

Recognized as an
American National Standard (ANSI)

IEEE Std 1364-1995

IEEE Standard Hardware Description Language Based on the Verilog[®] Hardware Description Language

Sponsor

**Design Automation Standards Committee
of the
IEEE Computer Society**

Approved 12 December 1995

IEEE Standards Board

Approved 1 August 1996

American National Standards Institute

Abstract: The Verilog[®] Hardware Description Language (HDL) is defined. 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, electronic systems, digital systems, hardware, hardware design, hardware description languages, HDL, programming language interface, PLI, Verilog HDL, Verilog PLI, Verilog[®]

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

Copyright © 1996 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 1996. Printed in the United States of America

ISBN 1-55937-727-5

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 Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. 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.

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.

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 all 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 technical 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 on standards and requests for interpretations should be addressed to:

Secretary, IEEE 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 all patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; (508) 750-8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Introduction

(This introduction is not a part of IEEE Std 1364-1995, IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language.)

The Verilog® Hardware Description Language (Verilog HDL) 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. The Verilog HDL was designed by Phil Moorby during the winter of 1983–1984, and it was introduced into the EDA market in 1985 as the cornerstone of a verification simulator product.

The Verilog HDL 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 registers. Continuous assignments, in which expressions of both registers and nets can continuously drive values onto nets, provide the basic structural construct. Procedural assignments, in which the results of calculations involving register and net values can be stored into registers, 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). The PLI 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 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. With many designers all over the world designing electronic circuits with Verilog HDL, this idea was enthusiastically received by the Verilog user community. When the Project Authorization Request (1364) was approved by the IEEE in 1993, a working group was formed and the first meeting was held on October 14, 1993.

Objective

The starting point for the IEEE P1364 Working Group were the OVI LRM version 2.0 and OVI PLI versions 1.0 and 2.0. The standardization process started with the clear objective of making it easier for the user to understand and use Verilog. The IEEE P1364 standard had to be clear, unambiguous, implementable, and not overly constraining. Since Verilog HDL has been in use for some time, it was quite robust enough to be presented to the user community without a great deal of enhancements. The working group, therefore, decided not to spend a lot of time extending the language, but, for the purpose of this standardization, to concentrate on clarifying the language.

Since Verilog HDL has been in widespread use and a number of ASIC vendors have built extensive libraries in Verilog HDL, it was very important to maintain the integrity of these existing models. With this in mind, it

was decided that the intent of the working group would be to maintain the integrity of the standard and every care would be taken not to invalidate existing models.

The standardization process

In order to clarify the language, many changes were proposed from a number of sources. The working group met 15 times over a period of 18 months and voted on nearly 400 motions. Four drafts of the document were generated and reviewed. It is a tribute to the hard work and dedication put forward by all the members of the working group that this standard was completed in the short span of 18 months.

Many new sections were created, one of which is the section on scheduling semantics. A number of sections were merged to form new sections. The two annexes containing compiler directives and system tasks were moved into main text as two sections. Every effort has been made to clarify all ambiguities, add explanations, and delete references that were deemed unnecessary.

Changes also included removing product specific references and restrictions. The minimum product requirements for implementing this standard were clarified. A number of examples, figures, and tables were retained in order to provide better context and explanation.

The PLI Task Force provided a clear and accurate description of OVI PLI 1.0 implementations already in existence, and revisited the OVI PLI 2.0 specification to ensure its accuracy and completeness. The baseline for the access routines and the task/function routines was the OVI PLI 1.0 specification. As there are a large number of OVI PLI 1.0 routines in widespread use that were not included in the OVI PLI 1.0 document, it was decided to consider additions to this document from the pool of existing OVI PLI 1.0 implementations. The access routines and the task/function routines provide full backwards compatibility with Verilog HDL software tools and PLI applications.

The baseline for the VPI routines was the existing OVI PLI 2.0 document. To this, the task force brought new experience from the implementations in progress, which helped prove the worthiness of the previously untested specification.

Acknowledgments

This standard is based on work originally developed by Cadence Design Systems, Inc. (in their Verilog LRM 1.6 and 2.0 and PLI documents) and Open Verilog International (in their Verilog LRM 2.0 and PLI 1.0 and 2.0). The IEEE is grateful to Cadence Design Systems and Open Verilog International for permission to use their materials as the basis for this standard.

The IEEE Std 1364-1995 working group organization

Many individuals from many different organizations participated directly or indirectly in the standardization process. The main body of the IEEE P1364 working group is located in the United States, with a subgroup in Japan. Over a period of 18 months many task forces were created, of which the PLI task force was prominent.

The members of the IEEE P1364 working group had voting privileges, and all motions had to be approved by this group to be implemented. All task forces and subgroups focused on some specific areas, and their recommendations were eventually voted on by the IEEE P1364 working group.

At the time this document was approved, the IEEE P1364 working group had the following membership:

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

Victor Berman
Leigh Brady
Clifford E. Cummings
Peter Eichenberger
Andrew T. Lynch

John Mancini
Michael McNamara
Elliot Mednick
Phil Moorby
Gabe Moretti

John Sanguinetti
Joseph P. Skudlarek
Stuart Sutherland
John R. Williamson
Alex N. Zamfirescu

The PLI task force consisted of the following members:

Andrew T. Lynch, PLI Task Force Leader
Stuart Sutherland, Technical Editor

Charles A. Dawson
Rajeev Madhavan

Joel Paston
Sathyam K. Pattanam
David Roberts

Marco Zelado
Guoqing Zhang

The IEEE P1364 Japan subgroup consisted of the following members:

Yoshiharu Furui, Vice-Chair, IEEE-1364 Working Group

Takaaki Akashi
Kasumi Hamaguchi
Masato Ikeda
Masaru Kakimoto
Kazuya Morii

Junichi Murayama
Masaharu Nakamura
Shouhei Oda
Fujio Otsuka

Toshiyuki Sakamoto
Hitomi Sato
Katsushida Seo
Mitsuhiro Yasuda
Kazuhiro Yoshinaga

The following persons were members of the balloting group:

Guy Adam
H. Gordon Adshead
Unmesh Agarwala
Anant Agrawal
John Ainscough
Takaaki Akashi
Tom Albers
Glen Anderson
Lawrence F. Arnstein
Michael Atkin
Venkata Atluri
Rick Bahr
Jim Ball
Jose Baradian
Daniel S. Barclay
David L. Barton
Jean-Michel Berge
Victor Berman
J. Bhasker
Ron Bianchini
William D. Billowitch
Ronald D. Blanton

Miriam Blatt
James Brandt
Dennis B. Brophy
Randal E. Bryant
John A. Busco
Ben Buzonas
L. Richard Carley
Thomas Chao
Daniel Chapiro
Clive R. Charlwood
Chin-Fu Chen
Mojj C. Chian
Kai Moon Chow
Michael D. Ciletti
Joseph C. Circello
Luc Claesen
George M. Cleveland
Edmond S. Cooley
Tedd Corman
David Crohn
Clifford E. Cummings
Godfrey Paul D'Souza
Brian A. Dalio

Carlos Dangelo
Hal Daseking
Timothy R. Davis
Charles A. Dawson
Willem De Lange
Rajiv Deshmukh
Caroline DeVore-Kenney
Allen Dewey
Bill Doss
Douglas D. Dunlop
Peter Eichenberger
Hazem El Tahawy
John A. Eldon
Bassam N. Elkhoury
Ted Elkind
Brian Erickson
Robert A. Flatt
Bob Floyd
Alain Blaise Fonkoua
Douglas W. Forehand
Paul Franzon
Bill Fuchs

Yoshiharu Furui
Vassilios Gerousis
Emil Girczye
Rita A. Glover
Timothy G. Goldsberry
Alan Goodrum
Suresh Gopalakrishnan
Harutaka Goto
Kenji Goto
Brian Griffin
Steve Grout
Kazuyuki Hagiwara
Michael J. Haney
James P. Hanna
Anne C. Harris
Akira Hasegawa
Stuart Hecht
Shankar Hemmady
John Hillawi
Chris N. Hinds
Kazuyuki Hirakawa
Fumiyasu Hirose
Lynn A. Horobin
Tamio Hoshino
May Huang
Sylvie Hurat
Masaharu Imai
Ann Irza
Mitsuaki Ishikawa
Yoshi Ishizaka
David Jakopac
Paul Jeffs
Roger Jennings
Eugene E. Jones
Richard Jones
Tetsuro Kage
Masaru Kakimoto
Osamu Karatsu
Jake Karrfalt
Kaoru Kawamura
Masamichi Kawarabayashi
Pratibha Kelapure
Khozema Khambati
Bruce Kim
Choon B. Kim
Tsutomu Kimoto
Chris Kingsley
Masayuki Koyama
Tokinori Kozawa
Sarangan K. Kumar
Ramachandra P. Kunda
Douglas Laird
Jean Lebrun
Bill Ledbetter
Hung-Yi Lee
Shawn Leonard
Oz Levia
George Lippincott

Herbert Lopez-Aguado
Jin-Qin Lu
Andrew T. Lynch
Viranjit S. Madan
Rajeev Madhavan
Naotaka Maeda
Serge Maginot
James Magro
Wojciech P. Maly
Maqsoodul Mannan
Guillermo Maturana
Michael McNamara
Paul J. Menchini
Jean Mermet
Gerald T. Michael
Glen S. Miranker
Shankha Mitra
Kristan Monsen
John T. Montague
Patrick Moore
Gabe Moretti
David S. Morris
Chandra Moturu
Wolfgang Mueller
Shankar Ranjan Mukherjee
Yoshiaki Nagashima
David Nagle
Hiroshi Nakamura
Hiroshi Nakamura
Seiji Nakamura
Zainalabedin Navabi
Sivaram K. Nayudu
Robert N. Newshutz
Jun Numata
John W. O'Leary
Tetsuya Okabe
Vincent Olive
Yoichi Onishi
Samir Palnitkar
Mark Papamarcos
David M. Parry
Rajesh Patil
Robert A. Pease
Mitchell Perilstein
Bruce Petrick
John Petry
Robert Piloty
Juan Pineda
Ron Poon
Jan Pukite
Selliah Rathnam
David Rich
John P. Ries
Hemant G. Rotithor
Jacques Rouillard
Paul Rowbottom
Jon Rubinstein
Stefan Rusu

Rob A. Rutenbar
Karem A. Sakallah
Toshiyuki Sakamoto
John Sanguinetti
Hitomi Sato
Larry F. Saunders
Quentin Schmierer
Michael L. Seavey
Alex Seibulescu
Shailesh Shah
Moe Shahdad
Ravi Shankar
Charles Shelor
John P. Shen
Hiroshi Shiraishi
Toru Shonai
Alexander A. Silbey
Supreet Singh
Joseph P. Skudlarek
David M. Smith
David R. Smith
William Bong H. Soon
Larry P. Soule
John Spittal
Chakra R. Srivatsa
Joseph J. Stanco
Alec G. Stanculescu
Jay K. Strosnider
Stuart Sutherland
Kinya Tabuchi
Atsushi Takahara
Donald Thomas
Partha Tirumalai
Jose A. Torres
Paul Traynar
Richard Trihy
Yatin Trivedi
Shunjen Tsay
Radha Vaidyanathan
Arie van Rhijn
Kerry Veenstra
Venkat V. Venkataraman
Sanjay Vishin
Robert A. Walker
Tsu-Hua Wang
John J. Watters
Ronald Waxman
J. Richard Weger
Paul Weil
John R. Williamson
John C. Willis
Claudia H. Ye
William R. Young
Tetsuo Yutani
Alex N. Zamfirescu
Guoqing Zhang

When the IEEE Standards Board approved this standard on 12 December 1995, it had the following membership:

E. G. “Al” Kiener, *Chair*

Donald C. Loughry, *Vice Chair*

Andrew G. Salem, *Secretary*

Gilles A. Baril
Clyde R. Camp
Joseph A. Cannatelli
Stephen L. Diamond
Harold E. Epstein
Donald C. Fleckenstein
Jay Forster*
Donald N. Heirman
Richard J. Holleman

Jim Isaak
Ben C. Johnson
Sonny Kasturi
Lorraine C. Kevra
Ivor N. Knight
Joseph L. Koepfinger*
D. N. “Jim” Logothetis
L. Bruce McClung

Marco W. Migliaro
Mary Lou Padgett
John W. Pope
Arthur K. Reilly
Gary S. Robinson
Ingo Rüsçh
Chee Kiow Tan
Leonard L. Tripp
Howard L. Wolfman

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal
Steve Sharkey
Robert E. Hebner
Chester C. Taylor

Mary Lynne Nielsen
IEEE Standards Project Editor

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

Contents

Section 1	Overview	1
Section 2	Lexical conventions	5
Section 3	Data types.....	13
Section 4	Expressions	27
Section 5	Scheduling semantics.....	45
Section 6	Assignments	50
Section 7	Gate and switch level modeling	55
Section 8	User-defined primitives (UDPs)	87
Section 8	Behavioral modeling.....	98
Section 10	Tasks and functions.....	125
Section 11	Disabling of named blocks and tasks.....	132
Section 12	Hierarchical structures	135
Section 13	Specify blocks.....	152
Section 14	System tasks and functions	172
Section 15	Value change dump (VCD) file	207
Section 16	Compiler directives.....	219
Section 17	PLI TF and ACC interface mechanism.....	228
Section 18	Using ACC routines.....	234
Section 19	ACC routine definitions.....	270
Section 20	Using TF routines	444
Section 21	TF routine definitions	449
Section 22	Using VPI routines.....	525
Section 23	VPI routine definitions.....	554
Annex A	Formal syntax definition	594
Annex B	List of keywords	604

Annex C	The acc_user.h file	605
Annex D	The veriusser.h file	615
Annex E	The vpi_user.h file	622
Annex F	System tasks and functions	635
Annex G	Compiler directives	642
Annex H	Bibliography	644

IEEE Standard Hardware Description Language Based on the Verilog[®] Hardware Description Language

Section 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
- 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
- Listings of header files for PLI

1.2 Conventions used in this standard

This standard is organized into sections, each of which focuses on some specific area of the language. There are sub-clauses within each section 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 verb “shall” is used through out this standard to indicate mandatory requirements, whereas the verb “can” is used to indicate optional features. These verbs denote different meanings to different readers of this standard:

- a) To the developers of tools that process the Verilog HDL, the verb “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 verb “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 verb “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 verb “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 sections and annexes is presented as a quick reference. There are 23 sections and 7 annexes. All the sections and annexes A through E are normative parts of this standard. Annexes F and G are included for informative purposes only.

- 1) **Overview**
This section discusses the conventions used in this standard and its contents.
- 2) **Lexical conventions**
This section describes how to specify and interpret the lexical tokens.
- 3) **Data types**
This section describes net and reg data types. This section also discusses the parameter data type for constant values and describes drive and charge strength of the values on nets.
- 4) **Expressions**
This section describes the operators and operands that can be used in expressions.
- 5) **Scheduling semantics**
This section describes the scheduling semantics of the Verilog HDL.
- 6) **Assignments**
This section 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.
- 7) **Gate and switch level modeling**
This section describes the gate and switch level primitives and logic strength modeling.
- 8) **User-defined primitives (UDPs)**
This section describes how a primitive can be defined in the Verilog HDL and how these primitives are included in Verilog HDL models.
- 9) **Behavioral modeling**
This section describes procedural assignments, procedural continuous assignments, and behavioral language statements.
- 10) **Tasks and functions**
This section 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.
- 11) **Disabling of named blocks and tasks**
This section describes how to disable the execution of a task and a block of statements that has a specified name.
- 12) **Hierarchical structures**
This section describes how hierarchies are created in the Verilog HDL and how parameter values declared in a module can be overridden.
- 13) **Specify blocks**
This section describes how to specify timing relationships between input and output ports of a module.
- 14) **System tasks and functions**
This section describes the system tasks and functions.
- 15) **Value change dump (VCD) file**
This section describes the system tasks associated with Value Change Dump (VCD) file, and the format of the file.
- 16) **Compiler directives**
This section describes the compiler directives.
- 17) **PLI TF and ACC interface mechanism**
This section 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.
- 18) **Using ACC routines**
This section describes the ACC routines in general, including how and why to use them.

- 19) **ACC routine definitions**
This section describes the specific ACC routines, explaining their function, syntax, and usage.
- 20) **Using TF routines**
This section provides an overview of the types of operations that are done with the TF routines.
- 21) **TF routine definitions**
This section describes the specific TF routines, explaining their function, syntax, and usage.
- 22) **Using VPI routines**
This section provides an overview of the types of operations that are done with the Verilog Programming Interface (VPI) routines.
- 23) **VPI routine definitions**
This section describes the VPI routines.
- A) **Formal syntax definition**
This normative annex describes, using BNF, the syntax of the Verilog HDL.
- B) **List of keywords**
This normative annex lists the Verilog HDL keywords.
- C) **The `acc_user.h` file**
This normative annex provides a listing of the contents of the `acc_user.h` file.
- D) **The `veriusers.h` file**
This normative annex provides a listing of the contents of the `veriusers.h` file.
- E) **The `vpi_user.h` file**
This normative annex provides a listing of the contents of the `vpi_user.h` file.
- F) **System tasks and functions**
This informative annex describes system tasks and functions that are frequently used, but that are not part of the standard.
- G) **Compiler directives**
This informative annex describes compiler directives that are frequently used, but that are not part of the standard.
- 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 C, D, and E for `veriusers.h`, `acc_user.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

Sections 17 through 23 and annexes C through E presuppose a working knowledge of the C programming language.

Section 2

Lexical conventions

This section 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 newline. A *block comment* shall start with `/*` and end with `*/`. Block comments shall not be nested. The one-line comment token `//` shall not have any special meaning in a block comment.

2.4 Operators

Operators are single-, double-, or triple-character sequences and are used in expressions. Section 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 or real constants.

```

number ::=
    decimal_number
  | octal_number
  | binary_number
  | hex_number
  | real_number
decimal_number ::=
    [ sign ] unsigned_number
  | [ size ] decimal_base unsigned_number
binary_number ::=
    [ size ] binary_base binary_digit { _ | binary_digit }
octal_number ::=
    [ size ] octal_base octal_digit { _ | octal_digit }
hex_number ::=
    [ size ] hex_base hex_digit { _ | hex_digit }
real_number ::=
    [ sign ] unsigned_number . unsigned_number
  | [ sign ] unsigned_number [ . unsigned_number ] e [ sign ] unsigned_number
  | [ sign ] unsigned_number [ . unsigned_number ] E [ sign ] unsigned_number
sign ::=
    + | -
size ::=
    unsigned_number
unsigned_number ::=
    decimal_digit { _ | decimal_digit }
decimal_base ::=
    'd' | 'D'
binary_base ::=
    'b' | 'B'
octal_base ::=
    'o' | 'O'
hex_base ::=
    'h' | 'H'
decimal_digit ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
binary_digit ::=
    x | X | z | Z | 0 | 1
octal_digit ::=
    x | X | z | Z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
hex_digit ::=
    x | X | z | Z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F

```

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 *sized 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 an unsigned decimal number. For example, the size specification for two hexadecimal digits is 8, because one hexadecimal digit requires 4 bits.

The second token, a base_format, shall consist of a letter specifying the base for the number, 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 *unsigned integers*.

A plus or a minus operator preceding the size constant is a sign for the constant number; the size constant does not take a sign. 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 table. See 8.1.4.

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:

Unsigned 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)
```

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
```

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)
```

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
```

Using underscore character in numbers

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

NOTES

1—A sized negative number is not sign-extended when assigned to a register data type.

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
```

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

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.

Examples:

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.

2.6.1 String variable declaration

String variables are variables of register 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 register 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 2-1 lists these characters in the right-hand column, with the escape sequence that represents the character in the left-hand column.

Table 2-1—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 ($0 \leq d \leq 7$)

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 shall be any sequence of letters, digits, dollar signs (\$), and underscore characters (_).

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

Examples:

```
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, newline). They provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal).

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

Examples:

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

2.7.2 Keywords

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

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

2.7.3 System tasks and functions

The \$ character introduces a language construct that enables development of user-defined tasks and functions. 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_or_function ::=
    $system_task_identifier [ ( list_of_arguments ) ] ;
    | $system_function_identifier [ ( list_of_arguments ) ] ;
list_of_arguments ::=
    argument { , [ argument ] }
argument ::=
    expression
```

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 Section 14.

- Additional \$identifier system tasks and functions defined using the PLI, as described in sections 17, 23, and 25.
- 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 Section 14 are part of this standard. Additional system tasks and functions with the \$identifier construct are not part of this standard.

Examples:

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

2.7.4 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 Section 16.
- 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 Section 16 are part of this standard. Additional compiler directives with the `identifier construct are not part of this standard.

Example:

```
`define wordsize 8
```

Section 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 Section 7.

3.2 Nets and registers

There are two main groups of data types: the register 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 Nets

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 Section 6 and Section 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.

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

```

net_declaration ::=
    net_type [ vector | scalar ] [range] [delay3] list_of_net_identifiers ;
    | trireg [ vector | scalar ] [charge_strength] [range] [delay3]
    list_of_net_identifiers ;
    | net_type [ vector | scalar ] [drive_strength] [range] [delay3]
    list_of_net_decl_assignments ;

net_type ::= wire | tri | tri1 | supply0 | wand | triand | tri0 | supply1 | wor | trior
range ::= [ msb_constant_expression : lsb_constant_expression ]
drive_strength ::=
    ( strength0 , strength1 )
    | ( strength1 , strength0 )
    | ( strength0 , highz1 )
    | ( strength1 , highz0 )
    | ( highz1 , strength0 )
    | ( highz0 , strength1 )
strength0 ::= supply0 | strong0 | pull0 | weak0
strength1 ::= supply1 | strong1 | pull1 | weak1
charge_strength ::= ( small ) | ( medium ) | ( large )
delay3 ::= # delay_value | # ( delay_value [ , delay_value [ , delay_value ] ] )
delay_value ::= unsigned_number | parameter_identifier |
    constant_mintypmax_expression
list_of_net_decl_assignments ::= net_decl_assignment { , net_decl_assignment }
net_decl_assignment ::= net_identifier = 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 Section 6.

3.2.2 Registers

A *register* is an abstraction of a data storage element. The keyword for the register data type is **reg**. A register 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 default initialization value for a **reg** data type shall be the unknown value, **x**.

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

```

reg_declaration ::= reg [range] list_of_register_identifiers ;
time_declaration ::= time list_of_register_identifiers ;
integer_declaration ::= integer list_of_register_identifiers ;
real_declaration ::= real list_of_real_identifiers ;
realtime_declaration ::= realtime list_of_real_identifiers ;
list_of_register_identifiers ::= register_name { , register_name }
register_name ::=
    register_identifier
    | memory_identifier [ upper_limit_constant_expression :
    lower_limit_constant_expression ]

```

Syntax 3-2—Syntax for reg declaration

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

CAUTION

Registers can be assigned negative values, but when a register is an operand in an expression, its value shall be treated as an unsigned (positive) value. For example, a minus one (-1) in a 4-bit register shall function as the number 15 if the register is an expression operand. See 4.1.3 for more information on numeric conventions in expressions.

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 register. 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 registers 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 registers shall be treated as unsigned quantities.

Examples:

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

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

Vector 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:

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

3.4 Strengths

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

charge strength Shall be used when declaring a net of type **triereg**

drive strength Shall 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 Section 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 triereg nets. A triereg 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 triereg net shall be **medium**.

A triereg 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 triereg net (see 7.15.2).

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 Section 6 for more details. Net strength properties are described in detail in Section 7.

3.5 Implicit declarations

The syntax shown in 3.2 shall be used to declare nets and registers explicitly. In the absence of an explicit declaration of a net or a register, statements for gate, user-defined primitive, and module instantiations shall assume an implicit net declaration. This happens when a name is used in the terminal list of an instance of a gate, a user-defined primitive, or a module that has not been explicitly declared previously in one of the declaration statements of the instantiating module. See 7.9.

These implicitly declared nets shall be treated as scalar nets of type **wire**. See Section 16 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 3-1.

Table 3-1—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 on a wire or a tri net result in unknown values unless the net is controlled by logic strength.

Table 3-2 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 7.10 for a discussion of logic strength modeling.

Table 3-2—Truth table for wire and tri nets

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

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. Tables 3-3 and 3-4 give the truth tables for wired nets. Note that they assume equal strengths for both drivers. See 7.10 for a discussion of logic strength modeling.

Table 3-3—Truth tables 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 3-4—Truth tables for wor and trior nets

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

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-code**, **pull**, or **weak**, depending on the strength of the driver.

Example:

Figure 3-1 shows a schematic that includes a trireg net whose size is **medium**, its driver, and the simulation results.

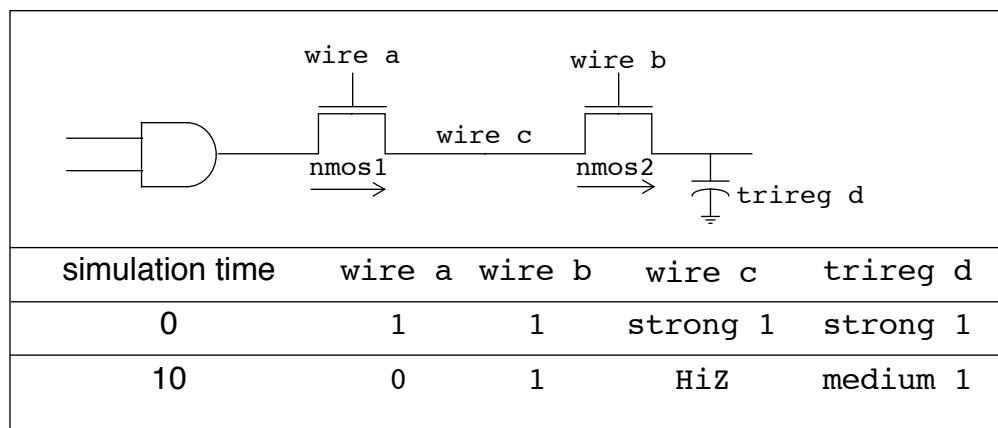


Figure 3-1 — Simulation values of a trireg and its driver

- 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.
- 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 3-2 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.

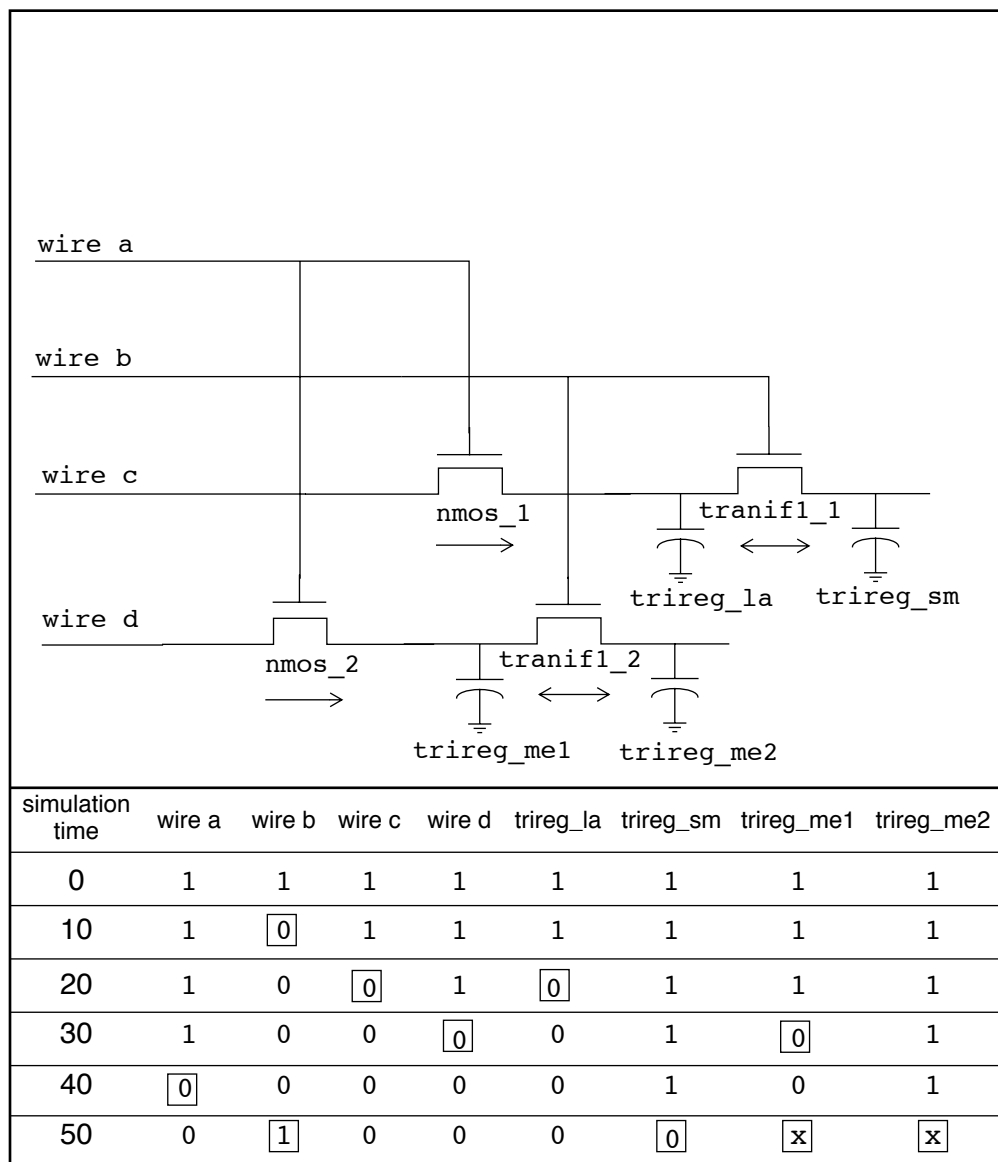


Figure 3-2—Simulation results of a capacitive network

In Figure 3-2, the capacitive strength of `trireg_la` net is **large**, `trireg_me1` and `trireg_me2` are **medium**, and `trireg_sm` is **small**. Simulation reports the following sequence of events:

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

- f) At simulation time 50, the value of wire b changes to 1.

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

This change of value in wire b also connects `triereg_me1` to `triereg_me2`; these triereg nets have the same size and stored different values. The connection causes both `triereg_me1` and `triereg_me2` to change value to x.

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

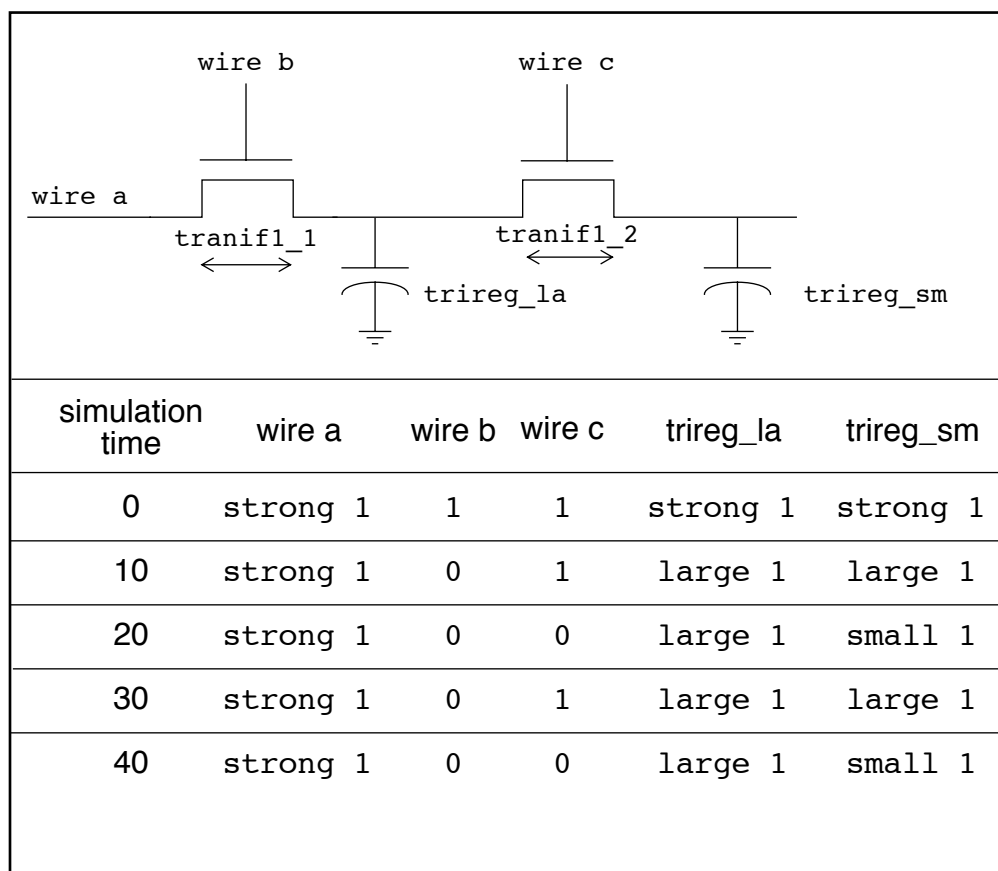


Figure 3-3—Simulation results of charge sharing

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

- At simulation time 0, the values of wire a, wire b, and wire c are 1, and wire a drives a **strong 1** into `triereg_la` and `triereg_sm`.
- At simulation time 10, the value of wire b changes to 0, disconnecting `triereg_la` and `triereg_sm` from wire a. The `triereg_la` and `triereg_sm` nets enter the capacitive state. Both triereg nets share the **large** charge of `triereg_la` because they remain connected through `tranif1_2`.

- c) At simulation time 20, the value of wire `c` changes to 0, disconnecting `triereg_sm` from `triereg_la`. The `triereg_sm` no longer shares **large** charge of `triereg_la` and now stores a **small** charge.
- d) At simulation time 30, the value of wire `c` changes to 1, connecting the two `triereg` nets. These `triereg` nets now share the same charge.
- e) At simulation time 40, the value of wire `c` changes again to 0, disconnecting `triereg_sm` from `triereg_la`. Once again, `triereg_sm` no longer shares the **large** charge of `triereg_la` and now stores a **small** charge.

3.7.3.2 Ideal capacitive state and charge decay

A *triereg* net can retain its value indefinitely or its charge can decay over time. The simulation time of charge decay is specified in the delay specification of the `triereg` net. See 7.15.2 for charge decay explanation.

3.7.4 Tri0 and tri1 nets

The *tri0* and *tri1* nets model nets with resistive *pulldown* and resistive *pullup* devices on them. When no driver drives a `tri0` net, its value is 0. When no driver drives a `tri1` net, its value is 1. The strength of this value is **pull**. See Section 7 for a description of strength modeling.

A truth table for `tri0` is shown in Table 3-5. A truth table for `tri1` is shown in Table 3-6.

Table 3-5—Truth table for `tri0` net

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

Table 3-6—Truth table for `tri1` net

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

3.7.5 Supply nets

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

3.8 Memories

An array of registers can be used to model read-only memories (ROMs), random access memories (RAMs), and reg-

ister files. Each register in the array is known as an *element* or *word* and is addressed by a single array index. There shall be no arrays with multiple dimensions.

Memories shall be declared in register declaration statements by specifying the element address range after the declared identifier. See 3.2.2. The expressions that specify the indices of the array shall be constant expressions. The value of the constant expression can be a positive integer, a negative integer, or zero.

One declaration statement can be used for declaring both registers and memories. This makes it convenient to declare both a memory and some registers that will hold data to be read from and written to the memory in the same declaration statement.

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

Examples:

Example 1—Memory declaration

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

parameter           // parameters are run-time constants - see 3.10
    wordsize = 16,
    memsize = 256;

// Declare 256 words of 16-bit memory plus two regs
reg [wordsize-1:0] writereg,           // equivalent to [15:0]
    readreg,
    mem [memsize-1:0]; // equivalent to [255:0]
```

Example 2—A memory of n 1-bit registers is different from an n -bit vector register

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

Example 3—Assignment to memory words

```
rega = 0; // Legal syntax
mema = 0; // Illegal syntax
mema[1] = 0; //Assigns 0 to the first element of mema
```

NOTE—Implementations may limit the maximum size of a register array, but they will at least be 16777216 (2^{24}).

3.9 Integers, reals, times, and realtimes

In addition to modeling hardware, there are other uses for registers in an HDL model. Although reg variables can be

used for general purposes such as counting the number of times a particular net changes value, the *integer* and *time* register data types are provided for convenience and to make the description more self-documenting.

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

```
integer_declaration ::= integer list_of_register_identifiers ;
time_declaration ::= time list_of_register_identifiers ;
real_declaration ::= real list_of_real_identifiers ;
realtime_declaration ::= realtime list_of_real_identifiers ;
list_of_register_identifiers ::= register_name { , register_name }
register_name ::=
    register_identifier
    | memory_identifier [ upper_limit_constant_expression :
        lower_limit_constant_expression ]
```

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

The syntax for list of register variables is defined in 3.2.2.

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

A *time* register is used for storing and manipulating simulation time quantities in situations where timing checks are required and for diagnostics and debugging purposes. This data type is typically used in conjunction with the **\$time** system function (see Section 14).

Arrays of integer and time registers shall be declared in the same manner as arrays of reg type (see 3.8).

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

The time registers shall behave the same as a register of at least 64 bits. They shall be unsigned quantities, and unsigned arithmetic shall be performed on them. In contrast, integer registers shall be treated as signed quantities. Arithmetic operations performed on integer registers shall produce 2's complement results.

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

- Not all Verilog HDL operators can be used with real number values. See Table 4-3 for lists of valid and invalid operators for real numbers and real registers.
- Real registers shall not use range in the declaration
- Real registers shall default to an initial value of zero.

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

Examples:

```
integer a[1:64];           // an array of 64 integer values
time chng_hist[1:1000];    // an array of 1000 time values
real float ;              // a register to store real value
realtime rtime ;          // a register to store time as a real value
```

NOTE—Implementations may limit the maximum size of an **integer** variable, but they shall at least be 32 bits.

3.9.1 Operators and real numbers

The result of using logical or relational operators on real numbers and real registers is a single-bit scalar value. Not all Verilog HDL operators can be used with expressions involving real numbers and real registers. Table 4-3 lists the valid operators for use with real numbers and real registers. Real number constants and real registers are also prohibited in the following cases:

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

3.9.2 Conversion

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

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

See Section 14 for a discussion of system tasks that perform explicit conversion.

3.10 Parameters

Verilog HDL parameters do not belong to either the register or the net group. Parameters are not variables, they are constants.

The syntax for parameter declarations is given in Syntax 3-4.

```
parameter_declaration ::= parameter list_of_param_assignments ;
list_of_param_assignments ::= param_assignment { , param_assignment }
param_assignment ::= parameter_identifier = constant_expression
```

Syntax 3-4—Syntax for parameter declaration

The list of param assignments shall be a comma-separated list of assignments, where the right-hand side of the assignment shall be a constant expression; that is, an expression containing only constant numbers and previously defined parameters.

Parameters represent constants; hence, it is illegal to modify their value at runtime. However, parameters can be modified at compilation time to have values that are different from those specified in the declaration assignment. This allows customization of module instances. A parameter can be modified with the **defparam** statement or in the module instance statement. Typical uses of parameters are to specify delays and width of variables. See Section 12 for details on parameter value assignment.

Examples:

```
parameter msb = 7;           // defines msb as a constant value 7
parameter e = 25, f = 9;     // defines two constant numbers
```

```
parameter r = 5.7;           // declares r as a real parameter
parameter byte_size = 8,
       byte_mask = byte_size - 1;
parameter average_delay = (r + f) / 2;
```

3.11 Name spaces

In Verilog HDL, there are six name spaces; two are global and four are local. The global name spaces are *definitions* and *text macros*. The *definitions name space* unifies all the **module** (see 12.1), **macromodule** (see 12.1), and **primitive** (see 8.1) definitions. That is, a module and a macromodule or a primitive cannot have the same name.

The *text macro name space* is global. Since text macro names are introduced and used with a leading ` character, they remain unambiguous with any other name space (see 16.3). The text macro names are defined in the linear order of appearance in the set of input files that make up the description of the design unit. Subsequent definitions of the same name override the previous definitions for the balance of the input files.

There are four local name spaces: *block*, *module*, *port*, and *specify block*.

The *block name space* is introduced by the named block (see 9.8), function (see 10.3), and task (see 10.2) constructs. It unifies the definitions of the named blocks, functions, tasks, and the register type of declaration (see 3.2.2). The register type of declaration includes the **reg**, **integer**, **time**, **real**, **realtime**, **event**, and **parameter** declarations.

The *module name space* is introduced by the **module**, **macromodule**, and **primitive** constructs. It unifies the definition of functions, tasks, named blocks, instance names, net type of declaration, and register type of declaration. The net type of declaration includes **wire**, **wor**, **wand**, **tri**, **trior**, **triand**, **tri0**, **tri1**, **triereg**, **supply0**, and **supply1** (see 3.7).

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

The *specify block name space* is introduced by the **specify** construct (see 13.2). A **specparam** name can be defined and used only in the specify block name space. Any other type of name cannot be defined in this name space.

Section 4

Expressions

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

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

Some statement constructs require an expression to be a *constant expression*. The operands of a constant expression consist of constant numbers, parameter names, constant bit-selects of parameters, and constant part-selects of parameters only, but they can use any of the operators defined in Table 4-1.

A *scalar expression* is an expression that evaluates to a scalar (single-bit) result. If the expression evaluates to a vector (multibit) result, then the least significant bit of the result is used as the scalar result.

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

An *operand* can be one of the following:

- Constant number (including real)
- Net
- Register variables of type reg, integer, time, real, and realtime
- Net bit-select
- Bit-select of type reg, integer, and time
- Net part-select
- Part-select of type reg, integer, and time
- Memory word
- A call to a user-defined function or system-defined function that returns any of the above

4.1 Operators

The symbols for the Verilog HDL operators are similar to those in the C programming language. Table 4-1 lists these operators.

Table 4-1—Operators in the Verilog HDL

{}, {}	Concatenation, replication
+ - * /	Arithmetic
%	Modulus
> >= < <=	Relational

Table 4-1—Operators in the Verilog HDL (continued)

!	Logical negation
&&	Logical and
	Logical or
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality
~	Bit-wise negation
&	Bit-wise and
	Bit-wise inclusive or
^	Bit-wise exclusive or
^~ or ~^	Bit-wise equivalence
&	Reduction and
~&	Reduction nand
	Reduction or
~	Reduction nor
^	Reduction xor
~^ or ^~	Reduction xnor
<<	Left shift
>>	Right shift
? :	Conditional
or	Event or

4.1.1 Operators with real operands

The operators shown in Table 4-2 shall be legal when applied to real operands. All other operators shall be considered illegal when used with real operands.

Table 4-2—Legal operators for use in real expressions

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

Table 4-2—Legal operators for use in real expressions (continued)

?:	Conditional
or	Event or

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

Table 4-3 lists operators that shall not be used to operate on real numbers.

Table 4-3—Operators not allowed for real expressions

{}, {{}}	Concatenate, replicate
%	Modulus
==, !=	Case equality
~, &, ^, ^~, ^~^	Bit-wise
^, ^~, ^~^ &, ~&, , ~	Reduction
<<, >>	Shift

See 3.9.1 for more information on use of real numbers.

4.1.2 Binary operator precedence

The precedence order of *binary operators* and the *conditional operator* (?:) is shown in Table 4-4. The Verilog HDL has two equality operators. They are discussed in 4.1.8.

Table 4-4—Precedence rules for operators

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

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

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

$$A + B - C$$

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

$$A + B / C$$

Parentheses can be used to change the operator precedence.

$$(A + B) / C \quad // \text{ not the same as } A + B / C$$

4.1.3 Using integer numbers in expressions

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

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

A negative value for an integer with no base specifier shall be interpreted differently than for an integer with a base specifier. An integer with no base specifier shall be interpreted as a signed value in 2's complement form. An integer with a base specifier shall be interpreted as an unsigned value.

Example:

This example shows two ways to write the expression “minus 12 divided by 3.” Note that -12 and -'d12 both evaluate to the same 2's complement bit pattern, but, in an expression, the -'d12 loses its identity as a signed negative number.

```
integer IntA;  
IntA = -12 / 3;           // The result is -4.  
  
IntA = -'d 12 / 3;       // The result is 1431655761.
```

4.1.4 Expression evaluation order

The operators shall follow the associativity rules while evaluating an expression as described in 4.1.2. However, if the final result of an expression can be determined early, the entire expression need not be evaluated. This is called *short-circuiting* an expression evaluation.

Example:

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

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

4.1.5 Arithmetic operators

The binary arithmetic operators are given in Table 4-5.

Table 4-5—Arithmetic operators defined

$a + b$	a plus b
$a - b$	a minus b
$a * b$	a multiply by b
a / b	a divide by b
$a \% b$	a modulo b

The integer division shall truncate any fractional part toward zero. The modulus operator, for example $y \% z$, gives the remainder when the first operand is divided by the second, and thus is zero when z divides y exactly. The result of a modulus operation shall take the sign of the first operand.

The unary arithmetic operators shall take precedence over the binary operators. The unary operators are given in Table 4-6.

Table 4-6—Unary operators defined

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

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

Example:

Table 4-7 gives examples of modulus operations.

Table 4-7—Examples of modulus operations

Modulus expression	Result	Comments
$10 \% 3$	1	$10/3$ yields a remainder of 1
$11 \% 3$	2	$11/3$ yields a remainder of 2
$12 \% 3$	0	$12/3$ yields no remainder
$-10 \% 3$	-1	The result takes the sign of the first operand
$11 \% -3$	2	The result takes the sign of the first operand
$-4'd12 \% 3$	1	$-4'd12$ is seen as a large, positive number that leaves a remainder of 1 when divided by 3

4.1.6 Arithmetic expressions with registers and integers

An arithmetic operation on a `reg` type register shall be treated differently than an arithmetic operation on an integer data type. A `reg` data type shall be treated as an *unsigned* value and an integer data type shall be treated as a *signed* value. Thus, if a sized constant with a negative value is stored in a `reg` type register, a positive constant, which is a 2's complement of the sized constant, shall be the value stored in the `reg` type register. When this register is used in an

arithmetic expression, the positive constant shall be used as the value of the register. In contrast, if a sized constant with a negative value is stored in an integer type register and used in an arithmetic expression, the expression shall evaluate using signed arithmetic.

Table 4-8 lists how arithmetic operators interpret each data type.

Table 4-8—Data type interpretation by arithmetic operators

Data type	Interpretation
net	Unsigned
reg	Unsigned
integer	Signed, 2's complement
time	Unsigned
real, realtime	Signed, floating point

Example:

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

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

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

regA = -4'd12; // regA is 65524
intA = regA / 3; // expression result is 21841,
                // regA is a reg data type

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

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

4.1.7 Relational operators

Table 4-9 lists and defines the relational operators.

Table 4-9—Definitions of the relational operators

$a < b$	a less than b
$a > b$	a greater than b
$a \leq b$	a less than or equal to b
$a \geq b$	a greater than or equal to b

An expression using these *relational operators* shall yield the scalar value 0 if the specified relation is *false* or the value 1 if it is *true*. If, due to unknown or high-impedance bits in the operands, the relation is *ambiguous*, then the

result shall be a 1-bit unknown value (**x**).

When two operands of unequal bit lengths are used, the smaller operand shall be zero filled on the most significant bit side to extend to the size of the larger operand.

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

Examples:

The following examples illustrate the implications of this precedence rule:

```

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

```

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

4.1.8 Equality operators

The *equality operators* shall rank lower in precedence than the relational operators. Table 4-10 lists and defines the equality operators.

Table 4-10—Definitions of the equality operators

<code>a === b</code>	a equal to b, including x and z
<code>a !== b</code>	a not equal to b, including x and z
<code>a == b</code>	a equal to b, result may be unknown
<code>a != b</code>	a not equal to b, result may be unknown

All four equality operators shall have the same precedence. These four operators compare operands bit for bit, with zero filling if the two operands are of unequal bit length. As with the relational operators, the result shall be 0 if comparison fails, 1 if it succeeds.

For the *logical equality* and *logical inequality* operators (`==` and `!=`), if either operand contains an **x** or a **z**, then the result shall be the unknown value (**x**).

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

4.1.9 Logical operators

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

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

Examples:

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

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

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

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

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

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

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

```
if (!inword)
```

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

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

4.1.10 Bit-wise operators

The *bit-wise operators* shall perform bit-wise manipulations on the operands—that is, the operator shall combine a bit in one operand with its corresponding bit in the other operand to calculate one bit for the result. Logic tables 4-11 through 4-15 show the results for each possible calculation.

Table 4-11—Bit-wise binary and operator

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

Table 4-13—Bit-wise binary exclusive or operator

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

Table 4-12—Bit-wise binary or operator

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

Table 4-14—Bit-wise binary exclusive nor operator

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

Table 4-15—Bit-wise unary negation operator

~	
0	1
1	0
x	x
z	x

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

4.1.11 Reduction operators

The *unary reduction operators* shall perform a bit-wise operation on a single operand to produce a single bit result. For *reduction and*, *reduction or*, and *reduction xor* operators, the first step of the operation shall apply the operator between the first bit of the operand and the second using logic tables 4-16 through 4-18. The second and subsequent steps shall apply the operator between the 1-bit result of the prior step and the next bit of the operand using the same logic table. For *reduction nand*, *reduction nor*, and *reduction xnor* operators, the result shall be computed by inverting the result of the reduction and, reduction or, and reduction xor operation respectively.

Table 4-16—Reduction unary and operator

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

Table 4-17—Reduction unary or operator

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

Table 4-18—Reduction unary exclusive or operator

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

Example:

Table 4-19 shows the results of applying reduction operators on different operands.

Table 4-19—Results of unary reduction operations

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

4.1.12 Shift operators

The *shift operators*, << and >>, shall perform left and right shifts of their left operand by the number of bit positions given by the right operand. Both shift operators shall fill the vacated bit positions with zeroes. If the right operand has an unknown or high-impedance value, then the result shall be unknown. The right operand is always treated as an unsigned number.

Example:

```

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

```

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

4.1.13 Conditional operator

The *conditional operator*, also known as *ternary operator*, shall be right associative and shall be constructed using three operands separated by two operators in the format given in Syntax 4-1.

$$\text{conditional_expression} ::= \text{expression1} \text{ ? } \text{expression2} \text{ : } \text{expression3}$$

Syntax 4-1—Syntax for conditional operator

The evaluation of a conditional operator shall begin with the evaluation of expression1. If expression1 evaluates to false (0), then expression3 shall be evaluated and used as the result of the conditional expression. If expression1 evaluates to true (known value other than 0), then expression2 is evaluated and used as the result. If expression1 evaluates to ambiguous value (x or z), then both expression2 and expression3 shall be evaluated and their results shall be combined, bit by bit, using Table 4-20 to calculate the final result unless expression2 or expression3 is real, in which case the result shall be 0. If the lengths of expression2 and expression3 are different, the shorter operand shall be lengthened to match the longer and zero-filled from the left (the high-order end).

Table 4-20—Ambiguous condition results for conditional operator

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

Example:

The following example of a tri-state output bus illustrates a common use of the conditional operator.

```

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

```

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

4.1.14 Concatenations

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

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

Examples:

This example concatenates four expressions:

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

and it is equivalent to the following example:

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

Concatenations can be expressed using a repetition multiplier as shown in this example:

```
{4{w}} // This is equivalent to {w, w, w, w}
```

If a repetition multiplier is used on a function call operand, the function need not be evaluated multiple times. For example:

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

may be computed as

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

or

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

This is another form of expression evaluation short-circuiting.

The next example illustrates nested concatenations:

```
{b, {3{a, b}}} // This is equivalent to {b, a, b, a, b, a, b}
```

The repetition multiplier shall be a constant expression.

4.1.15 Event or

The event **or** operator shall perform an or of events. See 9.7 for events and triggering of events.

Example:

The following example shows an assignment to `rega` when an event (change) occurs on `trig` or `enable`.

```
@(trig or enable) rega = regb ;
```

4.2 Operands

There are several types of operands that can be specified in expressions. The simplest type is a reference to a net or register in its complete form—that is, just the name of the net or register is given. In this case, all of the bits making up the net or register value shall be used as the operand.

If a single bit of a vector net or register is required, then a bit-select operand shall be used. A part-select operand shall be used to reference a group of adjacent bits in a vector net or register.

A memory word can be referenced as an operand. A concatenation of other operands (including nested concatenations) can be specified as an operand. A function call is an operand.

4.2.1 Net and register bit-select and part-select addressing

Bit-selects extract a particular bit from a vector net or register. The bit can be addressed using an expression. If the bit-select is out of the address bounds or the bit-select is **x** or **z**, then the value returned by the reference shall be **x**.

Several contiguous bits in a vector register or net can be addressed and are known as *part-selects*. A part-select of a vector register or net is given with the following syntax:

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

Both expressions shall be constant expressions. The first expression has to address a more significant bit than the second expression. If the part-select is out of the address bounds or the part-select is **x** or **z**, then the value returned by the reference shall be **x**.

The bit-select or part-select of a register declared as **real** or **realtime** shall be considered illegal.

Examples:

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

```
acc[index]
```

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

```
reg [15:0] acc;
reg [1:16] acc;
```

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

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

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

NOTES

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

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

4.2.2 Memory addressing

Declaration of memory is discussed in 3.8. This subclause discusses memory addressing.

Examples:

The next example declares a memory of 1024 8-bit words:

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

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

```
mem_name [ addr_expr ]
```

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

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

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

NOTE—There is no mechanism to express bit-selects or part-selects of memory words directly. If this is required, then the memory word has to be first transferred to an appropriately sized temporary register.

4.2.3 Strings

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

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

Example:

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