wire** 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 explicitly declared previously in one of the declaration statements of the instantiating module, 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, then an implicit scalar net of default net type shall be assumed. See 6.1.2 for a discussion of continuous assignment statements.

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

3.6 Net initialization

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**).

3.7 Net types

There are several distinct types of nets, as shown in Table 2.

Table 2—Net types

wire	tri	tri0	supply0
wand	triand	tri1	supply1
wor	trior	trireg	

3.7.1 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 3 is a truth table for resolving multiple drivers on wire and tri nets. Note that it assumes equal strengths for both drivers. Please refer to 7.9 for a discussion of logic strength modeling.

Table 3—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

Table 3—Truth table for wire and tri nets

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

3.7.2 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, such 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, such 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 and Table 5 give the truth tables for wired nets. Note that they assume equal strengths for both drivers. See 7.9 for a discussion of logic strength modeling.

Table 4—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 5—Truth table for wor and prior nets

wor/ prior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

3.7.3 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.

Examples:

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

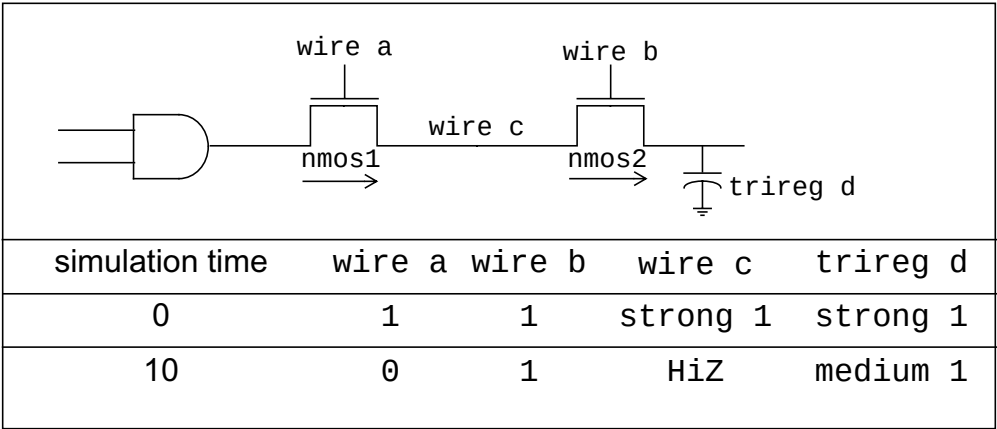


Figure 1—Simulation values of a trireg and its driver

- 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.

3.7.3.1 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.

Examples:

Figure 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.