

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

<p>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.</p>

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 Girczyc
 Rita A. Glover
 Timothy G. Goldsbury
 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
 Kareem 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 Rüschi
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:

Unsize constant numbers

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

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*ed and *scalare*d 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 **vector**ed 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.

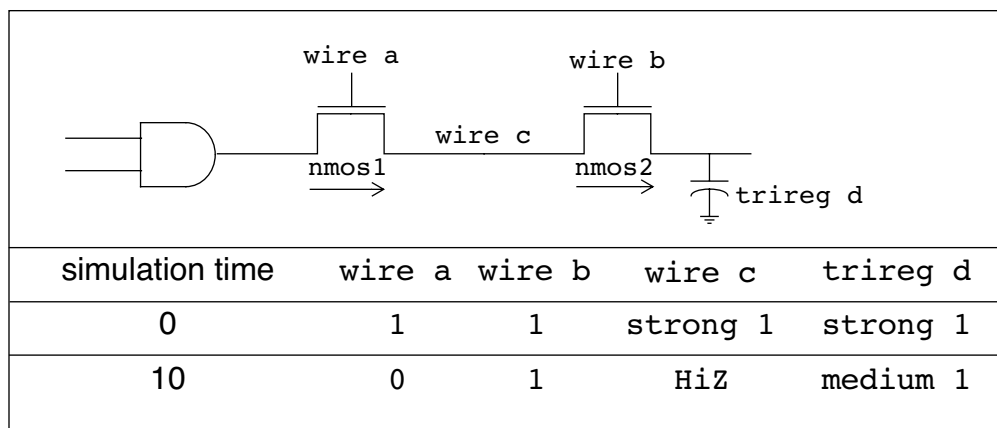


Figure 3-1 — Simulation values of a trireg and its driver

- 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.

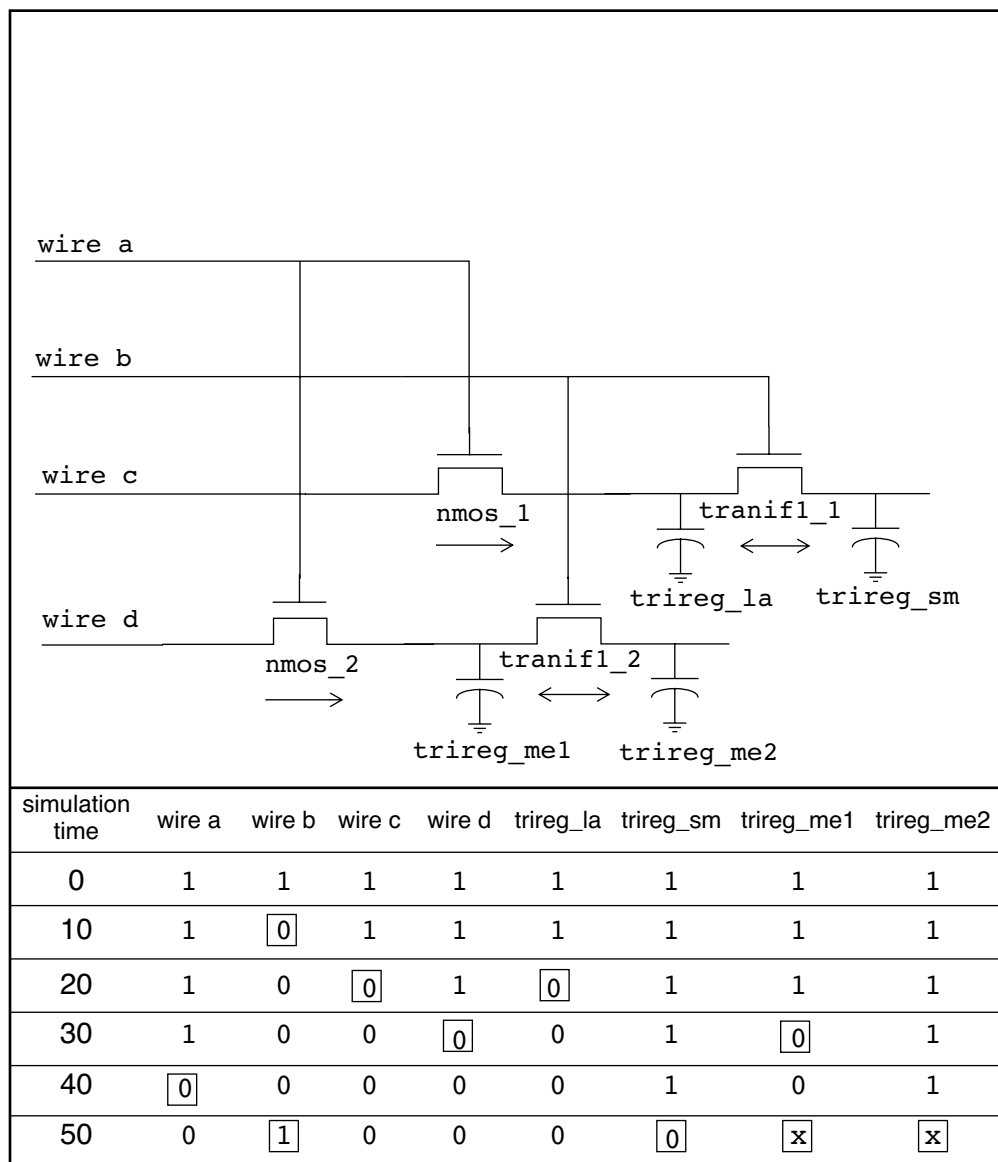


Figure 3-2—Simulation results of a capacitive network

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.

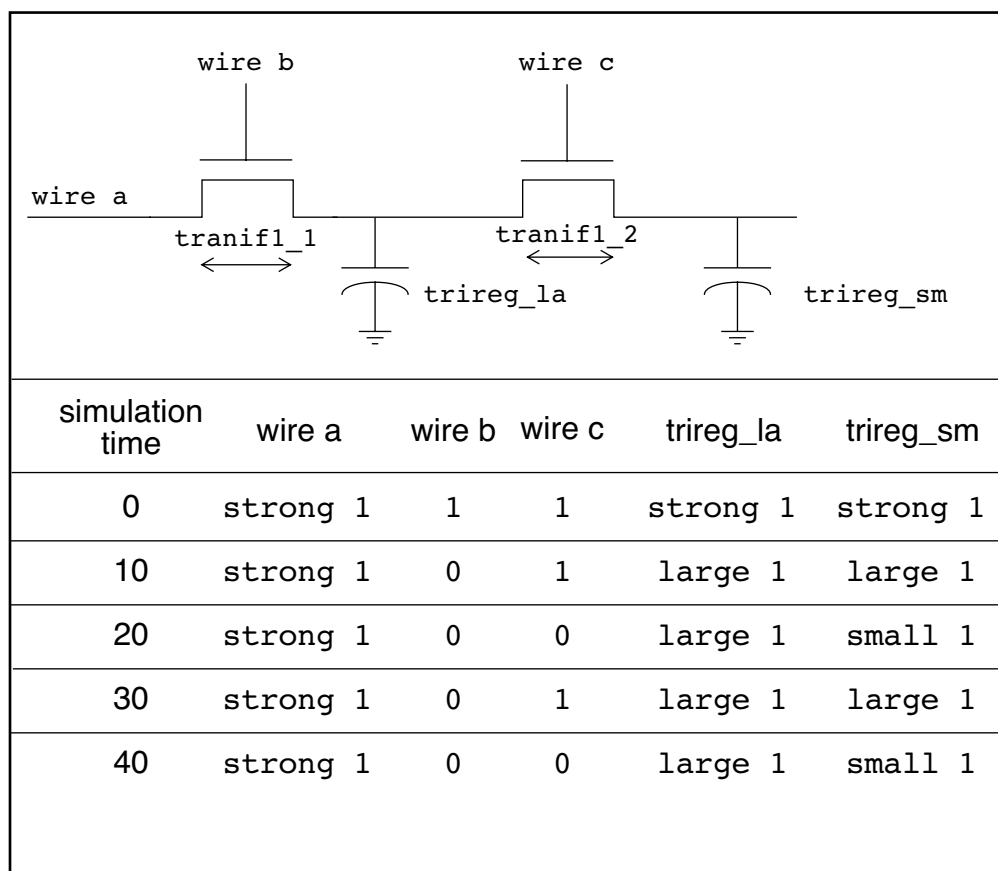


Figure 3-3—Simulation results of charge sharing

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` `// 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

$\wedge\sim$ $\sim\wedge$	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.

conditional_expression ::= expression1 ? expression2 : 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.

```

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

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

```

The result of simulating the above description is

```

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

```

4.2.3.1 String operations

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

When manipulating string values in vector variables, at least $8 * n$ bits shall be required in the vector, where n is the number of characters in the string.

4.2.3.2 String value padding and potential problems

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

Examples:

The following example illustrates the potential problem.

```

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

```

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

```

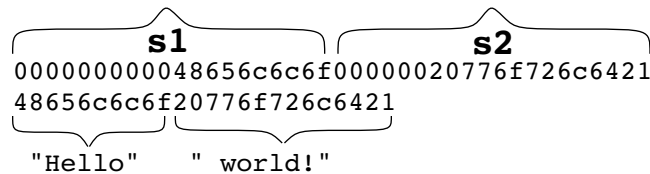
s1 = 000000000048656c6c6f
s2 = 00000020776f726c6421

```

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

```
000000000048656c6c6f00000020776f726c6421
```

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



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

4.2.3.3 Null string handling

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

4.3 Minimum, typical, and maximum delay expressions

Verilog HDL delay expressions can be specified as three expressions separated by colons. This is intended to represent minimum, typical, and maximum values—in that order. The syntax is given in Syntax 4-2.

```
mintypmax_expression ::=
    expression
    | expression : expression : expression
```

Syntax 4-2—Syntax for mintypmax expression

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

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

Examples:

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

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

Example 2—The next example shows some typical expressions that are used to specify min:typ:max format values:

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

4.4 Expression bit lengths

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

For example, should an arithmetic add of two 16-bit values perform the evaluation using 16 bits, or should the evaluation use 17 bits in order to allow for a possible carry overflow? The answer depends on the type of device being modeled, and whether that device handles carry overflow. The Verilog HDL uses the bit length of the operands to determine how many bits to use while evaluating an expression. The bit length rules are given in 4.4. In the case of the addition operator, the bit length of the largest operand, including the left-hand side of an assignment, shall be used.

Examples:

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

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

4.4.1 Rules for expression bit lengths

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

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

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

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

Table 4-21 shows how the form of an expression shall determine the bit lengths of the results of the expression. In Table 4-21, *i*, *j*, and *k* represent expressions of an operand, and $L(i)$ represents the bit length of the operand represented by *i*.

Table 4-21 — Bit lengths resulting from expressions

Expression	Bit length	Comments
Unsize constant number	Same as integer	
Sized constant number	As given	
<i>i op j</i> , where op is: + - / % & ^ ~ ^	$\max(L(i), L(j))$	
<i>i * j</i>	$L(i) + L(j)$	
op <i>i</i> , where op is: +, -, ~	$L(i)$	

Table 4-21 — Bit lengths resulting from expressions (continued)

Expression	Bit length	Comments
i op j, where op is: == != && > >= < <=	1 bit	All operands are self-determined
op i, where op is: & ~& ~ ^ ~^ ^~	1 bit	All operands are self-determined
i op j, where op is: >>, <<	L(i)	j is self-determined
i ? j : k	max(L(j),L(k))	i is self-determined
{i,...,j}	L(i)+..+L(j)	All operands are self-determined
{i{j,...,k}}	i * (L(j)+..+L(k))	All operands are self-determined

4.4.2 An example of an expression bit-length problem

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

Given the following declarations

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

The intent is to evaluate the expression

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

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

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

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

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

Section 5

Scheduling semantics

5.1 Execution of a model

The balance of the sections of this standard describe the behavior of each of the elements of the language. This section gives an overview of the interactions between these elements, especially with respect to the scheduling and execution of events.

The elements that make up the Verilog HDL can be used to describe the behavior, at varying levels of abstraction, of electronic hardware. An HDL has to be a parallel programming language. The execution of certain language constructs is defined by parallel execution of blocks or processes. It is important to understand what execution order is guaranteed to the user, and what execution order is indeterminate.

Although the Verilog HDL is used for more than simulation, the semantics of the language are defined for simulation, and everything else is abstracted from this base definition.

5.2 Event simulation

The Verilog HDL is defined in terms of a discrete event execution model. The discrete event simulation is described in more detail in this section to provide a context to describe the meaning and valid interpretation of Verilog HDL constructs. These resulting definitions provide the standard Verilog reference model for simulation, which all compliant simulators shall implement. Note, though, that there is a great deal of choice in the definitions that follow, and differences in some details of execution are to be expected between different simulators. In addition, Verilog HDL simulators are free to use different algorithms than those described in this section, provided the user-visible effect is consistent with the reference model.

A design consists of connected threads of execution or processes. Processes are objects that can be evaluated, that may have state, and that can respond to changes on their inputs to produce outputs. Processes include primitives, modules, initial and always procedural blocks, continuous assignments, asynchronous tasks, and procedural assignment statements.

Every change in value of a net or register in the circuit being simulated, as well as the named event, is considered an *update event*.

Processes are sensitive to update events. When an update event is executed, all the processes that are sensitive to that event are evaluated in an arbitrary order. The evaluation of a process is also an event, known as an *evaluation event*.

In addition to events, another key aspect of a simulator is time. The term *simulation time* is used to refer to the time value maintained by the simulator to model the actual time it would take for the circuit being simulated. The term *time* is used interchangeably with simulation time in this section.

Events can occur at different times. In order to keep track of the events and to make sure they are processed in the correct order, the events are kept on an *event queue*, ordered by simulation time. Putting an event on the queue is called *scheduling an event*.

5.3 The stratified event queue

The Verilog event queue is logically segmented into five different regions. Events are added to any of the five regions but are only removed from the *active* region.

- 1) Events that occur at the current simulation time and can be processed in any order. These are the *active* events.
- 2) Events that occur at the current simulation time, but that shall be processed after all the active events are processed. These are the *inactive* events.
- 3) Events that have been evaluated during some previous simulation time, but that shall be assigned at this simulation time after all the active and inactive events are processed. These are the *nonblocking assign update* events.
- 4) Events that shall be processed after all the active, inactive, and nonblocking assign update events are processed. These are the *monitor* events.
- 5) Events that occur at some future simulation time. These are the *future* events. Future events are divided into *future inactive events*, and *future nonblocking assignment update events*.

The processing of all the active events is called a *simulation cycle*.

The freedom to choose any active event for immediate processing is an essential source of nondeterminism in the Verilog HDL.

An *explicit zero delay* (#0) requires that the process be suspended and added as an inactive event for the current time so that the process is resumed in the next simulation cycle in the current time.

A nonblocking assignment (see 9.2.2) creates a nonblocking assign update event, scheduled for current or a later simulation time.

The **\$monitor** and **\$strobe** system tasks (see 14.1) create monitor events for their arguments. These events are continuously re-enabled in every successive time step. The monitor events are unique in that they cannot create any other events.

The call back procedures scheduled with PLI routines such as `tf_synchronize()` (see 21.58) or `vpi_register_cb(cb_readwrite)` (see 23.24) shall be treated as inactive events.

5.4 The Verilog simulation reference model

In all the examples that follow, T refers to the current simulation time, and all events are held in the event queue, ordered by simulation time.

```
while (there are events) {
    if (no active events) {
        if (there are inactive events) {
            activate all inactive events;
        } else if (there are nonblocking assign update events) {
            activate all nonblocking assign update events;
        } else if (there are monitor events) {
            activate all monitor events;
        } else {
            advance T to the next event time;
            activate all inactive events for time T;
        }
    }
}
```

```

    E = any active event;
    if (E is an update event) {
        update the modified object;
        add evaluation events for sensitive processes to event queue;
    } else { /* shall be an evaluation event */
        evaluate the process;
        add update events to the event queue;
    }
}

```

5.4.1 Determinism

This standard guarantees a certain scheduling order.

- 1) Statements within a **begin–end** block shall be executed in the order in which they appear in that **begin–end** block. Execution of statements in a particular **begin–end** block can be suspended in favor of other processes in the model; however, in no case shall the statements in a **begin–end** block be executed in any order other than that in which they appear in the source.
- 2) Nonblocking assignments shall be performed in the order the statements were executed. Consider the following example:

```

initial begin
    a <= 0;
    a <= 1;
end

```

When this block is executed, there will be two events added to the nonblocking assign update queue. The previous rule requires that they be entered on the queue in source order; this rule requires that they be taken from the queue and performed in source order as well. Hence, at the end of time step 1, the variable **a** will be assigned 0 and then 1.

5.4.2 Nondeterminism

One source of nondeterminism is the fact that active events can be taken off the queue and processed in any order. Another source of nondeterminism is that statements without time-control constructs in behavioral blocks do not have to be executed as one event. Time control statements are the **#** expression and **@** expression constructs (see 9.7). At any time while evaluating a behavioral statement, the simulator may suspend execution and place the partially completed event as a pending active event on the event queue. The effect of this is to allow the interleaving of process execution. Note that the order of interleaved execution is nondeterministic and not under control of the user.

5.5 Race conditions

Because the execution of expression evaluation and net update events may be intermingled, race conditions are possible:

```

assign p = q;
initial begin
    q = 1;
    #1 q = 0;
    $display(p);
end

```

The simulator is correct in displaying either a 1 or a 0. The assignment of 0 to **q** enables an update event for **p**. The

simulator may either continue and execute the \$display task or execute the update for *p*, followed by the \$display task.

5.6 Scheduling implication of assignments

Assignments are translated into processes and events as follows.

5.6.1 Continuous assignment

A continuous assignment statement (Section 6) corresponds to a process, sensitive to the source elements in the expression. When the value of the expression changes, it causes an active update event to be added to the event queue, using current values to determine the target.

5.6.2 Procedural continuous assignment

A procedural continuous assignment (which are the **assign** or **force** statement; see 9.3) corresponds to a process that is sensitive to the source elements in the expression. When the value of the expression changes, it causes an active update event to be added to the event queue, using current values to determine the target.

A **deassign** or a **release** statement deactivates any corresponding **assign** or **force** statement(s).

5.6.3 Blocking assignment

A blocking assignment statement with a delay computes the right-hand side value using the current values, then causes the executing process to be suspended and scheduled as a future event. If the delay is 0, the process is scheduled as an inactive event for the current time.

When the process is returned (or if it returns immediately if no delay is specified), the process performs the assignment to the left-hand side and enables any events based upon the update of the left-hand side. The values at the time the process resumes are used to determine the target(s). Execution may then continue with the next sequential statement or with other active events.

5.6.4 Nonblocking assignment

A nonblocking assignment statement always computes the updated value and schedules the update as a nonblocking assign update event, either in this time step if the delay is zero or as a future event if the delay is nonzero. The values in effect when the update is placed on the event queue are used to compute both the right-hand value and the left-hand target.

5.6.5 Switch (transistor) processing

The event-driven simulation algorithm described in 5.4 depends on unidirectional signal flow and can process each event independently. The inputs are read, the result is computed, and the update is scheduled.

The Verilog HDL provides switch-level modeling in addition to behavioral and gate-level modeling. Switches provide bi-directional signal flow and require coordinated processing of nodes connected by switches.

The Verilog HDL source elements that model switches are various forms of transistors, called **tran**, **tranif0**, **tranif1**, **rtran**, **rtranif0**, and **rtranif1**.

Switch processing shall consider all the devices in a bidirectional switch-connected net before it can determine the appropriate value for any node on the net, because the inputs and outputs interact. A simulator can do this using a relaxation technique. The simulator can process **tran** at any time. It can process a subset of **tran**-connected events at a particular time, intermingled with the execution of other active events.

Further refinement is required when some transistors have gate value *x*. A conceptually simple technique is to solve

the network repeatedly with these transistors set to all possible combinations of fully conducting and nonconducting transistors. Any node that has a unique logic level in all cases has steady-state response equal to this level. All other nodes have steady-state response x .

5.6.6 Port connections

Ports connect processes through implicit continuous assignment statements or implicit bidirectional connections. Bidirectional connections are analogous to an always-enabled tran connection between the two nets, but without any strength reduction. Port connection rules require that a value receiver be a net or a structural net expression.

Ports can always be represented as declared objects connected as follows:

- If an input port, then a continuous assignment from an outside expression to a local (input) net
- If an output port, then a continuous assignment from a local output expression to an outside net
- If an inout, then a nonstrength-reducing transistor connecting the local net to an outside net

Port connection rules are given in 12.3.7. Modules can have the following declaration:

```
module foo (.a(p), .b(p));
```

which makes *a* and *b* the external names of the ports and *p* the internal name of the port. This means *a*, *b*, and *p* are connected bidirectionally; hence, they always have the same value at the end of the time step.

5.6.7 Functions and tasks

Task and function parameter passing is by value, and it copies in on invocation and copies out on return. The copy out on the return function behaves in the same manner as does any blocking assignment.

Section 6

Assignments

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

- The *continuous assignment*, which assigns values to *nets*
- The *procedural assignment*, which assigns values to *registers*

There are two additional forms of assignments, called *procedural continuous assignments*, described in 9.3.

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

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

Statement type	Left-hand side (LHS)
Continuous assignment	Net (vector or scalar) Constant bit select of a vector net Constant part select of a vector net Concatenation of any of the above three LHS
Procedural assignment	Register (vector or scalar) Bit-select of a vector register Constant part select of a vector register Memory word Concatenation of any of the above four LHS

6.1 Continuous assignments

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

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

```

net_declaration ::=
    net_type [ vectored | scalared ] [range] [delay3] list_of_net_identifiers ;
    | triereg [ vectored | scalared ] [charge_strength] [range] [delay3]
    list_of_net_identifiers ;
    | net_type [ vectored | scalared ] [drive_strength] [range] [delay3]
    list_of_net_assignments ;

continuous_assignment ::=
    assign [ drive_strength ] [ delay3 ] list_of_net_assignments ;
list_of_net_assignments ::= net_assignment { , net_assignment }
net_assignment ::= net_lvalue = expression

```

Syntax 6-1—Syntax for continuous assignment

6.1.1 The net declaration assignment

The first two alternatives in the net declaration are discussed in Section 3 (see 3.2). The third alternative, the net declaration assignment, allows a continuous assignment to be placed on a net in the same statement that declares the net.

Example:

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

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

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

6.1.2 The continuous assignment statement

The continuous assignment statement shall place a continuous assignment on a net that has been previously declared, either explicitly by declaration or implicitly by using its name in the terminal list of a gate, user-defined primitive, or module instance (see 3.5).

Assignments on nets shall be continuous and automatic. This means that whenever an operand in the right-hand side expression changes value, the whole right-hand side shall be evaluated and assigned to the left-hand side.

Examples:

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

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

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

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

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

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

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

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

6.1.3 Delays

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

If the left-hand side references a vector net, then up to three delays can be applied. The following rules determine which delay controls the assignment:

- If the right-hand side makes a transition from nonzero to zero, then the falling delay shall be used.

- If the right-hand side makes a transition to **z**, then the turn-off delay shall be used.
- For all other cases, the rising delay shall be used.

Specifying the delay in a continuous assignment that is part of the net declaration shall be treated differently from specifying a net delay and then making a continuous assignment to the net. A delay value can be applied to a net in a net declaration, as in the following example:

```
wire #10 wireA;
```

This syntax, called a *net delay*, means that any value change that is to be applied to **wireA** by some other statement shall be delayed for ten time units before it takes effect. When there is a continuous assignment in a declaration, the delay is part of the continuous assignment and is *not* a net delay. Thus, it shall not be added to the delay of other drivers on the net. Furthermore, if the assignment is to a vector net, then the rising and falling delays shall not be applied to the individual bits if the assignment is included in the declaration.

In situations where a right-hand side operand changes before a previous change has had time to propagate to the left-hand side, then the latest value change shall be the only one to be applied. That is, only one assignment shall occur.

6.1.4 Strength

The driving strength of a continuous assignment can be specified by the user. This applies only to assignments to scalar nets of the following types:

wire	tri	triereg
wand	triand	tri0
wor	trior	tri1

Continuous assignments driving strengths can be specified in either a net declaration or in a stand-alone assignment, using the **assign** keyword. The strength specification, if provided, shall immediately follow the keyword (either the keyword for the net type or **assign**) and precede any delay specified. Whenever the continuous assignment drives the net, the strength of the value shall be simulated as specified.

A drive strength specification shall contain one strength value that applies when the value being assigned to the net is 1 and a second strength value that applies when the assigned value is 0. The following keywords shall specify the strength value for an assignment of 1:

supply1	strong1	pull1	weak1	highz1
----------------	----------------	--------------	--------------	---------------

The following keywords shall specify the strength value for an assignment of 0:

supply0	strong0	pull0	weak0	highz0
----------------	----------------	--------------	--------------	---------------

The order of the two strength specifications shall be arbitrary. The following two rules shall constrain the use of drive strength specifications:

- The strength specifications (**highz1**, **highz0**) and (**highz0**, **highz1**) shall be treated as illegal constructs.
- If drive strength is not specified, it shall default to (**strong1**, **strong0**).

6.2 Procedural assignments

The primary discussion of procedural assignments is in 9.2. However, a description of the basic ideas in this clause highlights the differences between continuous assignments and procedural assignments.

As stated in 6.1, continuous assignments drive nets in a manner similar to the way gates drive nets. The expression on the right-hand side can be thought of as a combinatorial circuit that drives the net continuously. In contrast, proce-

dural assignments put values in registers. The assignment does not have duration; instead, the register holds the value of the assignment until the next procedural assignment to that register.

Procedural assignments occur within procedures such as **always**, **initial** (see Section 9), **task**, and **function** (see Section 10) and can be thought of as “triggered” assignments. The trigger occurs when the flow of execution in the simulation reaches an assignment within a procedure. Reaching the assignment can be controlled by conditional statements. Event controls, delay controls, **if** statements, **case** statements, and looping statements can all be used to control whether assignments are evaluated. Section 9 gives details and examples.

Section 7

Gate and switch level modeling

This section describes the syntax and semantics of these built-in primitives and how a hardware design can be described using these primitives.

There are 14 logic gates and 12 switches predefined in the Verilog HDL to provide the *gate* and *switch* level modeling facility. Modeling with logic gates and switches has the following advantages:

- Gates provide a much closer one-to-one mapping between the actual circuit and the model.
- There is no continuous assignment equivalent to the bidirectional transfer gate.

7.1 Gate and switch declaration syntax

Syntax 7-1 shows the gate and switch declaration syntax.

A gate or a switch instance declaration shall have the following specifications:

- The keyword that names the type of gate or switch primitive
- An optional *drive strength*
- An optional *propagation delay*
- An optional identifier that names each gate or switch instance
- An optional range for *array of instances*
- The terminal connection list

Multiple instances of the one type of gate or switch primitive can be declared as a comma-separated list. All such instances shall have the same drive strength and delay specification.


```

gate_instantiation ::=
    n_input_gatetype [drive_strength] [delay2]
        n_input_gate_instance { , n_input_gate_instance } ;
    | n_output_gatetype [drive_strength] [delay2]
        n_output_gate_instance { , n_output_gate_instance } ;
    | enable_gatetype [drive_strength] [delay3] enable_gate_instance { ,
        enable_gate_instance } ;
    | mos_switchtype [delay3] mos_switch_instance { , mos_switch_instance } ;
    | pass_switchtype pass_switch_instance { , pass_switch_instance } ;
    | pass_en_switchtype [delay3] pass_en_switch_instance { , pass_en_switch_instance
    } ;
    | cmos_switchtype [delay3] cmos_switch_instance { , cmos_switch_instance } ;
    | pullup [pullup_strength] pull_gate_instance { , pull_gate_instance } ;
    | pulldown [pulldown_strength] pull_gate_instance { , pull_gate_instance } ;
n_input_gate_instance ::=
    [name_of_gate_instance] ( output_terminal , input_terminal { , input_terminal } )
n_output_gate_instance ::=
    [name_of_gate_instance] ( output_terminal { , output_terminal } , input_terminal )
enable_gate_instance ::=
    [name_of_gate_instance] ( output_terminal , input_terminal , enable_terminal )
mos_switch_instance ::=
    [name_of_gate_instance] ( output_terminal , input_terminal , enable_terminal )
pass_switch_instance ::=
    [name_of_gate_instance] ( inout_terminal , inout_terminal )
pass_enable_switch_instance ::=
    [name_of_gate_instance] ( inout_terminal , inout_terminal , enable_terminal )
cmos_switch_instance ::=
    [name_of_gate_instance] ( output_terminal , input_terminal ,
        ncontrol_terminal , pcontrol_terminal )
pull_gate_instance ::=
    [name_of_gate_instance] ( output_terminal )
name_of_gate_instance ::= gate_instance_identifier [range]
delay2 ::=
    # delay_value | # ( delay_value [ , delay_value ] )
pullup_strength ::=
    ( strength0 , strength1 ) | ( strength1 , strength0 ) | ( strength1 )
pulldown_strength ::=
    ( strength0 , strength1 ) | ( strength1 , strength0 ) | ( strength0 )
input_terminal ::= scalar_expression
enable_terminal ::= scalar_expression
ncontrol_terminal ::= scalar_expression
pcontrol_terminal ::= scalar_expression
output_terminal ::= terminal_identifier | terminal_identifier [ constant_expression ]
inout_terminal ::= terminal_identifier | terminal_identifier [ constant_expression ]
n_input_gatetype ::= and | nand | or | nor | xor | xnor
n_output_gatetype ::= buf | not
enable_gatetype ::= bufif0 | bufif1 | notif0 | notif1
mos_switchtype ::= nmos | pmos | rnmos | rpmos
pass_switchtype ::= tran | rtran
pass_en_switchtype ::= tranif0 | tranif1 | rtranif1 | rtranif0

```

Syntax 7-1 — Syntax for gate instantiation

7.1.1 The gate type specification

A gate or switch instance declaration shall begin with the keyword that specifies the gate or switch primitive being used by the instances that follow in the declaration. Table 7-1 lists the keywords that shall begin a gate or a switch instance declaration.

Table 7-1—Built-in gates and switches

n_input gates	n_output gates	tristate gates	pull gates	MOS switches	bidirectional switches
and	buf	bufif0	pulldown	cmos	rtran
nand	not	bufif1	pullup	nmos	rtranif0
nor		notif0		pmos	rtranif1
or		notif1		rcmos	tran
xnor				rnmos	tranif0
xor				rpmos	tranif1

Explanations of the built-in gates and switches shown in Table 7-1 begin in 7.2.

7.1.2 The drive strength specification

An optional drive strength specification shall specify the *strength* of the logic values on the output terminals of the gate instance. Only the instances of the gate primitives shown in Table 7-2 can have the drive strength specification.

Table 7-2—Valid gate types for strength specifications

and	nand	buf	not	pulldown
or	nor	bufif0	notif0	pullup
xor	xnor	bufif1	notif1	

The drive strength specification for a gate instance, with the exception of **pullup** and **pulldown**, shall have a *strength1* specification and a *strength0* specification. The *strength1* specification shall specify the strength of signals with a logic value 1, and the *strength0* specification shall specify the strength of signals with a logic value 0. The strength specification shall follow the gate type keyword and precede any delay specification. The *strength0* specification can precede or follow the *strength1* specification. The *strength1* and *strength0* specifications shall be separated by a comma and enclosed within a pair of parentheses.

The **pullup** gate can have only *strength1* specification; *strength0* specification shall be optional. The **pulldown** gate can have only *strength0* specification; *strength1* specification shall be optional.

The *strength1* specification shall be one of the following keywords:

supply1 strong1 pull1 weak1

The *strength0* specification shall be one of the following keywords:

supply0 strong0 pull0 weak0

Specifying **highz1** as *strength1* shall cause the gate or switch to output a logic value z in place of a 1. Specifying **highz0** shall cause the gate to output a logic value z in place of a 0. The strength specifications (**highz0**, **highz1**) and (**highz1**, **highz0**) shall be considered invalid.

In the absence of a strength specification, the instances shall have the default strengths **strong1** and **strong0**.

Example:

The following example shows a drive strength specification in a declaration of an open collector **nor** gate:

```
nor (highz1,strong0) n1(out1,in1,in2);
```

In this example, the **nor** gate outputs a **z** in place of a **1**.

Logic strength modeling is discussed in more detail in 7.10 through 7.14.

7.1.3 The delay specification

An optional delay specification shall specify the propagation delay through the gates and switches in a declaration. Gates and switches in declarations with no delay specification shall have no propagation delay. A delay specification can contain up to three delay values, depending on the gate type. The **pullup** and **pulldown** instance declarations shall not include delay specifications. Delays are discussed in more detail in 7.15.

7.1.4 The primitive instance identifier

An optional name can be given to a gate or switch instance. If multiple instances are declared as an array of instances, an identifier shall be used to name the instances.

7.1.5 The range specification

There are many situations when repetitive instances are required. These instances shall differ from each other only by the index of the vector to which they are connected.

In order to specify an array of instances, the instance name shall be followed by the range specification. The range shall be specified by two constant expressions, left-hand index (**lhi**) and right-hand index (**rhi**), separated by a colon and enclosed within a pair of square brackets. A [**lhi**:**rhi**] range specification shall represent an array of **abs(lhi-rhi)+1** instances. Neither of the two constant expressions are required to be zero, and **lhi** is not required to be larger than **rhi**. If both constant expressions are equal, only one instance shall be generated.

An array of instances shall have a continuous range. One instance identifier shall be associated with only one range to declare an array of instances.

The range specification shall be optional. If no range specification is given, a single instance shall be created.

Example:

A declaration shown below is illegal:

```
nand #2 t_nand[0:3] ( ... ), t_nand[4:7] ( ... );
```

It could be declared correctly as one array of eight instances, or two arrays with unique names of four elements each:

```
nand #2 t_nand[0:7]( ... );  
nand #2 x_nand[0:3] ( ... ), y_nand[4:7] ( ... );
```

7.1.6 Primitive instance connection list

The terminal list describes how the gate or switch connects to the rest of the model. The gate or switch type can limit these expressions. The connection list shall be enclosed in a pair of parentheses, and the terminals shall be separated by commas. The output or bidirectional terminals shall always come first in the terminal list, followed by the input

terminals.

The terminal connections for an array of instances shall follow these rules:

- The bit length of each port expression in the declared instance-array shall be compared with the bit length of each single-instance port or terminal in the instantiated module or primitive.
- For each port or terminal where the bit length of the instance-array port expression is the same as the bit length of the single-instance port, the instance-array port expression shall be connected to each single-instance port.
- If bit lengths are different, each instance shall get a part-select of the port expression as specified in the range, starting with the right-hand index.
- Too many or too few bits to connect to all the instances shall be considered an error.

An individual instance from an array of instances shall be referenced in the same manner as referencing an element of an array of registers.

Examples:

Example 1—The following declaration of `nand_array` declares four instances that can be referenced by `nand_array[1]`, `nand_array[2]`, `nand_array[3]`, and `nand_array[4]` respectively.

```
nand #2 nand_array[1:4]( ... ) ;
```

Example 2—The two module descriptions that follow are equivalent except for indexed instance names, and they demonstrate the range specification and connection rules for declaring an array of instances:

```
module driver (in, out, en);
input [3:0] in;
output [3:0] out;
input en;

bufif0 ar[3:0] (out, in, en); // array of tri-state buffers

endmodule

module driver_equiv (in, out, en);
input [3:0] in;
output [3:0] out;
input en;

bufif0 ar3 (out[3], in[3], en); // each buffer declared separately
bufif0 ar2 (out[2], in[2], en);
bufif0 ar1 (out[1], in[1], en);
bufif0 ar0 (out[0], in[0], en);

endmodule
```

Example 3—The two module descriptions that follow are equivalent except for indexed instance names, and they demonstrate how different instances within an array of instances are connected when the port sizes do not match.

```

module busdriver (busin, bushigh, buslow, enh, enl);
input [15:0] in;
output [7:0] bushigh, buslow;
input enh, enl;

  driver busar3 (busin[15:12], bushigh[7:4], enh);
  driver busar2 (busin[11:8], bushigh[3:0], enh);
  driver busar1 (busin[7:4], buslow[7:4], enl);
  driver busar0 (busin[3:0], buslow[3:0], enl);

endmodule

module busdriver_equiv (busin, bushigh, buslow, enh, enl);
input [15:0] busin;
output [7:0] bushigh, buslow;
input enh, enl;

  driver busar[3:0] (.out({bushigh, buslow}), .in(busin),
                    .en({enh, enh, enl, enl}));
endmodule

```

Example 4—This example demonstrates how a series of modules can be chained together. Figure 7-1 shows an equivalent schematic interconnection of DFF instances.

```

module dffn (q, d, clk);
parameter bits = 1;
input [bits-1:0] d;
output [bits-1:0] q;
input clk ;

  DFF dff[bits-1:0] (q, d, clk); // create a row of D flip-flops

endmodule

module MxN_pipeline (in, out, clk);
parameter M = 3, N = 4; // M=width,N=depth
input [M-1:0] in;
output [M-1:0] out;
input clk;
wire [M*(N-1):1] t;

  // #(M) redefines the bits parameter for dffn
  // create p[1:N] columns of dffn rows (pipeline)

  dffn #(M) p[1:N] ({out, t}, {t, in}, clk);

endmodule

```

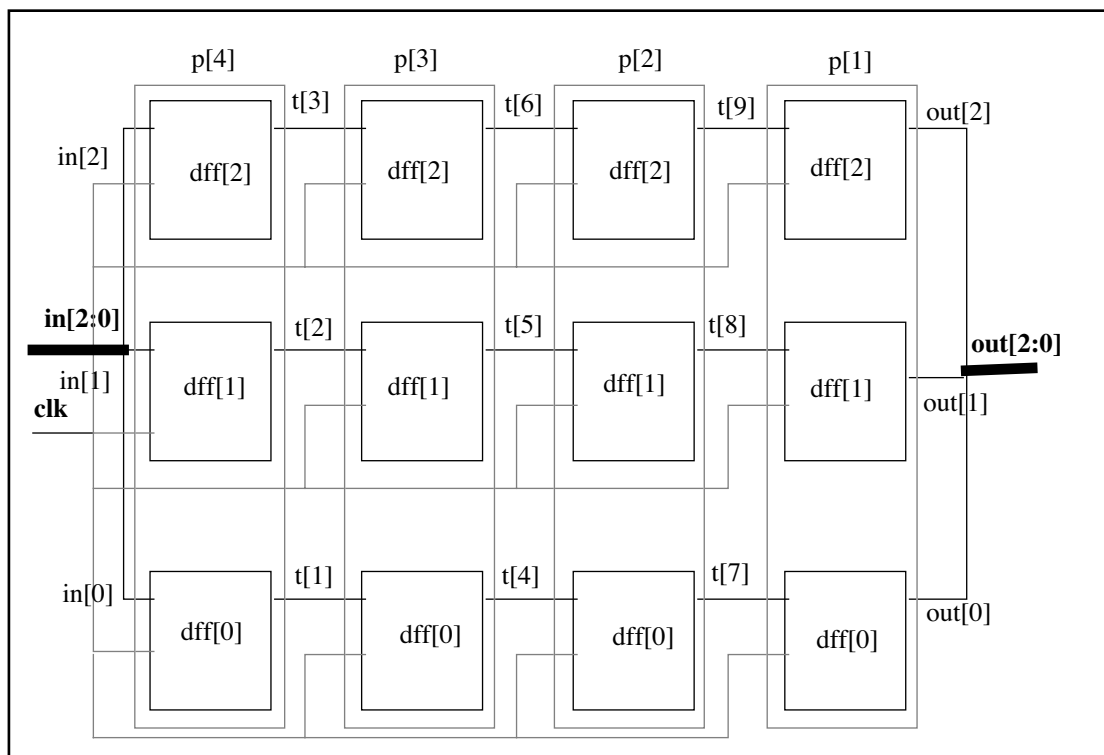


Figure 7-1 — Schematic diagram of interconnections in array of instances

7.2 And, nand, nor, or, xor, and xnor gates

The instance declaration of a multiple input logic gate shall begin with one of the following keywords:

and nand nor or xor xnor

The delay specification shall be zero, one, or two delays. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to x. If only one delay is specified, it shall specify both the rise delay and the fall delay. If there is no delay specification, there shall be no propagation delay through the gate.

These six logic gates shall have one output and one or more inputs. The first terminal in the terminal list shall connect to the output of the gate and all other terminals connect to its inputs.

The truth tables for these gates, showing the result of two input values, appear in Table 7-3.

Table 7-3—Truth tables for multiple input logic gates

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

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

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

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

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

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

Versions of these six logic gates having more than two inputs shall have a natural extension, but the number of inputs shall not alter propagation delays.

Example:

The following example declares a two input **and** gate:

```
and a1 (out, in1, in2);
```

The inputs are *in1* and *in2*. The output is *out*. The instance name is *a1*.

7.3 Buf and not gates

The instance declaration of a multiple output logic gate shall begin with one of the following keywords:

```
buf           not
```

The delay specification shall be zero, one, or two delays. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to *x*. If only one delay is specified, it shall specify both the rise delay and the fall delay. If there is no delay specification, there shall be no propagation delay through the gate.

These two logic gates shall have one input and one or more outputs. The last terminal in the terminal list shall connect to the input of the logic gate, and the other terminals shall connect to the outputs of the logic gate.

Truth tables for these logic gates with one input and one output are shown in Table 7-4.

Table 7-4—Truth tables for multiple output logic gates

buf		not	
input	output	input	output
0	0	0	1
1	1	1	0
x	x	x	x
z	x	z	x

Example:

The following example declares a two output buf:

```
buf b1 (out1, out2, in);
```

The input is `in`. The outputs are `out1` and `out2`. The instance name is `b1`.

7.4 Bufif1, bufif0, notif1, and notif0 gates

The instance declaration of a tri-state logic gate shall begin with one of the following keywords:

```
bufif0      bufif1      notif1      notif0
```

These four logic gates model three-state drivers. In addition to logic values 1 and 0, these gates can output `z`.

The delay specification shall be zero, one, two, or three delays. If the delay specification contains three delays, the first delay shall determine the rise delay, the second delay shall determine the fall delay, the third delay shall determine the delay of transitions to `z`, and the smallest of the three delays shall determine the delay of transitions to `x`. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to `x` and `z`. If only one delay is specified, it shall specify the delay for all output transitions. If there is no delay specification, there shall be no propagation delay through the gate.

Some combinations of data input values and control input values can cause these gates to output either of two values, without a preference for either value (see 7.11.2). These logic tables for these gates include two symbols representing such unknown results. The symbol `L` shall represent a result that has a value 0 or `z`. The symbol `H` shall represent a result that has a value 1 or `z`. Delays on transitions to `H` or `L` shall be treated the same as delays on transitions to `x`.

These four logic gates shall have one output, one data input, and one control input. The first terminal in the terminal list shall connect to the output, the second terminal shall connect to the data input, and the third terminal shall connect to the control input.

Table 7-5 presents the logic tables for these gates.

Table 7-5—Truth tables for tristate logic gates

bufif0	CONTROL				
		0	1	x	z
D	0	0	z	L	L
A	1	1	z	H	H
T	x	x	z	x	x
A	z	x	z	x	x

bufif1	CONTROL				
		0	1	x	z
D	0	z	0	L	L
A	1	z	1	H	H
T	x	z	x	x	x
A	z	z	x	x	x

notif0	CONTROL				
		0	1	x	z
D	0	1	z	H	H
A	1	0	z	L	L
T	x	x	z	x	x
A	z	x	z	x	x

notif1	CONTROL				
		0	1	x	z
D	0	z	1	H	H
A	1	z	0	L	L
T	x	z	x	x	x
A	z	z	x	x	x

Example:

The following example declares an instance of **bufif1**:

```
bufif1 bf1 (outw, inw, controlw);
```

The output is **outw**, the input is **inw**, and the control is **controlw**. The instance name is **bf1**.

7.5 MOS switches

The instance declaration of a MOS switch shall begin with one of the following keywords:

cmos **nmos** **pmos** **rcmos** **rnmos** **rpmos**

The **cmos** and **rcmos** switches are described in 7.7.

The **pmos** keyword stands for the P-type metal-oxide semiconductor (PMOS) transistor and the **nmos** keyword stands for the N-type metal-oxide semiconductor (NMOS) transistor. PMOS and NMOS transistors have relatively low impedance between their sources and drains when they conduct. The **rpmos** keyword stands for resistive PMOS transistor and the **rnmos** keyword stands for resistive NMOS transistor. Resistive PMOS and resistive NMOS transistors have significantly higher impedance between their sources and drains when they conduct than PMOS and NMOS transistors have. The load devices in static MOS networks are examples of **rpmos** and **rnmos** transistors. These four switches are *unidirectional channels* for data similar to the **bufif** gates.

The delay specification shall be zero, one, two, or three delays. If the delay specification contains three delays, the first delay shall determine the rise delay, the second delay shall determine the fall delay, the third delay shall determine the delay of transitions to z, and the smallest of the three delays shall determine the delay of transitions to x. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to x and z. If only one delay is specified, it shall specify the delay for all output transitions. If there is no delay specification, there shall be no propagation delay through the switch.

Some combinations of data input values and control input values can cause these switches to output either of two values, without a preference for either value. The logic tables for these switches include two symbols representing such unknown results. The symbol L represents a result that has a value 0 or z. The symbol H represents a result that has a value 1 or z. Delays on transitions to H and L shall be the same as delays on transitions to x.

These four switches shall have one output, one data input, and one control input. The first terminal in the terminal list shall connect to the output, the second terminal shall connect to the data input, and the third terminal shall connect to the control input.

The **nmos** and **pmos** switches shall pass signals from their inputs and through their outputs with a change in the strength of the signal in only one case, as discussed in 7.12. The **rnmos** and **rpmos** switches shall reduce the strength of signals that propagate through them, as discussed in 7.13.

Table 7-6 presents the logic tables for these switches.

Table 7-6—Truth tables for MOS switches

pmos rpmos	CONTROL				
		0	1	x	z
D	0	0	z	L	L
A	1	1	z	H	H
T	x	x	z	x	x
A	z	z	z	z	z

nmos rnmos	CONTROL				
		0	1	x	z
D	0	z	0	L	L
A	1	z	1	H	H
T	x	z	x	x	x
A	z	z	z	z	z

Example:

The following example declares a **pmos** switch:

```
pmos p1 (out, data, control);
```

The output is **out**, the data input is **data**, and the control input is **control**. The instance name is **p1**.

7.6 Bidirectional pass switches

The instance declaration of a bidirectional pass switch shall begin with one of the following keywords:

tran	tranif1	tranif0
rtran	rtranif1	rtranif0

The bidirectional pass switches shall not delay signals propagating through them. When **tranif0**, **tranif1**, **rtranif0**, or **rtranif1** devices are turned off they shall block signals, and when they are turned on they shall pass signals. The **tran**

and **rtran** devices cannot be turned off, and they shall always pass signals.

The delay specifications for **tranif1**, **tranif0**, **rtranif1**, and **rtranif0** devices shall be zero, one, or two delays. If the specification contains two delays, the first delay shall determine the *turn-on delay*, and the second delay shall determine the *turn-off delay*, and the smaller of the two delays shall apply to output transitions to **x** and **z**. If only one delay is specified, it shall specify both the turn-on and the turn-off delays. If there is no delay specification, there shall be no turn-on or turn-off delay for the bidirectional pass switch.

The bidirectional pass switches **tran** and **rtran** shall not accept delay specification.

The **tranif1**, **tranif0**, **rtranif1**, and **rtranif0** devices shall have three items in their terminal lists. The first two shall be bidirectional terminals that conduct signals to and from the devices, and the third terminal shall connect to a control input. The **tran** and **rtran** devices shall have terminal lists containing two bidirectional terminals. Both bidirectional terminals shall unconditionally conduct signals to and from the devices, allowing signals to pass in either direction through the devices. The bidirectional terminals of all six devices shall be connected only to scalar nets or bit-selects of vector nets.

The **tran**, **tranif0**, and **tranif1** devices shall pass signals with an alteration in their strength in only one case, as discussed in 7.12. The **rtran**, **rtranif0**, and **rtranif1** devices shall reduce the strength of the signals passing through them according to rules discussed in 7.13.

Example:

The following example declares an instance of **tranif1**:

```
tranif1 t1 (inout1,inout2,control);
```

The bidirectional terminals are **inout1** and **inout2**. The control input is **control**. The instance name is **t1**.

7.7 CMOS switches

The instance declaration of a CMOS switch shall begin with one of the following keywords:

cmos **rcmos**

The delay specification shall be zero, one, two, or three delays. If the delay specification contains three delays, the first delay shall determine the rise delay, the second delay shall determine the fall delay, the third delay shall determine the delay of transitions to **z**, and the smallest of the three delays shall determine the delay of transitions to **x**. Delays in transitions to **H** or **L** are the same as delays in transitions to **x**. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to **x** and **z**. If only one delay is specified, it shall specify the delay for all output transitions. If there is no delay specification, there shall be no propagation delay through the switch.

The **cmos** and **rcmos** switches shall have a data input, a data output, and two control inputs. In the terminal list, the first terminal shall connect to the data output, the second terminal shall connect to the data input, the third terminal shall connect to the n-channel control input, and the last terminal shall connect to the p-channel control input.

The **cmos** gate shall pass signals with an alteration in their strength in only one case, as discussed in 7.12. The **rcmos** gate shall reduce the strength of signals passing through it according to rules described in 7.13.

The **cmos** switch shall be treated as the combination of a **pmos** switch and an **nmos** switch. The **rcmos** switch shall be treated as the combination of an **rpmos** switch and an **rnmoss** switch. The combined switches in these configurations shall share data input and data output terminals, but they shall have separate control inputs.

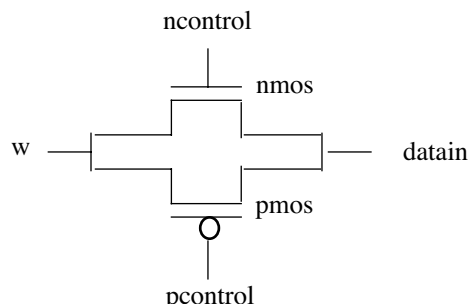
Example:

The equivalence of the **cmos** gate to the pairing of an **nmos** gate and a **pmos** gate is shown in the following example:

```
cmos (w, datain, ncontrol, pcontrol);
```

is equivalent to:

```
nmos (w, datain, ncontrol);
```



7.8 Pullup and pulldown sources

The instance declaration of a pullup or a pulldown source shall begin with one of the following keywords:

pullup

pulldown

A **pullup** source shall place a logic value 1 on the nets connected in its terminal list. A **pulldown** source shall place a logic value 0 on the nets connected in its terminal list. The signals that these sources place on nets shall have **pull** strength in the absence of a strength specification. If conflicting strength specification is declared, it shall be ignored. There shall be no delay specifications for these sources.

Example:

The following example declares two **pullup** instances:

```
pullup (strong1) p1 (neta), p2 (netb);
```

In this example, the p1 instance drives neta and the p2 instance drives netb.

7.9 Implicit net declarations

Including a previously unused identifier in a terminal list shall implicitly declare a new net of type **wire**, or of type defined by the compiler directive ``default_nettype` (see 16.2), with zero delay.

Each implicitly declared net shall connect to one or more of the following:

- Gate output
- Bidirectional terminal
- Module output port

7.10 Logic strength modeling

The Verilog HDL provides for accurate modeling of signal contention, bidirectional pass gates, resistive MOS devices, dynamic MOS, charge sharing, and other technology-dependent network configurations by allowing scalar net signal values to have a full range of unknown values and different levels of strength or combinations of levels of strength. This multiple-level logic strength modeling resolves combinations of signals into known or unknown values to represent the behavior of hardware with improved accuracy.

A strength specification shall have two components:

- a) The strength of the 0 portion of the net value, called strength0, designated as one of the following:

supply0 strong0 pull0 weak0 highz0

- b) The strength of the 1 portion of the net value, called strength1, designated as one of the following:

supply1 strong1 pull1 weak1 highz1

The combinations (**highz0, highz1**) and (**highz1, highz0**) shall be considered illegal.

Despite this division of the strength specification, it is helpful to consider strength as a property occupying regions of a continuum in order to predict the results of combinations of signals.

Table 7-7 demonstrates the continuum of strengths. The left column lists the keywords used in specifying strengths. The right column gives correlated strength levels.

Table 7-7—Strength levels for scalar net signal values

Strength name	Strength level
supply0	7
strong0	6
pull0	5
large0	4
weak0	3
medium0	2
small0	1
highz0	0
highz1	0
small1	1
medium1	2
weak1	3
large1	4
pull1	5
strong1	6
supply1	7

In Table 7-7, there are four *driving strengths*:

supply strong pull weak

Signals with driving strengths shall propagate from gate outputs and continuous assignment outputs.

In Table 7-7, there are three *charge storage strengths*:

large medium small

Signals with the charge storage strengths shall originate in the **triereg** net type.

It is possible to think of the strengths of signals in the preceding table as locations on the scale in Figure 7-2.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Figure 7-2—Scale of strengths

Discussions of signal combinations later in this section employs graphics similar to those used in Figure 7-2.

If the signal value of a net is known, all of its strength levels shall be in either the strength0 part of the scale represented by Figure 7-2, or all strength levels shall be in its strength1 part. If the signal value of a net is unknown, it shall have strength levels in both the strength0 and the strength1 parts. A net with a signal value *z* shall have a strength level only in one of the 0 subdivisions of the parts of the scale.

7.11 Strengths and values of combined signals

In addition to a signal value, a net shall have either a single unambiguous strength level or an ambiguous strength consisting of more than one level. When signals combine, their strengths and values shall determine the strength and value of the resulting signal in accordance with the principles in 7.11.1 through 7.11.4.

7.11.1 Combined signals of unambiguous strength

This subclause deals with combinations of signals in which each signal has a known value and a single strength level.

If two or more signals of unequal strength combine in a wired net configuration, the stronger signal shall dominate all the weaker drivers and determine the result. The combination of two or more signals of like value shall result in the same value with the greater of all the strengths. The combination of signals identical in strength and value shall result in the same signal.

The combination of signals with unlike values and the same strength can have three possible results. Two of the results occur in the presence of wired logic and the third occurs in its absence. Wired logic is discussed in 7.11.4. The result in the absence of wired logic is the subject of Figure 7-4.

Example:

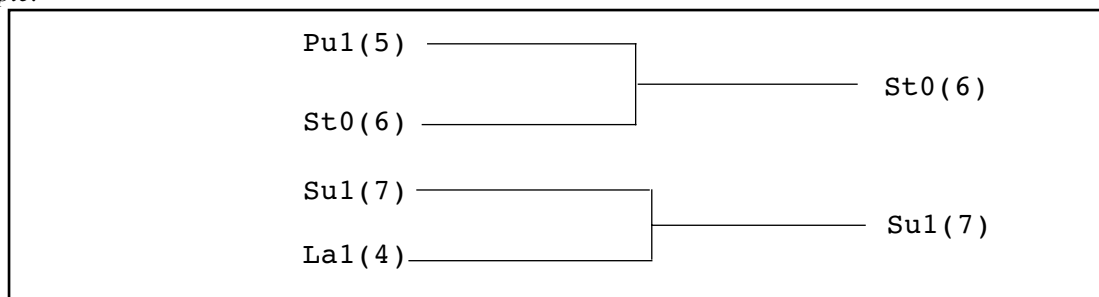


Figure 7-3—Combining unequal strengths

In Figure 7-3, the numbers in parentheses indicate the relative strengths of the signals. The combination of a **pull 1**

and a **strong** 0 results in a **strong** 0, which is the stronger of the two signals.

7.11.2 Ambiguous strengths: sources and combinations

There are several classifications of signals possessing ambiguous strengths:

- Signals with known values and multiple strength levels
- Signals with a value x , which have strength levels consisting of subdivisions of both the strength1 and the strength0 parts of the scale of strengths in Figure 7-2
- Signals with a value L , which have strength levels that consist of high impedance joined with strength levels in the strength0 part of the scale of strengths in Figure 7-2
- Signals with a value H , which have strength levels that consist of high impedance joined with strength levels in the strength1 part of the scale of strengths in Figure 7-2

Many configurations can produce signals of ambiguous strength. When two signals of equal strength and opposite value combine, the result shall be a value x , along with the strength levels of both signals and all the smaller strength levels.

Examples:

Figure 7-4 shows the combination of a **weak** signal with a value 1 and a **weak** signal with a value 0 yielding a signal with **weak** strength and a value x .

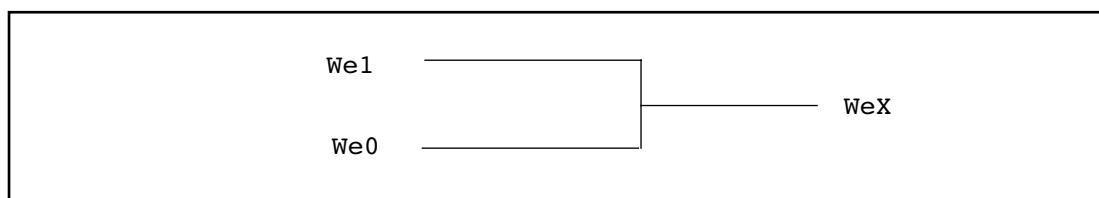


Figure 7-4—Combination of signals of equal strength and opposite values

This output signal is described in Figure 7-5.

strength0								strenght1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Figure 7-5—Weak x signal strength

An ambiguous signal strength can be a range of possible values. An example is the strength of the output from the tri-state drivers with unknown control inputs as shown in Figure 7-6.

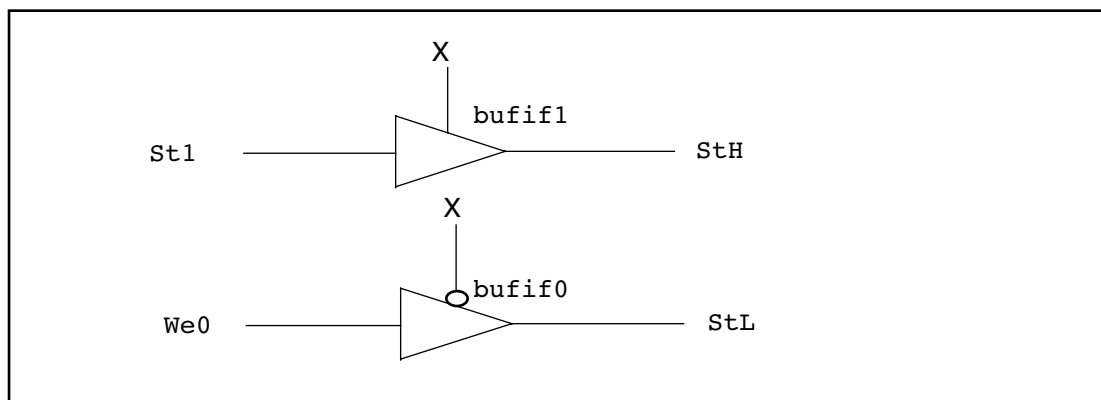


Figure 7-6—Bufifs with control inputs of x

The output of the **bufif1** in Figure 7-6 is a **strong H**, composed of the range of values described in Figure 7-7.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Figure 7-7—Strong H range of values

The output of the **bufif0** in Figure 7-6 is a **strong L**, composed of the range of values described in Figure 7-8.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Figure 7-8—Strong L range of values

The combination of two signals of ambiguous strength shall result in a signal of ambiguous strength. The resulting signal shall have a range of strength levels that includes the strength levels in its component signals. The combination of outputs from two tri-state drivers with unknown control inputs, shown in Figure 7-9, is an example.

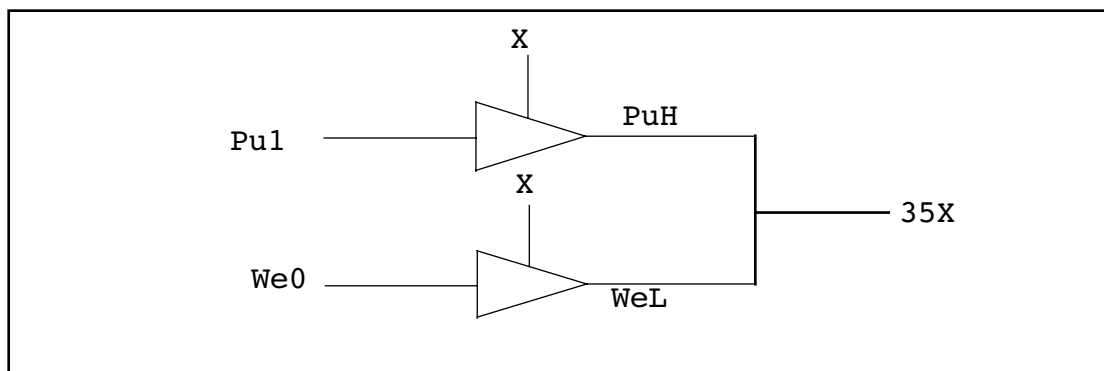


Figure 7-9—Combined signals of ambiguous strength

In Figure 7-9, the combination of signals of ambiguous strengths produces a range that includes the extremes of the signals and all the strengths between them, as described in Figure 7-10.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Figure 7-10—Range of strengths for an unknown signal

The result is a value x because its range includes the values 1 and 0. The number 35, which precedes the x , is a concatenation of two digits. The first is the digit 3, which corresponds to the highest strength0 level for the result. The second digit, 5, corresponds to the highest strength1 level for the result.

Switch networks can produce a ranges of strengths of the same value, such as the signals from the upper and lower configurations in Figure 7-11.

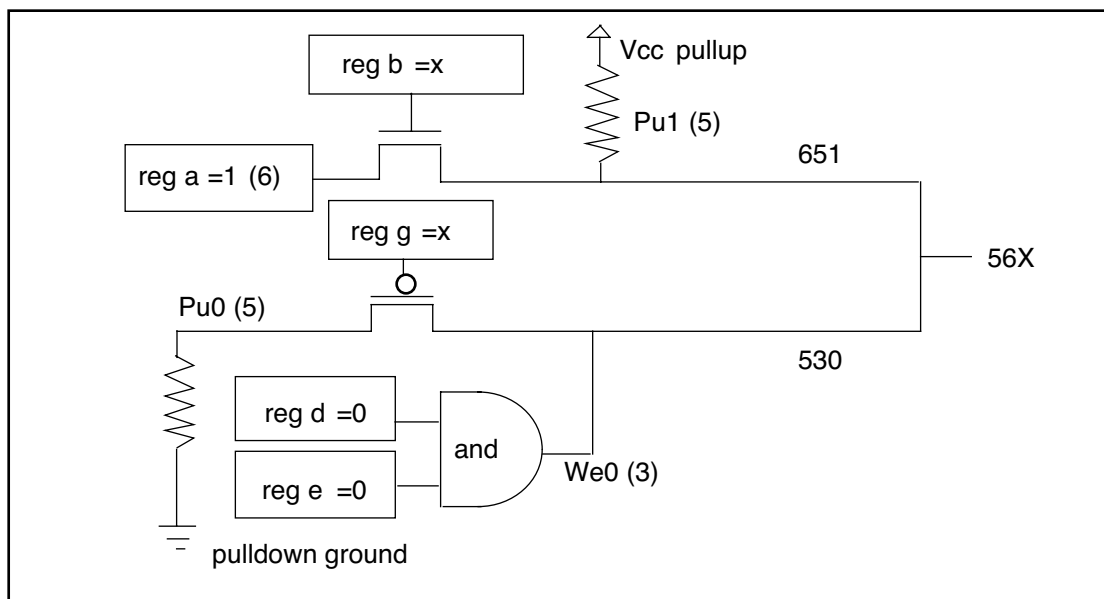


Figure 7-11—Ambiguous strengths from switch networks

In Figure 7-11, the upper combination of a register, a gate controlled by a register of unspecified value, and a pullup produces a signal with a value of 1 and a range of strengths (651) described in Figure 7-12.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Figure 7-12—Range of two strengths of a defined value

In Figure 7-11, the lower combination of a **pulldown**, a gate controlled by a register of unspecified value, and an **and** gate produces a signal with a value 0 and a range of strengths (530) described in Figure 7-13.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Figure 7-13—Range of three strengths of a defined value

When the signals from the upper and lower configurations in Figure 7-11 combine, the result is an unknown with a range (56x) determined by the extremes of the two signals shown in Figure 7-14.

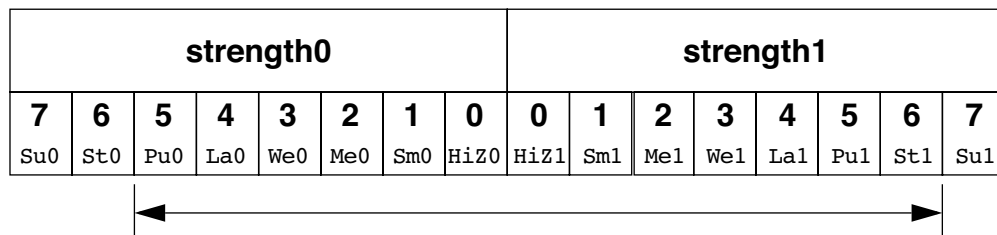


Figure 7-14—Unknown value with a range of strengths

In Figure 7-11, replacing the **pulldown** in the lower configuration with a **supply0** would change the range of the result to the range (StX) described in Figure 7-15.

The range in Figure 7-15 is **strong x**, because it is unknown and the extremes of both its components are **strong**. The extreme of the output of the lower configuration is **strong** because the lower **pmos** reduces the strength of the **supply0** signal. This modeling feature is discussed in 7.12.

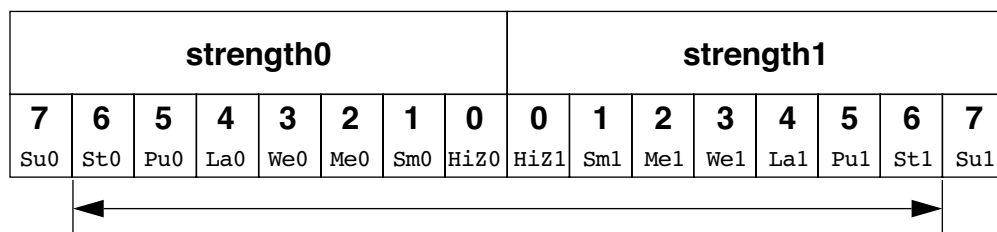


Figure 7-15—Strong X range

Logic gates produce results with ambiguous strengths as well as tri-state drivers. Such a case appears in Figure 7-16. The **and** gate N1 is declared with **highz0** strength, and N2 is declared with **weak0** strength.

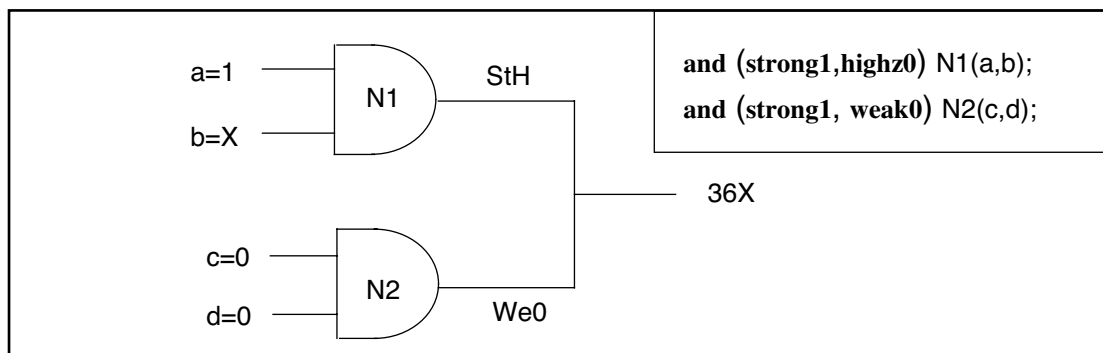


Figure 7-16—Ambiguous strength from gates

In Figure 7-16, register **b** has an unspecified value, so input to the upper **and** gate is **strong x**. The upper **and** gate has a strength specification including **highz0**. The signal from the upper **and** gate is a **strong H** composed of the values as described in Figure 7-17.

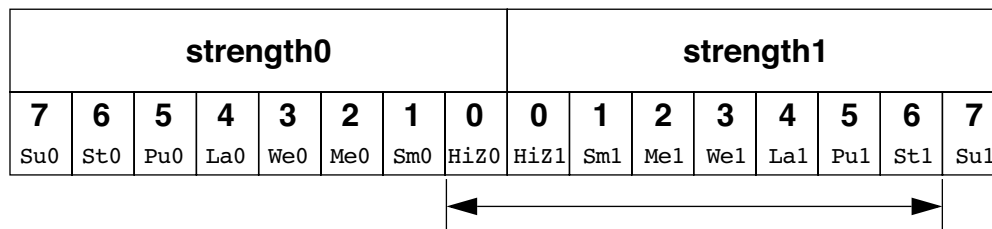


Figure 7-17—Ambiguous strength signal from a gate

HiZ0 is part of the result, because the strength specification for the gate in question specified that strength for an output with a value 0. A strength specification other than high impedance for the 0 value output results in a gate output value x. The output of the lower **and** gate is a **weak** 0 as described in Figure 7-18.

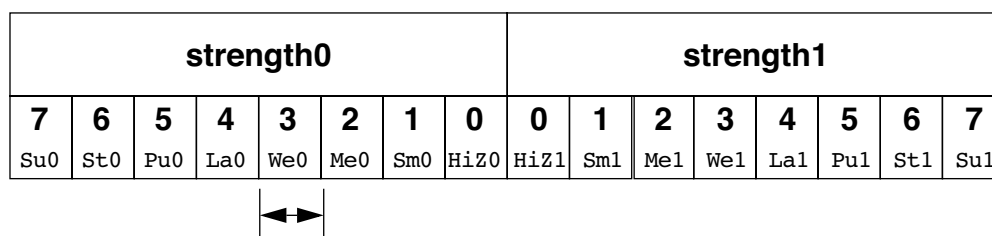


Figure 7-18—Weak 0

When the signals combine, the result is the range (36x) as described in Figure 7-19.

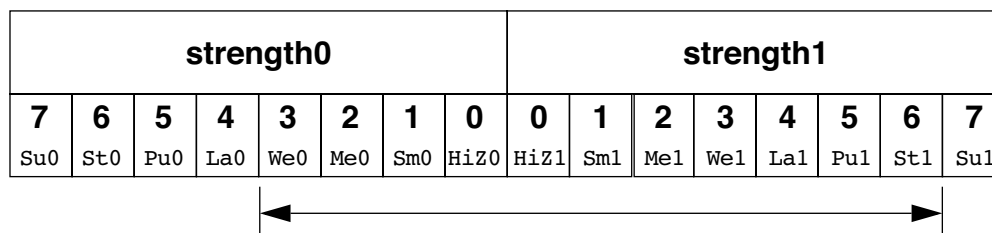


Figure 7-19—Ambiguous strength in combined gate signals

Figure 7-19 presents the combination of an ambiguous signal and an unambiguous signal. Such combinations are the topic of 7.11.3.

7.11.3 Ambiguous strength signals and unambiguous signals

The combination of a signal with unambiguous strength and known value with another signal of ambiguous strength presents several possible cases. To understand a set of rules governing this type of combination, it is necessary to consider the strength levels of the ambiguous strength signal separately from each other and relative to the unambiguous strength signal. When a signal of known value and unambiguous strength combines with a component of a signal of ambiguous strength, these shall be the effects:

- The strength levels of the ambiguous strength signal that are greater than the strength level of the unambiguous signal shall remain in the result.
- The strength levels of the ambiguous strength signal that are smaller than or equal to the strength level of the unambiguous signal shall disappear from the result, subject to rule c.

- c) If the operation of rule a and rule b results in a gap in strength levels because the signals are of opposite value, the signals in the gap shall be part of the result.

The following figures show some applications of the rules.

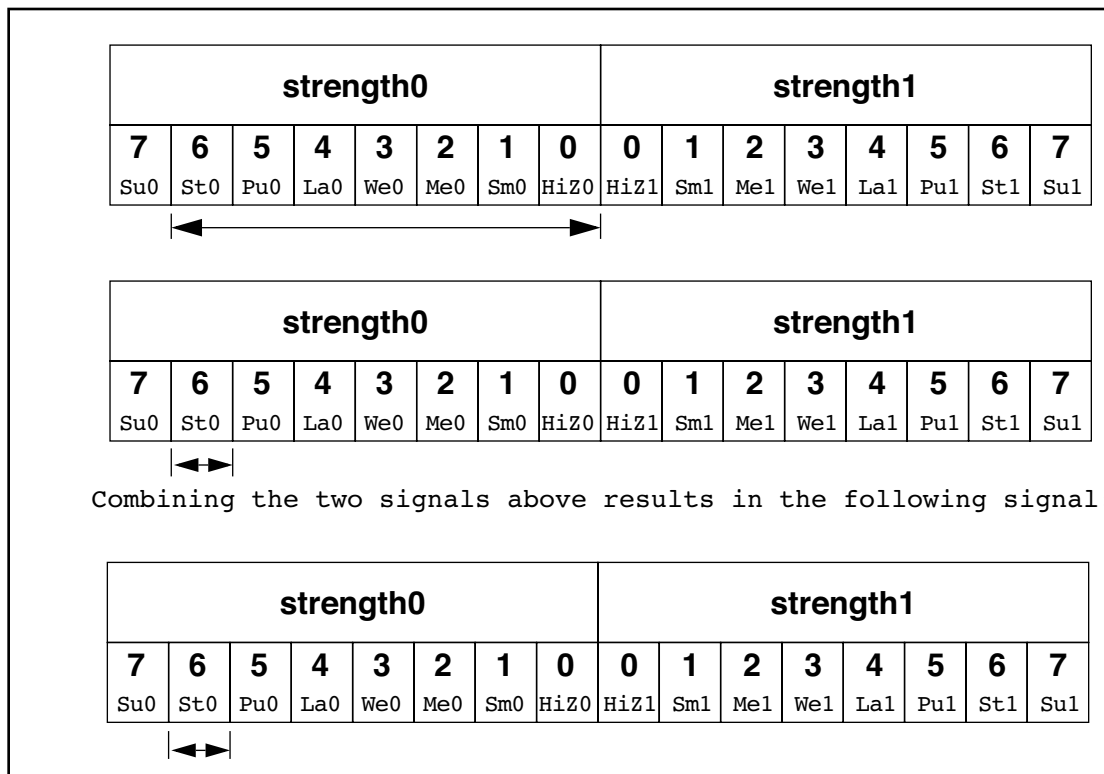


Figure 7-20—Elimination of strength levels

In Figure 7-20, the strength levels in the ambiguous strength signal that are smaller than or equal to the strength level of the unambiguous strength signal disappear from the result, demonstrating rule b.

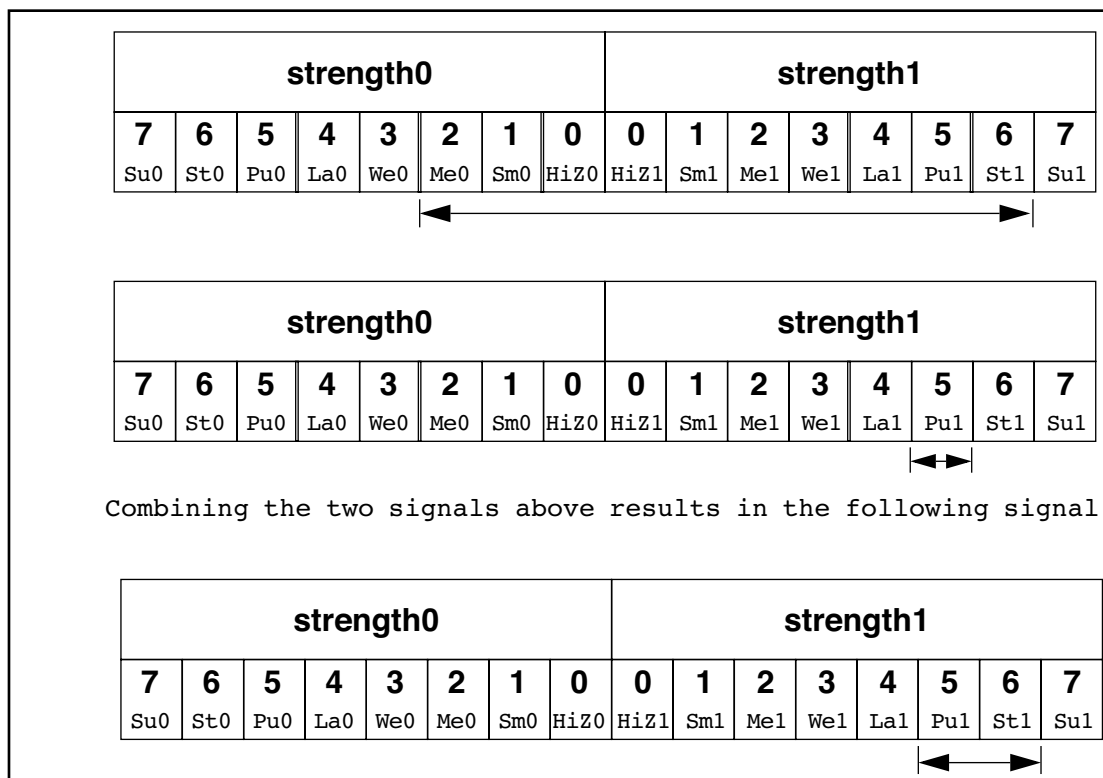


Figure 7-21 — Result demonstrating a range and the elimination of strength levels of two values

In Figure 7-21, rules a, b, and c apply. The strength levels of the ambiguous strength signal that are of opposite value and lesser strength than the unambiguous strength signal disappear from the result. The strength levels in the ambiguous strength signal that are less than the strength level of the unambiguous strength signal, and of the same value, disappear from the result. The strength level of the unambiguous strength signal and the greater extreme of the ambiguous strength signal define a range in the result.

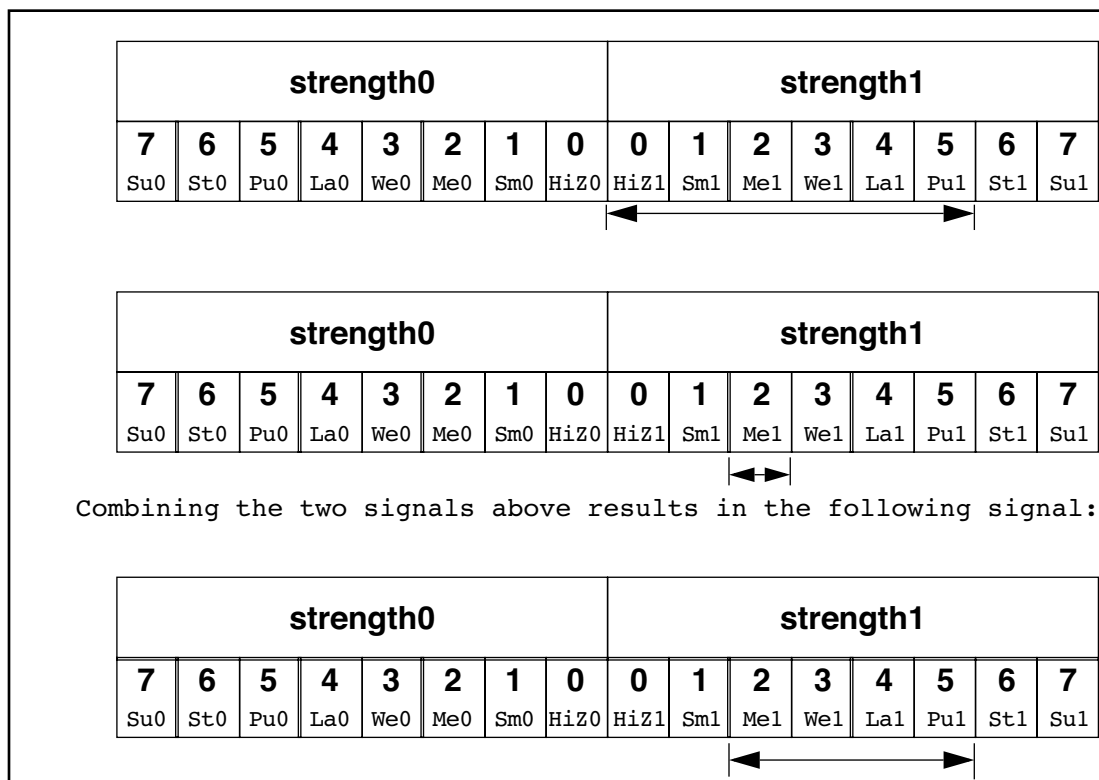


Figure 7-22—Result demonstrating a range and the elimination of strength levels of one value

In Figure 7-22, rules a and b apply. The strength levels in the ambiguous strength signal that are less than the strength level of the unambiguous strength signal disappear from the result. The strength level of the unambiguous strength signal and the strength level at the greater extreme of the ambiguous strength signal define a range in the result.

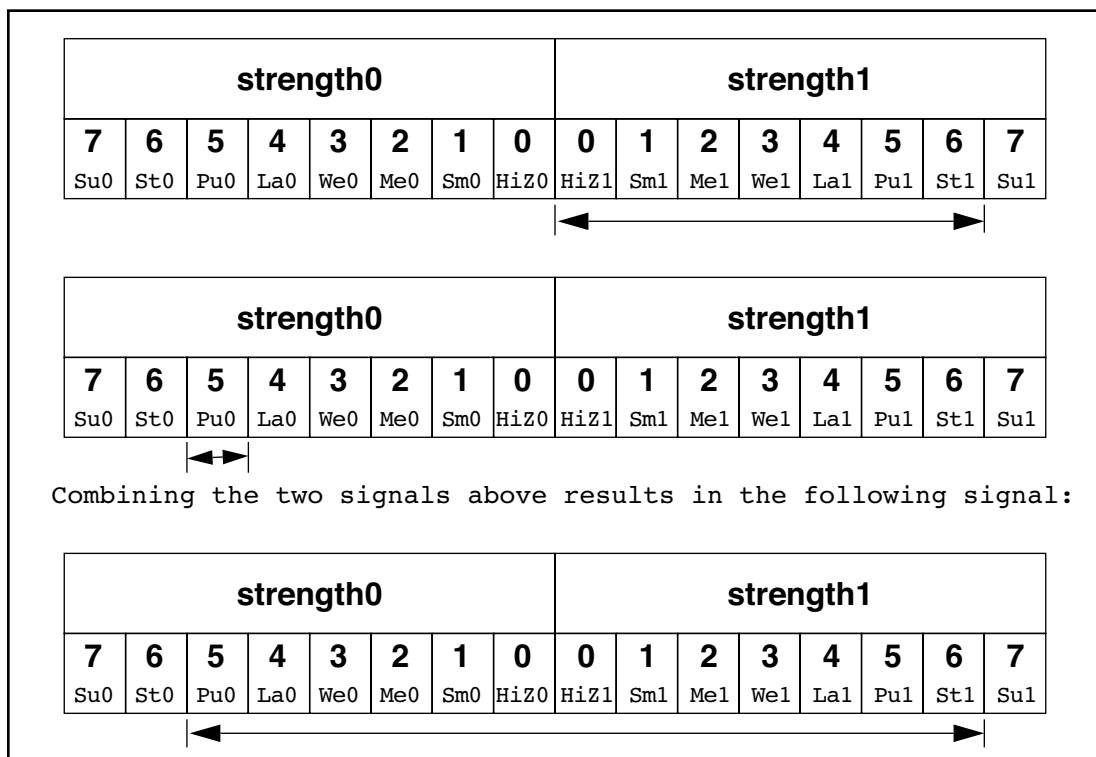


Figure 7-23—A range of both values

In Figure 7-23, rules a, b, and c apply. The greater extreme of the range of strengths for the ambiguous strength signal is larger than the strength level of the unambiguous strength signal. The result is a range defined by the greatest strength in the range of the ambiguous strength signal and by the strength level of the unambiguous strength signal.

7.11.4 Wired logic net types

The net types **triand**, **wand**, **trior**, and **wor** shall resolve conflicts when multiple drivers have the same strength. These net types shall resolve signal values by treating signals as inputs of logic functions.

Examples:

Consider the combination of two signals of unambiguous strength in Figure 7-24.

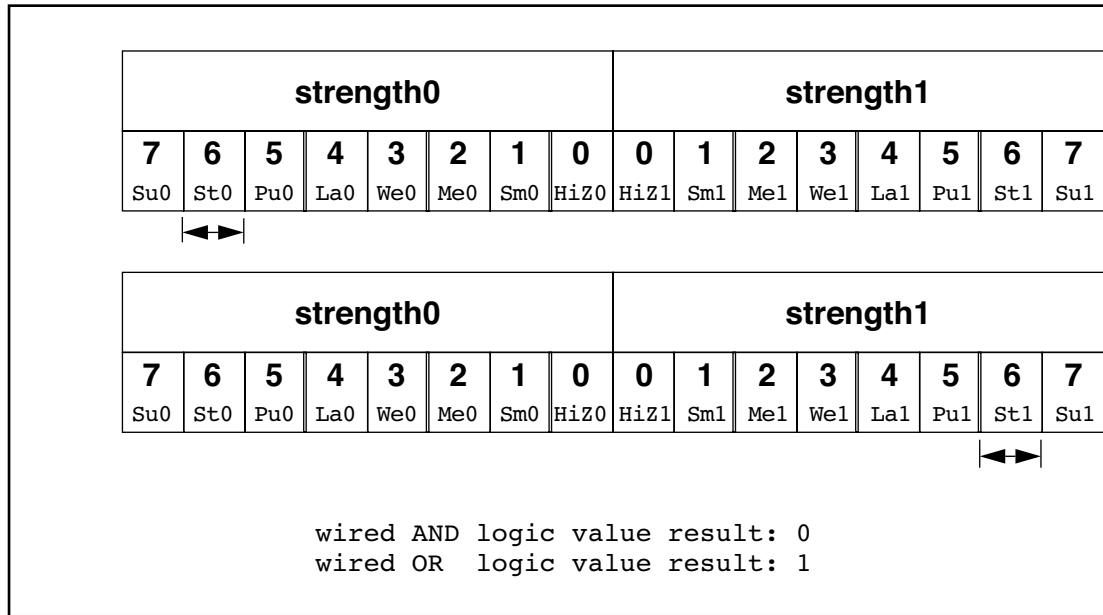


Figure 7-24—Wired logic with unambiguous strength signals

The combination of the signals in Figure 7-24, using *wired and* logic, produces a result with the same value as the result produced by an **and** gate with the value of the two signals as its inputs. The combination of signals using *wired or* logic produces a result with the same value as the result produced by an **or** gate with the values of the two signals as its inputs. The strength of the result is the same as the strength of the combined signals in both cases. If the value of the upper signal changes so that both signals in Figure 7-24 possess a value 1, then the results of both types of logic have a value 1.

When ambiguous strength signals combine in wired logic, it is necessary to consider the results of all combinations of each of the strength levels in the first signal with each of the strength levels in the second signal, as shown in Figure 7-25.

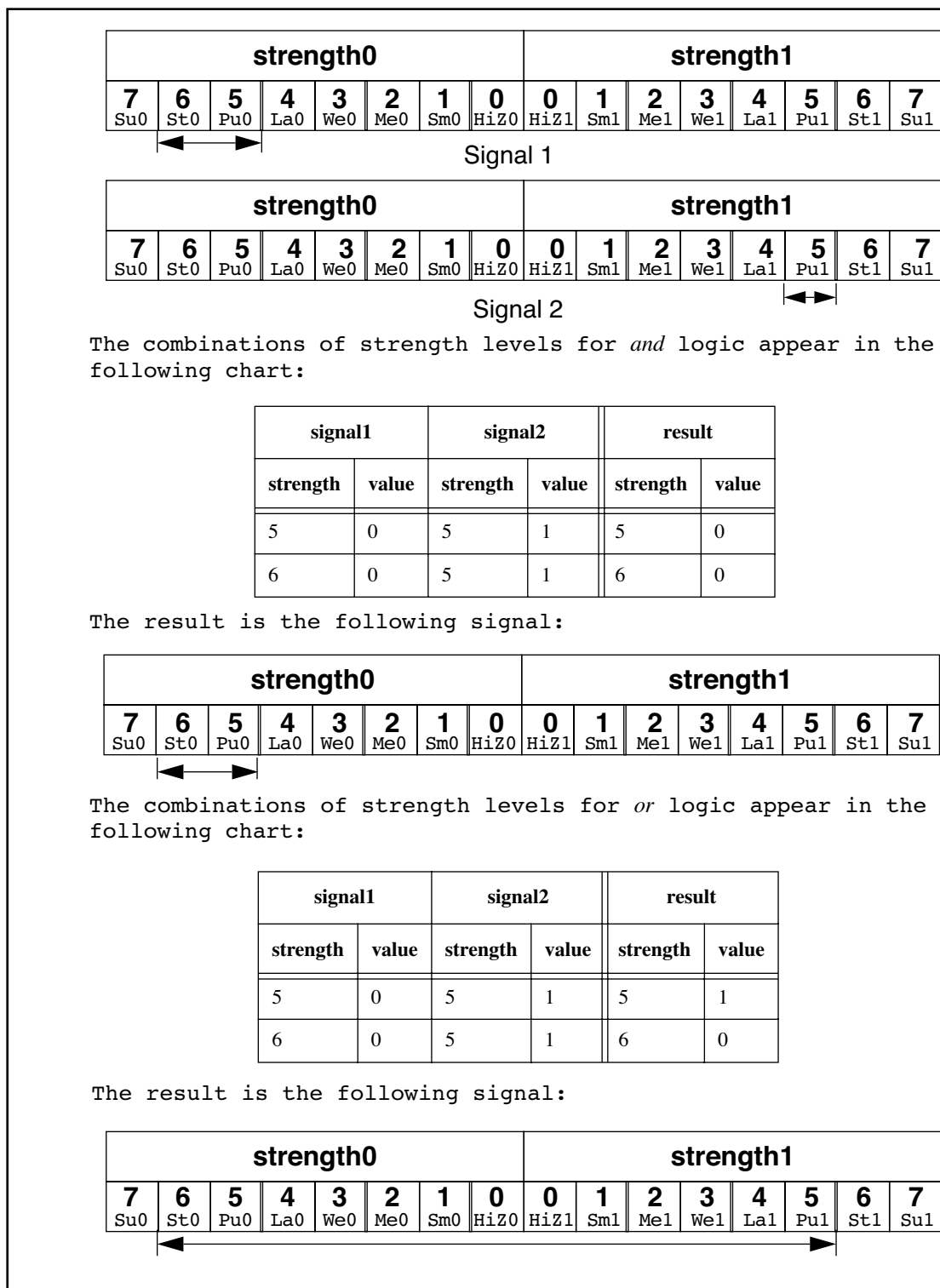


Figure 7-25—Wired logic and ambiguous strengths

7.12 Strength reduction by nonresistive devices

The **nmos**, **pmos**, and **cmos** switches shall pass the strength from the data input to the output, except that a **supply** strength shall be reduced to a **strong** strength.

The **tran**, **tranif0**, and **tranif1** switches shall not affect signal strength across the bidirectional terminals, except that a **supply** strength shall be reduced to a **strong** strength.

7.13 Strength reduction by resistive devices

The **rnmos**, **rpmos**, **rcmos**, **rtran**, **rtranif1**, and **rtranif0** devices shall reduce the strength of signals that pass through them according to Table 7-8.

Table 7-8—Strength reduction rules

Input strength	Reduced strength
Supply drive	Pull drive
Strong drive	Pull drive
Pull drive	Weak drive
Large capacitor	Medium capacitor
Weak drive	Medium capacitor
Medium capacitor	Small capacitor
Small capacitor	Small capacitor
High impedance	High impedance

7.14 Strengths of net types

The **tri0**, **tri1**, **supply0**, and **supply1** net types shall generate signals with specific strength levels. The **triereg** declaration can specify either of two signal strength levels other than a default strength level.

7.14.1 Tri0 and tri1 net strengths

The **tri0** net type models a net connected to a resistive **pulldown** device. In the absence of an overriding source, such a signal shall have a value 0 and a **pull** strength. The **tri1** net type models a net connected to a resistive **pullup** device. In the absence of an overriding source, such a signal shall have a value 1 and a **pull** strength.

7.14.2 Triereg strength

The **triereg** net type models charge storage nodes. The strength of the drive resulting from a **triereg** net that is in the charge storage state (that is, a driver charged the net and then went to high impedance) shall be one of these three strengths: **large**, **medium**, or **small**. The specific strength associated with a particular **triereg** net shall be specified by the user in the net declaration. The default shall be **medium**. The syntax of this specification is described in 3.4.1.

7.14.3 Supply0 and supply1 net strengths

The **supply0** net type models ground connections. The **supply1** net type models connections to power supplies. The **supply0** and **supply1** net types shall have **supply** driving strengths.

7.15 Gate and net delays

Gate and net delays provide a means of more accurately describing delays through a circuit. The *gate delays* specify the signal propagation delay from any gate input to the gate output. Up to three values per output representing rise, fall, and turn-off delays can be specified (see 7.2 through 7.8).

Net delays refer to the time it takes from any driver on the net changing value to the time when the net value is updated and propagated further. Up to three delay values per net can be specified.

For both gates and nets, the *default delay* shall be zero when no delay specification is given. When one delay value is given, then this value shall be used for all propagation delays associated with the gate or the net. When two delays are given, the first delay shall specify the rise delay and the second delay shall specify the fall delay. The delay when the signal changes to high impedance or to unknown shall be the lesser of the two delay values.

For a three-delay specification:

- The first delay refers to the transition to the 1 value (rise delay)
- The second delay refers to the transition to the 0 value (fall delay)
- The third delay refers to the transition to the high-impedance value

When a value changes to the unknown (x) value, the delay is the smallest of the three delays. The strength of the input signal shall not affect the propagation delay from an input to an output.

Table 7-9 summarizes the from-to propagation delay choice for the two- and three-delay specifications.

Table 7-9—Rules for propagation delays

From value:	To value:	Delay used if there are	
		2 delays	3 delays
0	1	d1	d1
0	x	min(d1, d2)	min(d1, d2, d3)
0	z	min(d1, d2)	d3
1	0	d2	d2
1	x	min(d1, d2)	min(d1, d2, d3)
1	z	min(d1, d2)	d3
x	0	d2	d2
x	1	d1	d1
x	z	min(d1, d2)	d3
z	0	d2	d2
z	1	d1	d1
z	x	min(d1, d2)	min(d1, d2, d3)

Examples:

Example 1—The following is an example of a delay specification with one, two, and three delays:

```
and #(10) a1 (out, in1, in2);           // only one delay
```

```

and #(10,12) a2 (out, in1, in2);    // rise and fall delays
bufif0 #(10,12,11) b3 (out, in, ctrl); // rise, fall, and turn-off delays

```

Example 2—The following example specifies a simple latch module with tri-state outputs, where individual delays are given to the gates. The propagation delay from the primary inputs to the outputs of the module will be cumulative, and it depends on the signal path through the network.

```

module tri_latch (qout, nqout, clock, data, enable);
output qout, nqout;
input clock, data, enable;
tri qout, nqout;

not    #5          n1 (ndata, data);
nand   #(3,5)      n2 (wa, data, clock),
          n3 (wb, ndata, clock);
nand   #(12,15)    n4 (q, nq, wa),
          n5 (nq, q, wb);
bufif1 #(3,7,13)   q_drive (qout, q, enable),
          nq_drive (nqout, nq, enable);

endmodule

```

7.15.1 Min:typ:max delays

The syntax for delays on gate primitives (including user-defined primitives; see Section 8), nets, and continuous assignments shall allow three values each for the rising, falling, and turn-off delays. The minimum, typical, and maximum values for each delay shall be specified as constant expressions separated by colons. There shall be no required relation (e.g., $\text{min} \leq \text{typ} \leq \text{max}$) between the expressions for minimum, typical, and maximum delays. These can be any three constant expressions.

Examples:

The following example shows min:typ:max values for rising, falling, and turn-off delays:

```

module iobuf (io1, io2, dir);
    ...
bufif0 #(5:7:9, 8:10:12, 15:18:21) b1 (io1, io2, dir);
bufif1 #(6:8:10, 5:7:9, 13:17:19) b2 (io2, io1, dir);
    ...
endmodule

```

The syntax for delay controls in procedural statements (see 9.7) also allows minimum, typical, and maximum values. These are specified by expressions separated by colons. The following example illustrates this concept.

```

parameter min_hi = 97, typ_hi = 100, max_hi = 107;
reg clk;

always begin
    #(95:100:105) clk = 1;
    #(min_hi:typ_hi:max_hi) clk = 0;
end

```

7.15.2 Trireg net charge decay

Like all nets, the delay specification in a **trireg** net declaration can contain up to three delays. The first two delays shall specify the delay for transition to the 1 and 0 logic states when the **trireg** net is driven to these states by a driver. The third delay shall specify the *charge decay time* instead of the delay in a transition to the z logic state. The charge decay time specifies the delay between when the drivers of a **trireg** net turn off and when its stored charge can no longer be determined.

A **trireg** net does not need a turn-off delay specification because a **trireg** net never makes a transition to the z logic state. When the drivers of a **trireg** net make transitions from the 1, 0, or x logic states to off, the **trireg** net shall retain the previous 1, 0, or x logic state that was on its drivers. The z value shall not propagate from the drivers of a **trireg** net to a **trireg** net. A **trireg** net can only hold a z logic state when z is the initial logic state of the **trireg** net or when the **trireg** net is forced to the z state with a **force** statement (see 9.3.2).

A delay specification for charge decay models a charge storage node that is not ideal—a charge storage node whose charge leaks out through its surrounding devices and connections.

The following subclauses describe the charge decay process and the delay specification for charge decay.

7.15.2.1 The charge decay process

Charge decay is the cause of transition of a 1 or 0 that is stored in a **trireg** net to an unknown value (x) after a specified delay. The charge decay process shall begin when the drivers of the **trireg** net turn off and the **trireg** net starts to hold charge. The charge decay process shall end under the following two conditions:

- a) The delay specified by charge decay time elapses and the **trireg** net makes a transition from 1 or 0 to x.
- b) The drivers of **trireg** net turn on and propagate a 1, 0, or x into the **trireg** net.

7.15.2.2 The delay specification for charge decay time

The third delay in a **trireg** net declaration shall specify the charge decay time. A three-valued delay specification in a **trireg** net declaration shall have the following form:

```
#(d1, d2, d3) // (rise_delay, fall_delay, charge_decay_time)
```

The charge decay time specification in a **trireg** net declaration shall be preceded by a rise and a fall delay specification.

Examples:

Example 1—The following example shows a specification of the charge decay time in a **trireg** net declaration:

```
trireg (large) #(0,0,50) cap1;
```

This example declares a **trireg** net named **cap1**. This **trireg** net stores a **large** charge. The delay specifications for the rise delay is 0, the fall delay is 0, and the charge decay time specification is 50 time units.

Example 2—The next example presents a source description file that contains a **triereg** net declaration with a charge decay time specification. Figure 7-26 shows an equivalent schematic for the source description.

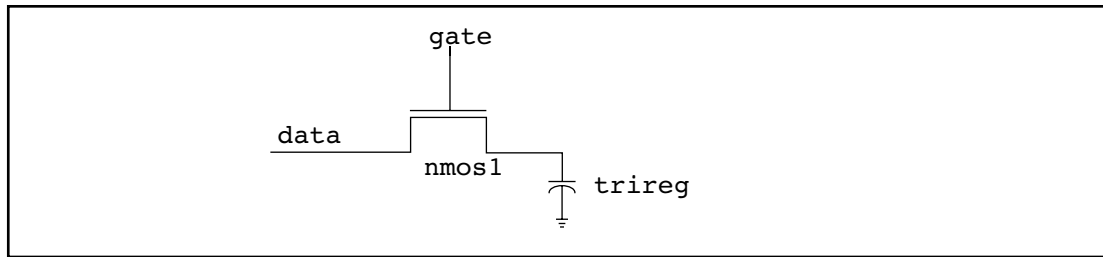


Figure 7-26—Triereg net with capacitance

```

module capacitor;
reg data, gate;

// triereg declaration with a charge decay time of 50 time units
triereg (large) #(0,0,50) cap1;

nmos nmos1 (cap1, data, gate); // nmos that drives the triereg

initial begin
    $monitor("%0d data=%v gate=%v cap1=%v", $time, data, gate, cap1);
    data = 1;
    // Toggle the driver of the control input to the nmos switch
        gate = 1;
    #10 gate = 0;
    #30 gate = 1;
    #10 gate = 0;
    #100 $finish;
end
endmodule

```

Section 8

User-defined primitives (UDPs)

This section describes a modeling technique to augment the set of predefined gate primitives by designing and specifying new primitive elements called user-defined primitives (UDPs). Instances of these new UDPs can be used in exactly the same manner as the gate primitives to represent the circuit being modeled.

The following two types of behavior can be represented in a user-defined primitive:

- a) Combinational—modeled by a combinational UDP
- b) Sequential—modeled by a sequential UDP

A combinational UDP uses the value of its inputs to determine the next value of its output. A sequential UDP uses the value of its inputs and the current value of its output to determine the value of its output. Sequential UDPs provide a way to model sequential circuits such as flip-flops and latches. A sequential UDP can model both level-sensitive and edge-sensitive behavior.

Each UDP has exactly one output, which can be in one of three states: 0, 1, or **x**. The tri-state value **z** is not supported. In sequential UDPs, the output always has the same value as the internal state.

The **z** values passed to UDP inputs shall be treated the same as **x** values.

8.1 UDP definition

UDP definitions are independent of modules; they are at the same level as module definitions in the syntax hierarchy. They can appear anywhere in the source text, either before or after they are instantiated inside a module. They shall not appear between the keywords **module** and **endmodule**.

NOTE—Implementations may limit the maximum number of UDP definitions in a model, but they shall allow at least 256.

The formal syntax of the UDP definition is given in Syntax 8-1.


```

udp_declaration ::=
    primitive udp_identifier ( udp_port_list ) ;
    udp_port_declaration { udp_port_declaration }
    udp_body
    endprimitive
udp_port_list ::= output_port_identifier , input_port_identifier { , input_port_identifier }
udp_port_declaration ::=
    output_declaration
    | input_declaration
    | reg_declaration
udp_body ::= combinational_body | sequential_body
combinational_body ::= table combinational_entry { combinational_entry } endtable
combinational_entry ::= level_input_list : output_symbol ;
sequential_body ::= [ udp_initial_statement ] table sequential_entry { sequential_entry }
endtable
udp_initial_statement ::= initial udp_output_port_identifier = init_val ;
init_val ::= 1'b0 | 1'b1 | 1'bx | 1'bX | 1'B0 | 1'B1 | 1'Bx | 1'BX | 1 | 0
sequential_entry ::= seq_input_list : current_state : next_state ;
seq_input_list ::= level_input_list | edge_input_list
level_input_list ::= level_symbol { level_symbol }
edge_input_list ::= { level_symbol } edge_indicator { level_symbol }
edge_indicator ::= ( level_symbol level_symbol ) | edge_symbol
current_state ::= level_symbol
next_state ::= output_symbol | -
output_symbol ::= 0 | 1 | x | X
level_symbol ::= 0 | 1 | x | X | ? | b | B

```

Syntax 8-1—Syntax for user-defined primitives

8.1.1 UDP header

A UDP definition shall begin with the keyword **primitive**, followed by an identifier, which is the name of the UDP. This in turn is followed by a comma-separated list of ports enclosed in parentheses, which is followed by a semicolon. The UDP definition header is followed by port declarations and a state table. The UDP definition shall be terminated by the keyword **endprimitive**.

UDPs have multiple input ports and exactly one output port; bidirectional inout ports are not permitted on UDPs. All ports of a UDP shall be scalar; vector ports are not permitted.

The output port shall be the first port in the port list.

8.1.2 UDP port declarations

UDPs shall contain input and output port declarations. The output port declaration begins with the keyword **output**, followed by one output port name. The input port declaration begins with the keyword **input**, followed by one or more input port names.

Sequential UDPs shall contain a **reg** declaration for the output port. Combinational UDPs cannot contain a **reg** declaration. The initial value of the output port can be specified in an **initial** statement in a sequential UDP (see 8.1.3).

NOTE—Implementations may limit the maximum number of inputs to a UDP, but they shall allow at least 9 inputs for sequential UDPs and 10 inputs for combinational UDPs.

8.1.3 Sequential UDP initial statement

The sequential UDP initial statement specifies the value of the output port when simulation begins. This statement begins with the keyword **initial**. The statement that follows shall be an assignment statement that assigns a single-bit literal value to the output port.

8.1.4 UDP state table

The state table defines the behavior of a UDP. It begins with the keyword **table** and is terminated with the keyword **endtable**. Each row of the table is terminated by a semicolon.

Each row of the table is created using a variety of characters (see Table 8-1), which indicate input values and output state. Three states—0, 1, and **x**—are supported. The **z** state is explicitly excluded from consideration in user-defined primitives. A number of special characters are defined to represent certain combinations of state possibilities. These are described in Table 8-1.

The order of the input state fields of each row of the state table is taken directly from the port list in the UDP definition header. It is not related to the order of the input port declarations.

Combinational UDPs have one field per input and one field for the output. The input fields are separated from the output field by a colon (:). Each row defines the output for a particular combination of the input values (see 8.2).

Sequential UDPs have an additional field inserted between the input fields and the output field. This additional field represents the current state of the UDP and is considered equivalent to the current output value. It is delimited by colons. Each row defines the output based on the current state, particular combinations of input values, and at most one input transition (see 8.4). A row such as the one shown below is illegal:

```
(01) (10) 0 : 0 : 1 ;
```

If all input values are specified as **x**, then the output state shall be specified as **x**.

It is not necessary to explicitly specify every possible input combination. All combinations of input values that are not explicitly specified result in a default output state of **x**.

It is illegal to have the same combination of inputs, including edges, specified for different outputs.

8.1.5 Z values in UDP

The **z** value in a table entry is not supported, and it is considered illegal. The **z** values passed to UDP inputs shall be treated the same as **x** values.

8.1.6 Summary of symbols

To improve the readability and to ease writing of the state table, several special symbols are provided. Table 8-1 summarizes the meaning of all the value symbols that are valid in the table part of a UDP definition.

Table 8-1 — UDP table symbols

Symbol	Interpretation	Comments
0	Logic 0	
1	Logic 1	
x	Unknown	Permitted in the input fields of all UDPs and in the current state field of sequential UDPs. Not permitted in the output field.
?	Iteration of 0, 1, and x	Not permitted in output field.
b	Iteration of 0 and 1	Permitted in the input fields of all UDPs and in the current state field of sequential UDPs. Not permitted in the output field.
-	No change	Permitted only in the output field of a sequential UDP.
(vw)	Value change from v to w	v and w can be any one of 0, 1, x, ?, or b, and are only permitted in the input field.
*	Same as (??)	Any value change on input.
r	Same as (01)	Rising edge on input.
f	Same as (10)	Falling edge on input.
p	Iteration of (01), (0 x) and (x1)	Potential positive edge on the input.
n	Iteration of (10), (1x) and (x0)	Potential negative edge on the input.

8.2 Combinational UDPs

In combinational UDPs, the output state is determined solely as a function of the current input states. Whenever an input state changes, the UDP is evaluated and the output state is set to the value indicated by the row in the state table that matches all the input states. All combinations of the inputs that are not explicitly specified will drive the output state to the unknown value x.

Examples:

The following example defines a multiplexer with two data inputs and a control input.

```

primitive multiplexer (mux, control, dataA, dataB);
output mux;
input control, dataA, dataB;
table
// control dataA dataB mux
    0      1      0 : 1 ;
    0      1      1 : 1 ;
    0      1      x : 1 ;
    0      0      0 : 0 ;
    0      0      1 : 0 ;
    0      0      x : 0 ;
    1      0      1 : 1 ;
    1      1      1 : 1 ;
    1      1      x : 1 ;
    1      0      0 : 0 ;
    1      1      0 : 0 ;
    1      x      0 : 0 ;
    x      0      0 : 0 ;
    x      1      1 : 1 ;
endtable
endprimitive

```

The first entry in this example can be explained as follows: when **control** equals 0, and **dataA** equals 1, and **dataB** equals 0, then output **mux** equals 1.

The input combination 0xx (**control**=0, **dataA**=x, **dataB**=x) is not specified. If this combination occurs during simulation, the value of output port **mux** will become x.

Using **?**, the description of a multiplexer can be abbreviated as

```

primitive multiplexer (mux, control, dataA, dataB);
output mux;
input control, dataA, dataB;
table
// control dataA dataB mux
    0      1      ? : 1 ;    // ? = 0 1 x
    0      0      ? : 0 ;
    1      ?      1 : 1 ;
    1      ?      0 : 0 ;
    x      0      0 : 0 ;
    x      1      1 : 1 ;
endtable
endprimitive

```

8.3 Level-sensitive sequential UDPs

Level-sensitive sequential behavior is represented the same way as combinational behavior, except that the output is declared to be of type **reg**, and there is an additional field in each table entry. This new field represents the current state of the UDP. The output field in a sequential UDP represents the next state.

Example:

Consider the example of a latch:

```

primitive latch (q, clock, data);
output q; reg q;
input clock, data;
table
//  clock data  q    q+
      0      1 : ? : 1 ;
      0      0 : ? : 0 ;
      1      ? : ? : - ;    // - = no change
endtable
endprimitive

```

This description differs from a combinational UDP model in two ways. First, the output identifier *q* has an additional **reg** declaration to indicate that there is an internal state *q*. The output value of the UDP is always the same as the internal state. Second, a field for the current state, which is separated by colons from the inputs and the output, has been added.

8.4 Edge-sensitive sequential UDPs

In level-sensitive behavior, the values of the inputs and the current state are sufficient to determine the output value. Edge-sensitive behavior differs in that changes in the output are triggered by specific transitions of the inputs. This makes the state table a transition table.

Each table entry can have a transition specification on at most one input. A transition is specified by a pair of values in parenthesis such as (01) or a transition symbol such as *r*. Entries such as the following are illegal:

```
(01)(01)0 : 0 : 1 ;
```

All transitions that do not affect the output shall be explicitly specified. Otherwise, such transitions cause the value of the output to change to *x*. All unspecified transitions default to the output value *x*.

If the behavior of the UDP is sensitive to edges of any input, the desired output state shall be specified for *all* edges of *all* inputs.

Example:

The following example describes a rising edge D flip-flop:

```

primitive d_edge_ff (q, clock, data);
output q; reg q;
input clock, data;
table
  // clock data      q      q+
  // obtain output on rising edge of clock
  (01)  0      :  ?  :  0  ;
  (01)  1      :  ?  :  1  ;
  (0?)  1      :  1  :  1  ;
  (0?)  0      :  0  :  0  ;
  // ignore negative edge of clock
  (?0)  ?      :  ?  :  -  ;
  // ignore data changes on steady clock
  ?      (??)  :  ?  :  -  ;
endtable
endprimitive

```

The terms such as (01) represent transitions of the input values. Specifically, (01) represents a transition from 0 to 1. The first line in the table of the preceding UDP definition is interpreted as follows: when clock changes value from 0 to 1, and data equals 0, the output goes to 0 no matter what the current state

The transition of clock from 0 to x with data equal to 0 and current state equal to 1 will result in the output q going to x.

8.5 Sequential UDP initialization

The initial value on the output port of a sequential UDP can be specified with an initial statement that provides a procedural assignment. The initial statement is optional.

Like initial statements in modules, the initial statement in UDPs begin with the keyword **initial**. The valid contents of initial statements in UDPs and the valid left-hand and right-hand sides of their procedural assignment statements differ from initial statements in modules. A partial list of differences between these two types of initial statements is described in Table 8-2.

Table 8-2—Initial statements in UDPs and modules

Initial statements in UDPs	Initial statements in modules
Contents limited to one procedural assignment statement	Contents can be one procedural statement of any type or a block statement that contains more than one procedural statement
The procedural assignment statement shall assign a value to a reg whose identifier matches the identifier of an output terminal	Procedural assignment statements in initial statements can assign values to a reg whose identifier does not match the identifier of an output terminal
The procedural assignment statement shall assign one of the following values: 1'b1, 1'b0, 1'bx, 1, 0	Procedural assignment statements can assign values of any size, radix, and value

Examples:

Example 1—The following example shows a sequential UDP that contains an initial statement.

```

primitive srff (q, s, r);
output q; reg q;
input s, r;
initial q = 1'b1;
table
//  s  r    q    q+
    1  0 : ? : 1 ;
    f  0 : 1 : - ;
    0  r : ? : 0 ;
    0  f : 0 : - ;
    1  1 : ? : 0 ;
endtable
endprimitive

```

The output *q* has an initial value of 1 at the start of the simulation; a delay specification on an instantiated UDP does not delay the simulation time of the assignment of this initial value to the output. When simulation starts, this value is the current state in the state table. Delays are not permitted in a UDP initial statement.

Example 2—The following example and figure show how values are applied in a module that instantiates a sequential UDP with an initial statement.

```

primitive dff1 (q, clk, d);
input clk, d;
output q; reg q;
initial q = 1'b1;
table
//  clk  d    q    q+
    r    0 : ? : 0 ;
    r    1 : ? : 1 ;
    f    ? : ? : - ;
    ?    * : ? : - ;
endtable
endprimitive

module dff (q, qb, clk, d);
input clk, d;
output q, qb;
    dff1  g1 (qi, clk, d);
    buf #3 g2 (q, qi);
    not #5 g3 (qb, qi);
endmodule

```

The UDP *dff1* contains an initial statement that sets the initial value of its output to 1. The module *dff* contains an instance of UDP *dff1*.

Figure 8-1 shows the schematic of the preceding module and the simulation propagation times of the initial value of the UDP output.

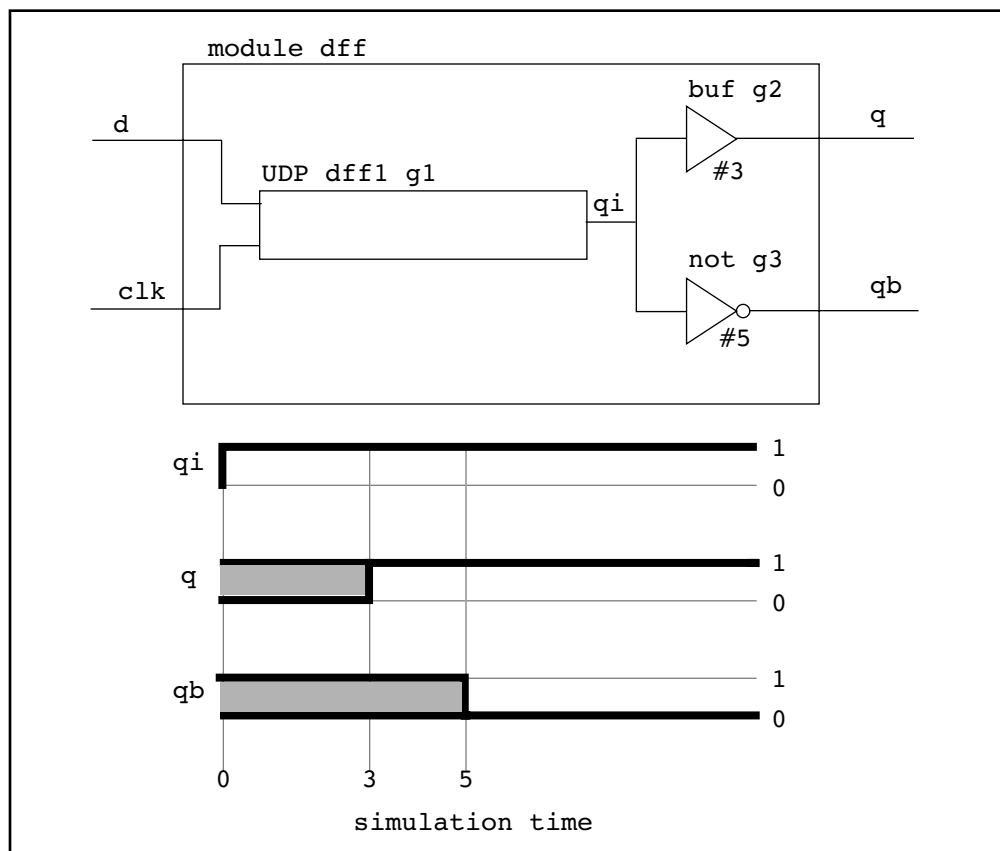


Figure 8-1—Module schematic and simulation times of initial value propagation

In Figure 8-1, the fanout from the UDP output `qi` includes nets `q` and `qb`. At simulation time 0, `qi` changes value to 1. That initial value of `qi` does not propagate to net `q` until simulation time 3, and it does not propagate to net `qb` until simulation time 5.

8.6 UDP instances

The syntax for creating a UDP instance is shown in Syntax 8-2.

```

udp_instantiation ::= udp_identifier [ drive_strength ] [ delay2 ] udp_instance { ,
                        udp_instance } ;
udp_instance ::= [ name_of_udp_instance ] ( output_port_connection ,
                        input_port_connection

```

Syntax 8-2—Syntax for UDP instances

Instances of user-defined primitives are specified inside modules in the same manner as gates (see 7.1). The instance name is optional, just as for gates. The port connection order is as specified in the UDP definition. Only two delays can be specified because `z` is not supported for UDPs. An optional range may be specified for an array of UDP instances. The port connection rules remain the same as outlined in 7.1.

Example:

The following example creates an instance of the D-type flip-flop `d_edge_ff` (defined in 8.4).

```

module flip;
reg clock, data;
parameter p1 = 10;
parameter p2 = 33;
parameter p3 = 12;

d_edge_ff #p3 d_inst (q, clock, data);

initial begin
    data = 1;
    clock = 1;
    #(20 * p1) $finish;
end
always #p1 clock = ~clock;
always #p2 data = ~data;
endmodule

```

8.7 Mixing level-sensitive and edge-sensitive descriptions

UDP definitions allow a mixing of the level-sensitive and the edge-sensitive constructs in the same table. When the input changes, the edge-sensitive cases are processed first, followed by level-sensitive cases. Thus, when level-sensitive and edge-sensitive cases specify different output values, the result is specified by the level-sensitive case.

Example:

```

primitive jk_edge_ff (q, clock, j, k, preset, clear);
output q; reg q;
input clock, j, k, preset, clear;
table
// clock jk pc state output/next state
?    ?? 01 : ? : 1 ; // preset logic
?    ?? *1 : 1 : 1 ;
?    ?? 10 : ? : 0 ; // clear logic
?    ?? 1* : 0 : 0 ;
r    00 00 : 0 : 1 ; // normal clocking cases
r    00 11 : ? : - ;
r    01 11 : ? : 0 ;
r    10 11 : ? : 1 ;
r    11 11 : 0 : 1 ;
r    11 11 : 1 : 0 ;
f    ?? ?? : ? : - ;
b    *? ?? : ? : - ; // j and k transition cases
b    ?* ?? : ? : - ;
endtable
endprimitive

```

In this example, the preset and clear logic is level-sensitive. Whenever the preset and clear combination is 01, the output has value 1. Similarly, whenever the preset and clear combination has value 10, the output has value 0.

The remaining logic is sensitive to edges of the clock. In the normal clocking cases, the flip-flop is sensitive to the rising clock edge, as indicated by an `r` in the clock field in those entries. The insensitivity to the falling edge of clock is indicated by a hyphen (-) in the output field (see Table 8-1) for the entry with an `f` as the value of clock. Remember that the desired output for this input transition shall be specified to avoid unwanted `x` values at the output. The last two entries show that the transitions in `j` and `k` inputs do not change the output on a steady low or high `clock`.

8.8 Level-sensitive dominance

Table 8-3 shows level-sensitive and edge-sensitive entries in the example from 8.7, their level-sensitive or edge-sensitive behavior, and a case of input values that each includes.

Table 8-3—Mixing of level-sensitive and edge-sensitive entries

Entry	Included case	Behavior
? ?? 01: ?; 1;	0 00 01: 0; 1;	Level-sensitive
f ?? ??: ?; -;	f 00 01: 0; 0;	Edge-sensitive

The included cases specify opposite next state values for the same input and current state combination. The level-sensitive included case specifies that when the inputs `clock`, `jk`, and `pc` values are 0, 00, and 01 and the current state is 0, the output changes to 1. The edge-sensitive included case specifies that when `clock` falls from 1 to 0, the other inputs `jk` and `pc` are 00 and 01, and the current state is 0, then the output changes to 0.

When the edge-sensitive case is processed first, followed by the level-sensitive case, the output changes to 1.

Section 9

Behavioral modeling

The language constructs introduced so far allow hardware to be described at a relatively detailed level. Modeling a circuit with logic gates and continuous assignments reflects quite closely the logic structure of the circuit being modeled; however, these constructs do not provide the power of abstraction necessary for describing complex high-level aspects of a system. The procedural constructs described in this section are well suited to tackling problems such as describing a microprocessor or implementing complex timing checks.

This section starts with a brief overview of a behavioral model to provide a context for many types of behavioral statements in the Verilog HDL.

9.1 Behavioral model overview

Verilog *behavioral models* contain *procedural statements* that control the simulation and manipulate variables of the data types previously described. These statements are contained within procedures. Each procedure has an activity flow associated with it.

The activity starts at the control constructs **initial** and **always**. Each *initial* construct and each *always* construct starts a separate activity flow. All of the activity flows are concurrent to model the inherent concurrence of hardware. These constructs are formally described in 9.9.

The following example shows a complete Verilog behavioral model.

```
module behave;
  reg [1:0] a, b;

  initial begin
    a = 'b1;
    b = 'b0;
  end
  always begin
    #50 a = ~a;
  end
  always begin
    #100 b = ~b;
  end
endmodule
```

During simulation of this model, all of the flows defined by the initial and always constructs start together at simulation time zero. The initial constructs execute once, and the always constructs execute repetitively.

In this model, the register variables a and b initialize to 1 and 0 respectively at simulation time zero. The initial construct is then complete and does not execute again during this simulation run. This initial construct contains a *begin-end block* (also called a *sequential block*) of statements. In this begin-end block a is initialized first, followed by b.

The always constructs also start at time zero, but the values of the variables do not change until the times specified by the delay controls (introduced by #) have elapsed. Thus, register **a** inverts after 50 time units and register **b** inverts after 100 time units. Since the always constructs repeat, this model will produce two square waves. The register **a** toggles with a period of 100 time units, and register **b** toggles with a period of 200 time units. The two always constructs proceed concurrently throughout the entire simulation run.

9.2 Procedural assignments

As described in Section 6, procedural assignments are used for updating **reg**, **integer**, **time**, **real**, **realtime**, and memory data types. There is a significant difference between procedural assignments and continuous assignments:

- *Continuous assignments* drive nets and are evaluated and updated whenever an input operand changes value.
- *Procedural assignments* update the value of registers under the control of the procedural flow constructs that surround them.

The right-hand side of a procedural assignment can be any expression that evaluates to a value. The left-hand side shall be a register that receives the assignment from the right-hand side. The left-hand side of a procedural assignment can take one of the following forms:

- **reg**, **integer**, **real**, **realtime**, or **time** register data type: an assignment to the name reference of one of these data types.
- Bit-select of a **reg**, **integer**, or **time** register data type: an assignment to a single bit that leaves the other bits untouched.
- Part-select of a **reg**, **integer**, or **time** register data type: a part-select of one or more contiguous bits that leaves the rest of the bits untouched. Only *constant* expressions shall be legal for specifying the part-select index.
- Memory word: a single word of a memory.
- Concatenation of any of the above: a concatenation of any of the previous four forms can be specified, which effectively partitions the result of the right-hand side expression and assigns the partition parts, in order, to the various parts of the concatenation.

NOTE—Assignment to a **reg** data type differs from assignment to a **real**, **realtime**, **time**, or **integer** variable when the right-hand side evaluates to fewer bits than the left-hand side. Assignment to a **reg** shall not sign-extend.

The Verilog HDL contains two types of procedural assignment statements:

- Blocking procedural assignment statements
- Nonblocking procedural assignment statements

Blocking and nonblocking procedural assignment statements specify different procedural flows in sequential blocks.

9.2.1 Blocking procedural assignments

A *blocking procedural assignment* statement shall be executed before the execution of the statements that follow it in a sequential block (see 9.8.1). A blocking procedural assignment statement shall not prevent the execution of statements that follow it in a parallel block (see 9.8.2).

The syntax for a blocking procedural assignment is given in Syntax 9-1.

```

blocking assignment ::=
    reg_lvalue = [ delay_or_event_control ] expression
delay_or_event_control ::=
    delay_control
    | event_control
    | repeat ( expression ) event_control
reg_lvalue ::=
    reg_identifier
    | reg_identifier [ expression ]
    | reg_identifier [ msb_constant_expression : lsb_constant_expression ]
    | reg_concatenation
delay_control ::=
    # delay_value
    | # ( mintypmax_expression )
event_control ::=
    @ event_identifier
    | @ ( event_expression )
event_expression ::=
    expression
    | event_identifier
    | posedge expression
    | negedge expression
    | event_expression or event_expression

```

Syntax 9-1—Syntax for blocking assignments

In this syntax, *reg_lvalue* is a data type that is valid for a procedural assignment statement, *=* is the assignment operator, and *delay_or_event_control* is the optional intra-assignment timing control. The control can be either a delay control (e.g., #6) or an event_control (e.g., @ (**posedge** *clk*)). The expression is the right-hand side value that shall be assigned to the left-hand side. If *reg_lvalue* requires an evaluation, it shall be evaluated at the time specified by the intra-assignment timing control.

The *=* assignment operator used by blocking procedural assignments is also used by procedural continuous assignments and continuous assignments.

Example:

The following examples show blocking procedural assignments.

```

rega = 0;
rega[3] = 1;           // a bit-select
rega[3:5] = 7;         // a part-select
mema[address] = 8'hff; // assignment to a mem element
{carry, acc} = rega + regb; // a concatenation

```

9.2.2 The nonblocking procedural assignment

The *nonblocking procedural assignment* allows assignment scheduling without blocking the procedural flow. The nonblocking procedural assignment statement can be used whenever several register assignments within the same time step can be made without regard to order or dependence upon each other.

The syntax for a nonblocking procedural assignment is given in Syntax 9-2.

```
non-blocking assignment ::=
    reg_lvalue <= [ delay_or_event_control ] expression
```

Syntax 9-2—Syntax for nonblocking assignments

In this syntax, `reg_lvalue` is a data type that is valid for a procedural assignment statement, `<=` is the nonblocking assignment operator, and `delay_or_event_control` is the optional intra-assignment timing control. If `reg_lvalue` requires an evaluation, it shall be evaluated at the same time as the expression on the right-hand side. The order of evaluation of the `reg_lvalue` and the expression on the right-hand side is undefined if timing control is not specified.

The nonblocking assignment operator is the same operator as the less-than-or-equal-to relational operator. The interpretation shall be decided from the context in which `<=` appears. When `<=` is used in an expression, it shall be interpreted as a relational operator, and when it is used in a nonblocking procedural assignment, it shall be interpreted as an assignment operator.

The nonblocking procedural assignments shall be evaluated in two steps as discussed in Section 5. These two steps are shown in the following example.

Example 1:

<pre>module evaluates2 (out); output out; reg a, b, c; initial begin a = 0; b = 1; c = 0; end always c = #5 ~c; always @(posedge c) begin a <= b; // evaluates, schedules, b <= a; // and executes in two steps end endmodule</pre>	<p>Step 1: The simulator evaluates the right-hand side of the non-blocking assignments and schedules the assignments of the new values at posedge c.</p> <p>Step 2: At posedge c, the simulator updates the left-hand side of each nonblocking assignment statement.</p>	<p>Nonblocking assignment schedules changes at time 5</p> <div style="border: 1px solid black; padding: 5px; margin: 10px 0; text-align: center;"> a = 0 b = 1 </div> <p>Assignment values are:</p> <div style="border: 1px solid black; padding: 5px; margin: 10px 0; text-align: center;"> a = 1 b = 0 </div>
--	--	---

At the end of the time step means that the nonblocking assignments are the last assignments executed in a time step—with one exception. Nonblocking assignment events can create blocking assignment events. These blocking assignment events shall be processed after the scheduled nonblocking events.

Unlike an event or delay control for blocking assignments, the nonblocking assignment does not block the procedural flow. The nonblocking assignment evaluates and schedules the assignment, but it does not block the execution of subsequent statements in a **begin-end** block.

Example 2:

<pre> //non_block1.v module non_block1; reg a, b, c, d, e, f; //blocking assignments initial begin a = #10 1; // a will be assigned 1 at time 10 b = #2 0; // b will be assigned 0 at time 12 c = #4 1; // c will be assigned 1 at time 16 end //non-blocking assignments initial begin d <= #10 1; // d will be assigned 1 at time 10 e <= #2 0; // e will be assigned 0 at time 2 f <= #4 1; // f will be assigned 1 at time 4 end endmodule </pre>	<p>scheduled changes at time 2</p> <div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 5px auto;">e = 0</div> <p>scheduled changes at time 4</p> <div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 5px auto;">f = 1</div> <p>scheduled changes at time 10</p> <div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 5px auto;">d = 1</div>
---	---

As shown in the previous example, the simulator evaluates and schedules assignments for the end of the current time step and can perform swapping operations with the nonblocking procedural assignments.

Example 3:

<pre> //non_block1.v module non_block1; reg a, b; initial begin a = 0; b = 1; a <= b; // evaluates, schedules, and b <= a; // executes in two steps end initial begin \$monitor (\$time, , "a = %b b = %b", a, b); #100 \$finish; end endmodule </pre>	<p>Step 1: The simulator evaluates the right-hand side of the nonblocking assignments and schedules the assignments for the end of the current time step.</p> <p>Step 2: At the end of the current time step, the simulator updates the left-hand side of each nonblocking assignment statement.</p>
--	--

assignment values are:

a = 1

b = 0

When multiple nonblocking assignments are scheduled to occur in the same register in a particular time slot, the order in which the assignments are evaluated is not guaranteed—the final value of the register is indeterminate. As shown in the following example, the value of register *a* is not known until the end of time step 4.

Example 4:

```
module multiple2 (out);
output out;
reg a;

initial a = 1;
// The assigned value of the register is indeterminate
initial begin
    a <= #4 0; // schedules a = 0 at time 4
    a <= #4 1; // schedules a = 1 at time 4
end // At time 4, a = ??
endmodule
```

If the simulator executes two procedural blocks concurrently, and if these procedural blocks contain nonblocking assignment operators to the same register, the final value of that register is indeterminate. For example, the value of register *a* is indeterminate in the following example.

Example 5:

```
module multiple3 ;
reg a;

initial a = 1;
initial a <= #4 0; // schedules 0 at time 4
initial a <= #4 1; // schedules 1 at time 4

// At time 4, a = ??
endmodule
```

When multiple nonblocking assignments with timing controls are made to the same register, the assignments are made without cancelling nonblocking assignments scheduled at other times. Scheduling an assignment to a register at the same time as a previously scheduled assignment to the same register shall result in an arbitrary order of assignment to that register, and, hence, the final value of that register cannot be predicted.

The following example shows how the value of *i[0]* is assigned to *r1* and how the assignments are scheduled to occur after each time delay.

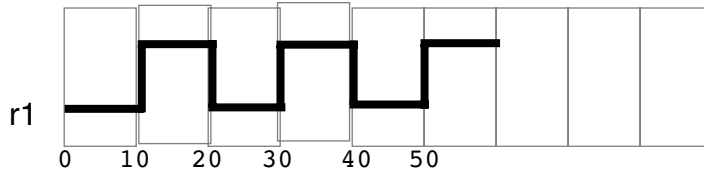
Example 6:

```

module multiple;
reg r1;
reg [2:0] i;

initial begin
  // starts at time 0, does not hold the block
  r1 = 0;
  // makes assignments to r1 without cancelling previous assignments
  for (i = 0; i <= 5; i = i+1)
    r1 <= # (i*10) i[0];
end
endmodule

```



9.3 Procedural continuous assignments

The *procedural continuous assignments* (using keywords **assign** and **force**) are procedural statements that allow expressions to be driven continuously onto registers or nets. The syntax for these statements is given in Syntax 9-3.

```

procedural_continuous_assignments ::=
    assign reg_assignment ;
    | deassign reg_lvalue ;
    | force reg_assignment ;
    | force net_assignment ;
    | release reg_lvalue ;
    | release net_lvalue ;
reg_assignment ::=
    reg_lvalue = expression
net_assignment ::=
    net_lvalue = expression

```

Syntax 9-3—Syntax for procedural continuous assignments

The left-hand side of the assignment in the *assign statement* shall be a register reference or a concatenation of registers. It shall not be a memory word (array reference) or a bit-select or a part-select of a register.

In contrast, the left-hand side of the assignment in the *force statement* can be a register reference or a net reference. It can be a concatenation of any of the above. Bit-selects and part-selects of vector registers or vector nets are not allowed.

9.3.1 The assign and deassign procedural statements

The *assign* procedural continuous assignment statement shall override all procedural assignments to a register. The *deassign* procedural statement shall end a procedural continuous assignment to a register. The value of the register shall remain the same until the register is assigned a new value through a procedural assignment or a procedural continuous assignment. The assign and deassign procedural statements allow, for example, modeling of asynchronous

clear/preset on a D-type edge-triggered flip-flop, where the clock is inhibited when the clear or preset is active.

If the keyword **assign** is applied to a register for which there is already a procedural continuous assignment, then this new procedural continuous assignment shall deassign the register before making the new procedural continuous assignment.

Example:

The following example shows a use of the assign and deassign procedural statements in a behavioral description of a D-type flip-flop with preset and clear inputs.

```

module dff (q, d, clear, preset, clock);
output q;
input d, clear, preset, clock;
reg q;

always @(clear or preset)
    if (!clear)
        assign q = 0;
    else if (!preset)
        assign q = 1;
    else
        deassign q;

always @(posedge clock)
    q = d;
endmodule

```

If either **clear** or **preset** is low, then the output **q** will be held continuously to the appropriate constant value and a positive edge on the **clock** will not affect **q**. When both the **clear** and **preset** are high, then **q** is deassigned.

9.3.2 The force and release procedural statements

Another form of procedural continuous assignment is provided by the *force* and *release* procedural statements. These statements have a similar effect to the assign-deassign pair, but a force can be applied to nets as well as to registers. The left-hand side of the assignment can be a register, a net, a constant bit-select of a vector net, a part-select of a vector net, or a concatenation. It cannot be a memory word (array reference) or a bit-select or a part-select of a vector register.

A *force* statement to a register shall override a procedural assignment or procedural continuous assignment that takes place on the register until a release procedural statement is executed on the register. After the *release* procedural statement is executed, the register shall not immediately change value (as would a net that is forced). The value specified in the force statement shall be maintained in the register until the next procedural assignment takes place, except in the case where a procedural continuous assignment is active on the register.

A force procedural statement on a net overrides all drivers of the net—gate outputs, module outputs, and continuous assignments—until a release procedural statement is executed on the net.

Releasing a register that currently has an active procedural continuous assignment shall re-establish that assignment.

Example:

```

module test;
reg a, b, c, d;
wire e;

and and1 (e, a, b, c);

initial begin
    $monitor ("%d d=%b,e=%b", $time, d, e);
    assign d = a & b & c;
    a = 1;
    b = 0;
    c = 1;
    #10;
    force d = (a | b | c);
    force e = (a | b | c);
    #10 $stop;
    release d;
    release e;
    #10 $finish;
end
endmodule

```

```

Results:
    0 d=0,e=0
   10 d=1,e=1
   20 d=0,e=0

```

In this example, an **and** gate instance **and1** is “patched” as an **or** gate by a force procedural statement that forces its output to the value of its logical or inputs, and an assign procedural statement of logical and values is “patched” as an assign procedural statement of logical or values.

The right-hand side of a procedural continuous assignment or a force statement can be an expression. This shall be treated just as a continuous assignment; that is, if any variable on the right-hand side of the assignment changes, the assignment shall be re-evaluated while the assign or force is in effect. For example:

```
force a = b + f(c) ;
```

Here, if **b** changes or **c** changes, **a** will be forced to the new value of the expression **b+f (c)**.

9.4 Conditional statement

The *conditional statement* (or *if-else* statement) is used to make a decision as to whether a statement is executed or not. Formally, the syntax is given in Syntax 9-4.

```

conditional_statement ::=
    if ( expression ) statement_or_null [ else statement_or_null ]
statement_or_null ::= statement | ;

```

Syntax 9-4—Syntax of if statement

If the expression evaluates to true (that is, has a nonzero known value), the first statement shall be executed. If it evaluates to false (has a zero value or the value is x or z), the first statement shall not execute. If there is an else statement and expression is false, the else statement shall be executed.

Since the numeric value of the `if` expression is tested for being zero, certain shortcuts are possible. For example, the following two statements express the same logic:

```
if (expression)
if (expression != 0)
```

Because the else part of an if-else is optional, there can be confusion when an else is omitted from a nested if sequence. This is resolved by always associating the else with the closest previous if that lacks an else. In the example below, the else goes with the inner if, as shown by indentation.

```
if (index > 0)
  if (rega > regb)
    result = rega;
  else          // else applies to preceding if
    result = regb;
```

If that association is not desired, a begin-end block statement shall be used to force the proper association, as shown below.

```
if (index > 0) begin
  if (rega > regb)
    result = rega;
end
else result = regb;
```

9.4.1 If-else-if construct

The following construction occurs so often that it is worth a brief separate discussion:

```
if_else_if_statement ::=
  if (expression) statement_or_null
  { else if (expression) statement_or_null }
  else statement
```

Syntax 9-5—Syntax of if-else-if construct

This sequence of if statements (known as an *if-else-if* construct) is the most general way of writing a multiway decision. The expressions shall be evaluated in order; if any expression is true, the statement associated with it shall be executed, and this shall terminate the whole chain. Each statement is either a single statement or a block of statements.

The last else part of the if-else-if construct handles the none-of-the-above or default case where none of the other conditions were satisfied. Sometimes there is no explicit action for the default; in that case, the trailing else statement can be omitted or it can be used for error checking to catch an impossible condition.

Example:

The following module fragment uses the if-else statement to test the variable `index` to decide whether one of three `modify_seg` registers has to be added to the memory address, and which increment is to be added to the `index` register. The first ten lines declare the registers and parameters.

```
// declare registers and parameters
reg [31:0] instruction, segment_area[255:0];
reg [7:0] index;
reg [5:0] modify_seg1,
    modify_seg2,
    modify_seg3;
parameter
    segment1 = 0, inc_seg1 = 1,
    segment2 = 20, inc_seg2 = 2,
    segment3 = 64, inc_seg3 = 4,
    data = 128;

// test the index variable
if (index < segment2) begin
    instruction = segment_area [index + modify_seg1];
    index = index + inc_seg1;
end
else if (index < segment3) begin
    instruction = segment_area [index + modify_seg2];
    index = index + inc_seg2;
end
else if (index < data) begin
    instruction = segment_area [index + modify_seg3];
    index = index + inc_seg3;
end
else
    instruction = segment_area [index];
```

9.5 Case statement

The *case* statement is a multiway decision statement that tests whether an expression matches one of a number of other expressions and branches accordingly. The case statement has the syntax shown in Syntax 9-6.

```
case_statement ::=
    | case ( expression ) case_item { case_item } endcase
    | casez ( expression ) case_item { case_item } endcase
    | casex ( expression ) case_item { case_item } endcase
case_item ::=
    expression { , expression } : statement_or_null
    | default [ : ] statement_or_null
```

Syntax 9-6—Syntax for case statement

The *default* statement shall be optional. Use of multiple default statements in one case statement shall be illegal.

The case expression and the case item expression can be computed at runtime; neither expression is required to be a constant expression.

Examples:

A simple example of the use of the case statement is the decoding of register `rega` to produce a value for `result` as follows:

```

reg [15:0] rega;
reg [9:0] result;

case (rega)
    16'd0: result = 10'b0111111111;
    16'd1: result = 10'b1011111111;
    16'd2: result = 10'b1101111111;
    16'd3: result = 10'b1110111111;
    16'd4: result = 10'b1111011111;
    16'd5: result = 10'b1111101111;
    16'd6: result = 10'b1111110111;
    16'd7: result = 10'b1111111011;
    16'd8: result = 10'b1111111101;
    16'd9: result = 10'b1111111110;
    default result = 'bx;
endcase

```

The *case item expressions* shall be evaluated and compared in the exact order in which they are given. During the linear search, if one of the **case** item expressions matches the case expression given in parentheses, then the statement associated with that case item shall be executed. If all comparisons fail and the default item is given, then the default item statement shall be executed. If the default statement is not given and all of the comparisons fail, then none of the case item statements shall be executed.

Apart from syntax, the **case** statement differs from the multiway if-else-if construct in two important ways:

- a) The conditional expressions in the if-else-if construct are more general than comparing one expression with several others, as in the case statement.
- b) The case statement provides a definitive result when there are **x** and **z** values in an expression.

In a case expression comparison, the comparison only succeeds when each bit matches exactly with respect to the values **0**, **1**, **x**, and **z**. As a consequence, care is needed in specifying the expressions in the **case** statement. The bit length of all the expressions shall be equal so that exact bit-wise matching can be performed. The length of all the **case** item expressions, as well as the case expression in the parentheses, shall be made equal to the length of the longest case expression and case item expression.

NOTE—The default length of **x** and **z** is same as the default length of an integer.

The reason for providing a case expression comparison that handles the **x** and **z** values is that it provides a mechanism for detecting such values and reducing the pessimism that can be generated by their presence.

Examples:

Example 1—The following example illustrates the use of a case statement to handle **x** and **z** values properly.

```
case (select[1:2])
  2'b00: result = 0;
  2'b01: result = flaga;
  2'b0x,
  2'b0z: result = flaga ? 'bx : 0;
  2'b10: result = flagb;
  2'bx0,
  2'bz0: result = flagb ? 'bx : 0;
  default result = 'bx;
endcase
```

In this example, if `select[1]` is 0 and `flaga` is 0, then whether the value of `select[2]` is `x` or `z`, `result` should be 0—which is resolved by the third case.

Example 2—The following example shows another way to use a case statement to detect `x` and `z` values.

```
case (sig)
  1'bz: $display("signal is floating");
  1'bx: $display("signal is unknown");
  default: $display("signal is %b", sig);
endcase
```

9.5.1 Case statement with don't-cares

Two other types of case statements are provided to allow handling of don't-care conditions in the case comparisons. One of these treats high-impedance values (`z`) as don't-cares, and the other treats both high-impedance and unknown (`x`) values as don't-cares.

These case statements can be used in the same way as the traditional case statement, but they begin with keywords **casez** and **casex** respectively.

Don't-care values (`z` values for **casez**, `z` and `x` values for **casex**) in any bit of either the case expression or the case items shall be treated as don't-care conditions during the comparison, and that bit position shall not be considered. The don't-care conditions in case expression can be used to control dynamically which bits should be compared at any time.

The syntax of literal numbers allows the use of the question mark (?) in place of `z` in these case statements. This provides a convenient format for specification of don't-care bits in case statements.

Examples:

Example 1—The following is an example of the **casez** statement. It demonstrates an instruction decode, where values of the most significant bits select which task should be called. If the most significant bit of `ir` is a 1, then the task `instruction1` is called, regardless of the values of the other bits of `ir`.

```

reg [7:0] ir;

casez (ir)
  8'b1??????? : instruction1(ir);
  8'b01??????? : instruction2(ir);
  8'b00010??? : instruction3(ir);
  8'b000001?? : instruction4(ir);
endcase

```

Example 2—The following is an example of the casex statement. It demonstrates an extreme case of how don't-care conditions can be dynamically controlled during simulation. In this case, if $r = 8'b01100110$, then the task `stat2` is called.

```

reg [7:0] r, mask;

mask = 8'bxx0x0x0x0;
casex (r ^ mask)
  8'b001100xx : stat1;
  8'b1100xx00 : stat2;
  8'b00xx0011 : stat3;
  8'bx010100 : stat4;
endcase

```

9.5.2 Constant expression in case statement

A constant expression can be used for case expression. The value of the constant expression shall be compared against case item expressions.

Example:

The following example demonstrates the usage by modeling a 3-bit priority encoder.

```

reg [2:0] encode ;

case (1)
  encode[2] : $display("Select Line 2") ;
  encode[1] : $display("Select Line 1") ;
  encode[0] : $display("Select Line 0") ;
  default : $display("Error: One of the bits expected ON");
endcase

```

Note that the case expression is a constant expression (1). The case items are expressions (bit-selects) and are compared against the constant expression for a match.

9.6 Looping statements

There are four types of looping statements. These statements provide a means of controlling the execution of a statement zero, one, or more times.

<i>forever</i>	Continuously executes a statement.
<i>repeat</i>	Executes a statement a fixed number of times. If the expression evaluates to unknown or high impedance, it shall be treated as zero, and no statement shall be executed.
<i>while</i>	Executes a statement until an expression becomes false. If the expression starts out false, the statement shall not be executed at all.
<i>for</i>	Controls execution of its associated statement(s) by a three-step process, as follows: <ul style="list-style-type: none"> a) Executes an assignment normally used to initialize a register that controls the number of loops executed. b) Evaluates an expression—if the result is zero, the for-loop shall exit, and if it is not zero, the for-loop shall execute its associated statement(s) and then perform step c. If the expression evaluates to an unknown or high-impedance value, it shall be treated as zero. c) Executes an assignment normally used to modify the value of the loop-control register, then repeats step b.

Syntax 9-7 shows the syntax for various looping statements.

```

looping_statements ::=
    forever statement
  | repeat ( expression ) statement
  | while ( expression ) statement
  | for ( reg_assignment ; expression ; reg_assignment ) statement

```

Syntax 9-7—Syntax for the looping statements

The rest of this clause presents examples for three of the looping statements. The forever loop should only be used in conjunction with the timing controls or the disable statement, therefore, this example is presented in 9.7.2.

Examples:

Example 1—Repeat statement: In the following example of a repeat loop, add and shift operators implement a multiplier.

```

parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;

begin : mult
    reg [longsize:1] shift_opa, shift_opb;
    shift_opa = opa;
    shift_opb = opb;
    result = 0;
    repeat (size) begin
        if (shift_opb[1])
            result = result + shift_opa;
        shift_opa = shift_opa << 1;
        shift_opb = shift_opb >> 1;
    end
end

```

Example 2—While statement: The following example counts the number of logic 1 values in `rega`.

```

begin : count1s
    reg [7:0] tempreg;
    count = 0;
    tempreg = rega;
    while (tempreg) begin
        if (tempreg[0])
            count = count + 1;
        tempreg = tempreg >> 1;
    end
end

```

Example 3—For statement: The for statement accomplishes the same results as the following pseudo-code that is based on the while loop:

```

begin
    initial_assignment;
    while (condition) begin
        statement
        step_assignment;
    end
end

```

The for loop implements this logic while using only two lines, as shown in the pseudo-code below.

```

for (initial_assignment; condition; step_assignment)
    statement

```

9.7 Procedural timing controls

The Verilog HDL has two types of explicit timing control over when procedural statements can occur. The first type is a *delay control*, in which an expression specifies the time duration between initially encountering the statement and when the statement actually executes. The delay expression can be a dynamic function of the state of the circuit, but it can be a simple number that separates statement executions in time. The delay control is an important feature when specifying stimulus waveform descriptions. It is described in 9.7.1 and 9.7.6.

The second type of timing control is the *event expression*, which allows statement execution to be delayed until the occurrence of some simulation event occurring in a procedure executing concurrently with this procedure. A simulation event can be a change of value on a net or register (an implicit event) or the occurrence of an explicitly named event that is triggered from other procedures (an explicit event). Most often, an event control is a positive or negative edge on a clock signal. Event control is discussed in 9.7.2 through 9.7.6.

The procedural statements encountered so far all execute without advancing simulation time. Simulation time can advance by one of the following three methods:

- A **delay** control, which is introduced by the symbol #
- An **event** control, which is introduced by the symbol @
- The **wait** statement, which operates like a combination of the event control and the while loop

Syntax 9-8 describes timing control in procedural statements.

```
procedural_timing_control_statement ::=
    delay_or_event_control statement_or_null
delay_or_event_control ::=
    delay_control
    | event_control
delay_control ::=
    # delay_value
    | # ( mintypmax_expression )
event_control ::=
    @ event_identifier
    | @ ( event_expression )
event_expression ::=
    expression
    | event_identifier
    | posedge expression
    | negedge expression
    | event_expression or event_expression
```

Syntax 9-8—Syntax for procedural timing control

The gate and net delays also advance simulation time, as discussed in Section 6. The next subclauses discuss the three procedural timing control methods.

9.7.1 Delay control

A procedural statement following the delay control shall be delayed in its execution with respect to the procedural statement preceding the delay control by the specified delay. If the delay expression evaluates to an unknown or high-impedance value, it shall be interpreted as zero delay. If the delay expression evaluates to a negative value, it shall be interpreted as a 2's complement unsigned integer of the same size as a time variable.

Examples:

Example 1—The following example delays the execution of the assignment by 10 time units:

```
#10 rega = regb;
```

Example 2—The next three examples provide an expression following the number sign (#). Execution of the assignment is delayed by the amount of simulation time specified by the value of the expression.

```
#d rega = regb;           // d is defined as a parameter
#((d+e)/2) rega = regb;    // delay is average of d and e
#regr regr = regr + 1;     // delay is the value in regr
```

9.7.2 Event control

The execution of a procedural statement can be synchronized with a value change on a net or register or the occurrence of a declared event. The value changes on nets and registers can be used as events to trigger the execution of a statement. This is known as detecting *an implicit event*. The event can also be based on the direction of the change—that is, towards the value 1 (**posedge**) or towards the value 0 (**negedge**). The behavior of posedge and negedge event is shown in Table 9-1 and can be described as follows:

- A *negedge* shall be detected on the transition from 1 to x, z, or 0, and from x or z to 0
- A *posedge* shall be detected on the transition from 0 to x, z, or 1, and from x or z to 1

Table 9-1—Detecting posedge and negedge

To From	0	1	x	z
0	No edge	posedge	posedge	posedge
1	negedge	No edge	negedge	negedge
x	negedge	posedge	No edge	No edge
z	negedge	posedge	No edge	No edge

If the expression evaluates to more than a 1-bit result, the edge transition shall be detected on the least significant bit of the result. The change of value in any of the operands without a change in the value of the least significant bit of the expression result shall not be detected as an edge.

Example:

The following example shows illustrations of edge-controlled statements.

```
@r rega = regb; // controlled by any value change in the register r
@(posedge clock) rega = regb; // controlled by posedge on clock
forever @(negedge clock) rega = regb; // controlled by negative edge
```

9.7.3 Named events

A new data type, in addition to net and register, called “event” can be declared. An identifier declared as an event data type is called a *named event*. A named event can be triggered explicitly. It can be used in an event expression to control the execution of procedural statements in the same manner as event control described in 9.7.1. Named events can be made to occur from a procedure. This allows control over the enabling of multiple actions in other procedures.

An event name shall be declared explicitly before it is used. Syntax 9-9 gives the syntax for declaring events.

```
event_declaration ::=  
    event event_identifier { , event_identifier } ;
```

Syntax 9-9—Syntax for event declaration

An event shall not hold any data. The following are the characteristics of a named event:

- It can be made to occur at any particular time
- It has no time duration
- Its occurrence can be recognized by using the event control syntax described in 9.7

A declared event is made to occur by the activation of an event triggering statement with the syntax given in Syntax 9-10.

```
event_trigger ::= -> event_identifier ;
```

Syntax 9-10—Syntax for event trigger

An event-controlled statement (for example, `@trig rega = regb;`) shall cause simulation of its containing procedure to wait until some other procedure executes the appropriate event-triggering statement (for example, `-> trig`).

Named events and event control give a powerful and efficient means of describing the communication between, and synchronization of, two or more concurrently active processes. A basic example of this is a small waveform clock generator that synchronizes control of a synchronous circuit by signalling the occurrence of an explicit event periodically while the circuit waits for the event to occur.

9.7.4 Event or operator

The logical or of any number of events can be expressed such that the occurrence of any one of the events triggers the execution of the procedural statement that follows it. The keyword **or** is used as an event logical or operator.

Examples:

The next two examples show the logical or of two and three events respectively.

```
@(trig or enable) rega = regb;          // controlled by trig or enable  
@(posedge clk_a or posedge clk_b or trig) rega = regb;
```

9.7.5 Level-sensitive event control

The execution of a procedural statement can also be delayed until a condition becomes true. This is accomplished using the *wait* statement, which is a special form of event control. The nature of the wait statement is level-sensitive, as opposed to basic event control (specified by the @ character), which is edge-sensitive.

The wait statement shall evaluate a condition, and, if it is false, the procedural statements following the wait statement shall remain blocked until that condition becomes true before continuing. The wait statement has the form given in Syntax 9-11.

```
wait_statement ::=
    wait ( expression ) statement_or_null
```

Syntax 9-11—Syntax for wait statement

Example:

The following example shows the use of the wait statement to accomplish level-sensitive event control.

```
begin
    wait (!enable) #10 a = b;
    #10 c = d;
end
```

If the value of *enable* is 1 when the block is entered, the wait statement will delay the evaluation of the next statement (*#10 a = b;*) until the value of *enable* changes to 0. If *enable* is already 0 when the *begin-end* block is entered, then the assignment “*a = b;*” is evaluated after a delay of 10 and no additional delay occurs.

9.7.6 Intra-assignment timing controls

The delay and event control constructs previously described precede a statement and delay its execution. In contrast, the *intra-assignment delay and event controls* are contained within an assignment statement and modify the flow of activity in a different way. This subclause describes the purpose of intra-assignment timing controls and the repeat timing control that can be used in intra-assignment delays.

An intra-assignment delay or event control shall delay the assignment of the new value to the left-hand side, but the right-hand side expression shall be evaluated before the delay, instead of after the delay. The syntax for intra-assignment delay and event control is given in Syntax 9-12.

```
intra_assignment_timing_control_statement ::=
    reg_lvalue = [ intra_assignment_timing_control ] expression ;
| reg_lvalue <= [ intra_assignment_timing_control ] expression ;
intra_assignment_timing_control ::=
    delay_control
| event_control
| repeat_event_control
repeat_event_control ::=
    repeat ( expression ) @ ( event_expression )
```

Syntax 9-12—Syntax for intra-assignment delay and event control

The intra-assignment delay and event control can be applied to both blocking assignments and nonblocking assignments. The event expression shall be resolved to a 1-bit value. The *repeat* event control shall specify an intra-assignment delay of a specified number of occurrences of an event. This construct is convenient when events have to be synchronized with counts of clock signals.

Examples:

Table 9-2 illustrates the philosophy of intra-assignment timing controls by showing the code that could accomplish the same timing effect without using intra-assignment.

Table 9-2—Intra-assignment timing control equivalence

Intra-assignment timing control	
With intra-assignment construct	Without intra-assignment construct
<code>a = #5 b;</code>	<pre>begin temp = b; #5 a = temp; end</pre>
<code>a = @(posedge clk) b;</code>	<pre>begin temp = b; @(posedge clk) a = temp; end</pre>
<pre>a = repeat(3) @(posedge clk) b;</pre>	<pre>begin temp = b; @(posedge clk); @(posedge clk); @(posedge clk) a = temp; end</pre>

The next three examples use the fork-join behavioral construct. All statements between the keywords **fork** and **join** execute concurrently. This construct is described in more detail in 9.8.2.

The following example shows a race condition that could be prevented by using intra-assignment timing control:

```
fork
    #5 a = b;
    #5 b = a;
join
```

The code in this example samples and sets the values of both **a** and **b** at the same simulation time, thereby creating a race condition. The intra-assignment form of timing control used in the next example prevents this race condition.

```
fork                                // data swap
    a = #5 b;
    b = #5 a;
join
```

Intra-assignment timing control works because the intra-assignment delay causes the values of **a** and **b** to be evaluated *before* the delay, and the assignments to be made *after* the delay. Some existing tools that implement intra-assignment timing control use temporary storage in evaluating each expression on the right-hand side.

Intra-assignment waiting for *events* is also effective. In the following example, the right-hand side expressions are evaluated when the assignment statements are encountered, but the assignments are delayed until the rising edge of the clock signal.

```

fork                                // data shift
    a = @(posedge clk) b;
    b = @(posedge clk) c;
join

```

The following is an example of a repeat event control as the intra-assignment delay of a nonblocking assignment:

```
a <= repeat(5) @(posedge clk) data;
```

Figure 9-1 illustrates the activities that result from this `repeat` event control.

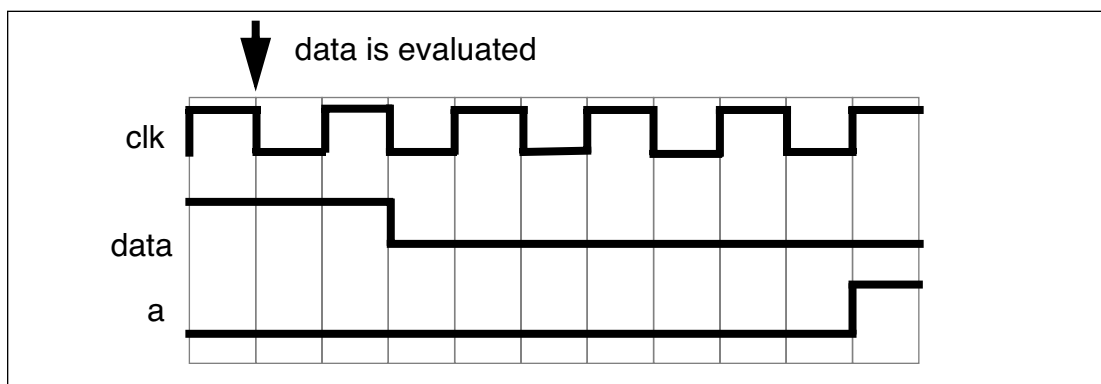


Figure 9-1 — Repeat event control utilizing a clock edge

In this example, the value of `data` is evaluated when the assignment is encountered. After five occurrences of `posedge clk`, `a` is assigned the value of `data`.

The following is an example of a repeat event control as the intra-assignment delay of a procedural assignment:

```
a = repeat(num) @(clk) data;
```

In this example, the value of `data` is evaluated when the assignment is encountered. After the number of transitions of `clk` equals the value of `num`, `a` is assigned the value of `data`.

The following is an example of a repeat event control with expressions containing operations to specify both the number of event occurrences and the event that is counted:

```
a <= repeat(a+b) @(posedge phi1 or negedge phi2) data;
```

In this example, the value of `data` is evaluated when the assignment is encountered. After the sum of the positive edges of `phi1` and the negative edges of `phi2` equals the sum of `a` and `b`, `a` is assigned the value of `data`. Even if `posedge phi1` and `negedge phi2` occurred at the same simulation time, each will be detected separately.

9.8 Block statements

The *block statements* are a means of grouping two or more statements together so that they act syntactically like a single statement. There are two types of blocks in the Verilog HDL:

- *Sequential block*, also called *begin-end block*
- *Parallel block*, also called *fork-join block*

The sequential block shall be delimited by the keywords **begin** and **end**. The procedural statements in sequential block shall be executed sequentially in the given order.

The parallel block shall be delimited by the keywords **fork** and **join**. The procedural statements in parallel block shall be executed concurrently.

9.8.1 Sequential blocks

A *sequential block* shall have the following characteristics:

- Statements shall be executed in sequence, one after another
- Delay values for each statement shall be treated relative to the simulation time of the execution of the previous statement
- Control shall pass out of the block after the last statement executes

Syntax 9-13 gives the formal syntax for a sequential block.

```
seq_block ::=  
    begin [ : block_identifier { block_item_declaration } ] { statement } end  
block_item_declaration ::=  
    parameter_declaration  
    | reg_declaration  
    | integer_declaration  
    | real_declaration  
    | time_declaration  
    | realtime_declaration  
    | event_declaration
```

Syntax 9-13—Syntax for the sequential block

Examples:

Example 1—A sequential block enables the following two assignments to have a deterministic result:

```
begin  
    areg = breg;  
    creg = areg;          // creg stores the value of breg  
end
```

The first assignment is performed and **areg** is updated before control passes to the second assignment.

Example 2—Delay control can be used in a sequential block to separate the two assignments in time.

```
begin  
    areg = breg;  
    @(posedge clock) creg = areg; // assignment delayed until  
end                               // posedge on clock
```

Example 3—The following example shows how the combination of the sequential block and delay control can be used to specify a time-sequenced waveform.

```

parameter d = 50;    // d declared as a parameter and
reg [7:0] r;         // r declared as an 8-bit register

begin    // a waveform controlled by sequential delay
    #d r = 'h35;
    #d r = 'hE2;
    #d r = 'h00;
    #d r = 'hF7;
    #d -> end_wave; // trigger an event called end_wave
end

```

9.8.2 Parallel blocks

A *parallel block* shall have the following characteristics:

- Statements shall execute concurrently
- Delay values for each statement shall be considered relative to the simulation time of entering the block
- Delay control can be used to provide time-ordering for assignments
- Control shall pass out of the block when the last time-ordered statement executes

Syntax 9-14 gives the formal syntax for a parallel block.

```

par_block ::=
    fork [ : block_identifier { block_item_declaration } ] { statement } join

```

Syntax 9-14—Syntax for the parallel block

The timing controls in a fork-join block do not have to be ordered sequentially in time.

Example:

The following example codes the waveform description shown in example 3 of 9.8.1 by using a parallel block instead of a sequential block. The waveform produced on the register is exactly the same for both implementations.

```

fork
    #50 r = 'h35;
    #100 r = 'hE2;
    #150 r = 'h00;
    #200 r = 'hF7;
    #250 -> end_wave;
join

```

9.8.3 Block names

Both sequential and parallel blocks can be named by adding : name_of_block after the keywords **begin** or **fork**.

The naming of blocks serves several purposes:

- It allows local registers to be declared for the block.
- It allows the block to be referenced in statements such as the disable statement (Section 11).

All registers shall be static—that is, a unique location exists for all registers and leaving or entering blocks shall not affect the values stored in them.

The block names give a means of uniquely identifying all registers at any simulation time.

9.8.4 Start and finish times

Both sequential and procedural blocks have the notion of a start and finish time. For sequential blocks, the start time is when the first statement is executed, and the finish time is when the last statement has been executed. For parallel blocks, the start time is the same for all the statements, and the finish time is when the last time-ordered statement has been executed.

Sequential and parallel blocks can be embedded within each other, allowing complex control structures to be expressed easily and with a high degree of structure. When blocks are embedded within each other, the timing of when a block starts and finishes is important. Execution shall not continue to the statement following a block until the finish time for the block has been reached—that is, until the block has completely finished executing.

Examples:

Example 1—The following example shows the statements from the example in 9.8.2 written in the reverse order and still producing the same waveform.

```

fork
    #250 -> end_wave;
    #200 r = 'hF7;
    #150 r = 'h00;
    #100 r = 'hE2;
    #50 r = 'h35;
join

```

Example 2—When an assignment is to be made after two separate events have occurred, known as the *joining of events*, a `fork-join` block can be useful.

```

begin
    fork
        @Aevent;
        @Bevent;
    join
        areg = breg;
end

```

The two events can occur in any order (or even at the same simulation time) and the `fork-join` block will complete and the assignment will be made. In contrast to this, if the `fork-join` block was a `begin-end` block and the `Bevent` occurred before the `Aevent`, then the block would be waiting for the next `Bevent`.

Example 3—This example shows two sequential blocks, each of which will execute when its controlling event occurs. Because the event controls are within a `fork-join` block, they execute in parallel and the sequential blocks can

therefore also execute in parallel.

```
fork
  @enable_a
  begin
    #ta wa = 0;
    #ta wa = 1;
    #ta wa = 0;
  end
  @enable_b
  begin
    #tb wb = 1;
    #tb wb = 0;
    #tb wb = 1;
  end
join
```

9.9 Structured procedures

All procedures in the Verilog HDL are specified within one of the following four statements:

- *initial* construct
- *always* construct
- Task
- Function

The initial and always constructs are enabled at the beginning of a simulation. The initial construct shall execute only once and its activity shall cease when the statement has finished. In contrast, the always construct shall execute repeatedly. Its activity shall cease only when the simulation is terminated. There shall be no implied order of execution between initial and always constructs. The initial constructs need not be scheduled and executed before the always constructs. There shall be no limit to the number of initial and always constructs that can be defined in a module.

Tasks and functions are procedures that are enabled from one or more places in other procedures. Tasks and functions are described in Section 10.

9.9.1 Initial construct

The syntax for the *initial construct* is given in Syntax 9-15.

<pre>initial_construct ::= initial statement</pre>

Syntax 9-15—Syntax for initial construct

Examples:

The following example illustrates use of the initial construct for initialization of variables at the start of simulation.

```
initial begin  
    areg = 0; // initialize a register  
    for (index = 0; index < size; index = index + 1)  
        memory[index] = 0; //initialize memory word  
end
```

Another typical usage of the initial construct is specification of waveform descriptions that execute once to provide stimulus to the main part of the circuit being simulated.

```
initial begin  
    inputs = 'b000000; //initialize at time zero  
    #10 inputs = 'b011001; // first pattern  
    #10 inputs = 'b011011; // second pattern  
    #10 inputs = 'b011000; // third pattern  
    #10 inputs = 'b001000; // last pattern  
end
```

9.9.2 Always construct

The *always construct* repeats continuously throughout the duration of the simulation. Syntax 9-16 shows the syntax for the always construct.

<pre>always_construct ::= always statement</pre>

Syntax 9-16—Syntax for always construct

The always construct, because of its looping nature, is only useful when used in conjunction with some form of timing control. If an always construct has no control for simulation time to advance, it will create a simulation deadlock condition. The following code, for example, creates a zero-delay infinite loop.

```
always areg = ~areg;
```

Providing a timing control to the above code creates a potentially useful description as shown in the following:

```
always #half_period areg = ~areg;
```

Section 10

Tasks and functions

Tasks and functions provide the ability to execute common procedures from several different places in a description. They also provide a means of breaking up large procedures into smaller ones to make it easier to read and debug the source descriptions. This section discusses the differences between tasks and functions, describes how to define and invoke tasks and functions, and presents examples of each.

10.1 Distinctions between tasks and functions

The following rules distinguish tasks from functions:

- A function shall execute in one simulation time unit; a task can contain time-controlling statements.
- A function cannot enable a task; a task can enable other tasks and functions.
- A function shall have at least one **input** type argument and shall not have an **output** or **inout** type argument; a task can have zero or more arguments of any type.
- A function shall return a single value; a task shall not return a value.

The purpose of a *function* is to respond to an input value by returning a single value. A *task* can support multiple goals and can calculate multiple result values. However, only the **output** or **inout** type arguments pass result values back from the invocation of a task. A function is used as an operand in an expression; the value of that operand is the value returned by the function.

Example:

Either a task or a function can be defined to switch bytes in a 16-bit word. The task would return the switched word in an output argument, so the source code to enable a task called `switch_bytes` could look like the following example:

```
switch_bytes (old_word, new_word);
```

The task `switch_bytes` would take the bytes in `old_word`, reverse their order, and place the reversed bytes in `new_word`.

A word-switching function would return the switched word as the return value of the function. Thus, the function call for the function `switch_bytes` could look like the following example:

```
new_word = switch_bytes (old_word);
```

10.2 Tasks and task enabling

A task shall be enabled from a statement that defines the argument values to be passed to the task and the registers that receive the results. Control shall be passed back to the enabling process after the task has completed. Thus, if a task has timing controls inside it, then the time of enabling a task can be different from the time at which the control is returned. A task can enable other tasks, which in turn can enable still other tasks—with no limit on the number of tasks enabled. Regardless of how many tasks have been enabled, control shall not return until all enabled tasks have completed.

10.2.1 Defining a task

The syntax for defining tasks is given in Syntax 10-1.

```
task_declaration ::=  
    task task_identifier ;  
    { task_item_declaration }  
    statement_or_null  
    endtask  
task_item_declaration ::=  
    block_item_declaration  
    | input_declaration  
    | output_declaration  
    | inout_declaration  
block_item_declaration ::=  
    parameter_declaration  
    | reg_declaration  
    | integer_declaration  
    | real_declaration  
    | time_declaration  
    | realtime_declaration  
    | event_declaration
```

Syntax 10-1—Syntax for task declaration

The task declaration shall begin with the keyword **task**, followed by a name for the task and a semicolon, and ending with the keyword **endtask**. Task item declarations can specify the following:

- Input arguments
- Output arguments
- Inout arguments
- All data types that can be declared in a procedural block

These declarations have the same syntax as the corresponding declarations in a module definition (see Section 12). The task declaration shall not declare a net data type. The body of the task shall contain zero or more behavioral statements (see Section 9).

10.2.2 Task enabling and argument passing

The task enabling statement shall pass arguments as a comma-separated list of expressions enclosed in parentheses. The formal syntax of the task enabling statement is given in Syntax 10-2.

```
task_enable ::= task_identifier [ ( expression { , expression } ) ] ;
```

Syntax 10-2—Syntax of the task enabling statement

The list of arguments for a task enabling statement shall be optional. If the list of arguments is provided, the list shall be an ordered list of expressions that has to match the order of the list of arguments in the task definition.

If an argument in the task is declared as an **input**, then the corresponding expression can be any expression. The order of evaluation of the expressions in the argument list is undefined. If the argument is declared as an **output** or an **inout**, then the expression shall be restricted to an expression that is valid on the left-hand side of a procedural assignment (see 9.2). The following items satisfy this requirement:

- reg, integer, real, realtime, and time registers
- Memory references
- Concatenations of reg, integer, real, realtime and time registers
- Concatenations of memory references
- Bit-selects and part-selects of reg, integer, and time registers

The execution of the task enabling statement shall pass input values from the registers listed in the enabling statement to the arguments specified within the task. Execution of the return from the task shall pass values from the task **output** and **inout** type arguments to the corresponding registers in the task enabling statement. All arguments to the task shall be passed by *value* rather than by reference (that is, a *pointer* to the value).

Examples:

Example 1—The following example illustrates the basic structure of a task definition with five arguments.

```
task my_task;
input a, b;
inout c;
output d, e;
begin
    . . .      // statements that perform the work of the task
    . . .
    c = foo1; // the assignments that initialize result registers
    d = foo2;
    e = foo3;
end
endtask
```

The following statement enables the task:

```
my_task (v, w, x, y, z);
```

The task enabling arguments (*v*, *w*, *x*, *y*, and *z*) correspond to the arguments (*a*, *b*, *c*, *d*, and *e*) defined by the task. At task enabling time, the **input** and **inout** type arguments (*a*, *b*, and *c*) receive the values passed in *v*, *w*, and *x*. Thus, execution of the task enabling call effectively causes the following assignments:

```
a = v;
b = w;
c = x;
```

As part of the processing of the task, the task definition for *my_task* shall place the computed result values into *c*, *d*, and *e*. When the task completes, the following assignments to return the computed values to the calling process are performed:

```
x = c;
y = d;
z = e;
```

Example 2—The following example illustrates the use of tasks by describing a traffic light sequencer.


```
module traffic_lights;
reg clock, red, amber, green;
parameter on = 1, off = 0, red_tics = 350,
          amber_tics = 30, green_tics = 200;

// initialize colors.
initial red = off;
initial amber = off;
initial green = off;

always begin                                // sequence to control the lights.
    red = on;                               // turn red light on
    light(red, red_tics);                   // and wait.
    green = on;                             // turn green light on
    light(green, green_tics);               // and wait.
    amber = on;                             // turn amber light on
    light(amber, amber_tics);               // and wait.
end

// task to wait for 'tics' positive edge clocks
// before turning 'color' light off.
task light;
output color;
input [31:0] tics;
begin
    repeat (tics) @ (posedge clock);
    color = off;                             // turn light off.
end
endtask

always begin                                // waveform for the clock.
    #100 clock = 0;
    #100 clock = 1;
end
endmodule // traffic_lights.
```

10.2.3 Task memory usage and concurrent activation

A task may be enabled more than once concurrently. A task (or function) may contain the definition of local registers of type **reg**, **integer**, **time**, **real**, **realtime**, or **event**. These registers shall be static in that there shall be a single register corresponding to each declared local register, regardless of the number of concurrent activations of the task. Thus, if a task is enabled more than once concurrently, all instances of the task would share the same local registers.

Because tasks can have nonzero time durations, each active task has a point of control. This point of control is unique to each active task instance. Thus, it is possible to write a recursive task (or function), but all local data shall remain static.

10.3 Functions and function calling

The purpose of a function is to return a value that is to be used in an expression. The rest of this clause explains how to define and use functions.

10.3.1 Defining a function

The syntax for defining a function is given in Syntax 10-3.

```

function_declaration ::=
    function [ range_or_type ] function_identifier ;
    function_item_declaration { function_item_declaration }
    endfunction
range_or_type ::= range | integer | real | realtime | time
function_item_declaration ::=
    input_declaration
    | block_item_declaration
block_item_declaration ::=
    | parameter_declaration
    | reg_declaration
    | integer_declaration
    | real_declaration
    | time_declaration
    | realtime_declaration
    | event_declaration

```

Syntax 10-3—Syntax for function declaration

A function declaration shall begin with the keyword **function**, followed by the range or type of the return value from the function, followed by the name of the function and a semicolon, and shall end with the keyword **endfunction**. The use of range or type shall be optional. A function specified without a range or a type defaults to a one-bit register for the return value. If used, range_or_type shall specify that the return value of the function is a **real**, an **integer**, a **time**, a **realtime**, or a value with a range of [n:m] bits. A function shall have at least one input declared.

Example:

The following example defines a function called `getbyte`, using a range specification.

```

function [7:0] getbyte;
input [15:0] address;
begin
    // code to extract low-order byte from addressed word
    . . .
    getbyte = result_expression;
end
endfunction

```

10.3.2 Returning a value from a function

The function definition shall implicitly declare a register, internal to the function, with the same name as the function. This register either defaults to a 1-bit register or is the same type as the type specified in the function declaration. The function definition initializes the return value from the function by assigning the function result to the internal register with the same name as the function. The following line from the example in 10.3.1 illustrates this concept:

```
getbyte = result_expression;
```

10.3.3 Calling a function

A function call is an operand within an expression. The function call has the syntax given in Syntax 10-4.

$\text{function_call} ::= \text{function_identifier} (\text{expression} \{ , \text{expression} \})$

Syntax 10-4—Syntax for function call

The order of evaluation of the arguments to a function call is undefined.

Example:

The following example creates a word by concatenating the results of two calls to the function `getbyte` (defined in 10.3.1).

```
word = control ? {getbyte(msbyte), getbyte(lsbyte)}:0;
```

10.3.4 Function rules

Functions are more limited than tasks. The following five rules govern their usage:

- a) A function definition shall not contain any time-controlled statements—that is, any statements introduced with `#`, `@`, or **wait**.
- b) Functions shall not enable tasks.
- c) A function definition shall contain at least one input argument.
- d) A function definition shall not have any argument declared as output or inout.
- e) A function definition shall include an assignment of the function result value to the internal register that has the same name as the function name.

Example:

This example defines a function called `factorial` that returns a 32-bit register value. The `factorial` function is called iteratively and the results are printed.

```
module tryfact;

// define the function
function [31:0] factorial;
input [3:0] operand;
reg [3:0] i;
begin
    factorial = 1;
    for (i = 2; i <= operand; i = i + 1)
        factorial = i * factorial;
end
endfunction

// test the function
integer result;
integer n;
initial begin
    for (n = 0; n <= 7; n = n+1) begin
        result = factorial(n);
        $display("%0d factorial=%0d", n, result);
    end
end
endmodule // tryfact
```

The simulation results are as follows:

```
0 factorial=1
1 factorial=1
2 factorial=2
3 factorial=6
4 factorial=24
5 factorial=120
6 factorial=720
7 factorial=5040
```

Section 11

Disabling of named blocks and tasks

The *disable* statement provides the ability to terminate the activity associated with concurrently active procedures, while maintaining the structured nature of Verilog HDL procedural descriptions. The disable statement gives a mechanism for terminating a task before it executes all its statements, breaking from a looping statement, or skipping statements in order to continue with another iteration of a looping statement. It is useful for handling exception conditions such as hardware interrupts and global resets.

The disable statement has the syntax form shown in Syntax 11-1.

<pre>disable_statement ::= disable task_identifier ; disable block_identifier ;</pre>

Syntax 11-1—Syntax of disable statement

Either form of disable statement shall terminate the activity of a task or a named block. Execution shall resume at the statement following the block or following the task enabling statement. All activities enabled within the named block or task shall be terminated as well. If task enable statements are nested—that is, one task enables another, and that one enables yet another—then disabling a task within the chain shall disable all tasks downward on the chain. If a task is enabled more than once, then disabling such a task shall disable all activations of the task.

The results of the following activities that may be initiated by a task are not specified if the task is disabled:

- Results of output and inout arguments
- Scheduled, but not executed, nonblocking assignments
- Procedural continuous assignments (**assign** and **force** statements)

The disable statement can be used within blocks and tasks to disable the particular block or task containing the disable statement. The disable statement cannot be used to disable functions.

Examples:

Example 1—This example illustrates how a block disables itself.

```
begin : block_name  
    rega = regb;  
    disable block_name;  
    regc = rega; // this assignment will never execute  
end
```

Example 2—This example shows the disable statement being used within a named block in a manner similar to a for-

ward *goto*. The next statement executed after the *disable* statement is the one following the named block.

```

begin : block_name
    ...
    ...
    if (a == 0)
        disable block_name;
    ...
end      // end of named block
// continue with code following named block
    ...

```

Example 3—This example shows the *disable* statement being used as an early return from a task. However, a task disabling itself using a *disable* statement is not a short-hand for the *return* statement found in programming languages.

```

task proc_a;
begin
    ...
    ...
    if (a == 0)
        disable proc_a; // return if true
    ...
    ...
end
endtask

```

Example 4—This example shows the *disable* statement being used in an equivalent way to the two statements *continue* and *break* in the C programming language. The example illustrates control code that would allow a named block to execute until a loop counter reaches *n* iterations or until the variable *a* is set to the value of *b*. The named block *break* contains the code that executes until *a* == *b*, at which point the *disable break;* statement terminates execution of that block. The named block *continue* contains the code that executes for each iteration of the *for* loop. Each time this code executes the *disable continue;* statement, the *continue* block terminates and execution passes to the next iteration of the *for* loop. For each iteration of the *continue* block, a set of statements executes if (*a* != 0). Another set of statements executes if (*a* != *b*).

```

begin : break
    for (i = 0; i < n; i = i+1) begin : continue
        @clk
            if (a == 0) // "continue" loop
                disable continue;
            statements
            statements
        @clk
            if (a == b) // "break" from loop
                disable break;
            statements
            statements
    end
end

```

Example 5—This example shows the `disable` statement being used to disable concurrently a sequence of timing controls and the task `action`, when the `reset` event occurs. The example shows a `fork/join` block within which is a named sequential block (`event_expr`) and a `disable` statement that waits for occurrence of the event `reset`. The sequential block and the wait for `reset` execute in parallel. The `event_expr` block waits for one occurrence of event `ev1` and three occurrences of event `trig`. When these four events have happened, plus a delay of `d` time units, the task `action` executes. When the event `reset` occurs, regardless of events within the sequential block, the `fork/join` block terminates—including the task `action`.

```
fork
  begin : event_expr
    @ev1;
    repeat (3) @trig;
    #d action (areg, breg);
  end
  @reset disable event_expr;
join
```

Example 6—The next example is a behavioral description of a retriggerable monostable. The named event `retrig` restarts the monostable time period. If `retrig` continues to occur within 250 time units, then `q` will remain at 1.

```
always begin : monostable
  #250 q = 0;
end

always @retrig begin
  disable monostable;
  q = 1;
end
```

Section 12

Hierarchical structures

The Verilog HDL supports a hierarchical hardware description structure by allowing modules to be embedded within other modules. Higher-level modules create instances of lower-level modules and communicate with them through input, output, and bidirectional ports. These module input/output ports can be scalar or vector.

As an example of a module hierarchy, consider a system consisting of printed circuit boards (PCBs). The system would be represented as the top-level module and would create instances of modules that represent the boards. The board modules would, in turn, create instances of modules that represent ICs, and the ICs could, in turn, create instances of modules such as flip-flops, mux's, and alu's.

To describe a hierarchy of modules, the user provides textual definitions of the various modules. Each module definition stands alone; the definitions are not nested. Statements within the module definitions create instances of other modules, thus describing the hierarchy.

12.1 Modules

This clause gives the formal syntax for a module definition and then gives the syntax for module instantiation, along with an example of a module definition and a module instantiation.

A module definition shall be enclosed between the keywords **module** and **endmodule**. The identifier following the keyword **module** shall be the name of the module being defined. The optional list of ports shall specify an ordered list of the ports of the module. The order used can be significant when instantiating the module (see 12.1.2). The identifiers in this list shall be declared in input, output, and inout statements within the module definition. The module items define what constitutes a module, and they include many different types of declarations and definitions; many of them have already been introduced.

The keyword **macromodule** can be used interchangeably with the keyword **module** to define a module. An implementation can choose to treat module definitions beginning with **macromodule** keyword differently.


```

module_declaration ::=
    module_keyword module_identifier [ list_of_ports ] ; { module_item }
endmodule
module_keyword ::= module | macromodule
list_of_ports ::= ( port { , port } )
port ::= [ port_expression ]
    | . port_identifier ( [ port_expression ] )
port_expression ::=
    port_reference
    | { port_reference { , port_reference } }
port_reference ::=
    port_identifier
    | port_identifier [ constant_expression ]
    | port_identifier [ msb_constant_expression : lsb_constant_expression ]
module_item ::=
    module_item_declaration
    | parameter_override
    | continuous_assign
    | gate_instantiation
    | udp_instantiation
    | module_instantiation
    | specify_block
    | initial_construct
    | always_construct
module_item_declaration ::=
    parameter_declaration
    | input_declaration
    | output_declaration
    | inout_declaration
    | net_declaration
    | reg_declaration
    | integer_declaration
    | real_declaration
    | time_declaration
    | realtime_declaration
    | event_declaration
    | task_declaration
    | function_declaration

```

Syntax 12-1—Syntax for module

See 12.3 for the definitions of ports.

12.1.1 Top-level modules

Top-level modules are modules that are included in the source text but are not instantiated, as described in 12.1.2.

12.1.2 Module instantiation

Instantiation allows one module to incorporate a copy of another module into itself. Module definitions do not nest. That is, one module definition shall not contain the text of another module definition within its **module-endmodule** keyword pair. A module definition nests another module by *instantiating* it. The *module instantiation statement* cre-

ates one or more named *instances* of a defined module.

For example, a counter module might instantiate a D flip-flop module to create multiple instances of the flip-flop.

Syntax 12-2 gives the syntax for specifying instantiations of modules.

```

module_instantiation ::=
    module_identifier [ parameter_value_assignment ] module_instance { ,
    module_instance } ;
parameter_value_assignment ::= # ( expression { , expression } )
module_instance ::= name_of_instance ( [ list_of_module_connections ] )
name_of_instance ::= module_instance_identifier [ range ]
list_of_module_connections ::=
    ordered_port_connection { , ordered_port_connection }
    | named_port_connection { , named_port_connection }
ordered_port_connection ::= [ expression ]
named_port_connection ::= . port_identifier ( [ expression ] )

```

Syntax 12-2—Syntax for module instantiation

The instantiations of modules can contain a range specification. This allows an array of instances to be created. The array of instances are described in 7.1. The syntax and semantics of arrays of instances defined for gates and primitives apply for modules as well.

One or more module instances (identical copies of a module) can be specified in a single module instantiation statement.

The list of module connections shall be provided only for modules defined with ports. The parentheses, however, are always required. When a list of module connections is given, the first element in the list shall connect to the first port, the second to the second port, and so on. See 12.3 for a more detailed discussion of ports and port connection rules.

A connection can be a simple reference to a register or a net identifier, an expression, or a blank. An expression can be used for supplying a value to a module input port. A blank module connection shall represent the situation where the port is not to be connected.

When connecting ports by name, an unconnected port can be indicated either by omitting it in the port list, or by providing no expression in the parentheses [i.e., `.port_name ()`].

Examples:

Example 1—The following example illustrates a circuit (the lower-level module) being driven by a simple waveform description (the higher-level module) where the circuit module is instantiated inside the waveform module.

```
// Lower level module:
// module description of a nand flip-flop circuit
module ffnand (q, qbar, preset, clear);
output q, qbar;           //declares 2 circuit output nets
input preset, clear;      //declares 2 circuit input nets

// declaration of two nand gates and their interconnections
nand g1 (q, qbar, preset),
      g2 (qbar, q, clear);
endmodule

// Higher-level module:
// a waveform description for the nand flip-flop
module ffnand_wave;
wire out1, out2;          //outputs from the circuit
reg in1, in2;             //variables to drive the circuit
parameter d = 10;

// instantiate the circuit ffnand, name it "ff",
// and specify the IO port interconnections
ffnand ff(out1, out2, in1, in2);

// define the waveform to stimulate the circuit
initial begin
    #d in1 = 0; in2 = 1;
    #d in1 = 1;
    #d in2 = 0;
    #d in2 = 1;
end
endmodule
```

Example 2—The following example creates two instances of the flip-flop module `ffnand` defined in example 1. It connects only to the `q` output in one instance and only to the `qbar` output in the other instance.

```

// a waveform description for testing
// the nand flip-flop, without the output ports
module ffnand_wave;
reg in1, in2; //variables to drive the circuit
parameter d = 10;
// make two copies of the circuit ffnand
// ff1 has qbar unconnected, ff2 has q unconnected
ffnand ff1(out1, , in1, in2),
        ff2(.qbar(out2), .clear(in2), .preset(in1), .q());
    // ff3(.q(out3),.clear(in1),,,); is illegal

// define the waveform to stimulate the circuit
initial begin
    #d in1 = 0; in2 = 1;
    #d in1 = 1;
    #d in2 = 0;
    #d in2 = 1;
end
endmodule

```

12.2 Overriding module parameter values

When one module instantiates another module, it can alter the values of any parameters declared within the instantiated module. There are two ways to alter parameter values: the *defparam statement*, which allows assignment to parameters using their hierarchical names, and the *module instance parameter value assignment*, which allows values to be assigned inline during module instantiation. The next two subclauses describe these two methods.

12.2.1 Defparam statement

Using the *defparam statement*, parameter values can be changed in any module instance throughout the design using the hierarchical name of the parameter. See 12.4 for hierarchical names.

The expression on the right-hand side of the defparam assignments shall be a constant expression involving only numbers and references to parameters. The referenced parameters (on the right-hand side of the **defparam**) shall be declared in the same module as the defparam statement.

The defparam statement is particularly useful for grouping all of the parameter value override assignments together in one module.

Example:

```
module top;
  reg clk;
  reg [0:4] in1;
  reg [0:9] in2;
  wire [0:4] o1;
  wire [0:9] o2;

  vdff m1 (o1, in1, clk);
  vdff m2 (o2, in2, clk);
endmodule

module vdff (out, in, clk);
  parameter size = 1, delay = 1;
  input [0:size-1] in;
  input clk;
  output [0:size-1] out;
  reg [0:size-1] out;

  always @(posedge clk)
    # delay out = in;
endmodule

module annotate;
  defparam
    top.m1.size = 5,
    top.m1.delay = 10,
    top.m2.size = 10,
    top.m2.delay = 20;
endmodule
```

The module `annotate` has the `defparam` statement which overrides `size` and `delay` parameter values for instances `m1` and `m2` in the top-level module `top`. The modules `top` and `annotate` would both be considered top-level modules.

12.2.2 Module instance parameter value assignment

An alternative method for assigning values to parameters within module instances is similar in appearance to the assignment of delay values to gate instances. It supplies values for particular instances of a module to any parameters that have been specified in the definition of that module.

The order of the assignments in the module instance parameter value assignment shall follow the order of declaration of the parameters within the module. It is not necessary to assign values to all of the parameters within a module when using this method. However, it is not possible to skip over a parameter. Therefore, to assign values to a subset of the parameters declared within a module, the declarations of the parameters that make up this subset shall precede the declarations of the remaining parameters. An alternative is to assign values to all of the parameters, but to use the default value (the same value assigned in the declaration of the parameter within the module definition) for those parameters that do not need new values.

Example:

Consider the following example, where the parameters within module instance `mod_a` are changed during instantiation.

```

module m;
reg clk;
wire[1:10] out_a, in_a;
wire[1:5] out_b, in_b;

    // create an instance and set parameters
    vdff #(10,15) mod_a(out_a, in_a, clk);
    // create an instance leaving default values
    vdff mod_b(out_b, in_b, clk);
endmodule

module vdff (out, in, clk);
parameter size = 1, delay = 1;
input [0:size-1] in;
input clk;
output [0:size-1] out;
reg [0:size-1] out;

    always @(posedge clk)
        # delay out = in;
endmodule

```

In this example, the name of the module being instantiated is `vdff`. The construct `#(10,15)` assigns values to parameters used in the `mod_a` instance of `vdff`. The parameter `size` is assigned the value 10 and the parameter `delay` is assigned the value 15 for the instance of module `vdff` called `mod_a`.

12.2.3 Parameter dependence

A parameter (for example, `memory_size`) can be defined with an expression containing another parameter (for example, `word_size`). Since `memory_size` depends on the value of `word_size`, a modification of `word_size` changes the value of `memory_size`. For example, in the following parameter declaration, an update of `word_size`, whether by `defparam` statement or in an instantiation statement for the module that defined these parameters, automatically updates `memory_size`.

```

parameter
    word_size = 32,
    memory_size = word_size * 4096;

```

12.3 Ports

Ports provide a means of interconnecting a hardware description consisting of modules, primitives, and macromodules. For example, module A can instantiate module B, using port connections appropriate to module A. These port names can differ from the names of the internal nets and registers specified in the definition of module B.

12.3.1 Port definition

The syntax for a port is given in Syntax 12-3 (this is the completion of the syntax presented in 12.1).

```
port ::=  
    [ port_expression ]  
    | .port_identifier ( [ port_expression ] )  
port_expression ::=  
    port_reference  
    | { port_reference { , port_reference } }  
port_reference ::=  
    port_identifier  
    | port_identifier [ constant_expression ]  
    | port_identifier [ msb_constant_expression : lsb_constant_expression ]
```

Syntax 12-3—Syntax for port

The port expression in the port definition can be one of the following:

- A simple identifier
- A bit-select of a vector declared within the module
- A part-select of a vector declared within the module
- A concatenation of any of the above

The two types of module port definitions shall not be mixed; the ports of a particular module definition shall all be defined by order or all by name. The port expression is optional because ports can be defined that do not connect to anything internal to the module.

12.3.2 Port declarations

Each port listed in the list of ports for the module definition shall be declared in the body of the module as an **input**, **output**, or **inout** (bidirectional). This is in addition to any other declaration for a particular port— for example, a **reg** or **wire**. The syntax for port declarations is given in Syntax 12-4.

```
input_declaration ::= input [ range ] list_of_port_identifiers ;  
output_declaration ::= output [ range ] list_of_port_identifiers ;  
inout_declaration ::= inout [ range ] list_of_port_identifiers ;  
list_of_port_identifiers ::= port_identifier { , port_identifier }
```

Syntax 12-4—Syntax for port declarations

A port can be declared in both a port declaration and a net or register declaration. If a port is declared as a vector, the range specification between the two declarations of a port shall be identical.

NOTE—Implementations may limit maximum number of ports in a module definition, but they will at least be 256.

12.3.3 Connecting module ports by ordered list

One method of making the connection between the ports listed in a module instantiation and the ports defined by the instantiated module is the ordered list—that is, the ports listed for the module instance shall be in the same order as the ports listed in the module definition.

Example:

The following example illustrates a top-level module (**topmod**) that instantiates a second module (**modB**). Module **modB** has ports that are connected by an ordered list. The connections made are as follows:

- Port **wa** in the **modB** definition connects to the bit-select **v[0]** in the **topmod** module.
- Port **wb** connects to **v[3]**.
- Port **c** connects to **w**.
- Port **d** connects to **v[4]**.

In the **modB** definition, ports **wa** and **wb** are declared as **inouts** while ports **c** and **d** are declared as **input**.

```

module topmod;
wire [4:0] v;
wire a,b,c,w;

modB b1 (v[0], v[3], w, v[4]);
endmodule

module modB (wa, wb, c, d);
inout wa, wb;
input c, d;

tranif1      g1 (wa, wb, cinvert);
not #(2, 6)  n1 (cinvert, int);
and #(6, 5)  g2 (int, c, d);
endmodule

```

During simulation of the **b1** instance of **modb**, the **and** gate **g2** activates first to produce a value on **int**. This value triggers the **not** gate **n1** to produce output on **cinvert**, which then activates the **tranif1** gate **g1**.

12.3.4 Connecting module ports by name

The second way to connect module ports consists of explicitly linking the two names for each side of the connection—the name used in the module definition, followed by the name used in the instantiating module. This compound name is then placed in the list of module connections. The name of port shall be the name specified in the module definition. The name of port cannot be a bit-select, a part-select, or a concatenation of ports.

The port expression shall be the name used by the instantiating module and can be one of the following:

- A simple identifier
- A bit-select of a vector declared within the module
- A part-select of a vector declared within the module
- A concatenation of any of the above

The port expression is optional so that the instantiating module can document the existence of the port without connecting it to anything. The parentheses are required.

The two types of module port connections shall not be mixed; connections to the ports of a particular module instance shall be all by order or all by name.

Examples:

Example 1—In the following example, the instantiating module connects its signals `topA` and `topB` to the ports `In1` and `Out` defined by the module `ALPHA`. At least one port provided by `ALPHA` is unused; it is named `In2`. There could be other unused ports not mentioned in the instantiation.

```
ALPHA instance1 (.Out(topB),.In1(topA),.In2());
```

Example 2—This example defines the modules `modB` and `topmod`, and then `topmod` instantiates `modB` using ports connected by name.

```
module topmod;
wire [4:0] v;
wire a,b,c,w;

modB b1 (.wb(v[3]),.wa(v[0]),.d(v[4]),.c(w));
endmodule

module modB(wa, wb, c, d);
inout wa, wb;
input c, d;

tranif1      g1(wa, wb, cinvert);
not #(6, 2)  n1(cinvert, int);
and #(5, 6)  g2(int, c, d);
endmodule
```

Since these connections are made by name, the order in which they appear is irrelevant.

12.3.5 Real numbers in port connections

The `real` data type shall not be directly connected to a port. It shall be connected indirectly, as shown in the following example. The system functions `$realtobits` and `$bitstoreal` shall be used for passing the bit patterns across module ports. (See 14.9 for a description of these system tasks.)

Example:

```
module driver (net_r);
output net_r;
real r;
wire [64:1] net_r = $realtobits(r);
endmodule

module receiver (net_r);
input net_r;
wire [64:1] net_r;
real r;

initial assign r = $bitstoreal(net_r);

endmodule
```

12.3.6 Connecting dissimilar ports

A port of a module can be viewed as providing a link or connection between two items (nets, registers, expressions, etc.)—one internal to the module instance and one external to the module instance.

Examination of the port connection rules described in 12.3.7 will show that the item receiving the value through the port (the internal item for inputs, the external item for outputs) shall be a structural net expression. The item that provides the value can be any expression.

NOTE—A port that is declared as input (output) but used as an output (input) or inout may be coerced to inout. If not coerced to inout, a warning has to be issued.

12.3.7 Port connection rules

The following rules shall govern the way module ports are declared and the way they are interconnected.

12.3.7.1 Rule 1

An input or inout port shall be of type net.

12.3.7.2 Rule 2

Each port connection shall be a continuous assignment of source to sink, where one connected item shall be a signal source and the other shall be a signal sink. The assignment shall be a continuous assignment from source to sink for input or output ports. The assignment is a nonstrength reducing transistor connection for inout ports. Only nets or structural net expressions shall be the sinks in an assignment.

A *structural net expression* is a port expression whose operands can be the following:

- A scalar net
- A vector net
- A constant bit-select of a vector net
- A part-select of a vector net
- A concatenation of structural net expressions

The following external items shall not be connected to the output or inout ports of modules:

- Registers
- Expressions other than
 - A scalar net
 - A vector net
 - A constant bit-select of a vector net
 - A part-select of a vector net
 - A concatenation of the expressions listed above

12.3.8 Net types resulting from dissimilar port connections

When different net types are connected through a module port, the nets on both sides of the port can take on the same type. The resulting net type can be determined as shown in Table 12-1. In the table, *external net* means the net specified in the module instantiation, and *internal net* means the net specified in the module definition. The net whose type is used is said to be the *dominating net*. The net whose type is changed is said to be the *dominated net*. It is permissible to merge the dominating and dominated nets into a single net, whose type shall be that of the dominating net. The resulting net is called the *simulated net*, and the dominated net is called a *collapsed net*.

The simulated net shall take the delay specified for the dominating net. If the dominating net is of the type **trireg**, any strength value specified for the trireg net shall apply to the simulated net.

12.3.8.1 Net type resolution rule

When the two nets connected by a port are of different net type, the resulting single net can be assigned one of the following:

- The dominating net type if one of the two nets is dominating, *or*
- The net type external to the module

When a dominating net type does not exist, the external net type shall be used.

12.3.8.2 Net type table

Table 12-1 shows the net type dictated by net type resolution rule.

The simulated net shall take the net type specified in the table and the delay specified for that net. If the simulated net selected is a **triereg**, any strength value specified for the triereg net applies to the simulated net.

Table 12-1 — Net types resulting from dissimilar port connections

Internal net	External net							
	wire, tri	wand, triand	wor, trior	triereg	tri0	tri1	supply0	supply1
wire, tri	ext	ext	ext	ext	ext	ext	ext	ext
wand, triand	int	ext	warn	warn	warn	warn	ext	ext
wor, trior	int	int	ext	warn	warn	warn	ext	ext
triereg	int	int	warn	ext	ext	ext	ext	ext
tri0	int	int	warn	int	ext	warn	ext	ext
tri1	int	int	warn	int	warn	ext	ext	ext
supply0	int	int	int	int	int	int	ext	warn
supply1	int	int	int	int	int	int	warn	ext

KEY

ext = The external net type is used

int = The internal net type is used

warn = A warning is issued and the external net type is used

12.4 Hierarchical names

Every identifier in a Verilog HDL description shall have a unique *hierarchical path name*. The hierarchy of modules and the definition of items such as tasks and named blocks within the modules shall define these names. The hierarchy of names can be viewed as a tree structure, where each module instance, task, function, or named **begin-end** or **fork-join** block defines a new hierarchical level, or scope, in a particular branch of the tree.

At the top of the name hierarchy are the names of modules of which no instances have been created. It is the *root* of the hierarchy. Inside any module, each module instance, task definition, function definition, and named **begin**–**end** or **fork**–**join** block shall define a new branch of the hierarchy. Named blocks within named blocks and within tasks and functions shall create new branches.

Each node in the hierarchical name tree shall be a separate scope with respect to identifiers. A particular identifier can be declared at most once in any scope. See 12.5 for a discussion of scope rules and 3.11 for a discussion of name spaces.

Any named Verilog object can be referenced uniquely in its full form by concatenating the names of the modules, tasks, functions, or blocks that contain it. The period character shall be used to separate each of the names in the hierarchy. The complete path name to any object shall start at a top-level module. This path name can be used from any level in the description. The first node name in a path name can also be the top of a hierarchy that starts at the level where the path is being used.

Examples:

Example 1—The code in this example defines a hierarchy of module instances and named blocks. Figure 12-1 illustrates the hierarchy implicit in this Verilog code. Figure 12-2 is a list of the hierarchical forms of the names of all the objects defined in the code.

```

module mod (in);
input in;

always @(posedge in) begin : keep
  reg hold;
  hold = in;
end
endmodule

module wave;
reg stim1, stim2;

  cct a(stim1, stim2); // instantiate cct

  initial begin :wave1
    #100 fork :innerwave
      reg hold;
      join
    #150 begin
      stim1 = 0;
    end
  end
endmodule

module cct (stim1, stim2);
input stim1, stim2;

  // instantiate mod
  mod amod(stim1), bmod(stim2);
endmodule

```

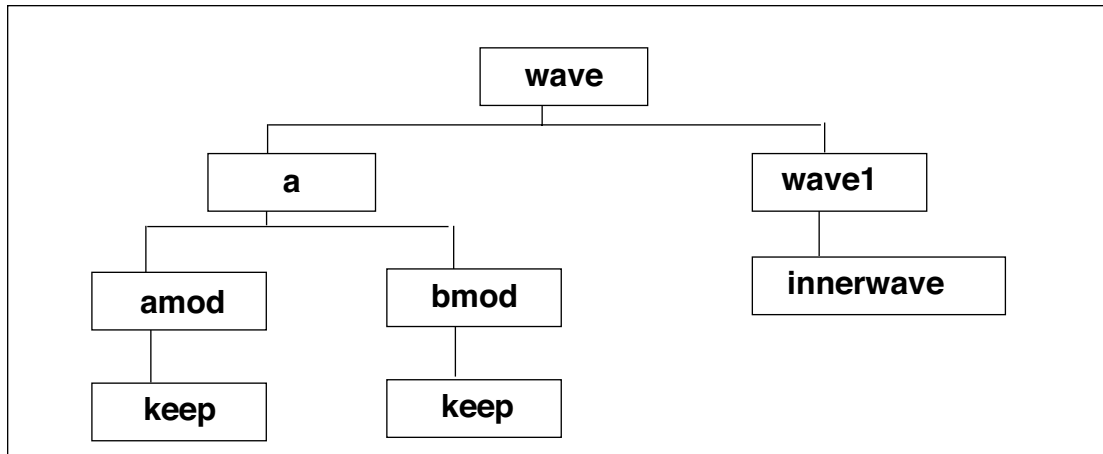


Figure 12-1—Hierarchy in a model

```

wave          wave.a.bmod
wave.stim1    wave.a.bmod.in
wave.stim2    wave.a.bmod.keep
wave.a        wave.a.bmod.keep.hold
wave.a.stim1  wave.wave1
wave.a.stim2  wave.wave1.innerwave
wave.a.amod   wave.wave1.innerwave.hold
wave.a.amod.in
wave.a.amod.keep
wave.a.amod.keep.hold
  
```

Figure 12-2—Hierarchical path names in a model

Hierarchical name referencing allows free data access to any object from any level in the hierarchy. If the unique hierarchical path name of an item is known, its value can be sampled or changed from anywhere within the description.

Example 2—The next example shows how a pair of named blocks can refer to items declared within each other.

```

begin
  fork :mod_1
    reg x;
    mod_2.x = 1;
  join

  fork :mod_2
    reg x;
    mod_1.x = 0;
  join
end
  
```

12.4.1 Upwards name referencing

A lower-level module can reference items in a module above it in the hierarchy if the name of the higher-level module is known. The syntax for an upward reference is given in Syntax 12-5.

```
upward_name_reference ::=  
    module_identifier.item_name  
item_name ::=  
    port_identifier  
    | reg_identifier  
    | net_identifier  
    | parameter_identifier  
    | function_identifier  
    | task_identifier  
    | named_block_identifier
```

Syntax 12-5—Syntax for upward name referencing

Upwards name references can also be done with names of the form

```
module_instance_name.item_name
```

A name of this form shall be resolved as follows:

- a) Look in the current module for a module instance named `module_instance_name`. If found, this name reference shall be treated as a downward reference, and the item name shall be resolved in the corresponding module.
- b) Look in the parent module for a module instance named `module_instance_name`. If found, the item name shall be resolved from that instance, which is the sibling of the module containing the reference.
- c) Repeat step b), going up the hierarchy.

There shall be no spaces within the hierarchical name reference.

Example:

In this example, there are four modules, **a**, **b**, **c**, and **d**. Each module contains an integer **i**. The highest-level modules in this segment of a model hierarchy are **a** and **d**. There are two copies of module **b** because module **a** and **d** instantiate **b**. There are four copies of **c.i** because each of the two copies of **b** instantiates **c** twice.

```
module a;
integer i;
b a_b1();
endmodule

module b;
integer i;
c b_c1(), b_c2();
initial // downward path references two copies of i:
    #10 b_c1.i = 2; // a.a_b1.b_c1.i, d.d_b1.b_c1.i
endmodule

module c;
integer i;
initial begin // local name references four copies of i:
    i = 1; // a.a_b1.b_c1.i, a.a_b1.b_c2.i,
           // d.d_b1.b_c1.i, d.d_b1.b_c2.i
    b.i = 1; // upward path references two copies of i:
             // a.a_b1.i, d.d_b1.i
end
endmodule

module d;
integer i;
b d_b1();
initial begin // full path name references each copy of i
    a.i = 1; d.i = 5;
    a.a_b1.i = 2; d.d_b1.i = 6;
    a.a_b1.b_c1.i = 3; d.d_b1.b_c1.i = 7;
    a.a_b1.b_c2.i = 4; d.d_b1.b_c2.i = 8;
end
endmodule
```

12.5 Scope rules

The following four elements define a new scope in Verilog:

- Modules
- Tasks
- Functions
- Named blocks

An identifier shall be used to declare only one item within a scope. This rule means it is illegal to declare two or more variables that have the same name, or to name a task the same as a variable within the same module, or to give a gate instance the same name as the name of the net connected to its output.

If an identifier is referenced directly (without a hierarchical path) within a task, function, or named block, it shall be declared either locally within the task, function, or named block, or within a module, task or named block that is higher in the same branch of the name tree that contains the task, function, or named block. If it is declared locally, then the local item shall be used; if not, the search shall continue upward until an item by that name is found or until a module boundary is encountered. The search shall cross named block, task, and function boundaries but not module

boundaries. This fact means that tasks and functions can use and modify the variables within the containing module by name, without going through their ports.

Because of the upward searching, path names that are not strictly on a downward path can be used.

Example:

Example 1—In Figure 12-3, each rectangle represents a local scope. The scope available to upward searching extends outward to all containing rectangles—with the boundary of the module A as the outer limit. Thus block G can directly reference identifiers in F, E, and A; it cannot directly reference identifiers in H, B, C, and D.

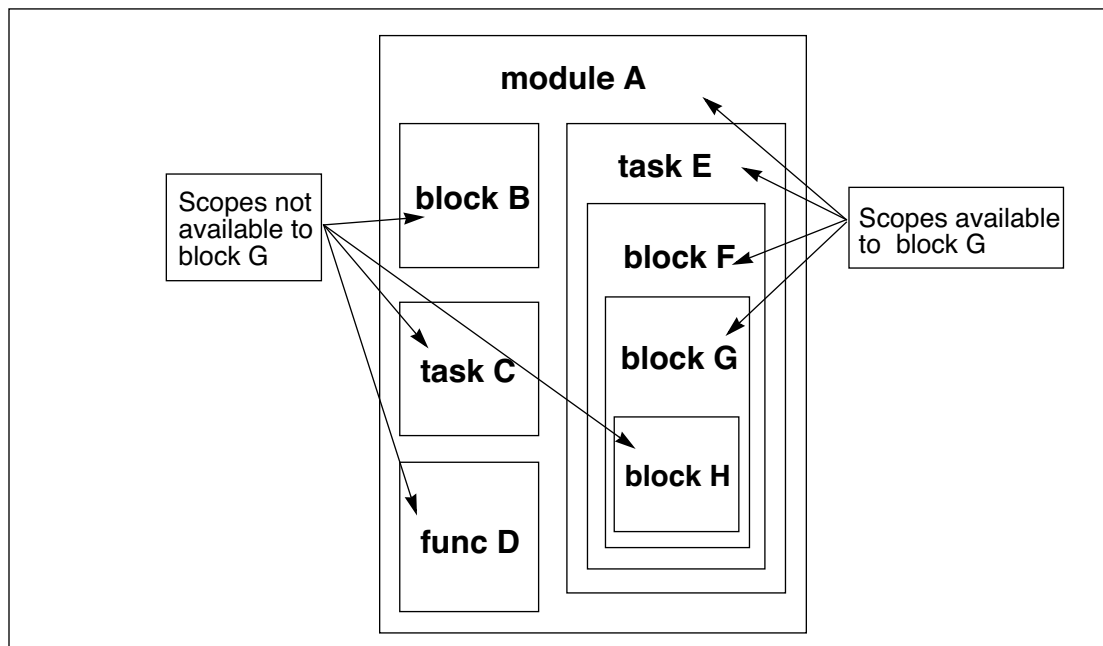


Figure 12-3—Scopes available to upward name referencing

Example 2—The following example shows an incompletely defined downward reference that can be accessed.

```
task t;
reg r, s;
begin : b
    // redundant assignments to reg r
    t.b.r = 0; // poorly defined but found by upward search
    t.s = 0;   // fully defined downward reference
end
endtask
```


Section 13

Specify blocks

Two types of HDL constructs are often used to describe delays for structural models such as ASIC cells. They are

- *Distributed delays*, which specify the time it takes events to propagate through gates and nets inside the module (see 7.15)
- *Module path delays*, which describe the time it takes an event at a source (input port or inout port) to propagate to a destination (output port or inout port)

This section describes how paths are specified in a module and how delays are assigned to these paths.

13.1 Specify block declaration

A block statement called the *specify block* is the vehicle for describing paths between a source and a destination and for assigning delays to these paths. The syntax for specify block is shown in Syntax 13-1.

```
specify_block ::= specify [ specify_item ] endspecify  
specify_item ::=  
    specparam_declaration  
    | path_declaration  
    | system_timing_check
```

Syntax 13-1—Syntax of specify block

The specify block shall be bounded by the keywords **specify** and **endspecify**, and it shall appear inside a module declaration. The specify block can be used to perform the following tasks:

- Describe various paths across the module.
- Assign delays to those paths.
- Perform timing checks to ensure that events occurring at the module inputs satisfy the timing constraints of the device described by the module. See 14.5.

The paths described in the specify block, called *module paths*, pair a signal source with a signal destination. The source may be unidirectional (an input port) or bidirectional (an inout port) and is referred to as the *module path source*. Similarly, the destination may be unidirectional (an output port) or bidirectional (an inout port) and is referred to as the *module path destination*.

Example:

```

specify
  specparam tRise_clk_q = 150, tFall_clk_q = 200;
  specparam tSetup = 70;

  (clk => q) = (tRise_clk_q, tFall_clk_q);

  $setup(d, posedge clk, tSetup);
endspecify

```

The first two lines following the keyword **specify** declare specify parameters, which are discussed in 13.2. The line following the declarations of specify parameters describes a module path and assigns delays to that module path. The specify parameters determine the delay assigned to the module path. Specifying module paths is presented in 13.3. Assigning delays to module paths is discussed in 13.4. The line preceding the keyword **endspecify** instantiates one of the system timing checks, which are discussed further in 14.5.

13.2 Declaring parameters in specify blocks

The keyword **specparam** declares parameters within specify blocks—called *specify parameters* or *specparams*, to distinguish them from *module parameters*. The syntax for declaring specify parameters is shown in Syntax 13-2.

```

specparam_declaration ::= specparam list_of_specparam_assignments ;
list_of_specparam_assignments ::= specparam_assignment { , specparam_assignment }
specparam_assignment ::=
    specparam_identifier = constant_expression
  | pulse_control_specparam

```

Syntax 13-2—Syntax of the specparam declaration

A specify parameter shall be declared and used only within the specify block. The value assigned to a specify parameter can be any constant expression. A specify parameter declared in the specify block can be used to construct a constant expression for a subsequent specify parameter declaration. A specify parameter shall not be visible outside the specify block in which it is declared.

The specify parameters and module parameters shall not be interchangeable. Table 13-1 summarizes the differences between the two types of parameter declarations.

Table 13-1—Differences between specparams and parameters

Specparams (Specify parameter)	Parameters (Module parameter)
Use keyword specparam	Use keyword parameter
Shall be declared <i>inside</i> specify blocks	Shall be declared <i>outside</i> specify blocks
May only be used inside specify blocks	May not be used inside specify blocks
Cannot use defparam to override values	Use defparam to override values

Example:

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

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

13.3 Module path declarations

There are two steps required to set up module path delays in a specify block:

- a) Describe the module paths
- b) Assign delays to those paths (see 13.4)

The syntax of the module path declaration is described in Syntax 13-3.

```
path_declaration ::=  
    simple_path_declaration ;  
    | edge_sensitive_path_declaration ;  
    | state-dependent_path_declaration ;
```

Syntax 13-3—Syntax of the module path declaration

A module path may be described as a *simple path*, an *edge sensitive path*, or a *state dependent path*. A module path shall be defined inside a specify block as a connection between a source signal and a destination signal. Module paths can connect any combination of vectors and scalars.

Example:

Figure 13-1 illustrates a circuit with module path delays. More than one source (A, B, C, and D) may have a module path to the same destination (Q), and different delays may be specified for each input to output path.

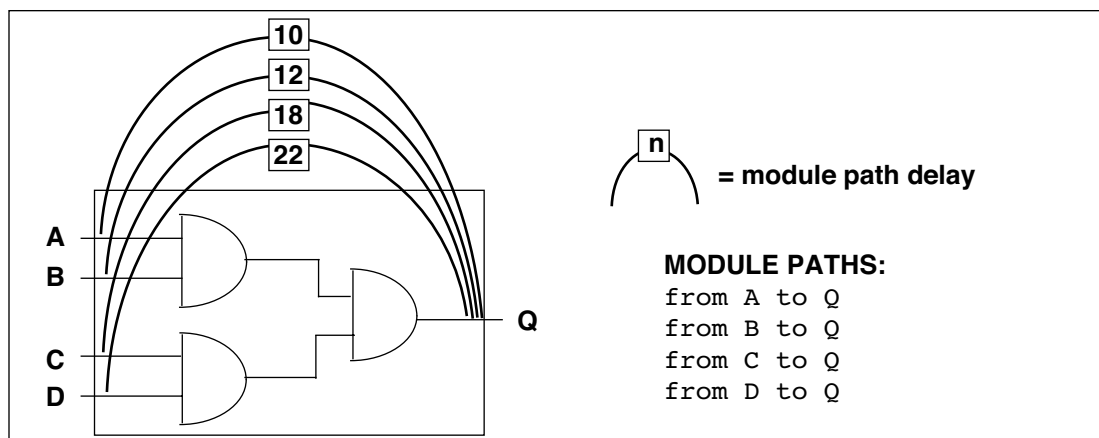


Figure 13-1—Module path delays

13.3.1 Module path restrictions

Module paths have the following restrictions:

- The module path source shall be a net that is connected to a module input port or inout port.
- The module path destination shall be a net or register that is connected to a module output port or inout port.
- The module path destination shall have only one driver inside the module.

13.3.2 Simple module paths

The syntax for specifying a simple module path is given in Syntax 13-4.

```

simple_path_declaration ::=
    parallel_path_description = path_delay_value
  | full_path_description = path_delay_value
parallel_path_description ::=
    ( specify_input_terminal_descriptor [ polarity_operator ] =>
      specify_output_terminal_descriptor )
full_path_description ::=
    ( list_of_path_inputs [ polarity_operator ] *> list_of_path_outputs )
list_of_path_inputs ::= specify_input_terminal_descriptor { ,
    specify_input_terminal_descriptor }
list_of_path_outputs ::= specify_output_terminal_descriptor { ,
    specify_output_terminal_descriptor }
specify_input_terminal_descriptor ::=
    input_identifier
  | input_identifier [ constant_expression ]
  | input_identifier [ msb_constant_expression : lsb_constant_expression ]
specify_output_terminal_descriptor ::=
    output_identifier
  | output_identifier [ constant_expression ]
  | output_identifier [ msb_constant_expression : lsb_constant_expression ]
input_identifier ::= input_port_identifier | inout_port_identifier

```

Syntax 13-4—Syntax for simple module path

Simple path can be declared in one of the two forms:

- source *> destination
- source => destination

The symbols *> and => each represent a different kind of connection between the module path source and the module path destination. The operator *> establishes a *full connection* between source and destination. The operator => establishes a *parallel connection* between source and destination. Refer to 13.3.5 for a description of full connection and parallel connection paths.

Example:

The following three examples illustrate valid simple module path declarations.

```
(A => Q) = 10;
(B => Q) = (12);
(C, D *> Q) = 18;
```

13.3.3 Edge-sensitive paths

When a module path is described using an edge transition at the source, it is called an *edge-sensitive path*. The edge-sensitive path construct is used to model the timing of input to output delays, which only occur when a specified edge occurs at the source signal.

The syntax of the edge-sensitive path declaration is shown in Syntax 13-5.

```
edge_sensitive_path_declaration ::=
    parallel_edge_sensitive_path_description = path_delay_value
  | full_edge_sensitive_path_description = path_delay_value
parallel_edge_sensitive_path_description ::=
    ( [ edge_identifier ] specify_input_terminal_descriptor =>
      specify_output_terminal_descriptor [ polarity_operator ] :
      data_source_expression ) )
full_edge_sensitive_path_description ::=
    ( [ edge_identifier ] list_of_path_inputs *>
      list_of_path_outputs [ polarity_operator ] : data_source_expression ) )
data_source_expression ::= expression
```

Syntax 13-5—Syntax of the edge-sensitive path declaration

The edge identifier may be one of the keywords **posedge** or **negedge**, associated with an input terminal descriptor, which may be any input port or inout port. If a vector port is specified as the input terminal descriptor, the edge transition shall be detected on the least significant bit. If the edge transition is not specified, the path shall be considered active on any transition at the input terminal.

An edge-sensitive path may be specified with full connections (*>) or parallel connections (=>). For parallel connections (=>), the destination shall be any scalar output or inout port or one of its bit-selects. For full connections (*>), the destination shall be a list of one or more of the vector or scalar output and inout ports, and bit-selects or part-selects of those ports. Refer to 13.3.5 for a description of parallel paths and full connection paths.

The data source expression is an arbitrary expression, which serves as a description of the flow of data to the path destination. This arbitrary data path description does not affect the actual propagation of data or events through the

model; how an event at the data path source propagates to the destination depends on the internal logic of the module. The polarity operator describes whether the data path is inverting or noninverting.

Examples:

Example 1—The following example demonstrates an edge-sensitive path declaration with a positive polarity operator:

```
( posedge clock => ( out +: in ) ) = (10, 8);
```

In this example, at the positive edge of `clock`, a module path extends from `clock` to `out` using a rise delay of 10 and a fall delay of 8. The data path is from `in` to `out`, and `in` is not inverted as it propagates to `out`.

Example 2—The following example demonstrates an edge-sensitive path declaration with a negative polarity operator:

```
( negedge clock[0] => ( out -: in ) ) = (10, 8);
```

In this example, at the negative edge of `clock[0]`, a module path extends from `clock[0]` to `out` using a rise delay of 10 and a fall delay of 8. The data path is from `in` to `out`, and `in` is inverted as it propagates to `out`.

Example 3—The following example demonstrates an edge-sensitive path declaration with no edge identifier:

```
( clock => ( out : in ) ) = (10, 8);
```

In this example, at any change in `clock`, a module path extends from `clock` to `out`.

13.3.4 State-dependent paths

A *state-dependent path* makes it possible to assign a delay to a module path that affects signal propagation delay through the path only if specified conditions are true.

A state-dependent path description includes the following items:

- A conditional expression that, when evaluated true, enables the module path
- A module path description
- A delay expression that applies to the module path

The syntax for the state-dependent path declaration is shown in Syntax 13-6.

```
state_dependent_path_declaration ::=
    if ( conditional_expression ) simple_path_declaration
  | if ( conditional_expression ) edge_sensitive_path_declaration
  | ifnone simple_path_declaration
```

Syntax 13-6—Syntax of state-dependent paths

13.3.4.1 Conditional expression

The operands in the conditional expression shall be constructed from the following:

- Scalar or vector module input ports or inout ports or their bit-selects or part-selects
- Locally defined registers or nets or their bit-selects or part-selects

- Compile time constants (constant numbers and specify parameters)

Table 13-2 contains a list of valid operators that may be used in conditional expressions:

Table 13-2—List of valid operators in state dependent path delay expression

Operator	Description	Operator	Description
~	bit-wise negation	&	reduction and
&	bit-wise and		reduction or
	bit-wise or	^	reduction xor
^	bit-wise xor	~&	reduction nand
^~ ~^	bit-wise xnor	~	reduction nor
==	logical equality	^~ ~^	reduction xnor
!=	logical inequality	{ }	concatenation
&&	logical and	{ { } }	replication
	logical or	?:	conditional
!	logical not		

A conditional expression shall evaluate to true (1) for the state-dependent path to be assigned a delay value. If the conditional expression evaluates to x or z, it shall be treated as true. If the conditional expression evaluates to multiple bits, the least significant bit shall represent the result. The conditional expression can have any number of operands and operators.

13.3.4.2 Simple state-dependent paths

If the path description of a state-dependent path is a simple path, then it is called a *simple state-dependent path*. The simple path description is discussed in 13.3.2.

Examples:

Example 1—The following example uses state-dependent paths to describe the timing of an XOR gate.

```

module XORgate (a, b, out);
input a, b;
output out;

xor x1 (out, a, b);

specify
  specparam noninvrise = 1, noninvfall = 2
  specparam invertrise = 3, invertfall = 4;
  if (a) (b=> out) = (invertrise, invertfall);
  if (b) (a=> out) = (invertrise, invertfall);
  if (~a)(b=> out) = (noninvrise, noninvfall);
  if (~b)(a=> out) = (noninvrise, noninvfall);
endspecify
endmodule

```

In this example, first two state-dependent paths describe a pair of output rise and fall delay times when the XOR gate (**x1**) inverts a changing input. The last two state-dependent paths describe another pair of output rise and fall delay times when the XOR gate buffers a changing input.

Example 2—The following example models a partial ALU. The state-dependent paths specify different delays for different ALU operations.

```

module ALU (o1, i1, i2, opcode);
input [7:0] i1, i2;
input [2:1] opcode;
output [7:0] o1;

//functional description omitted
specify
    // add operation
    if (opcode == 2'b00) (i1,i2 *> o1) = (25.0, 25.0);
    // pass-through i1 operation
    if (opcode == 2'b01) (i1 => o1) = (5.6, 8.0);
    // pass-through i2 operation
    if (opcode == 2'b10) (i2 => o1) = (5.6, 8.0);
    // delays on opcode changes
    (opcode => o1) = (6.1, 6.5);
endspecify
endmodule

```

In the preceding example, the first three path declarations declare paths extending from operand inputs **i1** and **i2** to the **o1** output. The delays on these paths are assigned to operations on the basis of the operation specified by the inputs on **opcode**. The last path declaration declares a path from the **opcode** input to the **o1** output.

13.3.4.3 Edge-sensitive state-dependent paths

If the path description of a state-dependent path describes an edge-dependent path, then the state-dependent path is called an *edge-sensitive state-dependent path*. The edge-sensitive paths are discussed in 13.3.3.

Different delays can be assigned to the same edge-sensitive path as long as the following criteria are met:

- The edge, condition, or both make each declaration unique.
- The port is referenced in the same way in all path declarations (entire port, bit-select, or part-select).

Examples:

Example 1

```

if ( !reset && !clear )
    ( posedge clock => ( out +: in ) ) = (10, 8) ;

```

In this example, if the positive edge of **clock** occurs when **reset** and **clear** are low, and a module path extends from **clock** to **out** using a rise delay of 10 and a fall delay of 8.

Example 2—The following example shows four edge-sensitive path declarations. Note that each path has a unique edge or condition.


```

specify
  ( posedge clk => ( q[0] : data ) ) = (10, 5);
  ( negedge clk => ( q[0] : data ) ) = (20, 12);

  if (reset)
    ( posedge clk => ( q[0] : data ) ) = (15, 8);
  if (!reset && cntrl)
    ( posedge clk => ( q[0] : data ) ) = (6, 2);
endspecify

```

Example 3—The two state-dependent path declarations shown below are not legal because even though they have different conditions, the destinations are not specified in the same way: the first destination is a part-select, the second is a bit-select.

```

specify
  if (reset)
    ( posedge clk => (q[3:0]:data) ) = (10,5);
  if (!reset)
    ( posedge clk => (q[0]:data) ) = (15,8);
endspecify

```

13.3.4.4 The **ifnone** condition

The **ifnone** keyword is used to specify a default state-dependant path delay when all other conditions for the path are false. The **ifnone** condition shall specify the same module path source and destination as the state-dependent module paths. The following rules apply to module paths specified with the **ifnone** condition:

- Only simple module paths may be described with an **ifnone** condition.
- The state-dependent paths that correspond to the **ifnone** path may be either simple module paths or edge-sensitive paths.
- If there are no corresponding state-dependent module paths to the **ifnone** module path, then the **ifnone** module path shall be treated the same as an unconditional simple module path.
- It is illegal to specify both an **ifnone** condition for a module path and an unconditional simple module path for the same module path.

Examples:

Example 1—The following are valid state-dependent path combinations.

```

if (C1) (IN => OUT) = (1,1);
ifnone (IN => OUT) = (2,2);

// add operation
if (opcode == 2'b00) (i1,i2 *> o1) = (25.0, 25.0);
// pass-through i1 operation
if (opcode == 2'b01) (i1 => o1) = (5.6, 8.0);
// pass-through i2 operation
if (opcode == 2'b10) (i2 => o1) = (5.6, 8.0);
// all other operations
ifnone (i2 => o1) = (15.0, 15.0);

(posedge CLK => (Q +: D)) = (1,1);
ifnone (CLK => Q) = (2,2);

```

Example 2—The following module path description combination is illegal because it combines a state-dependent path using an **ifnone** condition and an unconditional path for the same module path.

```

if (a) (b => out) = (2,2);
if (b) (a => out) = (2,2);
ifnone (a => out) = (1,1);
(a => out) = (1,1);

```

13.3.4.5 State-dependent and unconditional path precedence

A simple module path with no edge-sensitive or state-dependent conditions is an unconditional path. If both an unconditional path and a state-dependent path are specified for the same module path, then the unconditional path delay shall take precedence over the state-dependent path delay.

13.3.5 Full connection and parallel connection paths

The operator **>* shall be used to establish a *full connection* between source and destination. In a full connection, every bit in the source shall connect to every bit in the destination. The module path source need not have the same number of bits as the module path destination.

The full connection can handle most types of module paths, since it does not restrict the size or number of source signals and destination signals. The following situations require the use of full connections:

- To describe a module path between a vector and a scalar
- To describe a module path between vectors of different sizes
- To describe a module path with multiple sources or multiple destinations in a single statement (see 13.3.6)

The operator *=>* shall be used to establish a *parallel connection* between source and destination. In a parallel connection, each bit in the source shall connect to one corresponding bit in the destination. Parallel module paths can be created only between sources and destinations that contain the same number of bits.

Parallel connections are more restrictive than full connections. They only connect one source to one destination, where each signal contains the same number of bits. Therefore, a parallel connection may only be used to describe a module path between two vectors of the same size. Since scalars are one bit wide, either **>* or *=>* may be used to set up bit-to-bit connections between two scalars.

Examples:

Example 1—Figure 13-2 illustrates how a parallel connection differs from a full connection between two 4-bit vectors.

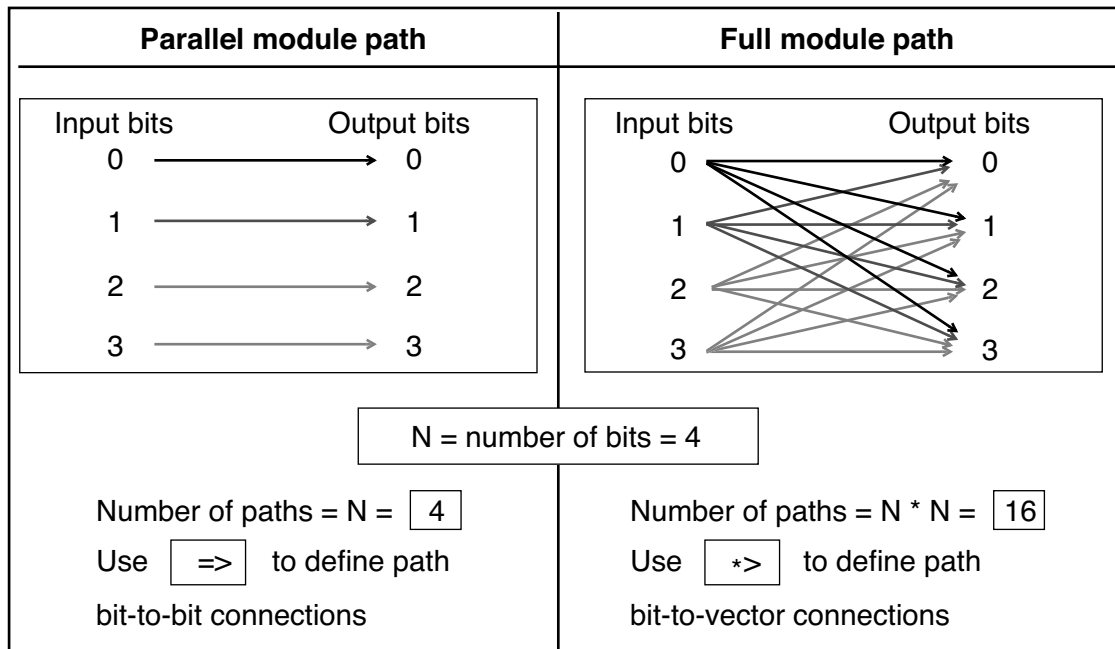


Figure 13-2—The difference between parallel and full connection paths

Example 2—The following example shows module paths for a 2:1 multiplexor with two 8-bit inputs and one 8-bit output.

```

module mux8 (in1, in2, s, q) ;
output [7:0] q;
input [7:0] in1, in2;
input s;
// Functional description omitted ...
specify
  (in1 => q) = (3, 4) ;
  (in2 => q) = (2, 3) ;
  (s *> q) = 1;
endspecify
endmodule

```

The module path from *s* to *q* uses a full connection (**>*) because it connects a scalar source—the 1-bit select line—to a vector destination—the 8-bit output bus. The module paths from both input lines *in1* and *in2* to *q* use a parallel connection (*=>*) because they set up parallel connections between two 8-bit buses.

13.3.6 Declaring multiple module paths in a single statement

Multiple module paths may be described in a single statement by using the symbol **>* to connect a comma-separated list of sources to a comma-separated list of destinations. When describing multiple module paths in one statement, the lists of sources and destinations may contain a mix of scalars and vectors of any size.

The connection in a multiple module path declaration is always a full connection.

Example:

```
(a, b, c *> q1, q2) = 10;
```

is equivalent to the following six individual module path assignments:

```
(a *> q1) = 10 ;
(b *> q1) = 10 ;
(c *> q1) = 10 ;
(a *> q2) = 10 ;
(b *> q2) = 10 ;
(c *> q2) = 10 ;
```

13.3.7 Module path polarity

The polarity of a module path is an arbitrary specification indicating whether or not the direction of a signal transition is inverted as it propagates from the input to the output. This arbitrary polarity description does not affect the actual propagation of data or events through the model; how a rise or a fall at the source propagates to the destination depends on the internal logic of the module.

Module paths may specify any of three polarities:

- Unknown polarity
- Positive polarity
- Negative polarity

13.3.7.1 Unknown polarity

By default, module paths shall have *unknown polarity*—that is, a transition at the path source may propagate to the destination in an unpredictable way, as follows:

- A rise at the source may cause either a rise transition, a fall transition, or no transition at the destination.
- A fall at the source may cause either a rise transition, a fall transition, or no transition at the destination.

A module path specified either as a full connection or a parallel connection, but without a polarity operator + or −, shall be treated as a module path with unknown polarity.

13.3.7.2 Positive polarity

For module paths with *positive polarity*, any transition at the source may cause the same transition at the destination, as follows:

- A rise at the source may cause either a rise transition or no transition at the destination.
- A fall at the source may cause either a fall transition or no transition at the destination.

A module path with positive polarity shall be specified by prefixing the + polarity operator to => or *>.

13.3.7.3 Negative polarity

For module paths with *negative polarity*, any transition at the source may cause the opposite transition at the destination, as follows:

- A rise at the source may cause either a fall transition or no transition at the destination.
- A fall at the source may cause either a rise transition or no transition at the destination.

A module path with negative polarity shall be specified by prefixing the – polarity operator to => or *>.

Examples:

The following examples show each type of path polarity:

```
// Positive polarity
(In1 +=> q) = In_to_q ;
(s    +*> q) = s_to_q ;

// Negative polarity
(In1 -=> q) = In_to_q ;
(s    -*> q) = s_to_q ;

// Unknown polarity
(In1 => q) = In_to_q ;
(s    *> q) = s_to_q ;
```

13.4 Assigning delays to module paths

The delays that occur at the module outputs where paths terminate shall be specified by assigning delay values to the module path descriptions. The syntax for specifying delay values is shown in Syntax 13-7.

```
path_delay_value ::=
    list_of_path_delay_expressions
    | ( list_of_path_delay_expressions )
list_of_path_delay_expressions ::=
    t_path_delay_expression
    | trise_path_delay_expression , tfall_path_delay_expression
    | trise_path_delay_expression , tfall_path_delay_expression ,
    tz_path_delay_expression
    | t01_path_delay_expression , t10_path_delay_expression ,
    t0z_path_delay_expression ,
    tz1_path_delay_expression , t1z_path_delay_expression ,
    tz0_path_delay_expression
    | t01_path_delay_expression , t10_path_delay_expression ,
    t0z_path_delay_expression ,
```

Syntax 13-7—Syntax for path delay value

In module path delay assignments, a module path description (see 13.3) is specified on the left-hand side, and one or more delay values are specified on the right-hand side. The delay values may be optionally enclosed in a pair of parentheses. There may be one, two, three, six, or twelve delay values assigned to a module path, as described in 13.4.1. The delay values shall be constant expressions containing literals or specparams, and there may be a delay expression of the form `min:typ:max`.

Example:

```

specify
  // Specify Parameters
  specparam tRise_clk_q = 45:150:270, tFall_clk_q=60:200:350;
  specparam tRise_Control = 35:40:45, tFall_control=40:50:65;

  // Module Path Assignments
  (clk => q) = (tRise_clk_q, tFall_clk_q);
  (clr, pre *> q) = (tRise_control, tFall_control);
endspecify

```

In the example above, the specify parameters declared following the **specparam** keyword specify values for the module path delays. The module path assignments assign those module path delays to the module paths.

13.4.1 Specifying transition delays on module paths

Each path delay expression may be a single value—representing the typical delay—or a colon-separated list of three values—representing a *minimum*, *typical*, and *maximum* delay, in that order. If the path delay expression results in a negative value, it shall be treated as zero. Table 13-3 describes how different path delay values shall be associated with various transitions. The path delay expression names refer to the names used in Syntax 13-7.

Table 13-3—Associating path delay expressions with transitions

Transitions	Number of path delay expressions specified				
	1	2	3	6	12
0 -> 1	t	trise	trise	t01	t01
1 -> 0	t	tfall	tfall	t10	t10
0 -> z	t	trise	tz	t0z	t0z
z -> 1	t	trise	trise	tz1	tz1
1 -> z	t	tfall	tz	t1z	t1z
z -> 0	t	tfall	tfall	tz0	tz0
0 -> x	*	*	*	*	t0x
x -> 1	*	*	*	*	tx1
1 -> x	*	*	*	*	t1x
x -> 0	*	*	*	*	tx0
x -> z	*	*	*	*	txz
z -> x	*	*	*	*	tzx

* See 13.4.2.

Example:

```
// one expression specifies all transitions
(C => Q) = 20;
(C => Q) = 10:14:20;

// two expressions specify rise and fall delays
specparam tPLH1 = 12, tPHL1 = 25;
specparam tPLH2 = 12:16:22, tPHL2 = 16:22:25;
(C => Q) = ( tPLH1, tPHL1 ) ;
(C => Q) = ( tPLH2, tPHL2 ) ;

// three expressions specify rise, fall, and z transition delays
specparam tPLH1 = 12, tPHL1 = 22, tPz1 = 34;
specparam tPLH2 = 12:14:30, tPHL2 = 16:22:40, tPz2 = 22:30:34;
(C => Q) = (tPLH1, tPHL1, tPz1);
(C => Q) = (tPLH2, tPHL2, tPz2);

// six expressions specify transitions to/from 0, 1, and z
specparam t01 = 12, t10 = 16, t0z = 13,
          tz1 = 10, t1z = 14, tz0 = 34 ;
(C => Q) = ( t01, t10, t0z, tz1, t1z, tz0 ) ;
specparam T01 = 12:14:24, T10 = 16:18:20, T0z = 13:16:30 ;
specparam Tz1 = 10:12:16, T1z = 14:23:36, Tz0 = 15:19:34 ;
(C => Q) = ( T01, T10, T0z, Tz1, T1z, Tz0 ) ;

// twelve expressions specify all transition delays explicitly
specparam t01=10, t10=12, t0z=14, tz1=15, t1z=29, tz0=36,
          t0x=14, tx1=15, t1x=15, tx0=14, txz=20, tzx=30 ;
(c => Q) = (t01, t10, t0z, tz1, t1z, tz0,
          t0x, tx1, t1x, tx0, txz, tzx) ;
```

13.4.2 Specifying x transition delays

If the x transition delays are not explicitly specified, the calculation of delay values for x transitions is based on the following two pessimistic rules:

- Transitions from a known state **s** to **x** shall occur as quickly as possible—that is, the shortest possible delay shall be used for any transition to **x**.
- Transitions from **x** to a known state shall take as long as possible—that is, the longest possible delay shall be used for any transition from **x**.

Table 13-4 presents the general algorithm for calculating delay values for x transitions, along with specific examples. The following two groups of x transitions are represented in the table:

- a) Transition from a known state **s** to **x**: $s \rightarrow x$
- b) Transition from **x** to a known state **s**: $x \rightarrow s$

Table 13-4—Calculating delays for x transitions

X transition	Delay value
General algorithm	
$s \rightarrow x$	minimum ($s \rightarrow$ other known signals)
$x \rightarrow s$	maximum (other known signals $\rightarrow s$)
Specific transitions	
$0 \rightarrow x$	minimum ($0 \rightarrow z$ delay, $0 \rightarrow 1$ delay)
$1 \rightarrow x$	minimum ($1 \rightarrow z$ delay, $1 \rightarrow 0$ delay)
$z \rightarrow x$	minimum ($z \rightarrow 1$ delay, $z \rightarrow 0$ delay)
$x \rightarrow 0$	maximum ($z \rightarrow 0$ delay, $1 \rightarrow 0$ delay)
$x \rightarrow 1$	maximum ($z \rightarrow 1$ delay, $0 \rightarrow 1$ delay)
$x \rightarrow z$	maximum ($1 \rightarrow z$ delay, $0 \rightarrow z$ delay)
Usage: (C \Rightarrow Q) = (5, 12, 17, 10, 6, 22) ;	
$0 \rightarrow x$	minimum (17, 5) = 5
$1 \rightarrow x$	minimum (6, 12) = 6
$z \rightarrow x$	minimum (10, 22) = 10
$x \rightarrow 0$	maximum (22, 12) = 22
$x \rightarrow 1$	maximum (10, 5) = 10
$x \rightarrow z$	maximum (6, 17) = 17

13.5 Mixing module path delays and distributed delays

If a module contains module path delays and distributed delays (delays on primitive instances within the module), the larger of the two delays for each path shall be used.

Examples:

Example 1—Figure 13-3 illustrates a simple circuit modeled with a combination of distributed delays and path delays (only the D input to Q output path is illustrated). Here, the delay on the module path from input D to output Q = 22, while the sum of the distributed delays = 0 + 1 = 1. Therefore, a transition on Q caused by a transition on D will occur 22 time units after the transition on D.

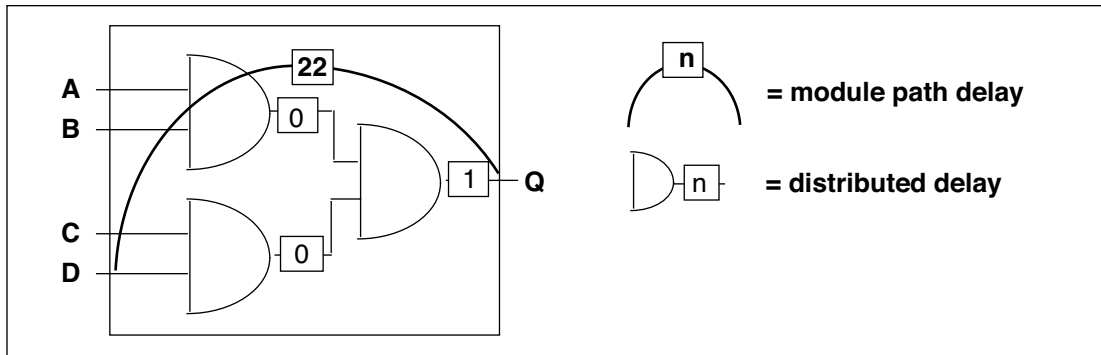


Figure 13-3—Module path delays longer than distributed delays

Example 2—In Figure 13-4, the delay on the module path from D to Q = 22, but the distributed delays along that module path now add up to $10 + 20 = 30$. Therefore, an event on Q caused by an event on D will occur 30 time units after the event on D.

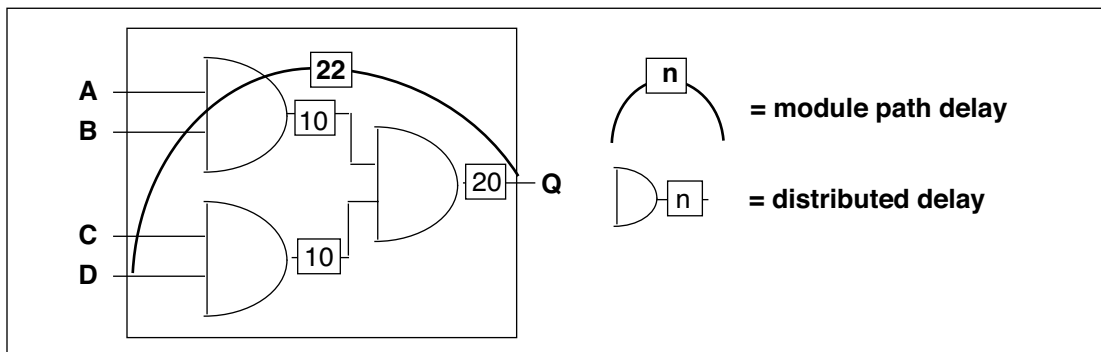


Figure 13-4—Module path delays shorter than distributed delays

13.6 Driving wired logic

Module path output nets shall not have more than one driver within the module. Therefore, wired logic is not allowed at module path outputs.

Figure 13-6 illustrates a violation of this wired-output rule and a method of avoiding the rule violation.

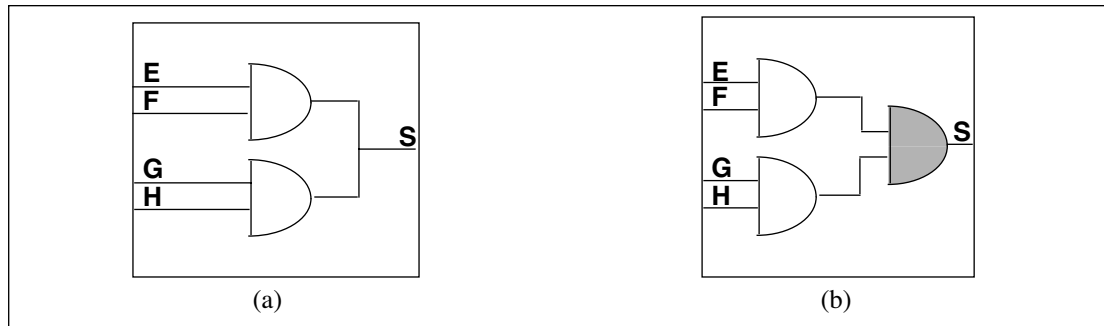


Figure 13-5—Legal and illegal module paths

In Figure 13-5 (a), any module path to S is illegal because the path destination has two drivers.

Assuming signal S in Figure 13-5 (a) is a *wired-and*, this limitation can be circumvented by replacing wired logic with gated logic to create a single driver to the output. Figure 13-5 (b) shows how adding a third **and** gate—the shaded gate—solves the problem for the module in Figure 13-5 (a).

The example in Figure 13-6 is also illegal. In this example, when the outputs Q and R are wired together, it creates a condition where both paths have multiple drivers from within the same module.

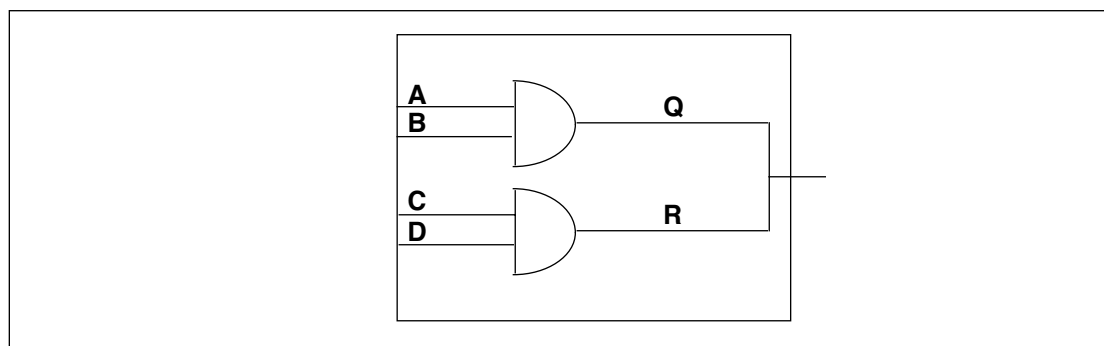


Figure 13-6—Illegal module paths

Although multiple output drivers to a path destination are prohibited *inside* the same module, they are allowed *outside* the module. The example in Figure 13-7 is legal since Q and R each have only one driver within the module in which the module paths are specified.

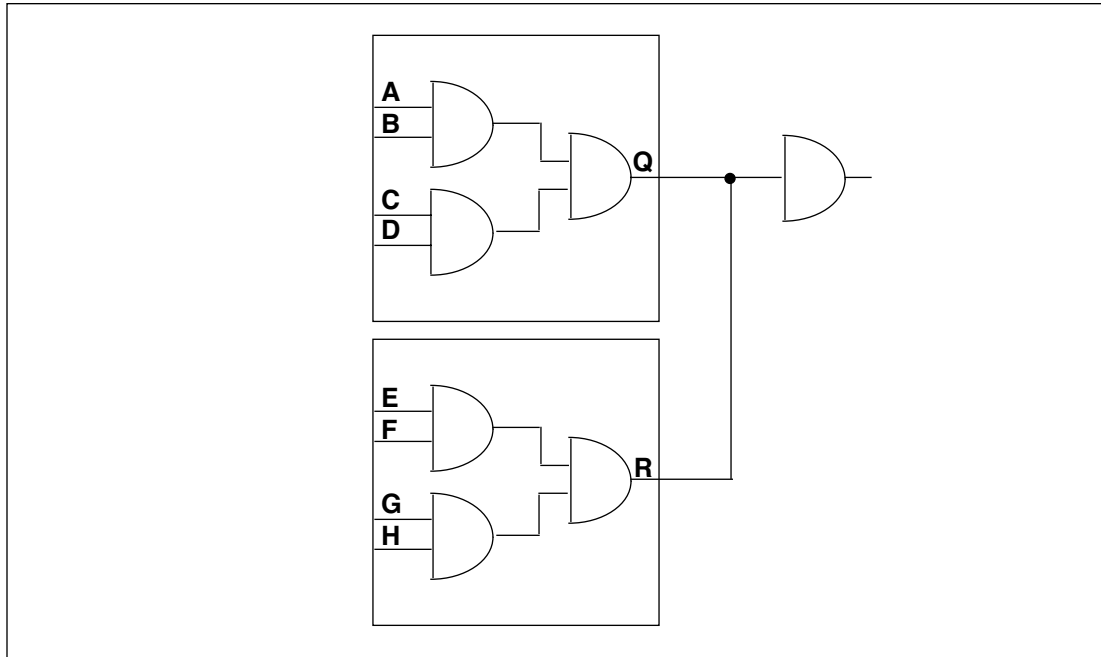


Figure 13-7—Legal module paths

13.7 Controlling pulses on module paths with PATHPULSE\$

A *pulse* is made up of two scheduled transitions on a module path destination that occur in a shorter period of time than the delay specified for that module path. By default, pulses on a path destination are rejected. That is, only transitions that are the same as or longer than the module path delay will propagate to the path destination. This is analogous to the inertial delay model of pulse propagation.

A special specparam, **PATHPULSE\$**, is used to specify a range of pulse widths that will propagate to the path destination. The following pulse width ranges are specified using **PATHPULSE\$**:

- A pulse width range for which a pulse shall be rejected
- A pulse width range for which a pulse shall be allowed to propagate to the path destination
- A pulse width range for which a pulse shall generate a logic x on the path destination

The syntax for specifying pulse control ranges is given in Syntax 13-8.

```

pulse_control_specparam ::=
    PATHPULSE$ = ( reject_limit_value [ , error_limit_value ] ) ;
    | PATHPULSE$specify_input_terminal_descriptor$specify_output_terminal_descriptor
      = ( reject_limit_value [ , error_limit_value ] ) ;
limit_value ::= constant_mintypmax_expression
    
```

Syntax 13-8—Syntax for **PATHPULSE\$** pulse control

The reject limit value defines a reject limit for the pulse control range. A pulse that is less than the reject limit shall not propagate to the module path destination. Any pulse that is greater than the reject limit shall propagate to the path destination as determined by the error limit value.

The error limit value defines an error limit for the pulse control range. A pulse that is less than the error limit but greater than or equal to the reject limit shall generate a logic x on the path destination. Any pulse that is greater than the error limit shall propagate to the path destination.

If only the reject limit value is specified, that value shall apply to both the reject limit and the error limit.

The reject limit and error limit may be specified for a specific module path. When no module path is specified, then the reject limit and error limit shall apply to all module paths defined in a module. If both path-specific PATHPULSE\$ specparams and a non-path-specific PATHPULSE\$ specparam appear in the same module, then the path-specific specparams shall take precedence for the specified paths.

The module path input terminals and output terminals shall conform to the rules for module path inputs and outputs, with the following restriction: the terminals may not be a bit-select or part-select of a vector.

If a module path declaration declares multiple paths, then the PATHPULSE\$ specparam shall only be specified for the first path input terminal and the first path output terminal. The reject limit and error limit specified shall apply to all other paths in the multiple path declaration.

Example:

In the following example, the path (`clk=>q`) acquires a reject limit of 2 and an error limit of 9, as defined by the first PATHPULSE\$ declaration. The paths (`clr*>q`) and (`pre*>q`) receive a reject limit of 0 and an error limit of 4, as specified by the second PATHPULSE\$ declaration. The path (`data=>q`) is not explicitly defined in any of the PATHPULSE\$ declarations, and so it acquires reject and error limit of 3, as defined by the last PATHPULSE\$ declaration.

```

specify
    (clk => q) = 12;
    (data => q) = 10;
    (clr, pre *> q) = 4;

    specparam
        PATHPULSE$clk$q = (2,9),
        PATHPULSE$clr$q = (0,4),
        PATHPULSE$ = 3;
endspecify

```

Section 14

System tasks and functions

This section describes system tasks and functions that are considered part of the Verilog HDL. These system tasks and functions are divided into ten categories as follows:

Display tasks		[14.1]	PLA modeling tasks		[14.6]
\$display	\$strobe		\$async\$and\$array	\$sync\$and\$plane	
\$displayb	\$strobeb		\$async\$nand\$array	\$sync\$nand\$plane	
\$displayh	\$strobeh		\$async\$or\$array	\$sync\$or\$plane	
\$displayo	\$strobeo		\$async\$nor\$array	\$sync\$nor\$plane	
\$monitor	\$write		\$async\$and\$array	\$sync\$and\$plane	
\$monitorb	\$writeb		\$async\$nand\$array	\$sync\$nand\$plane	
\$monitorh	\$writeh		\$async\$or\$array	\$sync\$or\$plane	
\$monitro	\$writeo		\$async\$nor\$array	\$sync\$nor\$plane	
\$monitoroff	\$monitoron				
File I/O tasks		[14.2]	Stochastic analysis tasks		[14.7]
\$fclose	\$fopen		\$q_initialize	\$q_add	
\$fdisplay	\$fstrobe		\$q_remove	\$q_full	
\$fdisplayb	\$fstrobeb		\$q_exam	\$random	
\$fdisplayh	\$fstrobeh		Simulation time functions		[14.8]
\$fdisplayo	\$fstrobeo		\$realtime	\$stime	
\$fmonitor	\$fwrite		\$time		
\$fmonitorb	\$fwriteb		Conversion functions for reals		[14.9]
\$fmonitorh	\$fwriteh		\$bitstoreal	\$realtobits	
\$fmonitro	\$fwriteo		\$itor	\$rtoi	
\$readmemb	\$readmemh				
Timescale tasks		[14.3]	Probabilistic distribution functions		[14.10]
\$prntimescale	\$timeformat		\$dist_chi_square	\$dist_erlang	
Simulation control tasks		[14.4]	\$dist_exponential	\$dist_normal	
\$finish	\$stop		\$dist_poisson	\$dist_t	
Timing check tasks		[14.5]	\$dist_uniform		
\$hold	\$nochange				
\$period	\$recovery				
\$setup	\$setuphold				
\$skew	\$width				

These utility tasks and functions provide some broadly useful capabilities. The following clauses describe the behavior of these tasks and functions. Additional tasks for value change dump (VCD) are described in Section 15.

14.1 Display system tasks

The display group of system tasks are divided into three categories: the display and write tasks, strobed monitoring tasks, and continuous monitoring tasks.

14.1.1 The display and write tasks

```
display_tasks ::= display_task_name ( list_of_arguments ) ;
display_task_name ::=
    $display | $displayb | $displayo | $displayh
    | $write | $writeb | $writeo | $writeh
```

Syntax 14-1—Syntax for \$display and \$write system tasks

These are the main system task routines for displaying information. The two sets of tasks are identical except that **\$display** automatically adds a newline character to the end of its output, whereas the **\$write** task does not.

The **\$display** and **\$write** tasks display their arguments in the same order as they appear in the argument list. Each argument can be a quoted string, an expression that returns a value, or a null argument.

The contents of string arguments are output literally except when certain escape sequences are inserted to display special characters or to specify the display format for a subsequent expression.

Escape sequences are inserted into a string in three ways:

- The special character `\` indicates that the character to follow is a literal or nonprintable character (see Table 14-1).
- The special character `%` indicates that the next character should be interpreted as a format specification that establishes the display format for a subsequent expression argument (see Table 14-2). For each `%` character that appears in a string, a corresponding expression argument shall be supplied after the string.
- The special character string `%%` indicates the display of the percent sign character `%` (see Table 14-1).

Any null argument produces a single space character in the display. (A null argument is characterized by two adjacent commas in the argument list.)

The **\$display** task, when invoked without arguments, simply prints a newline character. A **\$write** task supplied without parameters prints nothing at all.

14.1.1.1 Escape sequences for special characters

The escape sequences given in Table 14-1, when included in a string argument, cause special characters to be displayed.

Table 14-1—Escape sequences for printing special characters

<code>\n</code>	The newline character
<code>\t</code>	The tab character
<code>\\</code>	The <code>\</code> character

Table 14-1—Escape sequences for printing special characters (continued)

\"	The " character
\ddd	A character specified by 1 to 3 octal digits
%%	The % character

Example:

```

module disp;
initial begin
    $display( "\\t\\n\"123" );
end
endmodule

\      \
"S

```

14.1.1.2 Format specifications

Table 14-2 shows the escape sequences used for format specifications. Each escape sequence, when included in a string argument, specifies the display format for a subsequent expression. For each % character (except %m) that appears in a string, a corresponding expression shall follow the string in the argument list. The value of the expression replaces the format specification when the string is displayed.

Any expression argument that has no corresponding format specification is displayed using the default decimal format in **\$display** and **\$write**, binary format in **\$displayb** and **\$writeb**, octal format in **\$displayo** and **\$writeo**, and hexadecimal format in **\$displayh** and **\$writeh**.

Table 14-2—Escape sequences for format specifications

%h or %H	Display in hexadecimal format
%d or %D	Display in decimal format
%o or %O	Display in octal format
%b or %B	Display in binary format
%c or %C	Display in ASCII character format
%v or %V	Display net signal strength
%m or %M	Display hierarchical name
%s or %S	Display as a string
%t or %T	Display in current time format

The format specifications in Table 14-3 are used with real numbers and have the full formatting capabilities available in the C language. For example, the format specification %10.3g specifies a minimum field width of 10 with 3 fractional digits.

Table 14-3—Format specifications for real numbers

%e or %E	Display ‘real’ in an exponential format
%f or %F	Display ‘real’ in a decimal format
%g or %G	Display ‘real’ in exponential or decimal format, whichever format results in the shorter printed output

The net signal strength, hierarchical name, and string format specifications are described in 14.1.1.5 through 14.1.1.7.

The `%t` format specification works with the `$timeformat` system task to specify a uniform time unit, time precision, and format for reporting timing information from various modules that use different time units and precisions. The `$timeformat` task is described in 14.3.2.

Example:

```

module disp;
reg [31:0] rval;
pulldown (pd);
initial begin
    rval = 101;
    $display("rval = %h hex %d decimal",rval,rval);
    $display("rval = %o octal\nrval = %b bin",rval,rval);
    $display("rval has %c ascii character value",rval);
    $display("pd strength value is %v",pd);
    $display("current scope is %m");
    $display("%s is ascii value for 101",101);
    $display("simulation time is %t", $time);
end
endmodule

rval = 000000065 hex          101 decimal
rval = 00000000145 octal
rval = 0000000000000000000000001100101 bin
rval has e ascii character value
pd strength value is StX
current scope is disp
e is ascii value for 101
simulation time is 0

```

14.1.1.3 Size of displayed data

For expression arguments, the values written to the output file (or terminal) are sized automatically.

For example, the result of a 12-bit expression would be allocated three characters when displayed in hexadecimal format and four characters when displayed in decimal format, since the largest possible value for the expression is FFF (hexadecimal) and 4095 (decimal).

When displaying decimal values, leading zeros are suppressed and replaced by spaces. In other radices, leading zeros are always displayed.

The automatic sizing of displayed data may be overridden by inserting a zero between the % character and the letter

that indicates the radix, as shown in the following example.

```
$display( "d=%0h a=%0h", data, addr);
```

Example:

```
module printval;  
reg [11:0] r1;  
initial begin  
    r1 = 10;  
    $display( "Printing with maximum size - :%d: :%h:", r1,r1 );  
    $display( "Printing with minimum size - :%0d: :%0h:", r1,r1 );  
end  
endmodule  
  
Printing with maximum size - : 10: :00a:  
Printing with minimum size - :10: :a:
```

In this example, the result of a 12-bit expression is displayed. The first call to **\$display** uses the standard format specifier syntax and produces results requiring four and three columns for the decimal and hexadecimal radices, respectively. The second **\$display** call uses the %0 form of the format specifier syntax and produces results requiring two columns and one column, respectively.

14.1.1.4 Unknown and high impedance values

When the result of an expression contains an unknown or high impedance value, the following rules apply to displaying that value.

In decimal (%d) format:

- If all bits are at the unknown value, a single lowercase “x” character is displayed.
- If all bits are at the high impedance value, a single lowercase “z” character is displayed.
- If some, but not all, bits are at the unknown value, the uppercase “X” character is displayed.
- If some, but not all, bits are at the high impedance value, the uppercase “Z” character is displayed.
- Decimal numerals always appear right-justified in a fixed-width field.

In hexadecimal (%h) and octal (%o) formats:

- Each group of 4 bits is represented as a single hexadecimal digit; each group of 3 bits is represented as a single octal digit.
- If all bits in a group are at the unknown value, a lowercase “x” is displayed for that digit.
- If all bits in a group are at a high impedance state, a lowercase “z” is printed for that digit.
- If some, but not all, bits in a group are unknown, an uppercase “X” is displayed for that digit.
- If some, but not all, bits in a group are at a high impedance state, then an uppercase “Z” is displayed for that digit.

In binary (%b) format, each bit is printed separately using the characters 0, 1, x, and z.

Example:

STATEMENT	RESULT
<code>\$display ("%d", 1'bX);</code>	<code>x</code>
<code>\$display ("%h", 14'bX01010);</code>	<code>xxXa</code>
<code>\$display ("%h %o", 12'b001xxx101x01, 12'b001xxx101x01);</code>	<code>XXX 1x5X</code>

14.1.1.5 Strength format

The `%v` format specification is used to display the strength of scalar nets. For each `%v` specification that appears in a string, a corresponding scalar reference shall follow the string in the argument list.

The strength of a scalar net is reported in a three-character format. The first two characters indicate the strength. The third character indicates the current logic value of the scalar and may be any one of the values given in Table 14-4.

Table 14-4—Logic value component of strength format

0	For a logic 0 value
1	For a logic 1 value
X	For an unknown value
Z	For a high impedance value
L	For a logic 0 or high impedance value
H	For a logic 1 or high impedance value

The first two characters—the strength characters—are either a two-letter mnemonic or a pair of decimal digits. Usually, a mnemonic is used to indicate strength information; however, in less typical cases, a pair of decimal digits may be used to indicate a range of strength levels. Table 14-5 shows the mnemonics used to represent the various strength levels.

Table 14-5—Mnemonics for strength levels

Mnemonic	Strength name	Strength level
Su	Supply drive	7
St	Strong drive	6
Pu	Pull drive	5
La	Large capacitor	4
We	Weak drive	3
Me	Medium capacitor	2
Sm	Small capacitor	1
Hi	High impedance	0

Note that there are four driving strengths and three charge storage strengths. The driving strengths are associated with gate outputs and continuous assignment outputs. The charge storage strengths are associated with the **triereg** type net. (See Section 7 for strength modeling.)

For the logic values 0 and 1, a mnemonic is used when there is no range of strengths in the signal. Otherwise, the logic value is preceded by two decimal digits, which indicate the maximum and minimum strength levels.

For the unknown value, a mnemonic is used when both the 0 and 1 strength components are at the same strength level. Otherwise, the unknown value X is preceded by two decimal digits, which indicate the 0 and 1 strength levels respectively.

The high impedance strength cannot have a known logic value; the only logic value allowed for this level is Z.

For the values L and H, a mnemonic is always used to indicate the strength level.

Examples:

always

```
#15 $display($time, , "group=%b signals=%v %v %v", {s1,s2,s3}, s1, s2, s3);
```

The example below shows the output that might result from such a call, while Table 14-6 explains the various strength formats that appear in the output.

```
0 group=111 signals=St1 Pu1 St1
15 group=011 signals=Pu0 Pu1 St1
30 group=0xz signals=520 PuH HiZ
45 group=0xx signals=Pu0 65X StX
60 group=000 signals=Me0 St0 St0
```

Table 14-6—Explanation of strength formats

St1	Means a strong driving 1 value
Pu0	Means a pull driving 0 value
HiZ	Means the high-impedance state
Me0	Means a 0 charge storage of medium capacitor strength
StX	Means a strong driving unknown value
PuH	Means a pull driving strength of 1 or high-impedance value
65X	Means an unknown value with a strong driving 0 component and a pull driving 1 component
520	Means an 0 value with a range of possible strength from pull driving to medium capacitor

14.1.1.6 Hierarchical name format

The **%m** format specifier does not accept an argument. Instead, it causes the display task to print the hierarchical name of the module, task, function, or named block that invokes the system task containing the format specifier. This is use-

ful when there are many instances of the module that calls the system task. One obvious application is timing check messages in a flip-flop or latch module; the `%m` format specifier will pinpoint the module instance responsible for generating the timing check message.

14.1.1.7 String format

The `%s` format specifier is used to print ASCII codes as characters. For each `%s` specification that appears in a string, a corresponding parameter shall follow the string in the argument list. The associated argument is interpreted as a sequence of 8-bit hexadecimal ASCII codes, with each 8 bits representing a single character. If the argument is a variable, its value should be right-justified so that the rightmost bit of the value is the least-significant bit of the last character in the string. No termination character or value is required at the end of a string, and leading zeros are never printed.

14.1.2 Strobed monitoring

```
strobe_tasks ::= strobe_task_name ( list_of_arguments ) ;
strobe_task_name ::=
    $strobe | $strobeb | $strobeo | $strobeh
```

Syntax 14-2—Syntax for \$strobe system tasks

The system task **\$strobe** provides the ability to display simulation data at a selected time. That time is the end of the current simulation time, when all the simulation events that have occurred for that simulation time, just before simulation time is advanced. The arguments for this task are specified in exactly the same manner as for the **\$display** system task—including the use of escape sequences for special characters and format specifications (see 14.1.1).

Example:

```
forever @(negedge clock)
    $strobe ("At time %d, data is %h", $time, data);
```

In this example, **\$strobe** will write the time and data information to the standard output and the log file at each negative edge of the clock. The action will occur just before simulation time is advanced and after all other events at that time have occurred, so that the data written is sure to be the correct data for that simulation time.

14.1.3 Continuous monitoring

```
monitor_tasks ::=
    monitor_task_name [ ( list_of_arguments ) ] ;
    | $monitoron ;
    | $monitoroff ;
monitor_task_name ::=
    $monitor | $monitorb | $monitoro | $monitorh
```

Syntax 14-3—Syntax for \$monitor system tasks

The **\$monitor** task provides the ability to monitor and display the values of any variables or expressions specified as arguments to the task. The arguments for this task are specified in exactly the same manner as for the **\$display** system task—including the use of escape sequences for special characters and format specifications (see 14.1.1).

When a **\$monitor** task is invoked with one or more arguments, the simulator sets up a mechanism whereby each time a variable or an expression in the argument list changes value—with the exception of the **\$time**, **\$stime** or **\$realtime** system functions—the entire argument list is displayed at the end of the time step as if reported by the **\$display** task. If two or more arguments change value at the same time, only one display is produced that shows the new values.

Only one **\$monitor** display list can be active at any one time; however, a new **\$monitor** task with a new display list may be issued any number of times during simulation.

The **\$monitoron** and **\$monitoroff** tasks control a monitor flag that enables and disables the monitoring. Use **\$monitoroff** to turn off the flag and disable monitoring. The **\$monitoron** system task can be used to turn on the flag so that monitoring is enabled and the most recent call to **\$monitor** can resume its display. A call to **\$monitoron** shall produce a display immediately after it is invoked, regardless of whether a value change has taken place; this is used to establish the initial values at the beginning of a monitoring session. By default, the monitor flag is turned on at the beginning of simulation.

14.2 File input-output system tasks

The system tasks and functions for file-based operations are divided into three categories:

- Tasks that open and close files
- Tasks that output values into files
- Tasks that read values from files and load into memory

14.2.1 Opening and closing files

```
file_open_function ::=  
    integer multi_channel_descriptor = $fopen ( " file_name " ) ;  
  
file_close_task ::=  
    $fclose ( multi_channel_descriptor ) ;
```

Syntax 14-4—Syntax for \$fopen and \$fclose system tasks

The function **\$fopen** opens the file specified as an argument and returns a 32-bit unsigned multichannel descriptor that is uniquely associated with the file. It returns 0 if the file could not be opened for writing.

The multichannel descriptor should be thought of as a set of 32 flags, where each flag represents a single output channel. The least significant bit (bit 0) of a multichannel descriptor always refers to the standard output. The standard output is also called channel 0. The other bits refer to channels that have been opened by the **\$fopen** system function.

The first call to **\$fopen** opens channel 1 and returns a multichannel descriptor value of 2—that is, bit 1 of the descriptor is set. A second call to **\$fopen** opens channel 2 and returns a value of 4—that is, only bit 2 of the descriptor is set. Subsequent calls to **\$fopen** open channels 3, 4, 5, and so on and return values of 8, 16, 32, and so on, up to a maximum of 32 open channels. Thus, a channel number corresponds to an individual bit in a multichannel descriptor.

The **\$fclose** system task closes the channels specified in the multichannel descriptor and does not allow any further output to the closed channels. The **\$fopen** task will reuse channels that have been closed.

NOTE—The number of simultaneous output channels that may be active at any one time is dependent on the operating system.

14.2.2 File output system tasks

```

file_output_tasks ::=
    file_output_task_name ( multi_channel_descriptor , list_of_arguments ) ;
file_output_task_name ::=
    $fdisplay | $fdisplayb | $fdisplayh | $fdisplayf
    | $fwrite | $fwriteb | $fwriteh | $fwritef
    | $fstrobe | $fstrobebb | $fstrobeb | $fstrobeb
    | $fmonitor | $fmonitorb | $fmonitorh | $fmonitorf

```

Syntax 14-5—Syntax for file output system tasks

Each of the four formatted display tasks—**\$display**, **\$write**, **\$monitor**, and **\$strobe**—has a counterpart that writes to specific files as opposed to the standard output. These counterpart tasks—**\$fdisplay**, **\$fwrite**, **\$fmonitor**, and **\$fstrobe**—accept the same type of arguments as the tasks upon which they are based, with one exception: The first parameter shall be a multichannel descriptor that indicates where to direct the file output. A multichannel descriptor is either a variable or the result of an expression that takes the form of a 32-bit unsigned integer value. The value of the multichannel descriptor determines to which open files the task will write.

The **\$fstrobe** and **\$fmonitor** system tasks work just like their counterparts, **\$strobe** and **\$monitor**, except that they write to files using the multichannel descriptor for control. Unlike **\$monitor**, any number of **\$fmonitor** tasks can be set up to be simultaneously active. However, there is no counterpart to **\$monitoron** and **\$monitroff** tasks.

Example:

This example shows how to set up multichannel descriptors. In this example, three different channels are opened using the **\$fopen** function. The three multichannel descriptors that are returned by the function are then combined in a **bit-wise or** operation and assigned to the integer variable **messages**. The **messages** variable can then be used as the first parameter in a file output task to direct output to all three channels at once. To create a descriptor that directs output to the standard output as well, the **messages** variable is a **bit-wise** logical or with the constant 1, which effectively enables channel 0.

```

integer
    messages,      broadcast,
    cpu_chann,     alu_chann, mem_chann;
initial begin
    cpu_chann = $fopen("cpu.dat");
    if (cpu_chann == 0) $finish;
    alu_chann = $fopen("alu.dat");
    if (alu_chann == 0) $finish;
    mem_chann = $fopen("mem.dat");
    if (mem_chann == 0) $finish;
    messages = cpu_chann | alu_chann | mem_chann;
    // broadcast includes standard output
    broadcast = 1 | messages;
end
endmodule

```

The following file output tasks show how the channels opened in the preceding example might be used:

```

$fdisplay( broadcast, "system reset at time %d", $time );

$fdisplay( messages, "Error occurred on address bus",
           " at time %d, address = %h", $time, address );

forever @(posedge clock)
    $fdisplay( alu_chann, "acc= %h f=%h a=%h b=%h", acc, f, a, b );

```

14.2.3 Loading memory data from a file

```

load_memory_tasks ::=
    $readmemb ( " file_name " , memory_name [ , start_addr [ , finish_addr ] ] );
    | $readmemh ( " file_name " , memory_name [ , start_addr [ , finish_addr ] ] );

```

Syntax 14-6—Syntax for memory load system tasks

Two system tasks—**\$readmemb** and **\$readmemh**—read and load data from a specified text file into a specified memory. Either task may be executed at any time during simulation. The text file to be read shall contain only the following:

- White space (spaces, new lines, tabs, and form-feeds)
- Comments (both types of comment are allowed)
- Binary or hexadecimal numbers

The numbers shall have neither the length nor the base format specified. For **\$readmemb**, each number shall be binary. For **\$readmemh**, the numbers shall be hexadecimal. The unknown value (x or X), the high impedance value (z or Z), and the underscore (_) can be used in specifying a number as in a Verilog HDL source description. White space and/or comments shall be used to separate the numbers.

In the following discussion, the term “address” refers to an index into the array that models the memory.

As the file is read, each number encountered is assigned to a successive word element of the memory. Addressing is controlled both by specifying start and/or finish addresses in the system task invocation and by specifying addresses in the data file.

When addresses appear in the data file, the format is an “at” character (@) followed by a hexadecimal number as follows:

```
@hh...h
```

Both uppercase and lowercase digits are allowed in the number. No white space is allowed between the @ and the number. As many address specifications as needed within the data file may be used. When the system task encounters an address specification, it loads subsequent data starting at that memory address.

If no addressing information is specified within the system task, and no address specifications appear within the data file, then the default start address is the left-hand address given in the declaration of the memory. Consecutive words are loaded until either the memory is full or the data file is completely read. If the start address is specified in the task without the finish address, then loading starts at the specified start address and continues towards the right-hand

address given in the declaration of the memory.

If both start and finish addresses are specified as parameters to the task, then loading begins at the start address and continues toward the finish address, regardless of how the addresses are specified in the memory declaration.

When addressing information is specified both in the system task and in the data file, the addresses in the data file shall be within the address range specified by the system task parameters; otherwise, an error message is issued and the load operation is terminated.

A warning message is issued if the number of data words in the file differs from the number of words in the range implied by the start through finish addresses.

Example:

```
reg [ 7:0 ] mem[ 1:256 ];
```

Given this declaration, each of the following statements will load data into mem in a different manner:

```
initial $readmemh ( "mem.data", mem );
initial $readmemh ( "mem.data", mem, 16 );
initial $readmemh ( "mem.data", mem, 128, 1 );
```

The first statement will load up the memory at simulation time 0 starting at the memory address 1. The second statement will begin loading at address 16 and continue on towards address 256. For the third and final statement, loading will begin at address 128 and continue down towards address 1.

In the third case, when loading is complete, a final check is performed to ensure that exactly 128 numbers are contained in the file. If the check fails, a warning message is issued.

14.3 Timescale system tasks

The following system tasks display and set timescale information:

- a) **\$printtimescale**
- b) **\$timeformat**

14.3.1 \$printtimescale

The **\$printtimescale** system task displays the time unit and precision for a particular module.

Syntax:

```
$printtimescale [ ( hierarchical_name ) ] ;
```

This system task can be specified with or without an argument.

- When no argument is specified, **\$printtimescale** displays the time unit and precision of the module that is the current scope.
- When an argument is specified, **\$printtimescale** displays the time unit and precision of the module passed to it.

The timescale information appears in the following format:

```
Time scale of (module_name) is unit / precision
```


Example:

```

`timescale 1 ms / 1 us
module a_dat;
initial
    $printtimescale(b_dat.c1);
endmodule

`timescale 10 fs / 1 fs
module b_dat;
    c_dat c1 ();
endmodule

`timescale 1 ns / 1 ns
module c_dat;
    .
    .
    .
endmodule

```

In this example, module `a_dat` invokes the `$printtimescale` system task to display timescale information about another module `c_dat`, which is instantiated in module `b_dat`.

The information about `c_dat` is displayed in the following format:

```
Time scale of (b_dat.c1) is 1ns / 1ns
```

14.3.2 \$timeformat

Syntax:

```
$timeformat [ ( units_number , precision_number , suffix_string , minimum_field_width ) ] ;
```

The `$timeformat` system task performs the following two functions:

- It specifies how the `%t` format specification reports time information for the `$write`, `$display`, `$strobe`, `$monitor`, `$fwrite`, `$fdisplay`, `$fstrobe`, and `$fmonitor` group of system tasks.
- It specifies the time unit for delays entered interactively.

The units number argument shall be an integer in the range from 0 to -15. This argument represents the time unit as shown in Table 14-7.

Table 14-7—\$timeformat units_number arguments

Unit number	Time unit	Unit number	Time unit
0	1 s	-8	10 ns
-1	100 ms	-9	1 ns
-2	10 ms	-10	100 ps
-3	1 ms	-11	10 ps

Table 14-7—\$timeformat units_number arguments (continued)

Unit number	Time unit	Unit number	Time unit
-4	100 us	-12	1 ps
-5	10 us	-13	100 fs
-6	1 us	-14	10 fs
-7	100 ns	-15	1 fs

The **\$timeformat** system task performs the following two operations:

- It sets the time unit for all later-entered delays entered interactively.
- It sets the time unit, precision number, suffix string, and minimum field width for all **%t** formats specified in all modules that follow in the source description until another **\$timeformat** system task is invoked.

The default **\$timeformat** system task arguments are given in Table 14-8.

Table 14-8—\$timeformat default value for arguments

Argument	Default
units_number	The smallest time precision argument of all the `timescale compiler directives in the source description
precision_number	0
suffix_string	A null character string
minimum_field_width	20

Example:

The following example shows the use of **%t** with the **\$timeformat** system task to specify a uniform time unit, time precision, and format for timing information.

```

`timescale 1 ms / 1 ns
module cntrl;
initial
    $timeformat(-9, 5, " ns", 10);
endmodule

`timescale 1 fs / 1 fs
module a1_dat;
reg in1;
integer file;
buf #10000000 (o1,in1);
initial begin
    file = $fopen("a1.dat");
    #00000000 $fmonitor(file,"%m: %t in1=%d o1=%h", $realtime,in1,o1);
    #10000000 in1 = 0;
    #10000000 in1 = 1;
end
endmodule

`timescale 1 ps / 1 ps
module a2_dat;
reg in2;
integer file2;
buf #10000 (o2,in2);
initial begin
    file2=$fopen("a2.dat");
    #00000 $fmonitor(file2,"%m: %t in2=%d o2=%h", $realtime,in2,o2);
    #10000 in2 = 0;
    #10000 in2 = 1;
end
endmodule

```

The contents of file `a1.dat` are as follows:

```

a1_dat: 0.00000 ns in1= x o1=x
a1_dat: 10.00000 ns in1= 0 o1=x
a1_dat: 20.00000 ns in1= 1 o1=0
a1_dat: 30.00000 ns in1= 1 o1=1

```

The contents of file `a2.dat` are as follows:

```

a2_dat: 0.00000 ns in2=x o2=x
a2_dat: 10.00000 ns in2=0 o2=x
a2_dat: 20.00000 ns in2=1 o2=0
a2_dat: 30.00000 ns in2=1 o2=1

```

In this example, the times of events written to the files by the **\$fmonitor** system task in modules `a1_dat` and `a2_dat` are reported as multiples of 1 ns—even though the time units for these modules are 1 fs and 1 ps respectively—because the first argument of the **\$timeformat** system task is `-9` and the `%t` format specification is included in the arguments to **\$fmonitor**. This time information is reported after the module names with five fractional digits, followed by an “ns” character string in a space wide enough for 10 ASCII characters.

14.4 Simulation control system tasks

There are two simulation control system tasks:

- a) **\$finish**
- b) **\$stop**

14.4.1 \$finish

Syntax:

\$finish [(n)] ;

The **\$finish** system task simply makes the simulator exit and pass control back to the host operating system. If an expression is supplied to this task, then its value determines the diagnostic messages that are printed before the prompt is issued. If no argument is supplied, then a value of 1 is taken as the default.

Table 14-9—Diagnostic messages for \$stop and \$finish

Parameter value	Diagnostic message
0	Prints nothing
1	Prints simulation time and location
2	Prints simulation time, location, and statistics about the memory and CPU time used in simulation

14.4.2 \$stop

Syntax:

\$stop [(n)] ;

The **\$stop** system task causes simulation to be suspended. This task takes an optional expression argument (0, 1, or 2) that determines what type of diagnostic message is printed. The amount of diagnostic messages output increases with the value of the optional argument passed to **\$stop**.

14.5 Timing check system tasks

The timing check tasks may be invoked in specify blocks to verify the timing performance of a design by making sure critical events occur within given time limits.

Timing checks perform the following steps:

- a) Determine the elapsed time between two events.
- b) Compare the elapsed time to a specified limit.
- c) Report a timing violation (if the elapsed time does not fall within the specified time window).

The following system tasks may be used for performing timing checks:

\$setup	\$hold	\$setuphold	\$period
\$skew	\$recovery	\$width	\$nochange

Table 14-10 describes the arguments to these system tasks, which are explained with individual system task as well.

Table 14-10—Timing check arguments

Argument	Description	Type
reference_event	The transition at a control signal that establishes the reference time for tracking timing violations on the data_event	Module input or inout that is scalar or vector net
data_event	The signal change that initiates the timing check and is monitored for violations	Module input or inout that is scalar or vector net
limit	A time limit used to detect timing violations on the data_event	Constant expression or specparam
threshold	The largest pulse width that is ignored by the timing check \$width	Constant expression or specparam
setup_limit	A time limit used to detect timing violations on the data_event for \$setup	Constant expression or specparam
hold_limit	A time limit used to detect timing violations on the data_event for \$hold	Constant expression or specparam
notifier (optional)	An optional argument that “notifies” the simulator when a timing violation occurs	Register

14.5.1 \$setup

Syntax:

\$setup (data_event , reference_event , limit [, notifier]) ;

Table 14-11 defines the **\$setup** system task arguments.

Table 14-11—\$setup arguments

data_event	Lower bound event
reference_event	Upper bound event
limit	Positive constant expression or specparam
notifier (optional)	Register

The **\$setup** timing check reports a timing violation in the following case:

(time of reference event) - (time of data event) < limit

If the reference event and data event occur at the same simulation time, **\$setup** performs the timing check before it records the new data event value; therefore, no violation occurs.

14.5.2 \$hold

Syntax:

\$hold (reference_event , data_event , limit [, notifier]) ;

Table 14-12 defines the **\$hold** system task arguments.

Table 14-12—\$hold arguments

reference_event	Lower bound event
data_event	Upper bound event
limit	Positive constant expression or specparam
notifier (optional)	Register

\$hold system task reports a violation in the following case:

$$(\text{time of data event}) - (\text{time of reference event}) < \text{limit}$$

\$hold always records the new reference event time before it performs the timing check. Therefore, if reference and data events occur at the same simulation time, there will be a violation.

14.5.3 \$setuphold

Syntax:

```
$setuphold ( reference_event , data_event , setup_limit , hold_limit , [ notifier ] ) ;
```

Table 14-13 defines the **\$setuphold** system task arguments.

Table 14-13—\$setuphold arguments

reference_event	\$hold lower bound event \$setup upper bound event
data_event	\$hold upper bound event \$setup lower bound event
setup_limit	Constant expression or specparam
hold_limit	Constant expression or specparam
notifier (optional)	Register

Besides being a constant expression or a specparam, the setup limit and hold limit shall follow an additional restriction. Although individually each limit may be negative, the sum of the limits shall be positive. That is:

$$\text{setup_limit} + \text{hold_limit} > 0$$

The **\$setuphold** timing check is a shorthand way to combine the functionality of **\$setup** and **\$hold** into one system task call. Therefore, the following invocation:

```
$setuphold( posedge clk, data, tSU, tHLD );
```

is equivalent in functionality to the following, if tSU and tHLD are not negative:

```
$setup( data, posedge clk, tSU );  
$hold( posedge clk, data, tHLD );
```

14.5.4 \$width

Syntax:

\$width (reference_event , limit , threshold [, notifier]) ;

Table 14-14 defines the **\$width** system task arguments.

Table 14-14—\$width arguments

reference_event	Edge triggered event
limit	Positive constant expression or specparam
threshold (optional)	Positive constant expression or specparam
notifier (optional)	Register

The **\$width** timing check monitors the width of signal pulses by timing the duration of signal levels from one clock edge to the opposite clock edge. Since a data event is not passed to **\$width**, it is derived from the reference event, as follows:

data event = reference event signal with opposite edge

Because of the way the data event is derived for **\$width**, an edge triggered event has to be passed as the reference event. A compilation error will occur if the reference event is not an edge specification.

The **\$width** timing check reports a violation in the following case:

$\text{threshold} < (\text{time of data event}) - (\text{time of reference event}) < \text{limit}$

In other words, the pulse width has to be greater than or equal to limit in order to avoid a timing violation.

The data event and the reference event will never occur at the same simulation time because these events are triggered by opposite transitions.

The null arguments for **\$width** are not accepted. Therefore, the threshold argument shall be passed if the notifier argument is required. It is permissible, however, to drop both the threshold and notifier arguments when invoking **\$width**.

Example:

The following example demonstrates some examples of legal and illegal calls:

```
// Legal Calls
$width ( negedge clr, lim );
$width ( negedge clr, lim, thresh, notif );
$width ( negedge clr, lim, 0, notif );

// Illegal Calls
$width ( negedge clr, lim, , notif );
$width ( negedge clr, lim, notif );
```

14.5.5 \$period

Syntax:

\$period (reference_event , limit [, notifier]) ;

Table 14-15 defines the **\$period** system task arguments.

Table 14-15—: \$period arguments

reference_event	Edge triggered event
limit	Positive constant expression or specparam
notifier (optional)	Register

Since the data event is not passed as an argument to **\$period**, it is derived from the reference event, as follows:

data event = reference event signal with the same edge

Because of the way the data event is derived for **\$period**, an edge triggered event has to be passed as the reference event. A compilation error will occur if the reference event is not an edge specification.

The **\$period** timing check reports a violation in the following case:

(time of data event) - (time of reference event) < limit

14.5.6 \$skew

Syntax:

\$skew (reference_event , data_event , limit [, notifier]) ;

Table 14-16 defines the **\$skew** system task arguments.

Table 14-16—\$skew arguments

reference_event	Lower bound event
data_event	Upper bound event
limit	Positive constant expression or specparam
notifier (optional)	Register

The **\$skew** timing check reports a violation in the following case:

(time of data event) - (time of reference event) > limit

The **\$skew** timing check always records the new time of reference event before it performs the timing check. If the data event and the reference event occur at the same simulation time, **\$skew** does not report a timing violation.

14.5.7 \$recovery

Syntax:

\$recovery (reference_event , data_event , limit , [notifier]) ;

Table 14-17 defines the **\$recovery** system task arguments.

Table 14-17—\$recovery arguments

reference_event	Edge triggered event
data_event	Upper bound event
limit	Positive constant expression or specparam
notifier (optional)	Register

The reference event shall be specified as an edge triggered event using either the **posedge** or the **negedge** keyword. Not specifying an edge results in illegal specification of the reference event.

The **\$recovery** timing check system task reports a timing violation in the following case:

$$(\text{time of data event}) - (\text{time of reference event}) < \text{limit}$$

The **\$recovery** system task records the new reference event time before performing the timing check, so if a data event and a reference event occur at the same simulation time, a violation occurs.

14.5.8 \$nochange

Syntax:

\$nochange (reference_event , data_event , start_edge_offset , end_edge_offset [, notifier]) ;

Table 14-18 defines the **\$nochange** system task arguments.

Table 14-18—\$nochange arguments

reference_event	Edge triggered event
data_event	Upper bound event
start_edge_offset	Any constant expression or specparam
end_edge_offset	Any constant expression or specparam
notifier (optional)	Register

The **\$nochange** timing check system task reports a timing violation if the data event occurs during the specified level of the control signal (the reference event). The reference event may be specified with the **posedge** or the **negedge** keyword, but the edge control specifiers (see 14.5.9) can not be used.

The start edge and end edge offsets can expand or shrink the timing violation region, which is defined by the duration of the reference event signal edge. A positive offset for start edge extends the region by starting the timing violation region earlier; a negative offset for start edge shrinks the region by starting the region later. Similarly, a positive offset for the end edge extends the timing violation region by ending it later, while a negative offset for the end edge shrinks the region by ending it earlier. If both the offsets are zero, the size of the region will not change.

Example:

\$nochange(posedge clk, data, 0, 0) ;

In this example, **\$nochange** system task will report a violation if the **data** signal changes while **clk** is high.

14.5.9 Edge-control specifiers

The edge-control specifiers may be used to control events in timing checks based on specific edge transitions between 0, 1, and x.

```

edge_control_specifier ::= edge [ transition_pair { , transition_pair } ]

transition_pair ::= 01 | 0x | 10 | 1x | x0 | x1

```

Syntax 14-7—Syntax for edge control specifier

Edge-control specifiers contain the keyword **edge** followed by a square bracketed list of from one to six pairs of edge transitions between 0, 1 and x, as follows:

01	Transition from 0 to 1
0x	Transition from 0 to x
10	Transition from 1 to 0
1x	Transition from 1 to x
x0	Transition from x to 0
x1	Transition from x to 1

Edge transitions involving z are treated the same way as edge transitions involving x.

The **posedge** and **negedge** keywords may be used as a shorthand for certain edge-control specifiers. For example, the construct:

```
posedge clr
```

is equivalent to the following:

```
edge[ 01, 0x, x1 ] clr
```

Similarly, the construct:

```
negedge clr
```

is the same as the following:

```
edge[ 10, x0, 1x ] clr
```

However, edge-control specifiers offer the flexibility to declare edge transitions other than **posedge** and **negedge**.

14.5.10 Notifiers: user-defined responses to timing violations

Timing check notifiers detect timing check violations behaviorally, and, therefore, take an action as soon as a violation occurs. Such notifiers may be used to print an informative error message describing the violation or to propagate an x value at the output of the device that reported the violation.

The notifier is a register—declared in the module where timing check tasks are invoked—that is passed as the last argument to a system timing check. Whenever a timing violation occurs, the system task updates the value of the notifier.

The notifier is an optional argument to all system timing checks and can be omitted from the system task call without adversely affecting its operation.

Table 14-19 shows how the notifier values are toggled when timing violations occur.

Table 14-19—Notifier value changes

BEFORE violation	AFTER violation
x	0
0	1
1	0
z	z

Examples:

Example 1

```
$setup( data, posedge clk, 10, notify_reg ) ;
$width( posedge clk, 16, notify_reg ) ;
```

Example 2—Consider a more complex example of how to use notifiers in a behavioral model. The example that follows uses a notifier to set the D flip-flop output to x when a timing violation occurs in an edge-sensitive UDP.

```
primitive posdff_udp(q, clock, data, preset, clear, notifier);
output q; reg q;
input clock, data, preset, clear, notifier;
table
//clock data  p c notifier state q
//-----
r    0    1 1    ?    : ? : 0 ;
r    1    1 1    ?    : ? : 1 ;

p    1    ? 1    ?    : 1 : 1 ;
p    0    1 ?    ?    : 0 : 0 ;

n    ?    ? ?    ?    : ? : - ;
?    *    ? ?    ?    : ? : - ;

?    ?    0 1    ?    : ? : 1 ;
?    ?    * 1    ?    : 1 : 1 ;

?    ?    1 0    ?    : ? : 0 ;
?    ?    1 *    ?    : 0 : 0 ;
?    ?    ? ?    *    : ? : x ; // At any notifier event
                                   // output x
endtable
endprimitive
```

```

module dff(q, qbar, clock, data, preset, clear);
output q, qbar;
input clock, data, preset, clear;
reg notifier;

and (enable, preset,clear);
not (qbar, ffout);
buf (q, ffout);
posdff_udp (ffout, clock, data, preset, clear, notifier);

specify
  // Define timing check specparam values
  specparam tSU = 10, tHD = 1, tPW = 25, tWPC = 10, tREC = 5;
  // Define module path delay rise and fall min:typ:max values
  specparam tPLHc = 4:6:9 , tPHLc = 5:8:11;
  specparam tPLHpc = 3:5:6 , tPHLpc = 4:7:9;

  // Specify module path delays
  (clock *> q,qbar) = (tPLHc, tPHLc);
  (preset,clear *> q,qbar) = (tPLHpc, tPHLpc);

  // Setup time : data to clock, only when preset and clear are 1
  $setup(data, posedge clock &&& enable, tSU, notifier);

  // Hold time:clock to data,only when preset and clear are 1
  $hold(posedge clock, data &&& enable, tHD, notifier);

  // Clock period check
  $period(posedge clock, tPW, notifier);
  // Pulse width : preset, clear
  $width(negedge preset, tWPC, 0, notifier);
  $width(negedge clear, tWPC, 0, notifier);

  // Recovery time: clear or preset to clock
  $recovery(posedge preset, posedge clock, tREC, notifier);
  $recovery(posedge clear, posedge clock, tREC, notifier);
endspecify
endmodule

```

NOTE—This model applies to edge-sensitive UDPs only; for level-sensitive models, an additional UDP for **x** propagation has to be generated.

14.5.11 Enabling timing checks with conditioned events

A construct called a conditioned event ties the occurrence of timing checks to the value of a conditioning signal.

```
controlled_timing_check_event ::=  
    timing_check_event_control specify_terminal_descriptor [ &&&  
        timing_check_condition ]  
timing_check_condition ::=  
    scalar_expression  
    | ~scalar_expression  
    | scalar_expression == scalar_constant  
    | scalar_expression === scalar_constant  
    | scalar_expression != scalar_constant  
    | scalar_expression !== scalar_constant
```

Syntax 14-8—Syntax for controlled timing check event

The comparisons used in the condition may be deterministic, as in `===`, `!==`, `~`, or no operation, or nondeterministic, as in `==` or `!=`. When comparisons are deterministic, an `x` value on the conditioning signal will not enable the timing check. For nondeterministic comparisons, an `x` on the conditioning signal will enable the timing check.

The conditioning signal shall be a scalar net; if a vector net or an expression resulting in a multi-bit value is used, then the least significant bit of the vector net or the expression value is used.

If more than one conditioning signal is required for conditioning timing checks, appropriate logic shall be combined in a separate signal outside the `specify` block, which may be used as the conditioning signal.

Examples:

Example 1—To illustrate the difference between conditioned and unconditioned timing check events, consider the following example with unconditioned timing check:

```
$setup( data, posedge clk, 10 );
```

Here, a setup timing check will occur every time there is a positive edge on the signal `clk`.

To trigger the setup check on the positive edge on the signal `clk` only when the signal `clr` is high, rewrite the command as:

```
$setup( data, posedge clk &&& clr, 10 );
```

Example 2—This example shows two ways to trigger the same timing check as in example 1 (on the positive `clk` edge) only when `clr` is low. The second method uses the `===` operator, which makes the comparison deterministic.

```
$setup( data, posedge clk &&& (~clr), 10 );  
$setup( data, posedge clk &&& (clr===0), 10 );
```

Example 3—To perform the previous sample setup check on the positive `clk` edge only when `clr` and `set` are high, add the following statement outside the `specify` block:

```
and new_gate( clr_and_set, clr, set );
```

Then add the condition to the timing check using the signal `clr_and_set` as follows:

```
$setup( data, posedge clk &&& clr_and_set, 10 );
```

14.6 PLA modeling system tasks

The modeling of PLA devices is provided in the Verilog HDL by a group of system tasks. This clause describes the syntax and use of these system tasks and the formats of the logic array personality file.

```

pla_system_task ::= $ array_type $ logic $ format ( memory_name , input_terms ,
               output_terms ) ;
array_type ::= sync | async
logic ::= and | or | nand | nor
format ::= array | plane
memory_name ::= memory_identifier
input_terms ::= { scalar_variables }
output_terms ::= { scalar_variables }
scalar_variables ::= scalar_variable { , scalar_variable }

```

Syntax 14-9—Syntax for PLA modeling system task

The PLA syntax allows for the system tasks as shown in Table 14-20.

Table 14-20—PLA system tasks

\$async\$and\$array	\$sync\$and\$array	\$async\$and\$plane	\$sync\$and\$plane
\$async\$nand\$array	\$sync\$nand\$array	\$async\$nand\$plane	\$sync\$nand\$plane
\$async\$or\$array	\$sync\$or\$array	\$async\$or\$plane	\$sync\$or\$plane
\$async\$nor\$array	\$sync\$nor\$array	\$async\$nor\$plane	\$sync\$nor\$plane

14.6.1 Array types

The modeling of both synchronous and asynchronous arrays is provided by the PLA system tasks. The synchronous forms control the time at which the logic array will be evaluated and the outputs will be updated. For the asynchronous forms, the evaluations are automatically performed whenever an input term changes value or any word in the personality memory is changed.

For both the synchronous and asynchronous forms, the output terms are updated without any delay.

Examples:

An example of an asynchronous system call is as follows:

```
$async$and$array ( mem , { a1 , a2 , a3 , a4 , a5 , a6 , a7 } , { b1 , b2 , b3 } ) ;
```

An example of a synchronous system call is as follows:

```
$sync$or$plane ( mem , { a1 , a2 , a3 , a4 , a5 , a6 , a7 } , { b1 , b2 , b3 } ) ;
```

Note that the input terms and the output terms are always represented as concatenations.

14.6.2 Array logic types

The logic arrays are modeled with and, or, nand, and nor logic planes. This applies to all array types and formats.

Examples:

An example of a nor plane system call is as follows:

```
$async$nor$array (mem, {a1,a2,a3,a4,a5,a6,a7},{b1,b2,b3});
```

An example of a nand plane system call is as follows:

```
$sync$nand$plane (mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
```

14.6.3 Logic array personality declaration and loading

The logic array personality is declared as an array of registers that is as wide as the number of input terms and as deep as the number of output terms.

The personality of the logic array is normally loaded into the memory from a text data file using the system tasks **\$readmemb** or **\$readmemh**. Alternatively, the personality data may be written directly into the memory using the procedural assignment statements. PLA personalities may be changed dynamically at any time during simulation simply by changing the contents of the memory. The new personality will be reflected on the outputs of the logic array at the next evaluation.

Example:

The following example shows a logic array with *n* input terms and *m* output terms.

```
reg [1:n] mem[1:m];
```

NOTE—Put PLA input terms, output terms, and memory in ascending order, as shown in examples in this clause.

14.6.4 Logic array personality formats

Two separate personality formats are supported by the Verilog HDL and are differentiated by using either an array system call or a plane system call. The array system call allows for a 1 or 0 in the memory that has been declared. A 1 means take the input value and a 0 means do not take the input value.

The plane system call complies with the University of California at Berkeley format for Espresso. Each bit of the data stored in the array has the following meaning:

0	Take the complemented input value
1	Take the true input value
x	Take the “worst case” of the input value
z	Don’t-care; the input value is of no significance
?	Same as z

Examples:

Example 1—The following example illustrates an array with logic equations:

```
b1 = a1 & a2
b2 = a3 & a4 & a5
b3 = a5 & a6 & a7
```

The PLA personality is as follows:

```

1100000 in mem[1]
0011100 in mem[2]
0000111 in mem[3]

```

The module for the PLA is as follows:

```

module async_array(a1,a2,a3,a4,a5,a6,a7,b1,b2,b3);
input a1, a2, a3, a4, a5, a6, a7 ;
output b1, b2, b3;
reg [1:7] mem[1:3]; // memory declaration for array personality
reg b1, b2, b3;
initial begin
    // setup the personality from the file array.dat
    $readmemb("array.dat", mem);
    // setup an asynchronous logic array with the input
    // and output terms expressed as concatenations
    $asynb($and$array(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3}));
end
endmodule

```

Where the file `array.dat` contains the binary data for the PLA personality:

```

1100000
0011100
0000111

```

Example 2—An example of the usage of the plane format tasks follows. The logical function of this PLA is shown first, followed by the PLA personality in the new format, the Verilog HDL description using the `$asynb` and `$plane` system tasks, and finally the result of running the simulation.

The logical function of the PLA is as follows:

```

b[1] = a[1] & ~a[2];
b[2] = a[3];
b[3] = ~a[1] & ~a[3];
b[4] = 1;

```

The PLA personality is as follows:

```

3'b10?
3'b??1
3'b0?0
3'b???

```



```

module pla;
`define rows 4
`define cols 3
reg [1:`cols] a, mem[1:`rows];
reg [1:`rows] b;
initial begin
    // PLA system call
    $async$and$plane(mem, {a[1],a[2],a[3]}, {b[1],b[2],b[3],b[4]});
    mem[1] = 3'b10?;
    mem[2] = 3'b??1;
    mem[3] = 3'b0?0;
    mem[4] = 3'b???;
    // stimulus and display
    #10 a = 3'b111;
    #10 $displayb(a, " -> ", b);
    #10 a = 3'b000;
    #10 $displayb(a, " -> ", b);
    #10 a = 3'bxxx;
    #10 $displayb(a, " -> ", b);
    #10 a = 3'b101;
    #10 $displayb(a, " -> ", b);
end
endmodule

```

The output is as follows:

```

111 -> 0101
000 -> 0011
xxx -> xxx1
101 -> 1101

```

14.7 Stochastic analysis tasks

This clause describes a set of system tasks and functions that manage queues and generate random numbers with specific distributions. These tasks facilitate implementation of stochastic queueing models.

The set of tasks and functions that create and manage queues follow:

```

$q_initialize (q_id, q_type, max_length, status) ;
$q_add (q_id, job_id, inform_id, status) ;
$q_remove (q_id, job_id, inform_id, status) ;
$q_full (q_id, status) ;
$q_exam (q_id, q_stat_code, q_stat_value, status) ;

```

14.7.1 \$q_initialize

The **\$q_initialize** system task creates new queues. The `q_id` parameter is an integer input that shall uniquely identify the new queue. The `q_type` parameter is an integer input. The value of the `q_type` parameter specifies the type of the queue as shown in Table 14-21.

Table 14-21 — `q_type` parameter value

<code>q_type</code> value	Type of queue
1	first-in, first-out
2	last-in, first-out

The maximum length parameter is an integer input that specifies the maximum number of entries that will be allowed on the queue. The success or failure of the creation of the queue is returned as an integer value in `status`. The error conditions and corresponding values of `status` are described in Table 14-23.

14.7.2 \$q_add

The **\$q_add** system task places an entry on a queue. The `q_id` parameter is an integer input that indicates to which queue to add the entry. The `job_id` parameter is an integer input that identifies the job.

The `inform_id` parameter is an integer input that is associated with the queue entry. Its meaning is user-defined. For example, `inform_id` parameter can represent execution time for an entry in a CPU model. The `status` parameter reports on the success of the operation or error conditions as described in Table 14-23.

14.7.3 \$q_remove

The **\$q_remove** system task receives an entry from a queue. The `q_id` parameter is an integer input that indicates from which queue to remove. The `job_id` parameter is an integer output that identifies the entry being removed. The `inform_id` parameter is an integer output that the queue manager stored during **\$q_add**. Its meaning is user-defined. The `status` parameter reports on the success of the operation or error conditions as described in Table 14-23.

14.7.4 \$q_full

The **\$q_full** system function checks whether there is room for another entry on a queue. It returns 0 when the queue is not full and 1 when the queue is full.

14.7.5 \$q_exam

The **\$q_exam** system task provides statistical information about activity at the queue `q_id`. It returns a value in `q_stat_value` depending on the information requested in `q_stat_code`. The values of `q_stat_code` and the corresponding information returned in `q_stat_value` are described in Table 14-22.

Table 14-22 — The information received in `q_stat_value`

Value requested in <code>q_stat_code</code>	Information received back from <code>q_stat_value</code>
1	Current queue length
2	Mean interarrival time
3	Maximum queue length

Table 14-22—The information received in q_stat_value (continued)

Value requested in q_stat_code	Information received back from q_stat_value
4	Shortest wait time ever
5	Longest wait time for jobs still in the queue
6	Average wait time in the queue

14.7.6 Status codes

All of the queue management tasks and functions return an output status parameter. The status parameter values and corresponding information are described in Table 14-23.

Table 14-23—The status parameter values and corresponding information

Status parameter Values	What it means
0	OK
1	Queue full, cannot add
2	Undefined q_id
3	Queue empty, cannot remove
4	Unsupported queue type, cannot create queue
5	Specified length ≤ 0 , cannot create queue
6	Duplicate q_id, cannot create queue
7	Not enough memory, cannot create queue

14.8 Simulation time system functions

The following system functions provide access to current simulation time:

\$time **\$stime** **\$realtime**

14.8.1 \$time

Syntax:

integer \$time

The **\$time** system function returns an integer that is a 64-bit time, scaled to the timescale unit of the module that invoked it.

Example:

```

`timescale 10 ns / 1 ns
module test;
reg set;
parameter p = 1.55;
initial begin
    $monitor($time, , "set=", set);
    #p set = 0;
    #p set = 1;
end
endmodule

// The output from this example is as follows:
// 0 set=x
// 2 set=0
// 3 set=1

```

In this example, the reg `set` is assigned the value 0 at simulation time 16 ns, and the value 1 at simulation time 32 ns. Note that these times do not match the times reported by `$time`. The time values returned by the `$time` system function are determined by the following steps:

- a) The simulation times 16ns and 32 ns are scaled to 1.6 and 3.2 because the time unit for the module is 10 ns, so time values reported by this module are multiples of 10 ns.
- b) The value 1.6 is rounded to 2, and 3.2 is rounded to 3 because the `$time` system function returns an integer. The time precision does not cause rounding of these values.

14.8.2 \$time

Syntax:

integer \$time

The `$time` system function returns an unsigned integer that is a 32-bit time, scaled to the timescale unit of the module that invoked it. If the actual simulation time does not fit in 32 bits, the low order 32 bits of the current simulation time are returned.

14.8.3 \$realtime

Syntax:

real \$realtime

The `$realtime` system function returns a real number time that, like `$time`, is scaled to the time unit of the module that invoked it.

Example:

```

`timescale 10 ns / 1 ns
module test;
reg set;
parameter p = 1.55;
initial begin
    $monitor($realtime, , "set=", set);
    #p set = 0;
    #p set = 1;
end
endmodule

// The output from this example is as follows:
// 0 set=x
// 1.6 set=0
// 3.2 set=1

```

In this example, the event times in the register `set` are multiples of 10 ns because 10 ns is the time unit of the module. They are real numbers because `$realtime` returns a real number.

14.9 Conversion functions for reals

The following functions handle **real** values:

Syntax:

integer	<code>\$rtol(real_val) ;</code>
real	<code>\$itor(int_val) ;</code>
<code>[63:0]</code>	<code>\$realtobits(real_val) ;</code>
real	<code>\$bitstoreal(bit_val) ;</code>

\$rtol converts real values to integers by truncating the real value (for example, 123.45 becomes 123)

\$itor converts integers to real values (for example, 123 becomes 123.0)

\$realtobits passes bit patterns across module ports; converts from a real number to the 64-bit representation (vector) of that real number

\$bitstoreal is the reverse of **\$realtobits**; converts from the bit pattern to a real number.

The real numbers accepted or generated by these functions shall conform to the IEEE Std 754-1985 [B1] representation of the real number. The conversion shall round the result to the nearest valid representation.

Example:

The following example shows how the **\$realtobits** and **\$bitstoreal** functions are used in port connections.

```

module driver (net_r);
output net_r;
real r;
wire [64:1] net_r = $realtobits(r);
endmodule

module receiver (net_r);
input net_r;
wire [64:1] net_r;
real r;
initial assign r = $bitstoreal(net_r);
endmodule

```

14.10 Probabilistic distribution functions

There are a set of random number generators that return integer values distributed according to standard probabilistic functions.

14.10.1 \$random function

Syntax:

```
$random [ ( seed ) ] ;
```

The system function **\$random** provides a mechanism for generating random numbers. The function returns a new 32-bit random number each time it is called. The random number is a signed integer; it can be positive or negative. For further information on probabilistic random number generators, see 14.10.2.

The **seed** parameter controls the numbers that **\$random** returns. The **seed** parameter shall be either a register, an integer, or a time variable. The seed value should be assigned to this variable prior to calling **\$random**.

Examples:

Example 1—Where **b** is greater than 0, the expression **(\$random % b)** gives a number in the following range: **[(-b+1) : (b-1)]**. The following code fragment shows an example of random number generation between -59 and 59:

```

reg [23:0] rand;
rand = $random % 60;

```

Example 2—The following example shows how adding the concatenation operator to the preceding example gives **rand** a positive value from 0 to 59.

```

reg [23:0] rand;
rand = { $random } % 60;

```

14.10.2 \$dist_ functions

```
$dist_uniform (seed, start, end) ;  
$dist_normal (seed, mean, standard_deviation) ;  
$dist_exponential (seed, mean) ;  
$dist_poisson (seed, mean) ;  
$dist_chi_square (seed, degree_of_freedom) ;  
$dist_t (seed, degree_of_freedom) ;  
$dist_erlang (seed, k_stage, mean) ;
```

Syntax 14-10—Syntax for the probabilistic distribution functions

All parameters to the system functions are integer values. For the **exponential**, **poisson**, **chi-square**, **t**, and **erlang** functions, the parameters **mean**, **degree of freedom**, and **k_stage** shall be greater than 0.

Each of these functions returns a pseudo-random number whose characteristics are described by the function name. That is, **\$dist_uniform** returns random numbers uniformly distributed in the interval specified by its parameters.

For each system function, the **seed** parameter is an in-out parameter; that is, a value is passed to the function and a different value is returned. The system functions will always return the same value given the same seed. This facilitates debugging by making the operation of the system repeatable. The argument for the **seed** parameter should be an integer variable that is initialized by the user and only updated by the system function. This will ensure that the desired distribution is achieved.

In the **\$dist_uniform** function, the **start** and **end** parameters are integer inputs that bound the values returned. The **start** value should be smaller than the **end** value.

The **mean** parameter, used by **\$dist_normal**, **\$dist_exponential**, **\$dist_poisson**, and **\$dist_erlang**, is an integer input that causes the average value returned by the function to approach the value specified.

The **standard deviation** parameter used with the **\$dist_normal** function is an integer input that helps determine the shape of the density function. Larger numbers for **standard deviation** will spread the returned values over a wider range.

The **degree of freedom** parameter used with the **\$dist_chi_square** and **\$dist_t** functions is an integer input that helps determine the shape of the density function. Larger numbers will spread the returned values over a wider range.

Section 15

Value change dump (VCD) file

A *value change dump (VCD) file* contains information about value changes on selected variables in the design stored by value change dump system tasks.

This section describes how to generate a VCD file and its format.

15.1 Creating the value change dump file

The steps involved in creating the VCD file are listed below and illustrated in Figure 15-1.

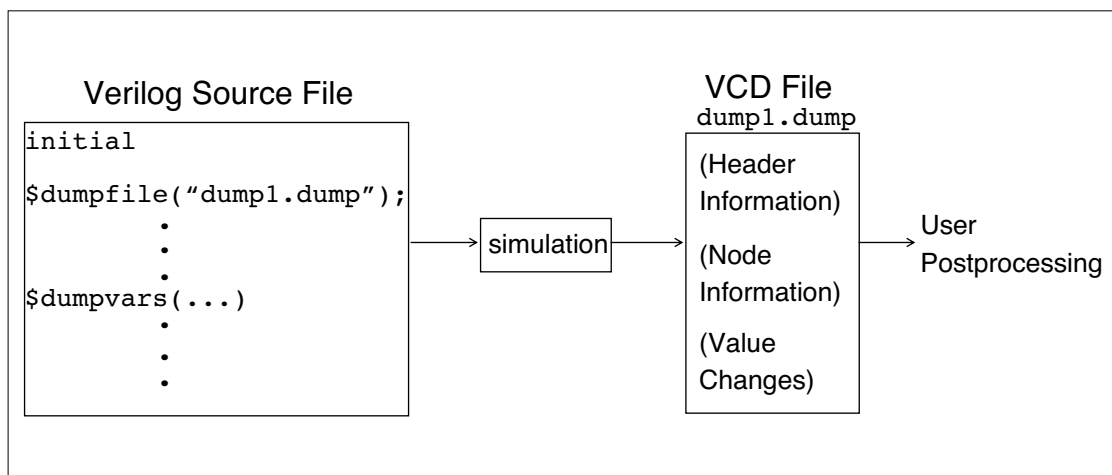


Figure 15-1—Creating the VCD file

- a) Insert the VCD system tasks in the Verilog source file to define the dump file name and to specify the variables to be dumped.
- b) Run the simulation.

A VCD file is an ASCII file that contains header information, variable definitions, and the value changes for all variables specified in the task calls.

Several system tasks can be inserted in the source description to create and control the VCD file.

15.1.1 Specifying the name of the dump file (\$dumpfile)

The **\$dumpfile** task shall be used to specify the name of the VCD file. The syntax for the task is as follows:

```
$dumpfile ( filename ) ;
```

The *filename* is optional.

Example:

```
initial $dumpfile ( "module1.dump" ) ;
```

15.1.2 Specifying the variables to be dumped (\$dumpvars)

The **\$dumpvars** task shall be used to list which variables to dump into the file specified by **\$dumpfile**. The **\$dumpvars** task may be invoked as often as desired throughout the model (for example, within various blocks), but the execution of all the **\$dumpvars** tasks shall be at the same simulation time.

The **\$dumpvars** task may be used with or without arguments. The syntax for the task without arguments is as follows:

```
$dumpvars ;
```

When invoked with no arguments, **\$dumpvars** task dumps all the variables in the model to the VCD file.

The syntax for the task with arguments is given in Syntax 15-1.

```
$dumpvars ( levels [ , list_of_modules_or_variables ] );  
list_of_modules_or_variables ::= module_or_variable { , module_or_variable }  
module_or_variable ::= module_identifier | variable_identifier
```

Syntax 15-1—Syntax for \$dumpvars system task

When the **\$dumpvars** task is specified with arguments, the first argument indicates how many *levels* of the hierarchy below each specified module instance to dump to the VCD file. Subsequent arguments specify which scopes of the model to dump to the VCD file. These arguments may specify entire modules or individual variables within a module.

Setting the first argument to 0 causes a dump of all variables in the specified module and in all module instances below the specified module. The argument 0 applies only to subsequent arguments that specify module instances, and not to individual variables.

Examples:

Example 1

```
$dumpvars (1, top);
```

Because the first argument is a 1, this invocation dumps all variables within the module **top**; it does not dump variables in any of the modules instantiated by module **top**.

Example 2

```
$dumpvars (0, top);
```

In this example, the **\$dumpvars** task will dump all variables in the module **top** and in all module instances below module **top** in the hierarchy.

Example 3

This example shows how the **\$dumpvars** task can specify both modules and individual variables:

```
$dumpvars (0, top.mod1, top.mod2.net1);
```

This call will dump all variables in module `mod1` and in all module instances below `mod1`, along with variable `net1` in module `mod2`. The argument 0 applies only to the module instance `top.mod1` and not to the individual variable `top.mod2.net1`.

15.1.3 Stopping and resuming the dump (\$dumpoff/\$dumpon)

Executing the **\$dumpvars** task causes the value change dumping to start at the end of the current simulation time unit. To suspend the dump, the **\$dumpoff** task may be invoked. To resume the dump, the **\$dumpon** task may be invoked. The syntax of these two tasks is as follows:

```
$dumpoff ;  
$dumpon ;
```

When the **\$dumpoff** task is executed, a checkpoint is made in which every selected variable is dumped as an **x** value. When the **\$dumpon** task is later executed, each variable is dumped with its value at that time. In the interval between **\$dumpoff** and **\$dumpon**, no value changes are dumped.

The **\$dumpoff** and **\$dumpon** tasks provide the mechanism to control the simulation period during which the dump will take place.

Example:

```
initial begin  
  #10    $dumpvars( . . . );  
  
  #200   $dumpoff;  
  
  #800   $dumpon;  
  
  #900   $dumpoff;  
end
```

This example starts the value change dump after 10 time units, stops it 200 time units later (at time 210), restarts it again 800 time units later (at time 1010), and stops it 900 time units later (at time 1910).

15.1.4 Generating a checkpoint (\$dumpall)

The **\$dumpall** task creates a checkpoint in the VCD file that shows the current value of all selected variables. The syntax is as follows:

```
$dumpall ;
```

When dumping is enabled, the value change dumper records the values of the variables that change during each time increment. Values of variables that do not change during a time increment are not dumped.

15.1.5 Limiting the size of the dump file (\$dumplimit)

The **\$dumplimit** task may be used to set the size of the VCD file. The syntax for this task is as follows:

```
$dumplimit ( filesize ) ;
```

The *filesize* argument that specifies the maximum size of the VCD file in bytes. When the size of VCD file reaches this number of bytes, the dumping stops and a comment is inserted in the VCD file indicating that the dump limit was reached.

15.1.6 Reading the dump file during simulation (\$dumpflush)

The **\$dumpflush** task may be used to empty the VCD file buffer of the operating system to ensure that all the data in that buffer is stored in the VCD file. After executing a **\$dumpflush** task, dumping is resumed as before so that no value changes are lost.

The syntax is as follows:

\$dumpflush ;

A common application is to call **\$dumpflush** to update the dump file so that an application program can read the VCD file during a simulation.

Examples:

Example 1—This example shows how the **\$dumpflush** task may be used in a Verilog HDL source file:

```
initial begin
    $dumpvars ;
    .
    .
    .

    $dumpflush ;

    $(applications program) ;

end
```

Example 2—The following is a simple source description example to produce a VCD file.

In this example, the name of the dump file is `verilog.dump`. It dumps value changes for all variables in the model. Dumping begins when an event `do_dump` occurs. The dumping continues for 500 clock cycles, then stops and waits for the event `do_dump` to be triggered again. At every 10000 time steps, the current values of all VCD variables are dumped.

```
module dump;
  event do_dump;

  initial $dumpfile("verilog.dump");
  initial @do_dump
    $dumpvars;          //dump variables in the design

  always @do_dump      //to begin the dump at event do_dump
  begin
    $dumpon;           //no effect the first time through
    repeat (500) @(posedge clock); //dump for 500 cycles
    $dumpoff;          //stop the dump
  end

  initial @(do_dump)
    forever #10000 $dumpall; //checkpoint all variables
endmodule
```

15.2 Format of the VCD file

The dump file is structured in a free format. White space is used to separate commands and to make the file easily readable by a text editor.

15.2.1 Syntax of the VCD file

The syntax of the VCD file is given in Syntax 15-2.

```

value_change_dump_definitions ::=
    { declaration_command } { simulation_command }
declaration_command ::= declaration_keyword [ command_text ] $end
simulation_command ::=
    simulation_keyword { value_change } $end
    | $comment [ comment_text ] $end
    | simulation_time
    | value_change
declaration_keyword ::=
    $comment | $date | $enddefinitions | $scope | $timescale | $upscope | $var |
    $version
simulation_keyword ::=
    $dumpall | $dumpoff | $dumpon | $dumpvars
simulation_time ::= # decimal_number
value_change ::=
    scalar_value_change
    | vector_value_change
scalar_value_change ::=
    value identifier_code
value ::= 0 | 1 | x | X | z | Z
vector_value_change ::=
    b binary_number identifier_code
    | B binary_number identifier_code
    | r real_number identifier_code
    | R real_number identifier_code

```

Syntax 15-2—Syntax of the output VCD file

The VCD file starts with header information giving the date, the version number of the simulator used for the simulation, and the timescale used. Next, the file contains definitions of the scope and type of variables being dumped, followed by the actual value changes at each simulation time increment. Only the variables that change value during a time increment are listed.

The simulation time recorded in VCD file is the absolute value of the simulation time for the changes in variable values that follow.

Value changes for real variables are specified by real numbers. Value changes for all other variables are specified in binary format by 0, 1, x, or z values. Strength information and memories are not dumped.

A real number is dumped using a `%.16g printf()` format. This preserves the precision of that number by outputting all 53 bits in the mantissa of a 64-bit IEEE Std 754-1985 [B1] double-precision number. Application programs can read a real number using a `%g` format to `scanf()`.

The value change dumper generates character identifier codes to represent variables. The identifier code is a code composed of the printable characters that are in the ASCII character set from ! to ~ (decimal 33 to 126).

NOTES

1—The VCD format does not support a mechanism to dump *part* of a vector. For example, bits 8 to 15 (8:15) of a 16-bit vector cannot be dumped in VCD file; instead, the entire vector (0:15) has to be dumped. In addition, expressions, such as `a + b`, cannot be dumped in the VCD file.

2— Data in the VCD file is case sensitive.

15.2.2 Formats of variable values

Variables may be either scalars or vectors. Each type is dumped in its own format. Dumps of value changes to scalar variables shall not have any white space between the value and the identifier code.

Dumps of value changes to vectors shall not have any white space between the base letter and the value digits, but they shall have one white space between the value digits and the identifier code.

The output format for each value is right-justified. Vector values appear in the shortest form possible: redundant bit values that result from left-extending values to fill a particular vector size are eliminated.

The rules for left-extending vector values are given in Table 15-1.

Table 15-1 — Rules for left-extending vector values

When the value is	VCD left-extends with
1	0
0	0
Z	Z
X	X

Table 15-2 shows how the VCD may shorten values.

Table 15-2 — How the VCD may shorten values

The binary value	Extends to fill a 4-bit register as	Appears in the VCD file as
10	0010	b10
X10	XX10	bX10
ZX0	ZZX0	bZX0
0X10	0X10	b0X10

Events are dumped in the same format as scalars; for example, 1*%. For events, however, the value (1 in this example) is irrelevant. Only the identifier code (*% in this example) is significant. It appears in the VCD file as a marker to indicate that the event was triggered during the time step.

Examples:

1*% No space between the value 1 and the identifier code *%

b1100x01z (k No space between the b and 1100x01z,
but a space between b1100x01z and (k

15.2.3 Description of keyword commands

The general information in the VCD file is presented as a series of sections surrounded by keywords. Keyword commands provide a means of inserting information in the VCD file. Keyword commands can be inserted either by the dumper or manually.

This subclause deals with the keyword commands given in Table 15-3.

Table 15-3—Keyword commands

Declaration keywords		Simulation keywords
\$comment	\$timescale	\$dumpall
\$date	\$upscope	\$dumpoff
\$enddefinitions	\$var	\$dumpon
\$scope	\$version	\$dumpvars

15.2.3.1 \$comment

The **\$comment** section provides a means of inserting a comment in the VCD file.

Syntax:

```
$comment comment_text $end
```

Examples:

```
$comment This is a single-line comment      $end  
$comment This is a  
multiple-line comment  
$end
```

15.2.3.2 \$date

The **\$date** section indicates the date on which the VCD file was generated.

Syntax:

```
$date date_text $end
```

Example:

```
$date  
June 25, 1989 09:24:35  
$end
```

15.2.3.3 \$enddefinitions

The **\$enddefinitions** section marks the end of the header information and definitions.

Syntax:

```
$enddefinitions $end
```

15.2.3.4 \$scope

The **\$scope** section defines the scope of the variables being dumped.

Syntax:

```
$scope scope_type scope_identifier $end
scope_type ::= module | task | function | begin | fork
```

The scope type indicates one of the following scopes:

<i>module</i>	Top-level module and module instances
<i>task</i>	Tasks
<i>function</i>	Functions
<i>begin</i>	Named sequential blocks
<i>fork</i>	Named parallel blocks

Example:

```
$scope
    module top
$end
```

15.2.3.5 \$timescale

The **\$timescale** keyword specifies what timescale was used for the simulation.

Syntax:

```
$timescale number time_unit $end
number ::= 1 | 10 | 100
time_unit ::= s | ms | us | ns | ps | fs
```

Example:

```
$timescale 10 ns $end
```

15.2.3.6 \$upscope

The **\$upscope** section indicates a change of scope to the next higher level in the design hierarchy.

Syntax:

```
$upscope $end
```

15.2.3.7 \$version

The **\$version** section indicates which version of the VCD writer was used to produce the VCD file.

Syntax:

```
$version version_text $end
```


Example:

```
$version
    VERILOG-SIMULATOR 1.0a
$end
```

15.2.3.8 \$var

The **\$var** section prints the names and identifier codes of the variables being dumped.

Syntax:

```
$var var_type size identifier_code reference $end

var_type ::=
    event | integer | parameter | real | reg | supply0 | supply1 | time
    | tri | triand | trior | trireg | tri0 | tri1 | wand | wire | wor
size ::= decimal_number
reference ::=
    identifier
    | identifier [ bit_select_index ]
    | identifier [ msb_index : lsb_index ]
index ::= decimal_number
```

Size specifies how many bits are in the variable.

The identifier code specifies the name of the variable using printable ASCII characters, as previously described.

- a) The msb index indicates the most significant index; the lsb index indicates the least significant index.
- b) More than one reference name may be mapped to the same identifier code. For example, net10 and net15 may be interconnected in the circuit and therefore may have the same identifier code.
- c) The individual bits of vector nets may be dumped individually.
- d) The identifier is the name of the variable being dumped in the model.

Example:

```
$var
    integer 32 (2 index
$end
```

15.2.3.9 \$dumpall

The **\$dumpall** keyword specifies current values of all variables dumped.

Syntax:

```
$dumpall { value_changes } $end
```

Example:

```
$dumpall 1*@ x*# 0*$ bx (k $end
```

15.2.3.10 \$dumpoff

The **\$dumpoff** keyword indicates all variables dumped with X values.

Syntax:

```
$dumpoff { value_changes } $end
```

Example:

```
$dumpoff x*@ x*# x*$ bx (k $end
```

15.2.3.11 \$dumpon

The **\$dumpon** keyword indicates resumption of dumping and lists current values of all variables dumped.

Syntax:

```
$dumpon { value_changes } $end
```

Example:

```
$dumpon x*@ 0*# x*$ b1 (k $end
```

15.2.3.12 \$dumpvars

The section beginning with **\$dumpvars** keyword lists initial values of all variables dumped.

Syntax:

```
$dumpvars { value_changes } $end
```

Example:

```
$dumpvars x*@ z*$ b0 (k $end
```

15.2.4 VCD file format example

The following example illustrates the format of the VCD file.

```

$date
    June 26, 1989 10:05:41
$end
$version
    VERILOG-SIMULATOR 1.0a
$end
$timescale
    1 ns
$end
$scope module top $end
$scope module m1 $end
$var trireg 1 *@ net1 $end
$var trireg 1 *# net2 $end
$var trireg 1 *$ net3 $end
$upscope $end
$scope task t1 $end
$var reg 32 (k accumulator[31:0] $end
$var integer 32 {2 index $end
$upscope $end
$upscope $end
$enddefinitions $end
$comment
    Note: $dumpvars was executed at time '#500'.
    All initial values are dumped at this time.
$end

#500
$dumpvars
x*@
x*#
x*$
bx (k
bx {2
$end
#505
0*@
1*#
1*$
b10zx1110x11100 (k
b1111000101z01x {2
#510
0*$
#520
1*$
#530
0*$
bz (k
#535
$dumpall    0*@    1*#    0*$

( Continued in right column )

```

(Continued from left column)

```

bz (k
b1111000101z01x {2
$end
#540
1*$
#1000
$dumpoff
x*@
x*#
x*$
bx (k
bx {2
$end
#2000
$dumpon
z*@
1*#
0*$
b0 (k
bx {2
$end
#2010
1*$

```

Section 16

Compiler directives

All Verilog compiler directives are preceded by the (`) character. This character is called accent grave. It is different from the character ('), which is the single quote character. The scope of compiler directives extends from the point where it is processed, across all files processed, to the point where another compiler directive supersedes it or the processing completes.

This section describes the following compiler directives:

<code>`celldefine</code>	[16.1]
<code>`default_nettype</code>	[16.2]
<code>`define</code>	[16.3]
<code>`else</code>	[16.4]
<code>`endcelldefine</code>	[16.1]
<code>`endif</code>	[16.4]
<code>`ifdef</code>	[16.4]
<code>`include</code>	[16.5]
<code>`nounconnected_drive</code>	[16.8]
<code>`resetall</code>	[16.6]
<code>`timescale</code>	[16.7]
<code>`unconnected_drive</code>	[16.8]
<code>`undef</code>	[16.3]

16.1 ``celldefine` and ``endcelldefine`

The directives ``celldefine` and ``endcelldefine` tag modules as cell modules. Cells are used by certain PLI routines for applications, such as delay calculations. It is advisable to pair each ``celldefine` with an ``endcelldefine`. More than one of these pairs may appear in a single source description.

These directives may appear anywhere in the source description, but it is recommended that the directives be specified outside the module definition.

16.2 ``default_nettype`

The directive ``default_nettype` controls the net type created for implicit net declarations (see 3.5). It can be used only outside of module definitions. It affects all modules that follow the directive, even across source file boundaries. Multiple ``default_nettype` directives are allowed. The latest occurrence of this directive in the source controls the type of nets that will be implicitly declared. Syntax 16-1 contains the syntax of the directive.

```
default_nettype_compiler_directive ::=  
    `default_nettype net_type  
net_type ::=  
    wire | tri | tri0 | wand | triand | tri1 | wor | trior | triereg
```

Syntax 16-1—Syntax for default nettype compiler directive

When no **`default_nettype** directive is present, implicit nets are of type **wire**.

16.3 **`define** and **`undef**

A text macro substitution facility has been provided so that meaningful names can be used to represent commonly used pieces of text. For example, in the situation where a constant number is repetitively used throughout a description, a text macro would be useful in that only one place in the source description would need to be altered if the value of the constant needed to be changed.

16.3.1 **`define**

The directive **`define** creates a macro for text substitution. This directive can be used both inside and outside module definitions. After a text macro is defined, it can be used in the source description by using the **(`)** character, followed by the macro name. The compiler shall substitute the text of the macro for the string **`macro_name**. All compiler directives shall be considered predefined macro names; it shall be illegal to redefine a compiler directive as a macro name.

A text macro can be defined with arguments. This allows the macro to be customized for each use individually.

The syntax for text macro definitions is given in Syntax 16-2.

```
text_macro_definition ::=  
    `define text_macro_name macro_text  
text_macro_name ::=  
    text_macro_identifier [ ( list_of_formal_arguments ) ]  
list_of_formal_arguments ::=  
    formal_argument_identifier { , formal_argument_identifier }
```

Syntax 16-2—Syntax for text macro definition

The macro text can be any arbitrary text specified on the same line as the text macro name. If more than one line is necessary to specify the text, the newline shall be preceded by a backslash (****). The first newline not preceded by a backslash shall end the macro text. The newline preceded by a backslash shall be replaced in the expanded macro with a newline (but without the preceding backslash character).

When formal arguments are used to define a text macro, the scope of the formal argument shall extend up to the end of the macro text. A formal argument can be used in the macro text in the same manner as an identifier.

If a one-line comment (that is, a comment specified with the characters **//**) is included in the text, then the comment shall not become part of the substituted text. The macro text can be blank, in which case the text macro is defined to be empty, and no text is substituted when the macro is used.

The syntax for using a text macro is given in Syntax 16-3.

```

text_macro_usage ::=
    `text_macro_identifier [ ( list_of_actual_arguments ) ]
list_of_actual_arguments ::=
    actual_argument { , actual_argument }
actual_argument ::=
    expression

```

Syntax 16-3—Syntax for text macro usage

For a macro without arguments, the text shall be substituted “as is” for every occurrence of `text_macro. However, a text macro with one or more arguments shall be expanded by substituting each formal argument with the expression used as the actual argument in the macro usage.

Once a text macro name has been defined, it can be used anywhere in a source description; that is, there are no scope restrictions. Text macros can be defined and used interactively.

The text specified for macro text shall not be split across the following lexical tokens:

- Comments
- Numbers
- Strings
- Identifiers
- Keywords
- Operators

Examples:

```

`define wordsize 8
reg [1:`wordsize] data;

//define a nand with variable delay
`define var_nand(dly) nand #dly

`var_nand(2) g121 (q21, n10, n11);
`var_nand(5) g122 (q22, n10, n11);

```

The following is illegal syntax because it is split across a string:

```

`define first_half "start of string
$display(`first_half end of string");

```

NOTES

1—Each actual argument is substituted for the corresponding formal argument literally. Therefore, when an expression is used as an actual argument, the expression will be substituted in its entirety. This may cause an expression to be evaluated more than once if the formal argument was used more than once in the macro text. For example,

```

`define max(a,b)((a) > (b) ? (a) : (b))
n = `max(p+q, r+s) ;

```

will expand as

```

n = ((p+q) > (r+s)) ? (p+q) : (r+s) ;

```

Here, the larger of the two expressions $p + q$ and $r + s$ will be evaluated twice.

2—The word **define** is known as a compiler directive keyword, and it is not part of the normal set of keywords. Thus, normal identifiers in a Verilog HDL source description can be the same as compiler directive keywords (although this is not recommended). The following problems should be considered:

- a) Text macro names may not be the same as compiler directive keywords.
- b) Text macro names can re-use names being used as ordinary identifiers. For example, `signal_name` and ``signal_name` are different.
- c) Redefinition of text macros is allowed; the latest definition of a particular text macro read by the compiler prevails when the macro name is encountered in the source text.

16.3.2 ``undef`

The directive ``undef` shall undefine a previously defined text macro. An attempt to undefine a text macro that was not previously defined using a ``define` compiler directive can result in a warning. The syntax for ``undef` compiler directive is given in Syntax 16-4.

<pre>undefine_compiler_directive ::= `undef text_macro_name</pre>

Syntax 16-4—Syntax for undef compiler directive

An undefined text macro has no value.

16.4 ``ifdef`, ``else`, ``endif`

These conditional compilation compiler directives are used to include optionally lines of a Verilog HDL source description during compilation. The ``ifdef` compiler directive checks for the definition of a variable name. If the variable name is defined, then the lines following the ``ifdef` directive are included. If the variable name is not defined and an ``else` directive exists, then this source is compiled.

These directives may appear anywhere in the source description.

Situations where the ``ifdef`, ``else`, and ``endif` compiler directives may be useful include:

- Selecting different representations of a module such as behavioral, structural, or switch level
- Choosing different timing or structural information
- Selecting different stimulus for a given run

The ``ifdef`, ``else`, and ``endif` compiler directives have the syntax shown in Syntax 16-5.

```
conditional_compilation_directive ::=
    `ifdef text_macro_name
        first_group_of_lines
    [ `else
        second_group_of_lines
    `endif ]
```

Syntax 16-5—Syntax for conditional compilation directives

The text macro name is a Verilog HDL identifier. The first group of lines and the second group of lines are parts of a Verilog HDL source description. The **`else** compiler directive and the second group of lines are optional.

The **`ifdef**, **`else**, and **`endif** compiler directives work in the following manner:

- When an **`ifdef** is encountered, the text macro name is tested to see if it is defined as a text macro name using **`define** within the Verilog HDL source description.
- If the text macro name is defined, the first group_of_lines is compiled as part of the description. If there is an **`else** compiler directive, the second group of lines is ignored.
- If the text macro name has not been defined, the first group of lines is ignored. If there is an **`else** compiler directive, the second group of lines is compiled.

NOTES

1—Any group of lines that the compiler ignores still has to follow the Verilog HDL lexical conventions for white space, comments, numbers, strings, identifiers, keywords, and operators.

2—These compiler directives may be nested.

Examples:

Example 1—The example below shows a simple usage of an **`ifdef** directive for conditional compilation. If the identifier **behavioral** is defined, a continuous net assignment will be compiled in; otherwise, an **and** gate will be instantiated.

```
module and_op (a, b, c);
    output a;
    input b, c;

    `ifdef behavioral
        wire a = b & c;
    `else
        and a1 (a,b,c);
    `endif

endmodule
```

Example 2—The following example shows usage of nested conditional compilation directives.


```
module test(out);
output out;
`define wow
`define nest_one
`define second_nest
`define nest_two
  `ifdef wow
    initial $display("wow is defined");
    `ifdef nest_one
      initial $display("nest_one is defined");
      `ifdef nest_two
        initial $display("nest_two is defined");
      `else
        initial $display("nest_two is not defined");
      `endif
    `else
      initial $display("nest_one is not defined");
    `endif
  `else
    initial $display("wow is not defined");
    `ifdef second_nest
      initial $display("nest_two is defined");
    `else
      initial $display("nest_two is not defined");
    `endif
  `endif
endmodule
```

16.5 `include

The file inclusion (**`include**) compiler directive is used to insert the entire contents of a source file in another file during compilation. The result is as though the contents of the included source file appear in place of the **`include** compiler directive. The **`include** compiler directive can be used to include global or commonly used definitions and tasks without encapsulating repeated code within module boundaries.

Advantages of using the **`include** compiler directive include the following:

- Providing an integral part of configuration management
- Improving the organization of Verilog HDL source descriptions
- Facilitating the maintenance of Verilog HDL source descriptions

The syntax for the **`include** compiler directive is given in Syntax 16-6.

<pre>include_compiler_directive ::= `include "filename"</pre>

Syntax 16-6—Syntax for include compiler directive

The compiler directive **`include** can be specified anywhere within the Verilog HDL description. The *filename* is the name of the file to be included in the source file. The *filename* can be a full or relative path name.

Only white space or a comment may appear on the same line as the **`include** compiler directive.

A file included in the source using the **`include** compiler directive may contain other **`include** compiler directives. The number of nesting levels for included files shall be finite.

Examples:

Examples of legal comments for the **`include** compiler directive are as follows:

```
`include "parts/count.v"
`include "fileB"
`include "fileB" // including fileB
```

NOTE—Implementations may limit the maximum number of levels to which include files can be nested, but the limit shall be at least 15.

16.6 **`resetall**

When **`resetall** compiler directive is encountered during compilation, all compiler directives are set to the default values. This is useful for ensuring that only those directives that are desired in compiling a particular source file are active.

The recommended usage is to place **`resetall** at the beginning of each source text file, followed immediately by the directives desired in the file.

16.7 **`timescale**

This directive specifies the time unit and time precision of the modules that follow it. The time unit is the unit of measurement for time values such as the simulation time and delay values.

To use modules with different time units in the same design, the following timescale constructs are useful:

- The **`timescale** compiler directive to specify the unit of measurement for time and precision of time in the modules in the design
- The **\$sprinttimescale** system task to display the time unit and precision of a module
- The **\$time** and **\$realtime** system functions, the **\$timeformat** system task, and the **%t** format specification to specify how time information is reported

The **`timescale** compiler directive specifies the unit of measurement for time and delay values and the degree of accuracy for delays in all modules that follow this directive until another **`timescale** compiler directive is read.

The syntax for the **`timescale** directive is given in Syntax 16-7.

```
timescale_compiler_directive ::=
    `timescale time_unit / time_precision
```

Syntax 16-7—Syntax for timescale compiler directive

The **time_unit** argument specifies the unit of measurement for times and delays.

The `time_precision` argument specifies how delay values are rounded before being used in simulation. The values used are accurate to within the unit of time that is specified here. The smallest `time_precision` argument of all the ``timescale` compiler directives in the design determines the time unit of the simulation.

The `time_precision` argument shall be at least as precise as the `time_unit` argument; it cannot specify a longer unit of time than `time_unit`.

The integers in these arguments specify an order of magnitude for the size of the value; the valid integers are 1, 10, and 100. The character strings represent units of measurement; the valid character strings are **s**, **ms**, **us**, **ns**, **ps**, and **fs**.

The units of measurement specified by these character strings are given in Table 16-1.

Table 16-1—Arguments of `time_precision`

Character string	Unit of measurement
s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds
ps	picoseconds
fs	femtoseconds

Examples:

The following example shows how this directive is used:

```
`timescale 1 ns / 1 ps
```

Here, all time values in the modules that follow the directive are multiples of 1 ns because the `time_unit` argument is “1 ns”. Delays are rounded to real numbers with three decimal places—or precise to within one thousandth of a nanosecond—because the `time_precision` argument is “1 ps,” or one thousandth of a nanosecond.

Consider the following example:

```
`timescale 10 us / 100 ns
```

The time values in the modules that follow this directive are multiples of 10 μ s because the `time_unit` argument is “10 us”. Delays are rounded to within one tenth of a microsecond because the `time_precision` argument is “100 ns,” or one tenth of a microsecond.

The following example shows a ``timescale` directive in the context of a module:

```
`timescale 10 ns / 1 ns
module test;
reg set;
parameter d = 1.55;

initial begin
    #d set = 0;
    #d set = 1;
end
endmodule
```

The ``timescale 10 ns / 1 ns` compiler directive specifies that the time unit for module `test` is 10 ns. As a result, the time values in the module are multiples of 10 ns, rounded to the nearest 1 ns and, therefore, the value stored in parameter `d` is scaled to a delay of 16 ns. This means that the value 0 is assigned to reg `set` at simulation time 16 ns (1.6×10 ns), and the value 1 at simulation time 32 ns.

Parameter `d` retains its value no matter what timescale is in effect.

These simulation times are determined as follows:

- a) The value of parameter `d` is rounded from 1.55 to 1.6 according to the time precision.
- b) The time unit of the module is 10 ns, and the precision is 1 ns, so the delay of parameter `d` is scaled from 1.6 to 16.
- c) The assignment of 0 to reg `set` is scheduled at simulation time 16 ns and the assignment of 1 at simulation time 32 ns. The time values are not rounded when the assignments are scheduled.

16.8 ``unconnected_drive` and ``nounconnected_drive`

All unconnected input ports of a module appearing between the directives ``unconnected_drive` and ``nounconnected_drive` are pulled up or pulled down instead of the normal default.

The directive ``unconnected_drive` takes one of two arguments—`pull1` or `pull0`. When `pull1` is specified, all unconnected input ports are automatically pulled up. When `pull0` is specified, unconnected ports are pulled down. These directives shall be specified in pairs, and outside of the module declarations.

Section 17

PLI TF and ACC interface mechanism

The interface mechanism described in this section provides a means for users to link applications based on PLI task/function (TF) routines and access (ACC) routines to Verilog software products. Through the interface mechanism, a user can

- Specify a user-defined system task or function name that can be included in Verilog HDL source descriptions; the user-defined system task or function name shall begin with a dollar sign (\$), such as **\$get_vector**
- Provide one or more PLI C applications to be called by a software product (such as a logic simulator)
- Define which PLI C applications are to be called—and when the applications should be called—when the user-defined system task or function name is encountered in the Verilog HDL source description
- Define whether the PLI applications should be treated as *functions* (which return a value) or *tasks* (analogous to subroutines in other programming languages)
- Define a data argument to be passed to the PLI applications each time they are called

NOTE—The PLI interface mechanism described in this section does not apply to applications that use the Verilog Procedural Interface (VPI) routines; these routines use the VPI registry mechanism described in sections 22 and 23.

17.1 PLI purpose and history

Sections 17 through 23 and annexes C through E describe the C-language procedural interface standard and interface mechanisms that are part of the Verilog HDL. This procedural interface, known as the Programming Language Interface, or PLI, provides a means for Verilog HDL users to access and modify data in an instantiated Verilog HDL data structure dynamically. An instantiated Verilog HDL data structure is the result of compiling Verilog HDL source descriptions and generating the hierarchy modeled by module instances, primitive instances, and other Verilog HDL constructs that represent scope. The PLI procedural interface provides a library of C-language functions that can directly access data within an instantiated Verilog HDL data structure.

A few of the many possible applications for the PLI procedural interface are

- C-language delay calculators for Verilog model libraries that can dynamically scan the data structure of a Verilog software product and then dynamically modify the delays of each instance of models from the library
- C-language applications that dynamically read test vectors or other data from a file and pass the data into a Verilog software product
- Custom graphical waveform and debugging environments for Verilog software products
- Source code decompilers that can generate Verilog HDL source code from the compiled data structure of a Verilog software product
- Simulation models written in the C language and dynamically linked into Verilog HDL simulations
- Interfaces to actual hardware, such as a hardware modeler, that dynamically interact with simulations

This document standardizes a public-domain Verilog PLI that has been in use since the mid-1980s. This standard comprises three primary generations of public-domain PLI routines.

- a) *Task/function* routines, called *TF* routines, make up the first generation of the PLI. These routines, most of which start with the characters **tf_**, are primarily used for operations involving user-defined task/function arguments, along with utility functions, such as setting up call-back mechanisms and writing data to output devices. The TF routines are sometimes referred to as *utility* routines

- b) *Access* routines, called *ACC* routines, form the second generation of the PLI. These routines, which all start with the characters **acc_**, provide an object-oriented access directly into a Verilog HDL structural description. *ACC* routines are used to access and modify information, such as delay values and logic values on a wide variety of objects that exist in a Verilog HDL description. There is some overlap in functionality between *ACC* routines and *TF* routines.
- c) *Verilog Procedural Interface* routines, called *VPI* routines, are the third generation of the PLI. These routines, all of which start with the characters **vpi_**, provide an object-oriented access for both Verilog HDL structural and behavioral objects. The *VPI* routines are a superset of the functionality of the *TF* routines and *ACC* routines.

17.2 User-defined task or function names

A user-defined task or function name is the name that will be used within a Verilog HDL source file to invoke specific PLI applications. The name shall adhere to the following rules:

- The first character of the name shall be the dollar sign character (\$)
- The remaining characters shall be letters, digits, the underscore character (_), or the dollar character (\$)
- Uppercase and lowercase letters shall be considered to be unique—the name is case sensitive
- The name can be any size, and all characters are significant

17.3 Overloading built-in system task and function names

Section 14 defines a number of built-in system tasks and functions that are part of the Verilog language. In addition, software products can include other built-in system tasks and functions specific to the product. These built-in system task and function names begin with the dollar sign character (\$), just as user-defined system task and function names.

If a user-provided PLI application is associated with the same name as a built-in system task or function (using the PLI interface mechanism), the user-provided C application shall overload the built-in system task/function, replacing its functionality with that of the user-provided C application. For example, a user could write a random number generator as a PLI application and then associate the application with the name **\$random**, thereby overriding the built-in **\$random** function with the user's application.

17.4 User-supplied PLI applications

User-supplied PLI applications are C-language functions that utilize the library of PLI C functions to access and interact dynamically with Verilog HDL software implementations as the Verilog HDL source code is executed.

These PLI applications are not independent C programs. They are C functions, which are linked into a software product, and become part of the product. This allows the PLI application to be called when the user-defined system task or function \$ name is compiled or executed in the Verilog HDL source code.

The PLI interface mechanism provides a means to have PLI applications called for various reasons when the associated system task or function \$ name is encountered in the Verilog HDL source description. For example, when a Verilog HDL simulator first compiles the Verilog HDL source description, a specific PLI application can be called that performs syntax checking to ensure the user-defined system task or function is being used correctly. Then, as simulation is executing, a different PLI application can be called to perform the operations required by the PLI application. Other PLI applications can be automatically called by the simulator for miscellaneous reasons, such as the end of a simulation time step or a logic value change on a specific signal.

The PLI interface mechanism for *TF* and *ACC* routines provides five classes of user-supplied PLI applications: *checktf* applications, *sizetf* applications, *calltf* applications, *misctf* applications, and *consumer* applications. The purpose of each of the PLI application classes is explained in the following subclauses.

17.4.1 The calltf class of PLI applications

A *calltf* PLI application shall be called each time the associated user-defined system task or function is executed within the Verilog HDL source code. For example, the following Verilog loop would call the PLI *calltf* application that is associated with the \$get_vector user-defined system task name 1024 times:

```
for (i = 1; i <= 1024; i = i + 1)
    @(posedge clk) $get_vector("test_vector.pat", input_bus);
```

In this example, the user-supplied PLI *calltf* application might read a test vector from a file called `test_vector.pat` (the first task/function argument), perhaps manipulate the vector to put it in a proper format for Verilog, and then assign the vector value to the second task/function argument called `input_bus`.

17.4.2 The checktf class of PLI applications

A *checktf* PLI application shall be called when the user-defined system task or function name is encountered during parsing or compiling the Verilog HDL source code. This application is typically used to check the correctness of any arguments used with the system task in the Verilog HDL source code. The *checktf* PLI application shall be called one time for each instance of a system task or function in the source description. Providing a *checktf* application is optional, but it is recommended that any arguments used with the system task or function be checked for correctness to avoid problems when the *calltf* or other PLI applications read and perform operations on the arguments.

17.4.3 The sizetf class of PLI applications

A *sizetf* PLI application can be used in conjunction with user-defined system *functions*. A function shall return a value, and software products that execute the system function may need to determine how many bits wide that return will be. The *sizetf* application shall be called one time for each instance of a system task or function in the source description, typically when the Verilog HDL source code is compiled; when called, the *sizetf* application shall return the number of bits of the system function return value. The *sizetf* application shall not be called for PLI system tasks.

17.4.4 The misctf class of PLI applications

A *misctf* PLI application shall be called by a Verilog software product for miscellaneous reasons while the Verilog HDL source description is being executed. Among these reasons can be the end of a simulation time step, a logic value change on a user-defined system task/function argument, or the execution of the **\$stop** and **\$finish** built-in system functions. When the software product calls the *misctf* PLI application, it shall pass in a reason argument, which can be used within the *misctf* application to determine why the application was called. The reason argument shall be a predefined integer constant. Table 17-1 lists the reasons the *misctf* application can be called. Note that the *misctf* application associated with a specific instance of a user-defined task or function \$ name shall not be activated until the instance of the task/function has been executed the first time.

17.4.5 The consumer class of PLI applications

A *consumer* PLI application shall be called through a PLI callback mechanism referred to as the Value Change Link (VCL). Using the VCL, another PLI application, typically the *calltf* application, can place VCL flags on objects within the Verilog HDL data structure, such as a specific net. Whenever an object with a VCL flag changes value during a simulation, the consumer PLI application shall be called and passed information about the change.

17.5 Associating PLI applications to a class and system task/function name

Each user-provided PLI application is a standard C-language function that makes use of the library of PLI functions. These user-provided PLI applications shall be associated with both the class of application (such as *calltf* or *checktf*) and the user-defined system task or function \$ name. In addition, the user-defined name shall be declared as either a system task or a system function.

The method of associating PLI applications with a class and system task/function name is not defined as part of this standard. Each software product vendor shall define an association mechanism specific to their product. Refer to the documentation provided by the vendor for instructions on associating PLI applications to classes and system task/function names and then linking the PLI applications into the software products of the vendor.

17.6 PLI application arguments

When the `calltf`, `checktf`, and `sizetf` PLI applications are called by a Verilog software implementation, they shall be passed two C arguments, *data* and *reason*, in that order. When the `misctf` application is called, it shall be passed three C arguments, *data*, *reason*, and *paramvc*, in that order. These arguments are defined in more detail in the following subclauses.

17.6.1 The data C argument

The *data* C argument shall be an integer value. The value is defined by the user at the time the PLI applications are associated with a user-defined system task/function name. This value can be used to allow several different system task/function names to use the same `calltf`, `checktf`, `sizetf`, or `misctf` applications. To do this, each system task/function name would be associated with the same PLI applications, but each would have a different value for the user-defined data argument. When a PLI application is called, it can then check the value of the data argument to determine which system task/function name was used to call the application.

17.6.2 The reason C argument

The *reason* C argument shall be a predefined integer constant that is passed to the `calltf`, `checktf`, `sizetf`, and `misctf` applications each time the applications are called. Generally, the `calltf`, `checktf`, and `sizetf` applications do not need to check the reason argument, since these applications can only be called under specific circumstances. The `misctf` application, however, can be called for a wide variety of reasons, and therefore it should always examine the reason argument to determine why the application was called. The value for the reason argument is defined in the PLI include file `veriusr.h`. The predefined constants that can be passed to the reason argument are listed in Table 17-1.

17.6.3 The paramvc C argument

The *paramvc* C argument shall be an integer value passed to the `misctf` application. The value of *paramvc* shall indicate which task/function argument changed value when the `misctf` application was called back after activating the utility routine `tf_asynchon()`. This routine shall cause the `misctf` application to be called with a reason argument of `reason_paramvc` or `reason_paramdrc`.

Table 17-1—Predefined reason integer constants

Reason value passed to	Predefined integer constant
<code>calltf</code> applications	<code>reason_calltf</code>
<code>checktf</code> applications	<code>reason_checktf</code>
<code>sizetf</code> applications	<code>reason_sizetf</code>
<code>misctf</code> applications for the end of Verilog source compilation/start of execution	<code>reason_endofcompile</code>
for a change of value on a user-defined system task or function argument parameter	<code>reason_paramvc</code>

Table 17-1—Predefined reason integer constants (continued)

Reason value passed to	Predefined integer constant
for a value change on the driver of a user-defined system task or function argument parameter	reason_paramdrc
for a the end of a time step flagged by tf_synchronize()	reason_synch
for a the end of a time step flagged by tf_rosynchronize()	reason_rosynch
for a simulation event scheduled by tf_setdelay()	reason_reactivate
for the execution of a procedural force or procedural continuous assignment on any net, register, or variable	reason_force
for the execution of a procedural release or procedural deassign on any net, register, or variable	reason_release
for the execution of a procedural disable statement	reason_disable
for the execution of the \$stop() built-in system task	reason_interactive
for the execution of the \$scope() built-in system task	reason_scope
for the start of execution of the \$save() built-in system task	reason_startofsave
for the completion of execution of the \$save() built-in system task	reason_save
for the execution of the \$restart() built-in system task	reason_restart
for the start of execution of the \$reset() built-in system task	reason_reset
for the completion of execution of the \$reset() built-in system task	reason_endofreset
for the \$finish() built-in system task executed	reason_finish

17.7 User-defined system task and function arguments

When a user-defined system task or function is used in a Verilog HDL source file, it can have arguments that can be used by the PLI applications associated with the system task or function. In the following example, the user-defined system task **\$get_vector** has two arguments:

```
$get_vector("test_vector.pat", input_bus);
```

The arguments to a system task or function are referred to as *task/function arguments* (often abbreviated as *tfargs*) by ACC routines, and as *task/function parameters* by TF routines. These arguments are not the same as C-language arguments. When the PLI applications associated with a user-defined system task or function are called, the task/function arguments are not passed to the PLI application. Instead, a number of PLI routines are provided that allow the PLI applications to read and write to the task/function arguments. Refer to the sections on ACC routines and TF routines for information on specific routines that work with task/function arguments.

Note that when PLI applications are called, they are passed two or three C arguments: data, reason, and paramvc. These arguments are not the same as the task/function arguments that appear in the Verilog HDL source code.

17.8 PLI include files for TF and ACC routines

PLI applications that use the TF routines shall include the file `veriusert.h`. PLI applications that use the ACC routines shall include the file `acc_user.h`. These files define constants, structures, and other data used by the library of PLI routines and the interface mechanism. The files are listed in annexes C and D.

Section 18

Using ACC routines

This section presents a general discussion of how and why to use PLI ACC routines. Section 19 defines the ACC routine syntax, listed in alphabetical order.

18.1 ACC routine definition

ACC routines are C programming language functions that provide procedural access to information within the Verilog HDL.

ACC routines perform one of two functions:

- a) Read data about particular objects in the Verilog HDL description directly from internal data structures.
- b) Write new information about certain objects in the Verilog HDL description into the internal data structures.

ACC routines shall read information about the following objects:

- Module instances
- Module ports
- Module paths
- Intermodule paths
- Top-level modules
- Primitive instances
- Primitive terminals
- Nets
- Registers
- Parameters
- Specparams
- Timing checks
- Named events
- Integer, real, and time variables

ACC routines shall read and write information on the following objects:

- Intermodule path delays
- Module path delays
- Module input port delays (MIPDs)
- Primitive instance delays
- Timing check limits
- Register logic values
- Sequential UDP logic values

18.2 The handle data type

A *handle* is a predefined data type that is a pointer to a specific object in the design hierarchy. Each handle conveys information to ACC routines about a unique instance of an accessible object—information about the type of the object, plus how and where to find data about the object.

Most ACC routines require a handle argument to indicate the objects about which they need to read or write information. The PLI provides two categories of ACC routines that return handles for objects: handle routines, which begin with the prefix **acc_handle_**, and next routines, which begin with the prefix **acc_next_**. Refer to 18.4.2 for a discussion of handle routines and 18.4.3 for more information about next routines.

Handles shall be passed to and from ACC routines through *handle variables*. To declare a handle variable, the keyword **handle** (all lowercase) shall be used, followed by the variable name, as in this example:

```
handle net_handle;
```

After declaring a handle variable, it can be passed to any ACC routine that requires a handle argument or be used to receive a handle returned by an ACC routine. The following C-language code fragment uses the variable `net_handle` to store the handle returned by the ACC routine **acc_handle_object()**:

```
handle net_handle;
net_handle = acc_handle_object("top.mod1.w3");
```

18.3 Using ACC routines

18.3.1 Header files

The header file `acc_user.h` shall be included in any C-language source file containing an application program that calls ACC routines. The `acc_user.h` file is listed in annex C.

18.3.2 Initializing ACC routines

The ACC routine **acc_initialize()** shall initialize the environment for ACC routines and shall be called from the C-language application program before the program invokes any other ACC routines.

18.3.3 Setting the development version

After initializing ACC routines, the configuration parameter **accDevelopmentVersion** can be set to indicate which version of ACC routines was used to develop the application. Configuring the **accDevelopmentVersion** parameter can ensure that future releases of ACC routines can run PLI application code the same as when the code was written.

To set this parameter, call **acc_configure()** after calling **acc_initialize()**. The following example sets **accDevelopmentVersion** to the IEEE Std 1364-1995 PLI version of ACC routines:

```
acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");
```

18.3.4 Exiting ACC routines

Before exiting a C-language application program that calls ACC routines, the ACC routine **acc_close()** should be called. This routine shall reset ACC routine configuration parameters back to their defaults, and it shall also free memory allocated by the ACC routines.

18.4 List of ACC routines by major category

The ACC routines are divided into the following major categories:

- *Fetch* routines
- *Handle* routines
- *Next* routines
- *Modify* routines
- *VCL* routines
- *Miscellaneous* routines

This clause contains a summary list of each major category. The ACC routines sorted by the types of objects they work with are listed in 18.5. Section 19 presents an alphabetical list of all ACC routines, with their functions, syntax, and usage.

18.4.1 Fetch routines

Fetch routines shall return a variety of information about different objects in the design hierarchy. The name of each routine begins with the prefix **acc_fetch_** and indicates the type of information desired. For example, **acc_fetch_fullname()** retrieves the full hierarchical path name for any named object, while **acc_fetch_paramval()** retrieves the value of a parameter or specparam.

Table 18-1 — List of fetch routines

ACC routine	Description
acc_fetch_argc()	Get the number of invocation command line arguments
acc_fetch_argv()	Get the invocation command line arguments
acc_fetch_attribute()	Get the value of a Verilog attribute, parameter or specparam
acc_fetch_defname()	Get the definition name of a module or primitive
acc_fetch_delays()	Get the existing delays for a primitive, module path, timing check, intermodule path, or module input port
acc_fetch_delay_mode()	Get the delay mode of a module instance
acc_fetch_direction()	Get the direction of a module port or primitive terminal
acc_fetch_edge()	Get the edge specifier of module path input terminal
acc_fetch_fullname()	Get the full hierarchical name of an object
acc_fetch_fulltype()	Get the full type description of an object as a predefined integer constant
acc_fetch_index()	Get the index number of a port or terminal
acc_fetch_location()	Get the location of an object in a Verilog source file
acc_fetch_name()	Get the local name of an object
acc_fetch_paramtype()	Get the data type of a parameter or specparam
acc_fetch_paramval()	Get the value of a parameter or specparam
acc_fetch_polarity()	Get the polarity of a module path or data path
acc_fetch_precision()	Get the simulation time precision
acc_fetch_pulsere()	Get the current pulse handling values of a module path
acc_fetch_range()	Get the range of a vector
acc_fetch_size()	Get the bit size of a vector or port

Table 18-1—List of fetch routines (continued)

ACC routine	Description
acc_fetch_tfarg()	Get the value of a system task/function argument as a double
acc_fetch_tfarg_int()	Get the value of a system task/function argument as an integer
acc_fetch_tfarg_str()	Get the value of a system task/function argument as a string
acc_fetch_timescale_info()	Get the timescale information for an object
acc_fetch_type()	Get the general type classification of an object as an integer constant
acc_fetch_type_str()	Get the string representation of a type or fulltype integer constant
acc_fetch_value()	Get the logic or strength value of a net, register, or variable

18.4.2 Handle routines

Handle routines can return handles to a variety of objects in the design hierarchy. The name of each routine begins with the prefix **acc_handle_** and indicates the type of handle desired. For example, **acc_handle_object()** retrieves a handle for a named object, while **acc_handle_conn()** retrieves a handle for a net connected to a particular terminal. Each handle routine shall return a handle to an object. This handle can, in turn, be passed as an argument to other ACC routines.

Table 18-2—List of handle routines

ACC routine	Description
acc_handle_by_name()	Get the handle to any named object
acc_handle_condition()	Get the handle to the condition of a module path, data path, or timing check
acc_handle_conn()	Get the handle to the net connected to a primitive, path, or timing check terminal
acc_handle_datapath()	Get the handle to a data path
acc_handle_hiconn()	Get the handle to the hierarchically higher net connected to a module port bit
acc_handle_interactive_scope()	Get the handle to the current simulation interactive scope
acc_handle_itfarg()	Get the handle to an argument of a specific system task/function instance
acc_handle_loconn()	Get the handle to the hierarchically lower net connected to a module port bit
acc_handle_modpath()	Get the handle to a module path delay
acc_handle_notifier()	Get the handle to the notifier argument of a timing check
acc_handle_object()	Get the handle to any named object
acc_handle_parent()	Get the handle to the parent of an object
acc_handle_path()	Get the handle to an intermodule path
acc_handle_pathin()	Get the handle to the first net connected to a module path source
acc_handle_pathout()	Get the handle to the first net connected to a module path destination
acc_handle_port()	Get the handle to a module port based on the port index

Table 18-2—List of handle routines (continued)

ACC routine	Description
acc_handle_scope()	Get the handle to the scope containing an object
acc_handle_simulated_net()	Get the handle to the net associated with a collapsed net
acc_handle_tchk()	Get the handle to a timing check
acc_handle_tchkarg1()	Get the handle to the first argument of a timing check
acc_handle_tchkarg2()	Get the handle to the second argument of a timing check
acc_handle_terminal()	Get the handle to terminal of a primitive based on the terminal index
acc_handle_tfarg()	Get a handle to the object named in a system task/function argument
acc_handle_tfinst()	Get the handle the current instance of a system task/function

18.4.3 Next routines

When used inside a C loop construct, next routines shall find each object of a given type that is related to a particular reference object in the design hierarchy. The name of each routine begins with the prefix **acc_next_** and indicates the type of object desired, known as the target object. For example, **acc_next_net()** retrieves each net in a module, while **acc_next_driver()** retrieves each terminal driving a net. Each call to a next routine returns a handle to the object it finds.

Most next routines require two arguments:

- The first argument shall be a handle to a *reference object*.
- The second argument shall be a handle that indicates whether to retrieve the first or next *target object*.

The *reference object* shall indicate where the next routine shall look for the target object. The *target object* is the type of object to be returned by a next routine.

Table 18-3 summarizes how next routines shall find each target object associated with a given reference object.

Table 18-3—How next routines use the target object argument

When	A next routine shall return
The <i>target object</i> is null	A handle to the first <i>target object</i> related to the <i>reference object</i>
The <i>target object</i> is a handle to the last <i>target object</i> returned	A handle to the next <i>target object</i> related to the <i>reference object</i>
No <i>target objects</i> remain for the <i>reference object</i>	A null handle
No <i>target objects</i> are found initially for the <i>reference object</i>	A null handle
An error occurs	A null handle

NOTE—Objects can be returned in an arbitrary order.

Each call to a next routine shall return only one handle. Therefore, to retrieve all target objects for a particular reference object, the following process can be used:

- a) Chose an appropriate ACC routine to retrieve the handle of the desired reference object.

- b) Set the target object handle variable to `null`. When a next routine is called with a `null` target handle, it shall return the first target associated with the reference.
- c) Call the next routine, assigning the return value to the same variable as the target object argument. This automatically updates the target object argument to point to the last object found.
- d) Place the next routine call inside a C `while` loop that terminates when the loop control value is `null`. When a next routine cannot access any more target objects, it shall return a `null`.

The following example, `display_net_names`, uses a next routine to display the names of all nets in a module.

```
#include "acc_user.h"

display_net_names()
{
    handle    module_handle;
    handle    net_handle;

    /*initialize environment for access routines*/
    acc_initialize();

    /*set the development version*/
    acc_configure( accDevelopmentVersion, "IEEE 1364 PLI" );

    /*get handle for module*/
    module_handle = acc_handle_tfang(1);

    /*display names of all nets in the module*/
    net_handle = null;
    while( net_handle = acc_next_net( module_handle, net_handle ) )
        io_printf("Net name is: %s\n", acc_fetch_fullname(net_handle) );
    acc_close();
}
```

Table 18-4—List of next routines

ACC routine	Description
<code>acc_next()</code>	Get handles to all objects of a set of types
<code>acc_next_bit()</code>	Get handles to all bits of a port or vector
<code>acc_next_cell()</code>	Get handles to all cell modules in the current hierarchy and below
<code>acc_next_cell_load()</code>	Get handles to all cell loads on a net
<code>acc_next_child()</code>	Get handles to all module instances within a module
<code>acc_next_driver()</code>	Get handles to all primitive terminals that drive a net
<code>acc_next_hiconn()</code>	Get handles to all nets connected hierarchically higher to a module port
<code>acc_next_input()</code>	Get handles to all input terminals of a module path or data path
<code>acc_next_load()</code>	Get handles to all primitive terminals driven by a net
<code>acc_next_loconn()</code>	Get handles to all nets connected hierarchically lower to a module port
<code>acc_next_modpath()</code>	Get handles to all path delays in a module
<code>acc_next_net()</code>	Get handles to all nets in a module

Table 18-4—List of next routines (continued)

ACC routine	Description
acc_next_output()	Get handles to all output terminals of a module path or data path
acc_next_parameter()	Get handles to all parameters in a module
acc_next_port()	Get handles to all ports of a module or connected to a net
acc_next_portout()	Get handles to all output ports of a module
acc_next_primitive()	Get handles to all primitive instances in a module
acc_next_scope()	Get handles to all hierarchy scopes within a scope
acc_next_specparam()	Get handles to all specify block parameters in a module
acc_next_tchk()	Get handles to all timing checks in a module
acc_next_terminal()	Get handles to all terminals of a primitive
acc_next_topmod()	Get handles to all top-level modules

18.4.4 Modify routines

Modify routines shall alter the values of a variety of objects in the design hierarchy. Table 18-5 lists the types of values that shall be modified for particular objects.

Table 18-5—Values that can be modified

Modify routines alter	For these objects
Delay values	Primitives Module paths Intermodule paths Module input ports Timing checks
Logic values	Register data types Net data types Sequential UDPs
Pulse handling values	Module paths

More details on using the **acc_append_delays()** and **acc_fetch_delays()** ACC routines are provided in 18.8.

Table 18-6—List of modify routines

ACC routine	Description
acc_append_delays()	Add delays to existing delays on primitives, module paths, timing checks, and module input ports
acc_append_pulsere()	Add to existing pulse control values of a module path
acc_replace_delays()	Replace existing delays on primitives, module paths, timing checks, inter-module paths, and module input ports

Table 18-6—List of modify routines (continued)

ACC routine	Description
acc_replace_pulse()	Replace existing values on pulse control values of a module path
acc_set_pulse()	Set pulse control values of a module path as a percentage of path delay
acc_set_value()	Set and propagate a logic value onto a register, variable or sequential UDP; continuously assign/deassign a register; force/release a net or register

18.4.5 Miscellaneous routines

Miscellaneous routines shall perform a variety of operations, such as initializing and configuring the ACC routine environment.

Table 18-7—List of miscellaneous routines

ACC routine	Description
acc_close()	Close ACC routine environment
acc_collect()	Collect an array of handles for a reference object
acc_compare_handles()	Determine if two handles are for the same object
acc_configure()	Set the ACC routine environment parameters
acc_count()	Count the number of objects related to a reference object
acc_free()	Free up memory allocated by acc_collect()
acc_initialize()	Initialize the ACC routine environment
acc_object_in_typedlist()	Determine if an object matches a set of types, fulltypes, or special properties
acc_object_of_type()	Determine if an object matches a specific type, fulltype, or special property
acc_product_type()	Get the type of software product being used
acc_product_version()	Get the version of software product being used
acc_release_object()	Release memory allocated by acc_next_input() or acc_next_output()
acc_reset_buffer()	Reset the string buffer
acc_set_interactive_scope()	Set the interactive scope of a software implementation
acc_set_scope()	Set the scope used by acc_handle_object()
acc_version()	Get the version of the ACC routines being used

18.4.6 VCL routines

The VCL shall allow a PLI application to monitor simulation value changes of selected objects. It consists of two ACC routines that instruct a Verilog simulator to start or stop informing an application when an object changes value. How the VCL routine is used is discussed in 18.10.

Table 18-8—List of VCL routines

ACC routine	Description
acc_vcl_add()	Add a value change callback on an object
acc_vcl_delete()	Remove a value change callback

18.5 Accessible objects

ACC routines shall access information about the following objects:

- Module instances
- Module ports
- Individual bits of a port
- Module or data paths
- Intermodule paths
- Top-level modules
- Primitive instances
- Primitive terminals
- Nets (scalars, vectors, and bit- or part-selects of vectors)
- Registers (scalars, vectors, and bit- or part-selects of vectors)
- Parameters
- Specparams
- Timing checks
- Named events
- Integer variables (and bit- or part-selects of integers)
- Real and time variables

The following tables summarize the operations that can be performed for each of the above object types.

18.5.1 ACC routines that operate on module instances

Table 18-9—Operations on module instances

To	Use
Obtain handles for module instances tagged as cells within a hierarchical scope and below	acc_next_cell()
Obtain handles for module instances within a particular module instance	acc_next_child()
Obtain a handle to the parent (the module that contains the instance)	acc_handle_parent()
Get the instance name	acc_fetch_name()
Get the full hierarchical name	acc_fetch_fullname()
Get the module definition name	acc_fetch_defname()

Table 18-9—Operations on module instances (continued)

To	Use
Get the fulltype of a module instance (cell instance, module instance, or top-level module)	acc_fetch_fulltype()
Get the delay mode of a module instance (none, zero, unit, distributed, or path)	acc_fetch_delay_mode()
Get timescale information for a module instance	acc_fetch_timescale_info()

18.5.2 ACC routines that operate on module ports**Table 18-10—Operations on module ports**

To	Use
Obtain handles for ports of a module instance	acc_next_port()
Obtain a handle for a particular port	acc_handle_port()
Obtain a handle to the parent (the module instance that contains the port)	acc_handle_parent()
Obtain handles to hierarchically higher-connected nets	acc_next_hiconn()
Obtain handles to hierarchically lower-connected nets	acc_next_loconn()
Obtain a handle to the hierarchically higher-connected net of a scalar module port or bit of a vector port	acc_handle_hiconn()
Obtain a handle to the hierarchically lower-connected net of a scalar module port or bit of a vector port	acc_handle_loconn()
Get the instance name	acc_fetch_name()
Get the full hierarchical name	acc_fetch_fullname()
Get the port direction	acc_fetch_direction()
Get the port index number	acc_fetch_index()
Get the fulltype of a module port	acc_fetch_fulltype()
Add VCL value change callback monitors	acc_vcl_add()
Delete VCL value change callback monitors	acc_vcl_delete()

18.5.3 ACC routines that operate on bits of a port

Table 18-11—Operations on bits of a port

To	Use
Obtain handles for bits of a module port	acc_next_bit()
Get the port name	acc_fetch_name()
Get the full hierarchical name	acc_fetch_fullname()
Get the fulltype of a port's bit	acc_fetch_fulltype()
Read MIPD	acc_fetch_delays()
Append to existing MIPD	acc_append_delays()
Replace existing MIPD	acc_replace_delays()
Add VCL value change callback monitors	acc_vcl_add()
Delete VCL value change callback monitors	acc_vcl_delete()

18.5.4 ACC routines that operate on module paths or data paths

Table 18-12—Operations on module paths and data paths

To	Use
Obtain handles for module paths within a scope	acc_next_modpath()
Obtain a handle to the first connected nets	acc_handle_pathin() acc_handle_pathout()
Obtain a handle to a module path	acc_handle_modpath()
Obtain a handle to a datapath	acc_handle_datapath()
Obtain a handle to a conditional expression for a path	acc_handle_condition()
Obtain handles for input terminals of a module path or data path	acc_next_input()
Obtain handles for output terminals of a module path or data path	acc_next_output()
Get the path name	acc_fetch_name()
Get the full hierarchical name	acc_fetch_fullname()
Get the polarity of a path	acc_fetch_polarity()
Get the edge specified for a path terminal	acc_fetch_edge()
Read path delays	acc_fetch_delays()
Append to existing path delays	acc_append_delays()
Replace existing path delays	acc_replace_delays()
Read path pulse handling	acc_fetch_pulsere()

Table 18-12—Operations on module paths and data paths

To	Use
Append to existing path pulse control values	acc_append_pulsere()
Specify path pulse control values	acc_set_pulsere()
Free memory allocated by acc_next_input() or acc_next_output()	acc_release_object()

18.5.5 ACC routines that operate on intermodule paths**Table 18-13—Operations on intermodule paths**

To	Use
Obtain a handle for an intermodule path	acc_handle_path()
Get the fulltype of an intermodule path	acc_fetch_fulltype()
Read intermodule path delays	acc_fetch_delays()
Modify intermodule path delays	acc_replace_delays()

18.5.6 ACC routines that operate on top-level modules**Table 18-14—Operations on top-level modules**

To	Use
Obtain handles for top-level modules in a design	acc_next_topmod()
Get the module name	acc_fetch_name() acc_fetch_fullname() acc_fetch_defname()

18.5.7 ACC routines that operate on primitive instances**Table 18-15—Operations on primitive instances**

To	Use
Obtain handles for primitive instances within a module instance	acc_next_primitive()
Obtain a handle to the parent (the module that contains the primitive)	acc_handle_parent()
Get the instance name	acc_fetch_name()
Get the full hierarchical name	acc_fetch_fullname()
Get the definition name	acc_fetch_defname()
Get the primitive <i>fulltype</i>	acc_fetch_fulltype()

Table 18-15—Operations on primitive instances

To	Use
Read delays	acc_fetch_delays()
Append to existing primitive delays	acc_append_delays()
Replace existing primitive delays	acc_replace_delays()

18.5.8 ACC routines that operate on primitive terminals**Table 18-16—Operations on primitive terminals**

To	Use
Obtain handles for terminals of a primitive instance	acc_next_terminal()
Obtain a handle to the net connected to the terminal	acc_handle_conn()
Obtain a handle to the parent (primitive instance containing the terminal)	acc_handle_parent()
Get the direction (input, output, inout)	acc_fetch_direction()
Get the terminal index number	acc_fetch_index()
Get the fulltype	acc_fetch_fulltype()
Add VCL value change callback monitors	acc_vcl_add()
Delete VCL value change callback monitors	acc_vcl_delete()

18.5.9 ACC routines that operate on nets**Table 18-17—Operations on nets**

To	Use
Obtain handles for nets within a module instance	acc_next_net()
Obtain a handle to the parent (the module instance that contains the net)	acc_handle_parent()
Determine if net is scalar, vector, collapsed, or expanded	acc_object_of_type()
Obtain handles to bits of a vector net	acc_next_bit()
Obtain handles to driving terminals of the net	acc_next_driver()
Obtain handles to load terminals of the net	acc_next_load()
Obtain handles to connected load terminals; only one per driven cell port	acc_next_cell_load()
Obtain a handle to the simulated net of a collapsed net	acc_handle_simulated_net()
Get the net name	acc_fetch_name()
Get the full hierarchical name	acc_fetch_fullname()

Table 18-17—Operations on nets

To	Use
Get the net vector size	<code>acc_fetch_size()</code>
Get the msb and lsb vector range	<code>acc_fetch_range()</code>
Get the net fulltype	<code>acc_fetch_fulltype()</code>
Get the net logic or strength value	<code>acc_fetch_value()</code>
Force or release the net logic value	<code>acc_set_value()</code>
Add VCL value change callback monitors	<code>acc_vcl_add()</code>
Delete VCL value change callback monitors	<code>acc_vcl_delete()</code>

18.5.10 ACC routines that operate on registers**Table 18-18—Operations on registers**

To	Use
Obtain handles to registers within a given scope	<code>acc_next()</code>
Obtain handles to bits of a vector register	<code>acc_next_bit()</code>
Obtain a handle to the parent (module instance containing the register)	<code>acc_handle_parent()</code>
Obtain handles to load terminals of the register	<code>acc_next_load()</code>
Determine if register is scalar or vector	<code>acc_object_of_type()</code>
Get the register name	<code>acc_fetch_name()</code>
Get the full hierarchical name	<code>acc_fetch_fullname()</code>
Get the register size	<code>acc_fetch_size()</code>
Get the msb and lsb vector range	<code>acc_fetch_range()</code>
Get the register value	<code>acc_fetch_value()</code>
Set the register value	<code>acc_set_value()</code>
Add VCL value change callback monitors	<code>acc_vcl_add()</code>
Delete VCL value change callback monitors	<code>acc_vcl_delete()</code>

18.5.11 ACC routines that operate on integer, real, and time variables**Table 18-19—Operations on integer, real, and time variables**

To	Use
Obtain handles to variables within a given scope	<code>acc_next()</code>
Obtain a handle to the parent (module instance containing the variable)	<code>acc_handle_parent()</code>
Get the variable name	<code>acc_fetch_name()</code>

Table 18-19—Operations on integer, real, and time variables

To	Use
Get the full hierarchical name	acc_fetch_fullname()
Get the variable value	acc_fetch_value()
Set the variable value	acc_set_value()
Add VCL value change callback monitors	acc_vcl_add()
Delete VCL value change callback monitors	acc_vcl_delete()

18.5.12 ACC routines that operate on named events

Table 18-20—Operations on named events

To	Use
Obtain handles to named events within a given scope	acc_next()
Obtain a handle to the parent (module instance containing the named event)	acc_handle_parent()
Get the named-event name	acc_fetch_name()
Get the full hierarchical name	acc_fetch_fullname()
Add VCL value change callback monitors	acc_vcl_add()
Delete VCL value change callback monitors	acc_vcl_delete()

18.5.13 ACC routines that operate on parameters and specparams

Table 18-21—Operations on parameters and specparams

To	Use
Obtain handles for parameters within a module instance	acc_next_parameter()
Obtain handles for specparams within a module instance	acc_next_specparam()
Obtain a handle to the parent (the module instance that contains the parameter)	acc_handle_parent()
Get the parameter or specparam name	acc_fetch_name()
Get the full hierarchical name	acc_fetch_fullname()
Get the parameter value data type (integer, floating point, string)	acc_fetch_paramtype() acc_fetch_fulltype()
Get the value of a parameter	acc_fetch_paramval()
Get the attribute value of a parameter defined with an attribute name	acc_fetch_attribute() acc_fetch_attribute_int() acc_fetch_attribute_str()

18.5.14 ACC routines that operate on timing checks**Table 18-22—Operations on timing checks**

To	Use
Obtain handles for timing checks within a module instance	acc_next_tchk()
Obtain a handle to a specific timing check	acc_handle_tchk()
Obtain a handle to a timing check terminal	acc_handle_tchkarg1() acc_handle_tchkarg2()
Get the timing check fulltype	acc_fetch_fulltype()
Get a timing check limit	acc_fetch_delays()
Append to an existing timing check limit	acc_append_delays()
Replace to an existing timing check limit	acc_replace_delays()

18.5.15 ACC routines that operate on timing check terminals**Table 18-23—Operations on timing check terminals**

To	Use
Obtain a handle to the net attached to timing check terminals	acc_handle_conn()
Obtain a handle to the condition on a timing check terminal	acc_handle_condition()
Get edge information on a timing check terminal	acc_fetch_edge()

18.5.16 ACC routines that operate on user-defined system task/function arguments**Table 18-24—Operations on user-defined system task/function arguments**

To	Use
Obtain a handle for an object named in a task/function argument	acc_handle_tfarg() acc_handle_itfarg()
Get the value of a task/function argument as a double	acc_fetch_tfarg() acc_fetch_itfarg()
Get the value of a task/function argument as an integer	acc_fetch_tfarg_int() acc_fetch_itfarg_int()
Get the value of a task/function argument as a string pointer	acc_fetch_tfarg_str() acc_fetch_itfarg_str()

18.6 ACC routine types and fulltypes

Many objects in the Verilog HDL can have both a *type* and a *fulltype* associated with them. A type shall be a general classification of an object, whereas a fulltype shall be a specific classification. The type and fulltype for a given object

can be different constants, or they can be the same constant. For example, an **and** logic gate has a type of **accPrimitive** and a fulltype of **accAndPrimitive**. The type and fulltype are predefined integer constants in the file **acc_user.h**. Several ACC routines either return a type or fulltype value, or use a type or fulltype value as an argument. Table 18-25 lists all type and fulltype constants that shall be supported by ACC routines.

Table 18-25—List of all predefined type and fulltype constants

type constant	fulltype constant	Description
accConstant	accConstant	Object is a constant
accDataPath	accDataPath	Object is a data path in a path delay
accNamedEvent	accNamedEvent	Object is declared as an event data type
accFunction	accFunction	Object is a Verilog HDL function
accIntegerVar	accIntegerVar	Object is declared as an integer data type
accModule		Object is a module instance
	accModuleInstance	Object is a module instance
	accCellInstance	Object is a module instance that has been defined as an ASIC cell
	accTopModule	Object is a top-level module
accNet		Object is declared as a net data type
	accSupply0	Object is declared as a supply0 net data type
	accSupply1	Object is declared as a supply1 net data type
	accTri	Object is declared as a tri net data type
	accTriand	Object is declared as a triand net data type
	accTrior	Object is declared as a trior net data type
	accTriage	Object is declared as a trireg net data type
	accTri0	Object is declared as a tri0 net data type
	accTri1	Object is declared as a tri1 net data type
	accWand	Object is declared as a wand net data type
	accWire	Object is declared as a wire net data type
	accWor	Object is declared as a wor net data type
accNetBit	accNetBit	Object is a bit-select of a net data type
accOperator	accOperator	Object is a Verilog HDL operator
accParameter		Object is a parameter
	accIntegerParam	Object is a parameter with an integer value
	accRealParam	Object is a parameter with a real value
	accStringParam	Object is a parameter with a string value
accPartSelect	accPartSelect	Object is a part-select of a vector

Table 18-25—List of all predefined type and fulltype constants (continued)

type constant	fulltype constant	Description
accPath		Object is a path
	accInterModPath	Object is an intermodule path
	accModPath	Object is a module path
accPathTerminal		Object is a terminal of a module path
	accPathInput	Object is an input terminal of a module path
	accPathOutput	Object is an output terminal of a module path
accPort		Object is a module port
	accConcatPort	Object is a module port concatenation
	accScalarPort	Object is a scalar module port
	accBitSelectPort	Object is a bit-select of a module port
	accPartSelectPort	Object is a part-select of a module port
	accVectorPort	Object is a vector module port
accPortBit	accPortBit	Object is a bit of a module port

Table 18-25—List of all predefined type and fulltype constants (continued)

type constant	fulltype constant	Description
accPrimitive		Object is a primitive
	accAndGate	Object is an and primitive
	accBufGate	Object is a buf primitive
	accBufif0Gate	Object is a bufif0 primitive
	accBufif1Gate	Object is a bufif1 primitive
	accCmosGate	Object is a cmos primitive
	accCombPrim	Object is a combinational logic UDP
	accNandGate	Object is a nand primitive
	accNmosGate	Object is an nmos primitive
	accNorGate	Object is a nor primitive
	accNotGate	Object is a not primitive
	accNotif0Gate	Object is a notif0 primitive
	accNotif1Gate	Object is a notif1 primitive
	accOrGate	Object is an or primitive
	accPmosGate	Object is a pmos primitive
	accPulldownGate	Object is a pulldown primitive
	accPullupGate	Object is a pullup primitive
	accRcmosGate	Object is an rcmos primitive
	accRnmosGate	Object is an rnmos primitive
	accRpmosGate	Object is an rpmos primitive
	accRtranGate	Object is an rtran primitive
	accRtranif0Gate	Object is an rtranif0 primitive
	accRtranif1Gate	Object is an rtranif1 primitive
	accSeqPrim	Object is a sequential logic UDP
	accTranGate	Object is a tran primitive
	accTranif0Gate	Object is a tranif0 primitive
	accTranif1Gate	Object is a tranif1 primitive
	accXnorGate	Object is an xnor primitive
	accXorGate	Object is an xor primitive
accRealVar	accRealVar	Object is declared as a real data type
accReg	accReg	Object is declared as a reg data type
accRegBit	accRegBit	Object is a bit-select of a reg data type

Table 18-25—List of all predefined type and fulltype constants (continued)

type constant	fulltype constant	Description
accSpecparam		Object is a specparam
	accIntegerParam	Object is a specparam with an integer value
	accRealParam	Object is a specparam with a real value
	accStringParam	Object is a specparam with a string value
accStatement	accStatement	Object is a procedural statement
	accNamedBeginStat	Object is a named begin statement
	accNamedForkStat	Object is a named fork statement
accSystemTask	accSystemTask	Object is a built-in system task
accSystemFunction	accSystemFunction	Object is a built-in system function with a scalar or vector return
accSystemRealFunction	accSystemRealFunction	Object is a built-in system function with a real value return
accTask	accTask	Object is a Verilog HDL task
accTchk		Object is a timing check
	accHold	Object is a \$hold timing check
	accNochange	Object is a \$nochange timing check
	accPeriod	Object is a \$period timing check
	accRecovery	Object is a \$recovery timing check
	accSetup	Object is a \$setup timing check
	accSetuphold	Object is a \$setuphold timing check
	accSkew	Object is a \$skew timing check
	accWidth	Object is a \$width timing check
accTchkTerminal	accTchkTerminal	Object is a timing check terminal
accTerminal		Object is a primitive terminal
	accInputTerminal	Object is a primitive input terminal
	accOutputTerminal	Object is a primitive output terminal
	accInoutTerminal	Object is a primitive inout terminal
accTimeVar	accTimeVar	Object is declared as a time data type
accUserTask	accUserTask	Object is a user-defined system task
accUserFunction	accUserFunction	Object is a user-defined system function with a scalar or vector return
accUserRealFunction	accUserRealFunction	Object is a user-defined system function with a real value return
accWirePath	accIntermodPath	Object is an intermodule path (from a module output to a module input)

18.7 Error handling

When an ACC routine detects an error, it shall perform the following operations:

- a) Set the global error flag **acc_error_flag** to non-zero
- b) Display an error message at run time to standard output
- c) Return an exception value

When an ACC routine is called, it automatically resets **acc_error_flag** to 0.

18.7.1 Suppressing error messages

By default, ACC routines shall display error messages. Error messages can be suppressed using the ACC routine **acc_configure()** to set the configuration parameter **accDisplayErrors** to "false".

18.7.2 Enabling warnings

By default, ACC routines shall not display warning messages. To enable warning messages, use the ACC routine **acc_configure()** to set the configuration parameter **accDisplayWarnings** to "true".

18.7.3 Testing for errors

If automatic error reporting is suppressed, error handling can be performed by checking the **acc_error_flag** explicitly after calling an ACC routine. This procedure is described in Figure 18-1.

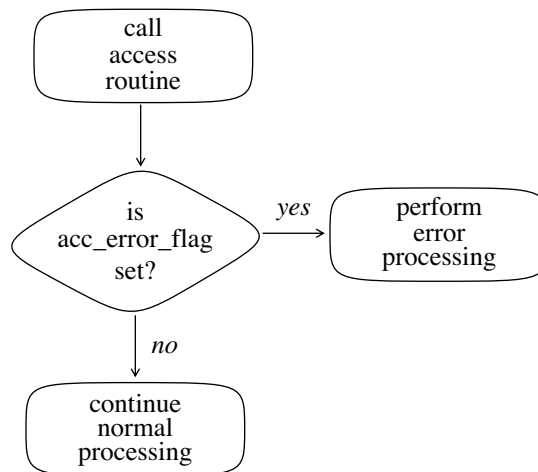


Figure 18-1—Using **acc_error_flag** to detect errors

18.7.4 Example

The following example shows a C-language application that performs error checking for ACC routines. This example uses **acc_configure()** to suppress automatic error reporting. Instead, it checks **acc_error_flag** explicitly and displays its own specialized error message.

```

#include  "acc_user.h"

check_new_timing()
{
    handle    gate_handle;

    /* initialize and configure access routines */
    acc_initialize();

    /* suppress error reporting by access routines */
    acc_configure( accDisplayErrors, "false" );

    /* set development version */
    acc_configure( accDevelopmentVersion, "IEEE 1364 PLI" );

    /* check type of first argument, the object */
    gate_handle = acc_handle_tfarg( 1 );

    /* check for valid argument */
    if (acc_error_flag)
        tf_error("Cannot derive handle from argument\n");
    else
        /* argument is valid */
        /* make sure it is a primitive */
        if ( acc_fetch_type(gate_handle) != accPrimitive )
            tf_error("Invalid argument type:not a primitive\n");
    acc_close();
}

```

18.7.5 Exception values

ACC routines shall return one of three exception values when an error occurs, unless specified differently in the syntax of a specific ACC routine.

Table 18-26—Exception values returned by ACC routines on errors

When routine returns	The exception value shall be
int or double values	0.0
pointers or handles	null
bool (boolean) values	false

Because ACC routines can return valid values that are the same as exception values, the only definitive way to detect errors explicitly is to check **acc_error_flag**.

Note that **null** and **false** are predefined constants, declared in **acc_user.h**.

18.8 Reading and writing delay values

This clause explains how ACC routines that read and modify delays are used. The ACC routines **acc_fetch_delays()**, **acc_replace_delays()**, and **acc_append_delays()** can read or modify delay values in a Verilog software implementation data structure. Refer to Section 19 for the complete syntax of each of these routines.

18.8.1 Number of delays for Verilog HDL objects

There are a variety of objects in a Verilog HDL source description that can model delays. These objects can have a single delay that represents all possible logic transitions, or multiple delays that represent different logic transitions. Table 18-27 lists the objects that can have delays and the number of different delays for each object.

Table 18-27—Number of possible delays for Verilog HDL objects

Verilog HDL Objects	Number of delays	Description
2-state primitives	1	One delay for: all transitions
	2	Separate delays for: rise, fall
3-state primitives	1	One delay for: all transitions
	2	Separate delays for: rise, fall
	3	Separate delays for: rise, fall, toZ
Module path delays	1	One delay for: all transitions
	2	Separate delays for: rise, fall
	3	Separate delays for: rise, fall, toZ
	6	Separate delays for: 0->1, 1->0, 0->Z, Z->1, 1->Z, Z->0
	12	Separate delays for: 0->1, 1->0, 0->Z, Z->1, 1->Z, Z->0, 0->X, X->1, 1->X, X->0, X->Z, Z->X
Module ports Module port bits Intermodule paths	3	Separate delays for: rise, fall, toZ
Timing checks	1	One delay for: timing limit

NOTE—The routine **acc_append_delays()** does not support intermodule path delays.

In addition to the number of delays, each delay can be represented as a single delay for each transition or as a minimum:typical:maximum delay set for each transition. Thus, a module path with 1 delay might have one value or three values, and a module path with 12 delays can have 12 delay values or 36 delay values.

18.8.2 ACC routine configuration

The PLI shall use configuration parameters to set up the delay ACC routines to work with the variations of Verilog objects and the number of possible delays. These parameters shall be set using the routine **acc_configure()**. The

parameters that configure the delay ACC routines are summarized in Table 18-28. How these configuration parameters are used is presented in 18.8.3. Refer to 19.6 for details on using **acc_configure()**.

Table 18-28— Configuration parameters for delay ACC routines

Configuration parameter	Description
accMinTypMaxDelays	When “false”, each delay shall be represented by one value. When “true”, each delay shall be represented by a min:typ:max value. The default shall be “false”.
accToHiZDelay	Set to “average”, “max”, “min”, or “from user”, which allows the delay modify ACC routines to calculate the toZ delay for 3-state objects when accPathDelayCount is set to “2”. The default shall be “from user”, which means the toZ delay shall not be calculated. Note that this parameter shall be ignored when accMinTypMaxDelays is set to “true”.
accPathDelayCount	Sets the number of delay arguments to be used by the ACC routines for module path delays. Shall be set to “1”, “2”, “3”, “6”, or “12”. The default shall be “6”.

18.8.3 Determining the number of arguments for ACC delay routines

The ACC routines **acc_fetch_delays()**, **acc_replace_delays()**, and **acc_append_delays()** shall require a different number of arguments based on

- The type of object handle
- The setting of configuration parameters

The following subclauses discuss how these factors affect the number of arguments for delay ACC routines.

18.8.3.1 Single delay value mode

When the configuration parameter **accMinTypMaxDelays** is “false” (the default), a single value shall be used for each delay transition. In this mode, the routines **acc_fetch_delays()**, **acc_replace_delays()**, and **acc_append_delays()** shall require each delay value as a separate argument. For **acc_replace_delays()** and **acc_append_delays()**, the arguments shall be a literal value of type double or variables of type double. For **acc_fetch_delays()**, the arguments shall be pointers to variables of type double.

The number of arguments required is determined by the type of object handle passed to the delay ACC routine, as shown in Table 18-29.

Table 18-29— Number of delay arguments in single delay mode

Object handle type	Configuration parameters	Number and order of delay arguments
Timing check		1 argument: timing check limit
2-state primitive		2 arguments: rise, fall transitions
3-state primitive Module port Module port bit Intermodule path	accToHiZDelay set to “min”, “max”, or “average”	2 arguments: rise, fall transitions (toZ delay is calculated; see 18.8.3.3)
	accToHiZDelay set to “from_user”	3 arguments: rise, fall, toZ transitions

Table 18-29—Number of delay arguments in single delay mode (continued)

Object handle type	Configuration parameters	Number and order of delay arguments
Module path	accPathDelayCount set to "1"	1 argument: all transitions
	accPathDelayCount set to "2"	2 arguments: rise, fall transitions
	accPathDelayCount set to "3"	3 arguments: rise, fall, toZ transitions
	accPathDelayCount set to "6"	6 arguments: 0->1, 1->0, 0->Z, Z->1, 1->Z, Z->0
	accPathDelayCount set to "12"	12 arguments: 0->1, 1->0, 0->Z, Z->1, 1->Z, Z->0 0->X, X->1, 1->X, X->0, X->Z, Z->X

18.8.3.2 Min:typ:max delay value mode

When the configuration parameter **accMinTypMaxDelays** is "true", a three-value set shall be used for each delay transition. In this mode, the routines **acc_fetch_delays()**, **acc_replace_delays()**, and **acc_append_delays()** shall require the delay argument to be a pointer of an array of variables of type double. The number of elements in the array shall be determined by the type of object handle passed to the delay ACC routine, as shown in Table 18-30.

Table 18-30—Number of delay elements in min:typ:max delay mode

Object handle type	Configuration parameters	Size and order of the delay array
Timing check		3 elements: array[0] = min limit array[1] = typ limit array[2] = max limit
2-state primitive 3-state primitive Module port Module port bit Intermodule path		9 elements: array[0] = min rise delay array[1] = typ rise delay array[2] = max rise delay array[3] = min fall delay array[4] = typ fall delay array[5] = max fall delay array[6] = min toZ delay array[7] = typ toZ delay array[8] = max toZ delay (an array of size 9 shall be declared, even if toZ delays are not used by the object)

Table 18-30—Number of delay elements in min:typ:max delay mode (continued)

Object handle type	Configuration parameters	Size and order of the delay array
Module path	accPathDelayCount set to "1"	3 elements: array[0] = min delay array[1] = typ delay array[2] = max delay
	accPathDelayCount set to "2"	6 elements: array[0] = min rise delay array[1] = typ rise delay array[2] = max rise delay array[3] = min fall delay array[4] = typ fall delay array[5] = max fall delay
	accPathDelayCount is set to "3"	9 elements: array[0] = min rise delay array[1] = typ rise delay array[2] = max rise delay array[3] = min fall delay array[4] = typ fall delay array[5] = max fall delay array[6] = min toZ delay array[7] = typ toZ delay array[8] = max toZ delay
	accPathDelayCount set to "6"	18 elements: array[0] = min 0->1 delay array[1] = typ 0->1 delay array[2] = max 0->1 delay array[3] = min 1->0 delay array[4] = typ 1->0 delay array[5] = max 1->0 delay array[6] = min 0->Z delay array[7] = typ 0->Z delay array[8] = max 0->Z delay array[9] = min Z->1 delay array[10] = typ Z->1 delay array[11] = max Z->1 delay array[12] = min 1->Z delay array[13] = typ 1->Z delay array[14] = max 1->Z delay array[15] = min Z->0 delay array[16] = typ Z->0 delay array[17] = max Z->0 delay

Table 18-30—Number of delay elements in min:typ:max delay mode (continued)

Object handle type	Configuration parameters	Size and order of the delay array
Module path (continued)	accPathDelayCount set to “12”	36 elements: array[0] = min 0->1 delay array[1] = typ 0->1 delay array[2] = max 0->1 delay array[3] = min 1->0 delay array[4] = typ 1->0 delay array[5] = max 1->0 delay array[6] = min 0->Z delay array[7] = typ 0->Z delay array[8] = max 0->Z delay array[9] = min Z->1 delay array[10] = typ Z->1 delay array[11] = max Z->1 delay array[12] = min 1->Z delay array[13] = typ 1->Z delay array[14] = max 1->Z delay array[15] = min Z->0 delay array[16] = typ Z->0 delay array[17] = max Z->0 delay array[18] = min 0->X delay array[19] = typ 0->X delay array[20] = max 0->X delay array[21] = min X->1 delay array[22] = typ X->1 delay array[23] = max X->1 delay array[24] = min 1->X delay array[25] = typ 1->X delay array[26] = max 1->X delay array[27] = min X->0 delay array[28] = typ X->0 delay array[29] = max X->0 delay array[30] = min X->Z delay array[31] = typ X->Z delay array[32] = max X->Z delay array[33] = min Z->X delay array[34] = typ Z->X delay array[35] = max Z->X delay

18.8.3.3 Calculating turn-off delays from rise and fall delays

In single delay mode (**accMinTypMaxDelays** set to “false”), the routines **acc_replace_delays()** and **acc_append_delays()** can be instructed to calculate automatically the turn-off delays from rise and fall delays. How the calculation shall be performed is controlled by the configuration parameter **accToHiZDelay**, as shown in Table 18-31.

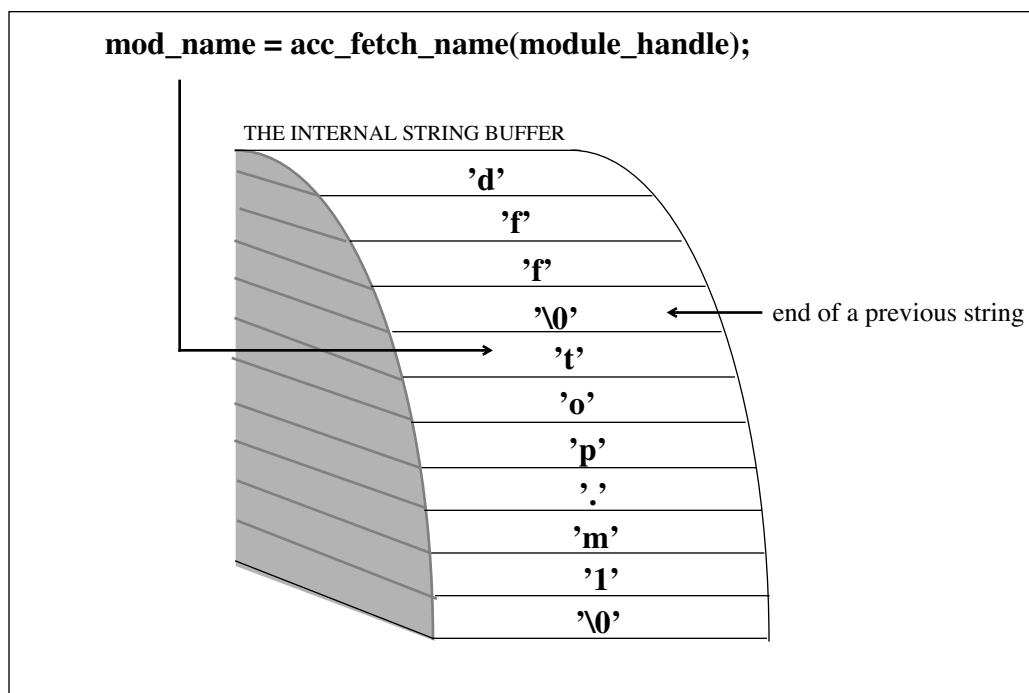
Table 18-31—Configuring accToHiZDelay to determine the toZ delay

Configuration of accToHiZDelay	Value of the toZ delay
"average"	The toZ turn-off delay shall be the average of the rise and fall delays.
"min"	The toZ turn-off delay shall be the smaller of the rise and fall delays.
"max"	The toZ turn-off delay shall be the larger of the rise and fall delays.
"from_user" (the default)	The toZ turn-off delay shall be set to the value passed as a user-supplied argument.

18.9 String handling

18.9.1 ACC routines share an internal string buffer

ACC routines that return pointers to strings can share an internal buffer to store string values. These routines shall return a pointer to the location in the buffer that contains the first character of the string, as illustrated in Figure 18-2. In this example, `mod_name` points to the location in the buffer where `top.m1` (the name of the module associated with `module_handle`) is stored.

**Figure 18-2—How ACC routines store strings in the internal buffer**

18.9.2 String buffer reset

ACC routines shall place strings at the next available sequential location in the string buffer, which stores at least 4096 characters. If there is not enough room to store an entire string starting at the next location, a condition known as *buffer reset* shall occur.

When buffer reset occurs, ACC routines shall place the next string starting at the beginning of the buffer, overwriting data already stored there. The result can be a loss of data, as illustrated in Figure 18-3.

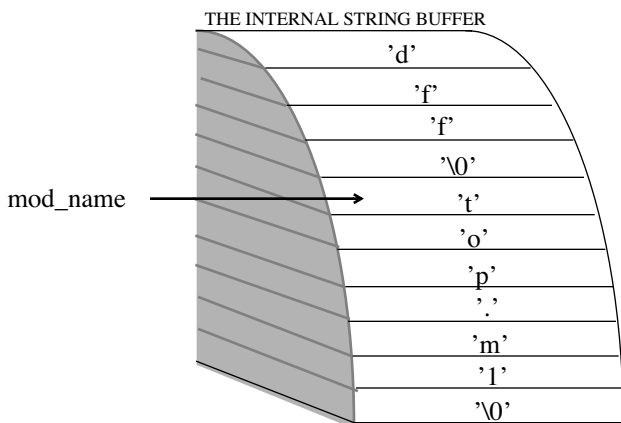
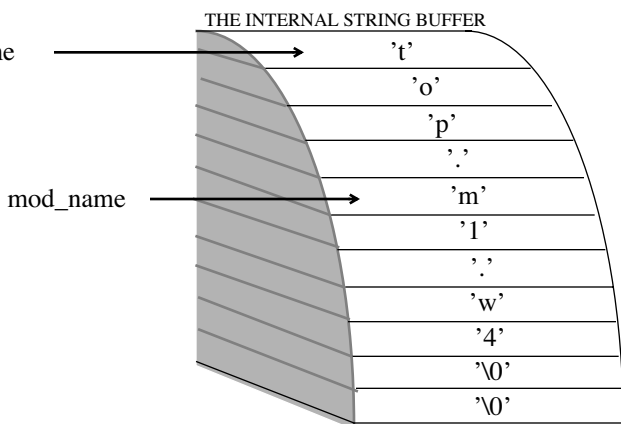
<i>Action:</i>	<i>Results:</i>
<p>mod_name = acc_fetch_fullname(module_handle);</p>  <p>THE INTERNAL STRING BUFFER</p> <p>mod_name</p> <p>'d'</p> <p>'f'</p> <p>'f'</p> <p>'\0'</p> <p>'t'</p> <p>'o'</p> <p>'p'</p> <p>'.'</p> <p>'m'</p> <p>'l'</p> <p>'\0'</p>	<p>mod_name points to the string "top.ml".</p> <p>The string happens to be stored near the end of the buffer.</p>
<p>net_name = acc_fetch_fullname(net_handle);</p>  <p>THE INTERNAL STRING BUFFER</p> <p>net_name</p> <p>mod_name</p> <p>'t'</p> <p>'o'</p> <p>'p'</p> <p>'.'</p> <p>'m'</p> <p>'l'</p> <p>'w'</p> <p>'4'</p> <p>'\0'</p> <p>'\0'</p>	<p>acc_fetch_fullname() cannot place the next string at the end of the buffer. Therefore, a buffer reset occurs.</p> <p>net_name points to the string "top.ml.w4"</p> <p>The data at the beginning of the buffer is overwritten; The old mod_name pointer now points to corrupted data, which in this example is "ml.w4".</p>

Figure 18-3—Buffer reset causes data in the string buffer to be overwritten

18.9.2.1 The buffer reset warning

ACC routines shall issue a warning whenever the internal string buffer resets. To view the warning message, the configuration parameter **accDisplayWarnings** shall be set to **true**, using the ACC routine **acc_configure()**.

18.9.3 Preserving string values

Applications that use strings immediately—for example, to print names of objects—do not need to be concerned about overwrites after a string buffer reset. Applications that have to preserve string values while calling other ACC routines that write to the string buffer should preserve the string value before it is overwritten. To preserve a string value, the C routine `strcpy` can be used to copy the string to a local character array.

18.9.4 Example of preserving string values

The following example code illustrates preserving string values. If the module in this example contains many cells, one of the calls to `acc_fetch_name()` could eventually overwrite the module name in the string buffer with a cell name. To preserve the module name, `strcpy` is used to store it locally in an array called `mod_name`.

```
#include "acc_user.h"
#define NAME_SIZE 256
void display_cells_in_module(mod)
handle mod;
{
    handle cell;
    char mod_name[NAME_SIZE];
    /*save the module name in local buffer mod_name*/
    strcpy( mod_name, acc_fetch_fullname( mod ) );
    cell = null;
    while (cell = acc_next_cell( mod, cell ) )
        io_printf( "%s.%s\n", mod_name, acc_fetch_name( cell ) );
}
```

strcpy saves the full module name in array **mod_name**

the ACC routine call is passed as the second argument to **strcpy**

18.10 Using VCL ACC routines

The VCL routines add or delete value change monitors on a specified object. Then, whenever the object changes logic value or strength, a PLI consumer routine shall be called.

The ACC routine `acc_vcl_add()` adds a value change monitor on an object in the Verilog HDL source description. The arguments for `acc_vcl_add()` specify

- A handle to an object in the Verilog HDL structure
- The name of a consumer routine
- A user_data value
- A VCL reason_flag

The following example illustrates the usage of `acc_vcl_add()`.

```
acc_vcl_add(net, netmon_consumer, net_name, vcl_verilog_logic);
```

The purpose of each of these arguments is described in the following paragraphs. Refer to 19.97 for the full syntax and usage of `acc_vcl_add()` and its arguments.

The *handle* argument shall be a handle to any object type in the list in 18.10.1.

The *consumer routine* argument shall be the name of a C application that shall be called for the reasons specified by the *reason_flag*, such as a logic value change. When a consumer routine is called, it shall be passed a pointer to a C record, called **vc_record**. This record shall contain information about the object, including the simulation time of the change and the new logic value of the object. The **vc_record** is defined in the file **acc_user.h** and is listed in Figure 18-4.

The *user_data* argument shall be a character pointer. The value of the *user_data* argument shall be passed to the consumer routine as part of the **vc_record**. The *user_data* argument can be used to pass a single value to the consumer routine, or it can be used to pass a pointer to information. For example, the name of the object could be stored in a global character string array, and a pointer to that array could be passed as the *user_data* argument. The consumer routine could then have access to the object name. Another example is to create a global user-defined structure with several values that need to be passed to the consumer routine. A pointer to a variable using the user-defined structure is then passed as the *user_data* argument. Note that the *user_data* argument is defined as a character pointer; therefore, any other data type should be cast to a character pointer.

The VCL *reason_flag* argument is one of two predefined constants that sets up the VCL callback mechanism to call the consumer routine under specific circumstances. The constant **vcl_verilog_logic** sets up the VCL to call the consumer routine whenever the monitored object changes logic value. The constant **vcl_verilog_strength** sets up the VCL to call the consumer routine when the monitored object changes logic value or logic strength.

An object can have any number of VCL monitors associated with it, as long as each monitor is unique in some way. VCL monitors can be deleted using the ACC routine **acc_vcl_delete()**.

18.10.1 VCL objects

The VCL shall monitor value changes for the following objects:

- Scalar, vector, and bit-selects of registers
- Scalar, vector, and bit-selects of nets
- Integer, real, and time variables
- Module ports
- Primitive output or inout terminals
- Events

18.10.2 The VCL record definition

Each time a consumer routine is called, it shall be passed a pointer to a record structure called **vc_record**. This structure shall contain information about the most recent change that occurred on the monitored object. The **vc_record** structure is defined in **acc_user.h** and is listed in Figure 18-4.

```
typedef struct t_vc_record
{
    int      vc_reason;
    int      vc_hightime;
    int      vc_lowtime;
    char     *user_data;
    union    {
        unsigned char logic_value;
        double         real_value;
        handle         vector_handle;
        s_strengths    strengths_s;
    } out_value;
} s_vc_record, *p_vc_record;
```

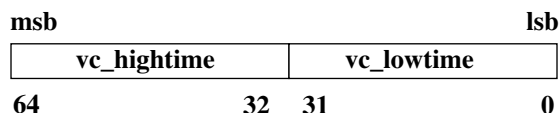
Figure 18-4—The VCL **s_vc_record** structure

The *vc_reason* field of **vc_record** shall contain a predefined integer constant that shall describe what type of change occurred. The constants that can be passed in the *vc_reason* field are described in Table 18-32.

Table 18-32—Predefined *vc_reason* constants

Predefined <i>vc_reason</i> constant	Description
logic_value_change	A scalar net or bit-select of a vector net changed logic value.
strength_value_change	A scalar net or bit-select of a vector net changed logic value or strength.
vector_value_change	A vector net or part-select of a vector net changed logic value.
sregister_value_change	A scalar register changed logic value.
vregister_value_change	A vector register or part-select of a vector register changed logic value.
integer_value_change	An integer variable changed value.
real_value_change	A real variable changed value.
time_value_change	A time variable changed value.
event_value_change	An event variable changed.

The *vc_hightime* and *vc_lowtime* fields of **vc_record** shall be 32-bit integers that shall contain the simulation time the change occurred, as follows:



The *user_data* field of **vc_record** shall be a character pointer, and it shall contain the value specified as the *user_data* argument in the **acc_vcl_add()** ACC routine.

The *out_value* field of **vc_record** shall be a union of several data types. Only one data type shall be passed in the structure, based on the reason the callback occurred, as shown Table 18-33.

Table 18-33—Predefined *out_value* constants

If <i>vc_reason</i> is	The <i>out_value</i> shall be a type of	Description
logic_value_change	unsigned char	A predefined constant, from the following: vcl0 vcl1 vclX vclx vclZ vclz
strength_value_change	s_strengths structure pointer	A structure with logic and strength, as shown in Figure 18-5
vector_value_change	handle	A handle to a vector net or part-select of a vector net
sregister_value_change	unsigned char	A predefined constant, from the following: vcl0 vcl1 vclX vclx vclZ vclz
vregister_value_change	handle	A handle to a vector register or part-select of a vector register

Table 18-33—Predefined out_value constants (continued)

If vc_reason is	The out_value shall be a type of	Description
integer_value_change	handle	A handle to an integer variable
real_value_change	double	The value of a real variable
time_value_change	handle	A handle to a time variable
event_value_change	none	Event types have no value

When the *vc_reason* field of the *vc_record* is **strength_value_change**, a pointer to an *s_strengths* structure shall be passed in the *out_value* field of *vc_record*. This structure shall contain three fields, as shown in Figure 18-5.

```
typedef struct t_strengths
{
    unsigned char logic_value;
    unsigned char strength1;
    unsigned char strength2;
} s_strengths;
```

Figure 18-5—The VCL s_strengths structure

The values of the *s_strengths* structure fields are defined in Table 18-34.

Table 18-34—Predefined out_value constants

If s_strengths is	C data type	Description
logic_value	unsigned char	A predefined constant, from the following: vcl0 vcl1 vclX vclx vclZ vclz
strength1 strength2	unsigned char	A predefined constant, from the following: vclSupply vclWeak vclStrong vclMedium vclPull vclSmall vclLarge vclHighZ

The *strength1* and *strength2* fields of the *s_strengths* structure can represent

- A known strength—when *strength1* and *strength2* contain the same value, the signal strength shall be that value.
- An ambiguous strength with a known *logic_value*—when *strength1* and *strength2* contain different values and the *logic_value* contains either **vcl0** or **vcl1**, the signal strength shall be an ambiguous strength, where the *strength1* value shall be the maximum possible strength and *strength2* shall be the minimum possible strength.
- An ambiguous strength with an unknown *logic_value*—when *strength1* and *strength2* contain different values and the *logic_value* contains **vclX**, the signal strength shall be an ambiguous strength, where the *strength1* value shall be the logic 1 component and *strength2* shall be the logic 0 component.

18.10.3 Affects of `acc_initialize()` and `acc_close()` on VCL consumer routines

The ACC routines `acc_initialize()` and `acc_close()` shall reset all configuration parameters set by the routine `acc_configure()` back to default values. Care should be taken to ensure that the VCL consumer routine does not depend on any configuration parameters, as these parameters might not have the same value when a VCL callback occurs. Refer to 19.6 on `acc_configure()` for a list of routines that are affected by configuration parameters.

18.10.4 An example of using VCL ACC routines

The following example contains three PLI routines: a checktf application, a calltf application, and a consumer routine. The example is based on the checktf and calltf applications both being associated with two user-defined system tasks, using the PLI interface mechanism described in Section 17.

```
$net_monitor(<net_name>,<net_name>*);
$net_monitor_off(<net_name>,<net_name>*);
```

The checktf application, `netmon_checktf`, is shown below. This application performs syntax checking on instances of the user-defined system tasks to ensure there is at least one argument and that the arguments are valid net names.

```
#include "veriusers.h"
#include "acc_user.h"

int netmon_checktf()
{
    int i;
    int arg_cnt = tf_nump();

    /* initialize the environment for access routines */
    acc_initialize();

    /* set the development version */
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /* check number and type of task/function arguments */
    if (tf_nump() == 0)
        tf_error("$net_monitor[_off] must have at least one argument");
    else
        for (i = 1; i <= arg_cnt; i++)
            if (acc_fetch_type(acc_handle_tfarg(1)) != accNet)
                tf_error("$net_monitor[_off] arg %d is not a net type",i);

    acc_close();
}
```

The calltf application, `netmon_calltf`, follows. This application gets a handle to each task function argument and either adds or deletes a VCL monitor on the net. The application checks the data C argument associated with each system task name to determine whether the application was called by `$net_monitor` or `$net_monitor_off`.

```
#include "veriusers.h"
#include "acc_user.h"
#include <malloc.h>

int netmon_calltf(data)
int data;
{
    handle net;
    int netmon_consumer();
    char *net_name;
    int tfnum;

    #define ADD      0 /* data value associated with $net_monitor */
    #define DELETE  1 /* data value associated with $net_monitor_off */

    /* initialize the environment for access routines */
    acc_initialize();

    /* set the development version */
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    switch (data) /* see which system task name called this application */
    {
        case ADD: /* called by $net_monitor */
            /* add a VCL flag to each net in the task/function argument list */
            tfnum = 1;
            while ((net = acc_handle_tfarg(tfnum++)) != null)
            {
                /* allocate memory for the net name of this argument */
                net_name = malloc(strlen+1(acc_fetch_name(net)));

                /* preserve the net name in a character string */
                strcpy(net_name, acc_fetch_name(net));

                /* add a VCL monitor; pass net name pointer as user_data argument*/
                acc_vcl_add(net, netmon_consumer, net_name, vcl_verilog_logic);
            }
            break;

        case DELETE: /* called by $net_monitor_off */
            tfnum = 1;
            while ((net = acc_handle_tfarg(tfnum++)) != null)
            {
                /* delete the VCL monitor */
                acc_vcl_delete(net, netmon_consumer, net_name, vcl_verilog);
            }
            break;
    }
    acc_close();
}
```

The consumer routine, `netmon_consum`, is shown in the following example. The consumer routine is called by the VCL callback mechanism. Since the `checktf` application only permits net data types to be used, the consumer routine only needs to check for scalar and vector net value changes when it is called.

```
#include "veriusers.h"
#include "acc_user.h"

int netmon_consumer(vc_record)
p_vc_record vc_record; /* record type passed to consumer routine */
{
    char    net_value, value;
    handle  vector_value;

    /* check reason VCL call-back occurred */
    switch (vc_record->vc_reason)
    {
        case logic_value_change : /* scalar signal changed logic value */
        {
            net_value = vc_record->out_value.logic_value;
            /* convert logic value constant to a character for printing */
            switch (net_value)
            {
                case vcl0 : value = '0'; break;
                case vcl1 : value = '1'; break;
                case vclX : value = 'X'; break;
                case vclZ : value = 'Z'; break;
            }
            io_printf("%d : %s = %c\n",
                    vc_record->vc_lowtime, vc_record->user_data, value);
            break;
        }
        case vector_value_change : /* vector signal changed logic value */
        {
            vector_value = vc_record->out_value.vector_handle;
            io_printf("%d : %s = %s\n",
                    vc_record->vc_lowtime, vc_record->user_data,
                    acc_fetch_value(vector_value, "%b") );
            break;
        }
    }
}
```

Section 19

ACC routine definitions

This section describes the PLI access (ACC) routines, explaining their function, syntax, and usage. The routines are listed in alphabetical order.

The following conventions are used in the definitions of the PLI routines described in sections 19, 21, and 23.

Synopsis: A brief description of the PLI routine functionality, intended to be used as a quick reference when searching for PLI routines to perform specific tasks.

Syntax: The exact name of the PLI routine and the order of the arguments passed to the routine.

Returns: The definition of the value returned when the PLI routine is called, along with a brief description of what the value represents. The return definition contains the fields

- **Type:** The data type of the C value that is returned. The data type is either a standard ANSI C type or a special type defined within the PLI.
- **Description:** A brief description of what the value represents.

Arguments: The definition of the arguments passed with a call to the PLI routine. The argument definition contains the fields

- **Type:** The data type of the C values that are passed as arguments. The data type is either a standard ANSI C type, or a special type defined within the PLI.
- **Name:** The name of the argument used in the Syntax definition.
- **Description:** A brief description of what the value represents.

All arguments shall be considered mandatory unless specifically noted in the definition of the PLI routine. Two tags are used to indicate arguments that may not be required:

- **Conditional:** Arguments tagged as conditional shall be required only if a previous argument is set to a specific value, or if a call to another PLI routine has configured the PLI to require the arguments. The PLI routine definition explains when conditional arguments are required.
- **Optional:** Arguments tagged as optional may have default values within the PLI, but they may be required if a previous argument is set to a specific value, or if a call to another PLI routine has configured the PLI to require the arguments. The PLI routine definition explains the default values and when optional arguments are required.

Related routines: A list of PLI routines that are typically used with, or provide similar functionality to, the PLI routine being defined. This list is provided as a convenience to facilitate finding information in this standard. It is not intended to be all-inclusive, and it does not imply that the related routines have to be used.

19.1 acc_append_delays()

acc_append_delays() for single delay values (accMinTypMaxDelays set to "false")			
Synopsis:	Add delays to existing delay on primitives, module paths, timing checks, and module input ports.		
Syntax: Primitives Ports Port bits	acc_append_delays(object_handle, rise_delay, fall_delay, z_delay)		
Module paths	acc_append_delays(object_handle, d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12)		
Timing checks	acc_append_delays(object_check_handle, limit)		
Type		Description	
Returns:	bool	1 if successful; 0 if an error occurred	
Arguments:	Type	Name	Description
	handle	object_handle	Handle of a primitive, module path, timing check, module input port or bit of a module input port
Conditional	double	rise_delay fall_delay	Rise and fall transition delay for 2-state primitives, 3-state primitives, module input ports, and module input port bits
	double	z_delay	If accToHiZDelay is set to "from_user": turn-off (to Z) transition delay for 3-state primitives, module input ports, and module input port bits
	double	d1	If accPathDelayCount is set to "1": delay for all transitions for module paths If accPathDelayCount is set to "2" or "3": rise transition delay for module paths If accPathDelayCount is set to "6" or "12": 0→1 transition delay for module paths
Conditional	double	d2	If accPathDelayCount is set to "2" or "3": fall transition delay for module paths If accPathDelayCount is set to "6" or "12": 1→0 transition delay for module paths
Conditional	double	d3	If accPathDelayCount is set to "3": turn-off transition delay for module paths If accPathDelayCount is set to "6" or "12": 0→Z transition delay for module paths
Conditional	double	d4 d5 d6	If accPathDelayCount is set to "6" or "12": d4 is Z→1 transition delay for module paths d5 is 1→Z transition delay for module paths d6 is Z→0 transition delay for module paths
Conditional	double	d7 d8 d9 d10 d11 d12	If accPathDelayCount is set to "12": d7 is 0→X transition delay for module paths d8 is X→1 transition delay for module paths d9 is 1→X transition delay for module paths d10 is X→0 transition delay for module paths d11 is X→Z transition delay for module paths d12 is Z→X transition delay for module paths
	double	limit	Limit of timing check

acc_append_delays() for min:typ:max delays (accMinTypMaxDelays set to "true")			
Synopsis:	Add min:typ:max delay values to existing delay values for primitives, module paths, timing checks or module input ports; the delay values are contained in an array.		
Syntax:	acc_append_delays(object_handle, array_ptr)		
Type		Description	
Returns:	bool	1 if successful; 0 if an error is encountered	
Type		Name	Description
Arguments:	handle	object_handle	Handle of a primitive, module path, timing check, module input port or bit of a module input port
	double address	array_ptr	Pointer to array of min:typ:max delay values; the size of the array depends on the type of object and the setting of accPathDelayCount (see 18.8)

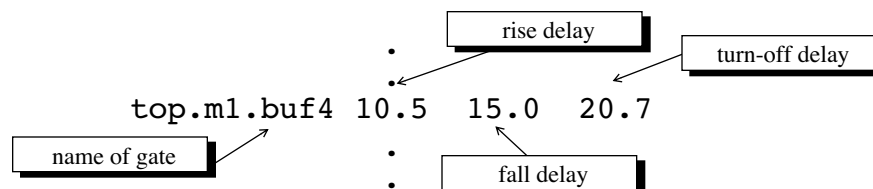
The ACC routine `acc_append_delays()` shall work differently depending on how the configuration parameter `accMinTypMaxDelays` is set. When this parameter is set to `false`, a single delay per transition shall be assumed, and delays shall be passed as individual arguments. For this single delay mode, the first syntax table in this clause shall apply.

When **accMinTypMaxDelays** is set to **true**, **acc_append_delays()** shall pass one or more sets of minimum:typical:maximum delays contained in an array, rather than single delays passed as individual arguments. For this min:typ:max delay mode, the second syntax table in this clause shall apply.

The number delay values appended by **acc_append_delays()** shall be determined by the type of object and the setting of configuration parameters. Refer to 18.8 for a description of how the number of delay values are determined.

The **acc_append_delays()** routine shall write delays in the timescale of the module that contains the object_handle.

The example shown in Figure 19-1 is an example of backannotation. It reads new delay values from a file called `primdelay.dat` and uses `acc_append_delays()` to add them to the current delays on a gate. The format of the file is shown below.



```

#include <stdio.h>
#include "acc_user.h"

write_gate_delays()
{
    FILE      *infile;
    char      full_gate_name[NAME_SIZE];
    double    rise,fall,toz;
    handle    gate_handle;

    /*initialize the environment for ACC routines*/
    acc_initialize();

    /*set development version*/
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*read delays from file - "r" means read only*/
    infile = fopen("primdelay.dat","r");
    while(fscanf(infile, "%s %lf %lf %lf",
                  full_gate_name, rise, fall, toz) != EOF)
    {

        /*get handle for the gate*/
        gate_handle = acc_handle_object(full_gate_name);

        /*add new delays to current values for the gate*/
        acc_append_delays(gate_handle, rise, fall, toz);
    }
    acc_close();
}

```

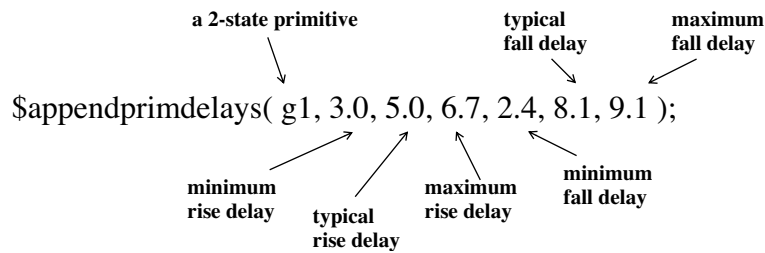
Figure 19-1—Using `acc_append_delays()` in single delay value mode

The example shown in Figure 19-2 shows how to append min:typ:max delays for a 2-state primitive (no high-impedance state). The C application follows these steps:

- Declares an array of nine double-precision floating-point values to hold three sets of min:typ:max values, one set each for rising transitions, falling transitions, and transitions to Z.
- Sets the configuration parameter **accMinTypMaxDelays** to **true** to instruct **acc_append_delays()** to write delays in min:typ:max format.
- Calls **acc_append_delays()** with a valid primitive handle and the array pointer.

Since the primitive to be used in this example does not have a high-impedance state, **acc_append_delays()** automatically appends just the rise and fall delay value sets. The last three array elements for the toZ delay values are not used. However, even though the last three array elements are not used with a 2-state primitive, the syntax for using min:typ:max delays requires that the array contain all nine elements.

For this example, the C application, `append_mintypmax_delays`, is associated through the ACC interface mechanism with a user-defined system task called `$appendprimdelays`. A primitive with no Z state and new delay values are passed as task/function arguments to `$appendprimdelays` as follows:



```
#include "acc_user.h"

append_mintypmax_delays()
{
    handle    prim;
    double    delay_array[9];
    int       i;

    acc_configure(accMinTypMaxDelays, "true");

    /* get handle for primitive */
    prim = acc_handle_tfarg(1);

    /* store new delay values in array */
    for (i = 0; i < 9; i++)
        delay_array[i] = acc_fetch_tfarg(i+2);

    /* append min:typ:max delays */
    acc_append_delays(prim, delay_array);
}
```

delay_array has to be large enough to hold *nine* values to handle both 2-state primitives and 3-state primitives

Figure 19-2—Using `acc_append_delays()` in min:typ:max mode

19.2 acc_append_pulsere()

acc_append_pulsere()			
Synopsis:	Add delays to existing pulse handling <i>reject_value</i> and <i>e_value</i> for a module path.		
Syntax:	acc_append_pulsere(path,r1,e1, r2,e2, r3,e3, r4,e4, r5,e5, r6,e6, r7,e7, r8,e8, r9,e9, r10,e10, r11,e11, r12,e12)		
Type		Description	
Returns:	bool	1 if successful; 0 if an error is encountered	
Type		Name	Description
Arguments:	handle	path	Handle of module path
	double	r1...r12	<i>reject_limit</i> values; the number of arguments is determined by accPathDelayCount
	double	e1...e12	<i>e_limit</i> values; the number of arguments is determined by accPathDelayCount
Related routines:	Use acc_fetch_pulsere() to get current pulse handling values Use acc_replace_pulsere() to replace existing pulse handling values Use acc_set_pulsere() to set pulse handling values as a percentage of the path delay		

The ACC routine **acc_append_pulsere()** shall add to an existing pulse handling *reject_limit* value and *e_limit* value for a module path. The *reject_limit* and *e_limit* values are used to control how pulses are propagated through module paths.

A *pulse* is defined as two transitions on the same path that occur in a shorter period of time than the path delay. Pulse control values determine whether a pulse should be rejected, propagated through to the output, or considered an error. The pulse control values consist of a *reject_limit* and an *e_limit* pair of values, where

- The *reject_limit* shall set a threshold for determining when to reject a pulse—any pulse less than the *reject_limit* shall not propagate to the output of the path.
- The *e_limit* shall set a threshold for determining when a pulse is considered to be an error—any pulse less than the *e_limit* and greater than or equal to the *reject_limit* shall propagate a logic **x** to the path output.
- A pulse that is greater than or equal to the *e_limit* shall propagate through to the path output.

Table 19-1 illustrates the relationship between the *reject_limit* and the *e_limit*.

Table 19-1 — Path pulse control example

When	The pulse on a module path output shall be
<i>reject_limit</i> = 10.5 <i>e_limit</i> = 22.6	Rejected if < 10.5 An error if ≥ 10.5 and < 22.6 Passed if ≥ 22.6

The following rules shall apply when specifying pulse handling values:

- a) The value of *reject_limit* shall be less than or equal to the value of *e_limit*.

- b) The reject_limit and e_limit shall not be greater than the path delay.

The number of pulse control values that **acc_append_pulsere()** sets shall be controlled using the ACC routine **acc_configure()** to set the delay count configuration parameter **accPathDelayCount**, as shown in Table 19-2.

Table 19-2—How the value of accPathDelayCount affects acc_append_pulsere()

When accPathDelayCount is	acc_append_pulsere() shall write
"1"	One pair of reject_limit and e_limit values: one pair for all transitions, r1 and e1
"2"	Two pairs of reject_limit and e_limit values: one pair for rise transitions, r1 and e1 one pair for fall transitions, r2 and e2
"3"	Three pairs of reject_limit and e_limit values: one pair for rise transitions, r1 and e1 one pair for fall transitions, r2 and e2 one pair for turn-off transitions, r3 and e3
"6" (the default)	Six pairs of reject_limit and e_limit values—a different pair for each possible transition among 0, 1, and Z: one pair for 0->1 transitions, r1 and e1 one pair for 1->0 transitions, r2 and e2 one pair for 0->Z transitions, r3 and e3 one pair for Z->1 transitions, r4 and e4 one pair for 1->Z transitions, r5 and e5 one pair for Z->0 transitions, r6 and e6
"12"	Twelve pairs of reject_limit and e_limit values—a different pair for each possible transition among 0, 1, X, and Z: one pair for 0->1 transitions, r1 and e1 one pair for 1->0 transitions, r2 and e2 one pair for 0->Z transitions, r3 and e3 one pair for Z->1 transitions, r4 and e4 one pair for 1->Z transitions, r5 and e5 one pair for Z->0 transitions, r6 and e6 one pair for 0->X transitions, r7 and e7 one pair for X->1 transitions, r8 and e8 one pair for 1->X transitions, r9 and e9 one pair for X->0 transitions, r10 and e10 one pair for X->Z transitions, r11 and e11 one pair for Z->X transitions, r12 and e12

The minimum number of pairs of reject_limit and e_limit arguments to pass to **acc_append_pulsere()** has to equal the value of **accPathDelayCount**. Any unused reject_limit and e_limit argument pairs shall be ignored by **acc_append_pulsere()** and can be dropped from the argument list.

If **accPathDelayCount** is not set explicitly, it shall default to six; therefore, six pairs of pulse reject_limit and e_limit arguments have to be passed when **acc_append_pulsere()** is called. Note that the value assigned to **accPathDelayCount** also affects **acc_append_delays()**, **acc_fetch_delays()**, **acc_replace_delays()**, **acc_fetch_pulsere()**, and **acc_replace_pulsere()**.

Pulse control values shall be appended using the timescale of the module that contains the path.

19.3 **acc_close()**

acc_close()		
Synopsis:	Free internal memory used by ACC routines; reset all configuration parameters to default values.	
Syntax:	acc_close()	
	Type	Description
Returns:	void	No return
	Type	Name Description
Arguments:	None	
Related routines:	Use acc_initialize() to initialize the ACC routine environment	

The ACC routine **acc_close()** shall free internal memory used by ACC routines and reset all configuration parameters to default values. No other ACC routines should be called after calling **acc_close()**; in particular, ACC routines that are affected by **acc_configure()** should not be called.

Potentially, multiple PLI applications running in the same simulation session can interfere with each other because they share the same set of configuration parameters. To guard against application interference, both **acc_initialize()** and **acc_close()** reset all configuration parameters to their default values.

The example shown in Figure 19-3 presents a C-language routine that calls **acc_close()** before exiting.

```
#include "acc_user.h"

show_versions()
{
    handle    module_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*show version of ACC routines and simulator */
    io_printf("Running %s with %s\n",acc_version(),acc_product_version() );

    acc_close();
}
```

Figure 19-3—Using acc_close()

19.4 **acc_collect()**

acc_collect()			
Synopsis:	Obtain an array of handles for all objects related to a particular reference object; get the number of objects collected.		
Syntax:	<code>acc_collect(acc_next_routine_name, object_handle, number_of_objects)</code>		
Type		Description	
Returns:	handle array address	An address pointer to an array of handles of the objects collected	
Arguments:	Type	Name	Description
	pointer to acc_next_routine	acc_next_routine_name	Actual name (unquoted) of the acc_next_routine that finds the objects to be collected
	handle	object_handle	Handle of the reference object for acc_next_routine
	int *	number_of_objects	Integer pointer where the count of objects collected shall be written
Related routines:	All acc_next_routines Use acc_free() to free memory allocated by acc_collect()		

The ACC routine **acc_collect()** shall scan through a reference object, such as a module, and collect handles to all occurrences of a specific target object. The collection of handles shall be stored in an array, which can then be used by other ACC routines.

The object associated with object_handle shall be a valid type of handle for the reference object required by the acc_next routine to be called.

The routine **acc_collect()** should be used in the following situations:

- To retrieve data that can be used more than once
- Instead of using nested or concurrent calls to **acc_next_loconn()**, **acc_next_hiconn()**, **acc_next_load()**, and **acc_next_cell_load()** routines

Otherwise, it can be more efficient to use the an acc_next_routine directly.

The routine **acc_collect()** shall allocate memory for the array of handles it returns. When the handles are no longer needed, the memory can be freed by calling the routine **acc_free()**.

The ACC routine **acc_next_topmod()** does not work with **acc_collect()**. However, top-level modules can be collected by passing **acc_next_child()** with a null reference object argument. For example:

```
acc_collect(acc_next_child, null, &count);
```

The example shown in Figure 19-4 presents a C-language routine that uses **acc_collect()** to collect and display all nets in a module.

```
#include "acc_user.h"

display_nets()
{
    handle    *list_of_nets, module_handle;
    int       net_count, i;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*set development version*/
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for the module*/
    module_handle = acc_handle_tfarg(1);

    /*collect all nets in the module*/
    list_of_nets = acc_collect(acc_next_net, module_handle, &net_count);

    /*display names of net instances*/
    for(i=0; i < net_count; i++)
        io_printf("Net name is: %s\n", acc_fetch_name(list_of_nets[i]));

    /*free memory used by array list_of_nets*/
    acc_free(list_of_nets);

    acc_close();
}
```

Figure 19-4—Using acc_collect()

19.5 acc_compare_handles()

acc_compare_handles()			
Synopsis:	Determine if two handles refer to the same object.		
Syntax:	acc_compare_handles(handle1, handle2)		
Returns:	Type	Description	
	bool	true if handles refer to the same object; false if different objects	
Arguments:	Type	Name	Description
	handle	handle1	Handle to any objects
	handle	handle2	Handle to any objects

The ACC routine **acc_compare_handles()** shall determine if two handles refer to the same object. In some cases, two different handles might reference the same object if each handle is retrieved in a different way—for example, if an **acc_next** routine returns one handle and **acc_handle_object()** returns the other.

The C `==` operator cannot be used to determine if two handles reference the same object.

```
if (handle1 == handle2)    /* this does not work */
```

The example shown in Figure 19-5 uses **acc_compare_handles()** to determine if a primitive drives the specified output of a scalar port of a module.

```
#include "acc_user.h"

bool  prim_drives_scalar_port(prim, mod, port_num)
handle  prim, mod;
int     port_num;
{
    /* retrieve net connected to scalar port */
    handle  port = acc_handle_port(mod, port_num);
    handle  port_conn = acc_next_loconn(port, null);

    /* retrieve net connected to primitive output */
    handle  out_term = acc_handle_terminal(prim, 0);
    handle  prim_conn = acc_handle_conn(out_term);

    /* compare handles */
    if (acc_compare_handles(port_conn, prim_conn) )
        return(true);
    else
        return(false);
}
```

If port_conn and prim_conn refer to the same connection, then the prim drives port

Figure 19-5—Using acc_compare_handles()

19.6 acc_configure()

acc_configure()			
Synopsis:	Set parameters that control the operation of various ACC routines.		
Syntax:	<code>acc_configure(configuration_parameter, configuration_value)</code>		
Type		Description	
Returns:	bool	1 if successful; 0 if an error occurred	
Arguments:	Type	Name	Description
	integer constant	configuration_parameter	One of the following predefined constants: accDefaultAttr0 accDevelopmentVersion accDisplayErrors accDisplayWarnings accEnableArgs accMapToMipd accMinTypMaxDelays
	quoted string	configuration_value	One of a fixed set of string values for the configuration_parameter
Related routines:	For accDefaultAttr0 acc_fetch_attribute() For accDevelopmentVersion all ACC routines For accDisplayErrors all ACC routines For accDisplayWarnings all ACC routines For accEnableArgs acc_handle_modpath() acc_handle_tchk() acc_set_scope()	For accMapToMipd acc_append_delays() acc_replace_delays() For accMinTypMaxDelays acc_append_delays() acc_fetch_delays() acc_replace_delays() For accPathDelayCount acc_append_delays() acc_fetch_delays() acc_replace_delays() acc_append_pulsere() acc_fetch_pulsere()	For accPathDelimStr acc_fetch_attribute() acc_fetch_fullname() acc_fetch_name() For accToHiZDelay acc_append_delays() acc_replace_delays()

The ACC routine **acc_configure()** shall set parameters that control the operation of various ACC routines. Tables 19-3 through 19-12 describe each parameter and its set of values. Note that a call to either **acc_initialize()** or **acc_close()** shall set each configuration parameter back to its default value.

Table 19-3—accDefaultAttr0 configuration parameter

	Set of values	Effect	Default
accDefaultAttr0	"true"	acc_fetch_attribute() shall return zero when it does not find the attribute requested and shall ignore the default_value argument	"false"
	"false"	acc_fetch_attribute() shall return the value passed as the default_value argument when it does not find the attribute requested	

Table 19-4—accDevelopmentVersion configuration parameter

	Set of values	Effect	Default
accDevelopmentVersion	Quoted string of letters, numbers, and the period character that form a valid PLI version, such as: "IEEE 1364 PLI" Software vendors can define version strings specific to their products	Can help ensure backward compatibility by indicating which version of ACC routines was used to develop a PLI application	Current version of ACC routines

Table 19-5—accDisplayErrors configuration parameter

	Set of values	Effect	Default
accDisplayErrors	"true"	ACC routines shall display error messages	"true"
	"false"	ACC routines shall not display error messages	

Table 19-6—accDisplayWarnings configuration parameter

	Set of values	Effect	Default
accDisplayWarnings	"true"	ACC routines shall display warning messages	"false"
	"false"	ACC routines shall not display warning messages	

Table 19-7—accEnableArgs configuration parameter

	Set of values	Effect	Default
accEnableArgs	"acc_handle_modpath"	acc_handle_modpath() shall recognize its optional arguments	"no_acc_handle_modpath" "no_acc_handle_tchk" "no_acc_set_scope"
	"no_acc_handle_modpath"	acc_handle_modpath() shall ignore its optional arguments	
	"acc_handle_tchk"	acc_handle_tchk() shall recognize its optional arguments	
	"no_acc_handle_tchk"	acc_handle_tchk() shall ignore its optional arguments	
	"acc_set_scope"	acc_set_scope() shall recognize its optional arguments	
	"no_acc_set_scope"	acc_set_scope() shall ignore its optional arguments	

Table 19-8—accMapToMipd configuration parameter

	Set of values	Effect	Default
accMapToMipd	"max"	acc_replace_delays() shall map the longest intermodule path delay to the MIPD	"max"
	"min"	acc_replace_delays() shall map the shortest intermodule path delay to the MIPD	
	"latest"	acc_replace_delays() shall map the last intermodule path delay to the MIPD	

Table 19-9—accMinTypMaxDelays configuration parameter

	Set of values	Effect	Default
accMinTypMaxDelays	"true"	acc_append_delays() , acc_fetch_delays() , acc_replace_delays() , acc_append_pulsere() , acc_fetch_pulsere() , and acc_replace_pulsere() shall use min:typ:max delay sets	"false"
	"false"	acc_append_delays() , acc_fetch_delays() , acc_replace_delays() , acc_append_pulsere() , acc_fetch_pulsere() , and acc_replace_pulsere() shall use a single delay value	

Table 19-10—accPathDelayCount configuration parameter

	Set of values	Effect	Default
accPathDelayCount	"1"	acc_append_delays() , acc_fetch_delays() , acc_replace_delays() , acc_append_pulsere() , acc_fetch_pulsere() , and acc_replace_pulsere() shall use 1 delay value or value set	"6"
	"2"	acc_append_delays() , acc_fetch_delays() , acc_replace_delays() , acc_append_pulsere() , acc_fetch_pulsere() , and acc_replace_pulsere() shall use 2 delay values or value sets	
	"3"	acc_append_delays() , acc_fetch_delays() , acc_replace_delays() , acc_append_pulsere() , acc_fetch_pulsere() , and acc_replace_pulsere() shall use 3 delay values or value sets	
	"6"	acc_append_delays() , acc_fetch_delays() , acc_replace_delays() , acc_append_pulsere() , acc_fetch_pulsere() , and acc_replace_pulsere() shall use 6 delay values or value sets	
	"12"	acc_append_delays() , acc_fetch_delays() , acc_replace_delays() , acc_append_pulsere() , acc_fetch_pulsere() , and acc_replace_pulsere() shall use 12 delay values or value sets	

Table 19-11—accPathDelimStr configuration parameter

	Set of values	Effect	Default
accPathDelimStr	Quoted string of letters, numbers, \$ or _	acc_fetch_name() , acc_fetch_fullname() , and acc_fetch_attribute() shall use the string literal as the delimiter separating the source and destination in module path names	"\$"

Table 19-12—accToHiZDelay configuration parameter

	Set of values	Effect	Default
accToHiZDelay	"average"	acc_append_delays() and acc_replace_delays() shall derive turn-off delays from the average of the rise and fall delays	"from_user"
	"max"	acc_append_delays() and acc_replace_delays() shall derive turn-off delays from the larger of the rise and fall delays	
	"min"	acc_append_delays() and acc_replace_delays() shall derive turn-off delays from the smaller of the rise and fall delays	
	"from_user"	acc_append_delays() and acc_replace_delays() shall derive turn-off delays from user-supplied argument(s)	

The example shown in Figure 19-6 presents a C-language application that obtains the load capacitance of all scalar nets connected to the ports in a module. This application uses **acc_configure()** to direct **acc_fetch_attribute()** to return zero if a load capacitance is not found for a net; as a result, the third argument, *default_value*, can be dropped from the call to **acc_fetch_attribute()**.

```
#include "acc_user.h"

display_load_capacitance()
{
    handle    module_handle, port_handle, net_handle;
    double    cap_val;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*set development version*/
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*configure acc_fetch_attribute to return 0 when it does not find*/
    /* the attribute*/
    acc_configure(accDefaultAttr0, "true");

    /*get handle for module*/
    module_handle = acc_handle_tfarg(1);

    /*scan all ports in module; display load capacitance*/
    port_handle = null;
    while(port_handle = acc_next_port(module_handle, port_handle) )
    {
        /*ports are scalar, so pass "null" to get single net connection*/
        net_handle = acc_next_loconn(port_handle, null);

        /*since accDefaultAttr0 is "true", drop default_value argument*/
        cap_val = acc_fetch_attribute(net_handle, "LoadCap_");

        if (!acc_error_flag)
            io_printf("Load capacitance of net #%d = %lf\n",
                      acc_fetch_index(port_handle), cap_val);
    }
    acc_close();
}
```

default_value
argument is dropped

Figure 19-6—Using **acc_configure() to set **accDefaultAttr0****

The example shown in Figure 19-7 presents a C-language application that displays the name of a module path. It uses **acc_configure()** to set **accEnableArgs** and, therefore, forces **acc_handle_modpath()** to ignore its null name arguments and recognize its optional handle arguments, **src_handle** and **dst_handle**.

```
#include "acc_user.h"

get_path()
{
    handle    path_handle, mod_handle, src_handle, dst_handle;

    /*initialize the environment for ACC routines*/
    acc_initialize();

    /*set development version*/
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*set accEnableArgs for acc_handle_modpath*/
    acc_configure(accEnableArgs, "acc_handle_modpath");

    /*get handles to the three system task arguments:*/
    /*    arg 1 is module name */
    /*    arg 2 is module path source */
    /*    arg 3 is module path destination*/
    mod_handle = acc_handle_tfarg(1);
    src_handle = acc_handle_tfarg(2);
    dst_handle = acc_handle_tfarg(3);

    /*display name of module path*/
    path_handle = acc_handle_modpath(mod_handle,
                                     null, null,
                                     src_handle, dst_handle);
    io_printf("Path is %s \n", acc_fetch_fullname(path_handle) );

    acc_close();
}
```

acc_handle_modpath() uses optional handle arguments *src_handle* and *dst_handle* because:

accEnableArgs is set and the name arguments are *null*

Figure 19-7—Using `acc_configure()` to set `accEnableArgs`

The example shown in Figure 19-8 fetches the rise and fall delays of each path in a module and backannotates the maximum delay value as the delay for all transitions. The value of **accPathDelayCount** specifies the minimum number of arguments that have to be passed to routines that read or write delay values. By setting **accPathDelayCount** to the minimum number of arguments needed for **acc_fetch_delays()** and again for **acc_replace_delays()**, all unused arguments can be eliminated from each call.

```

#include "acc_user.h"

set_path_delays()
{
    handle    mod_handle;
    handle    path_handle;
    double    rise_delay, fall_delay, max_delay;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*set development version*/
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle to module*/
    mod_handle = acc_handle_tfarg(1);

    /*fetch rise delays for all paths in module "top.m1"*/
    path_handle = null;
    while(path_handle = acc_next_modpath(mod_handle, path_handle) )
    {
        /*configure accPathDelayCount for rise and fall delays only*/
        acc_configure(accPathDelayCount, "2");
        acc_fetch_delays(path_handle, &rise_delay, &fall_delay);

        /*find the maximum of the rise and fall delays*/
        max_delay = (rise_delay > fall_delay) ? rise_delay : fall_delay;

        /*configure accPathDelayCount to apply one delay for all transitions*/
        acc_configure(accPathDelayCount, "1");
        acc_replace_delays(path_handle, max_delay);
    }
    acc_close();
}

```

only 2 delay arguments are needed

only 1 delay argument is needed

Figure 19-8—Using acc_configure() to set accPathDelayCount

The example shown in Figure 19-9 shows how **accToHiZDelay** is used to direct **acc_replace_delays()** to derive the turn-off delay for a Z-state primitive automatically as the larger of its rise and fall delays.

```
#include "acc_user.h"

set_buf_delays()
{
    handle    primitive_handle;
    handle    path_handle;
    double    added_rise, added_fall;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*set development version*/
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*configure accToHiZDelay so acc_append_delays derives turn-off */
    /* delay from the smaller of the rise and fall delays*/
    acc_configure(accToHiZDelay, "min");

    /*get handle to Z-state primitive*/
    primitive_handle = acc_handle_tfarg(1);

    /*get delay values*/
    added_rise = tf_getrealp(2);
    added_fall = tf_getrealp(3);

    acc_append_delays(primitive_handle, added_rise, added_fall);

    acc_close();
}
```

Figure 19-9—Using acc_configure() to set accToHiZDelay

19.7 acc_count()

acc_count()			
Synopsis:	Count the number of objects related to a particular reference object.		
Syntax:	<code>acc_count(acc_next_routine_name, object_handle)</code>		
Type		Description	
Returns:	int	Number of objects	
Arguments:	Type	Name	Description
	pointer to an acc_next_routine	acc_next_routine_name	Actual name (unquoted) of the acc_next_routine that finds the objects to be counted
	handle	object_handle	Handle of the reference object for the acc_next_routine
Related routines:	All acc_next_routines except acc_next_topmod()		

The ACC routine **acc_count()** shall find the number of objects that exist for a specific acc_next_routine with a given reference object. The object associated with object_handle shall be a valid reference object for the type acc_next_routine to be called.

Note that the ACC routine **acc_next_topmod()** does not work with **acc_count()**. However, top-level modules can be counted using **acc_next_child()** with a null reference object argument. For example:

```
acc_count(acc_next_child, null);
```

The example shown in Figure 19-10 uses **acc_count()** to count the number of nets in a module.

```
#include "acc_user.h"
count_nets()
{
    handle    module_handle;
    int        number_of_nets;

    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for module*/
    module_handle = acc_handle_tfarg(1);

    /*count and display number of nets in the module*/
    number_of_nets = acc_count(acc_next_net, module_handle);
    io_printf("number of nets = %d\n", number_of_nets);

    acc_close();
}
```

Figure 19-10—Using acc_count()

19.8 acc_fetch_argc()

acc_fetch_argc()			
Synopsis:	Get the number of command-line arguments supplied with a Verilog software tool invocation.		
Syntax:	acc_fetch_argc()		
Returns:	Type	Description	
	int	Number of command-line arguments	
Arguments:	Type	Name	Description
	None		
Related routines:	Use acc_fetch_argv() to get a character string array of the invocation options		

The ACC routine **acc_fetch_argc()** shall obtain the number of command-line arguments given on a Verilog software product invocation command line.

The example shown in Figure 19-11 uses **acc_fetch_argc()** to determine the number of invocation arguments used.

```
#include "acc_user.h"
#include <string.h> /* string.h is implementation dependent */

char* my_scan_plusargs(str)
char *str;
{
    int i;
    int length = strlen(str);
    char *curStr;
    char **argv = acc_fetch_argv();

    for (i = acc_fetch_argc()-1; i>0; i--)
    {
        curStr = argv[i];
        if ((curStr[0] == '+') && (!strncmp(curStr+1,str,length)))
        {
            char *retVal;

            length = strlen(&(curStr[length]) + 1);
            retVal = (char *)malloc(sizeof(char) * length);
            strcpy(retVal, &(curStr[length]));
            return(retVal);
        }
    }
    return(null);
}
```

Figure 19-11—Using acc_fetch_argc()

19.9 acc_fetch_argv()

acc_fetch_argv()		
Synopsis:	Get an array of character pointers that make up the command-line arguments for a Verilog software product invocation.	
Syntax:	acc_fetch_argv()	
	Type	Description
Returns:	char **	An array of character pointers that make up the command-line arguments
	Type	Name
Arguments:	None	
Related routines:	Use acc_fetch_argc() to get a count of the number of invocation arguments	

The ACC routine **acc_fetch_argv()** shall obtain an array of character pointers that make up the command-line arguments.

The example shown in Figure 19-12 uses **acc_fetch_argv()** to retrieve the invocation arguments used.

```
#include "acc_user.h"
#include <string.h> /* string.h is implementation dependent */

char* my_scan_plusargs(str)
char *str;
{
    int i;
    int length = strlen(str);
    char *curStr;
    char **argv = acc_fetch_argv();
    for (i = acc_fetch_argc()-1; i>0; i--)
    {
        curStr = argv[i];
        if ((curStr[0] == '+') && (!strncmp(curStr+1,str,length)))
        {
            char *retVal;

            length = strlen(&(curStr[length]) + 1);
            retVal = (char *)malloc(sizeof(char) * length);
            strcpy(retVal, &(curStr[length]));
            return(retVal);
        }
    }
    return(null);
}
```

Figure 19-12—Using acc_fetch_argv()

19.10 acc_fetch_attribute()

acc_fetch_attribute()			
Synopsis:	Get the value of a parameter or specparam named as an attribute in the Verilog source description.		
Syntax:	<code>acc_fetch_attribute(object_handle, attribute_string, default_value)</code>		
Returns:	Type	Description	
	double	Value of the parameter or specparam	
Arguments:	Type	Name	Description
	handle	object_handle	Handle of a named object
	quoted string or char *	attribute_string	Literal string or character string pointer with the <i>attribute</i> portion of the parameter or specparam declaration
Optional	double	default_value	Double-precision value to be returned if the attribute is not found (depends on <i>accDefaultAttr0</i>)
Related routines:	Use <code>acc_fetch_attribute_int()</code> to get an attribute value as an integer Use <code>acc_fetch_attribute_str()</code> to get an attribute value as a string Use <code>acc_configure(accDefaultAttr0...)</code> to set default value returned when attribute is not found Use <code>acc_fetch_paramtype()</code> to data type of the parameter value Use <code>acc_fetch_paramval()</code> to get parameters or specparam values not declared in attribute/object format		

The ACC routine **acc_fetch_attribute()** shall obtain the value of a parameter or specparam that is declared as an attribute in the Verilog HDL source description. The value shall be returned as a double.

Any parameter or specparam can be an attribute by naming it in one of the following ways:

- As a general attribute associated with more than one object in the module where the parameter or specparam attribute is declared
- As a specific attribute associated with a particular object in the module where the parameter or specparam attribute is declared

Each of these methods uses its own naming convention, as described in Table 19-13. For either convention, *attribute_string* shall name the attribute and shall be passed as the second argument to **acc_fetch_attribute()**. The *object_name* shall be the actual name of a design object in a Verilog HDL source description.

Table 19-13—Naming conventions for attributes

For	Naming convention	Example
A general attribute	attribute_string A mnemonic name that describes the attribute	specparam DriveStrength\$ = 2.8; <i>attribute_string</i> is DriveStrength\$
A specific attribute associated with a particular object	attribute_string—object_name Concatenate a mnemonic name that describes the attribute with the name of the object	specparam DriveStrength\$g1 = 2.8; <i>attribute_string</i> is DriveStrength\$ <i>object_name</i> is g1

The ACC routine **acc_fetch_attribute()** shall identify module paths in terms of their sources and destinations in the following format:

source	path_delimiter	destination
--------	----------------	-------------

The **acc_fetch_attribute()** routine shall look for module path names in this format, and **acc_fetch_name()** and **acc_fetch_fullname()** shall return names of module paths in this format. Therefore, the same naming convention should be used when associating an attribute with a module path. Note that names of module paths with multiple sources or destinations shall be derived from the first source or destination only.

By default, the *path_delimiter* used in path names is the “\$” character. This default can be changed by using the ACC routine **acc_configure()** to set the delimiter parameter **accPathDelimStr** to another character string.

The examples in Table 19-14 show how to name module paths using different delimiter strings.

Table 19-14—Example module path names using delimiter strings

For module path	If accPathDelimStr is	Then the module path name is
(a => q) = 10;	"\$"	a\$q
(b *> q1,q2) = 8;	"_\$_"	b\$_q1
(d,e,f *> r,s)= 8;	"_"	d_r

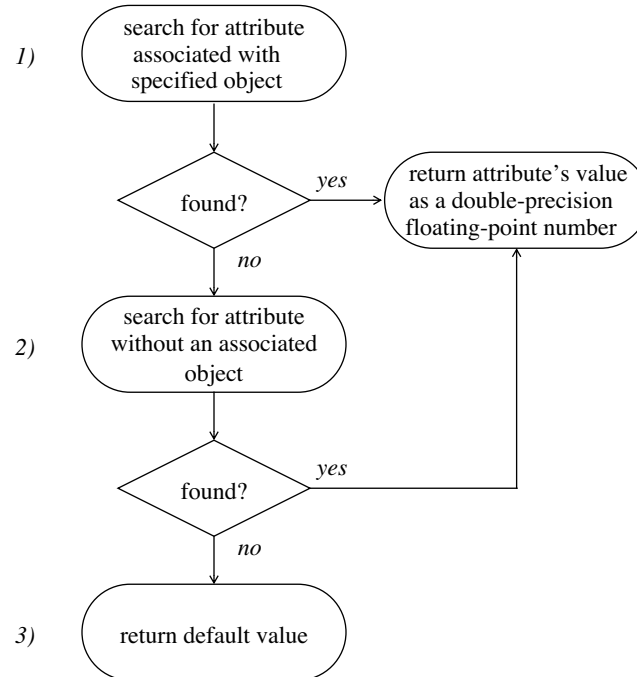
The following example shows an attribute name for a particular module path object:

Given the module path: (a => q) = 10;

An attribute name is: specparam RiseStrength\$a\$q = 20;

In this example, the *attribute_string* is RiseStrength\$, the *object_name* is a\$q, and the *path_delimiter* is \$ (the default path delimiter).

The following flowchart illustrates how **acc_fetch_attribute()** shall work:



This flowchart shows that when **acc_fetch_attribute()** finds the attribute requested, it returns the value of the attribute as a double-precision floating-point number.

- 1) The routine shall first look for the attribute name that concatenates *attribute_string* with the name associated with *object_handle*. For example, to find an attribute `InputLoad$` for a net `n1`, **acc_fetch_attribute()** would search for `InputLoad$n1`.
- 2) If **acc_fetch_attribute()** does not find the attribute associated with the object specified with *object_handle*, the routine shall then search for a name that matches *attribute_string*. Assume that, in the previous example, **acc_fetch_attribute()** does not find `InputLoad$n1`. It would then look for `InputLoad$`. Other variants of that name, such as `InputLoad$n3` or `InputLoad$n`, shall not be considered matches.
- 3) Failing both search attempts, the routine **acc_fetch_attribute()** shall return a default value. The default value is controlled by using the ACC routine **acc_configure()** to set or reset the configuration parameter **accDefaultAttr0** as shown in Table 19-15.

Table 19-15—Controlling the default value returned by **acc_fetch_attribute()**

When accDefaultAttr0 is	acc_fetch_attribute() shall return
"true"	Zero when the attribute is not found; the <i>default_value</i> argument can be dropped
"false"	The value passed as the <i>default_value</i> argument when the attribute is not found

The example shown in Figure 19-13 presents a C-language application that uses **acc_fetch_attribute()** to obtain the load capacitance of all scalar nets connected to the ports in a module. Note that **acc_fetch_attribute()** does not require its third argument, *default_value*, because **acc_configure()** is used to set **accDefaultAttr0** to **true**.

```
#include "acc_user.h"
display_load_capacitance()
{
    handle    module_handle, port_handle, net_handle;
    double    cap_val;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*set development version*/
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*configure acc_fetch_attribute to return 0 when it does not find*/
    /*the attribute*/
    acc_configure(accDefaultAttr0, "true");

    /*get handle for module*/
    module_handle = acc_handle_tfarg(1);

    /*scan all ports in module; display load capacitance*/
    port_handle = null;
    while(port_handle = acc_next_port(module_handle, port_handle) )
    {
        /*ports are scalar, so pass "null" to get single net connection*/
        net_handle = acc_next_loconn(port_handle, null);

        /*since accDefaultAttr0 is "true", drop default_value argument*/
        cap_val = acc_fetch_attribute(net_handle, "LoadCap_");

        if (!acc_error_flag)
            io_printf("Load capacitance of net #%d = %lf\n",
                      acc_fetch_index(port_handle), cap_val);
    }
    acc_close();
}
```

Figure 19-13—Using acc_fetch_attribute()

19.11 acc_fetch_attribute_int()

acc_fetch_attribute_int()			
Synopsis:	Get the integer value of a parameter or specparam named as an attribute in the Verilog source description.		
Syntax:	<code>acc_fetch_attribute_int(object_handle, attribute_string, default_value)</code>		
Returns:	Type		Description
	int		Value of the parameter or specparam
Arguments:	Type	Name	Description
	handle	object_handle	Handle of a named object
	quoted string or char *	attribute_string	Literal string or character string pointer with the <i>attribute</i> portion of the parameter or specparam declaration
Optional	int	default_value	Integer value to be returned if the attribute is not found (depends on accDefaultAttr0)
Related routines:	Use acc_fetch_attribute() to get an attribute value as a double Use acc_fetch_attribute_str() to get an attribute value as a string Use acc_configure(accDefaultAttr0...) to set default value returned when attribute is not found Use acc_fetch_paramtype() to get the data type of the parameter value Use acc_fetch_paramval() to get parameters or specparam values not declared in attribute/object format		

The ACC routine **acc_fetch_attribute_int()** shall obtain the value of a parameter or specparam that is declared as an attribute in the Verilog HDL source description. The value shall be returned as an integer.

Any parameter or specparam can be an attribute. Refer to 19.10 for a description of attribute naming and how attribute values are fetched.

19.12 acc_fetch_attribute_str()

acc_fetch_attribute_str()			
Synopsis:	Get the integer value of a parameter or specparam named as an attribute in the Verilog source description.		
Syntax:	acc_fetch_attribute_str(object_handle, attribute_string, default_value)		
Returns:	Type		Description
	char *	Value of the parameter or specparam	
Arguments:	Type	Name	Description
	handle	object_handle	Handle of a named object
	quoted string or char *	attribute_string	Literal string or character string pointer with the <i>attribute</i> portion of the parameter or specparam declaration
Optional	quoted string or char *	default_value	Character string value to be returned if the attribute is not found (depends on accDefaultAttr0)
Related routines:	Use acc_fetch_attribute() to get an attribute value as a double Use acc_fetch_attribute_int() to get an attribute value as an integer Use acc_configure(accDefaultAttr0...) to set default value returned when attribute is not found Use acc_fetch_paramtype() to get the data type of the parameter value Use acc_fetch_paramval() to get parameters or specparam values not declared in attribute/object format		

The ACC routine **acc_fetch_attribute_str()** shall obtain the value of a parameter or specparam that is declared as an attribute in the Verilog HDL source description. The value shall be returned as a pointer to a character string. The return value for this routine is placed in the ACC internal string buffer. See 18.9 for explanation of strings in ACC routines.

Any parameter or specparam can be an attribute. Refer to 19.10 for a description of attribute naming and how attribute values are fetched.

19.13 acc_fetch_defname()

acc_fetch_defname()			
Synopsis:	Get the definition name of a module instance or primitive instance.		
Syntax:	acc_fetch_defname(object_handle)		
Returns:	Type	Description	
	char *	Pointer to a character string containing the definition name	
Arguments:	Type	Name	Description
	handle	object_handle	Handle of the module instance or primitive instance
Related routines	Use acc_fetch_name() to display the instance name of an object		

The ACC routine **acc_fetch_defname()** shall obtain the definition name of a module instance or primitive instance. The *definition name* is the declared name of the object as opposed to the *instance name* of the object. In the illustration shown below, the definition name is "dff", and the instance name is "i15".

```

dff i15 (q, clk, d); //instance of a module or primitive

```

The return value for this routine is placed in the ACC internal string buffer. See 18.9 for explanation of strings in ACC routines.

The example shown in Figure 19-14 presents a C-language application that uses **acc_fetch_defname()** to display the definition names of all primitives in a module.

```

#include "acc_user.h"

get_primitive_definitions(module_handle)
handle    module_handle;
{
    handle    prim_handle;

    /*get and display defining names of all primitives in the module*/
    prim_handle = null;
    while(prim_handle = acc_next_primitive(module_handle,prim_handle))
        io_printf("primitive definition is %s\n",
                  acc_fetch_defname(prim_handle) );
}

```

Figure 19-14—Using acc_fetch_defname()

19.14 acc_fetch_delay_mode()

acc_fetch_delay_mode()			
Synopsis:	Get the delay mode of a module instance.		
Syntax:	acc_fetch_delay_mode(module_handle)		
Returns:	Type	Description	
	int	A predefined integer constant representing the delay mode of the module instance: <div> accDelayModeNone accDelayModeZero accDelayModeUnit accDelayModePath accDelayModeDistrib accDelayModeMTM </div>	
Arguments:	Type	Name	Description
	handle	module_handle	Handle to a module instance

The ACC routine **acc_fetch_delay_mode()** shall return the delay mode of a module or cell instance. The delay mode determines how delays are stored for primitives and paths within the module or cell. The routine shall return one of the predefined constants given in Table 19-16.

Table 19-16—Predefined constants used by acc_fetch_delay_mode()

Predefined constant	Description
accDelayModeNone	No delay mode specified.
accDelayModeZero	All primitive delays are zero; all path delays are ignored.
accDelayModeUnit	All primitive delays are one; all path delays are ignored.
accDelayModeDistrib	If a logical path has both primitive delays and path delays specified, the primitive delays shall be used.
accDelayModePath	If a logical path has both primitive delays and path delays specified, the path delays shall be used.
accDelayModeMTM	If this property is true, Minimum:Typical:Maximum delay sets for each transition are being stored; if this property is false, a single delay for each transition is being stored.

Figure 19-15 uses **acc_fetch_delay_mode()** to retrieve the delay mode of all children of a specified module.

```
#include "acc_user.h"
display_delay_mode()
{
    handle    mod, child;

    /*reset environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get module passed to user-defined system task*/
    mod = acc_handle_tfarg(1);

    /*find and display delay mode for each module instance*/
    child = null;
    while(child = acc_next_child(mod, child))
    {
        io_printf("Module %s set to: ", acc_fetch_fullname(child));
        switch(acc_fetch_delay_mode(child) )
        {
            case accDelayModePath:
                io_printf(" path delay mode\n");
                break;
            case accDelayModeDistrib:
                io_printf(" distributed delay mode\n");
                break;
            . . .
        }
    }
}
```

Figure 19-15—Using acc_fetch_delay_mode()

19.15 acc_fetch_delays()

acc_fetch_delays() for single delay values (accMinTypMaxDelays set to "false")			
Synopsis:	Get existing delays for primitives, module paths, timing checks, module input ports, and intermodule paths.		
Syntax: Primitives Ports Port bits Intermodule paths	<code>acc_fetch_delays(object_handle, rise_delay, fall_delay, z_delay)</code>		
Module paths	<code>acc_fetch_delays(object_handle, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12)</code>		
Timing checks	<code>acc_fetch_delays(object_check_handle, limit)</code>		
Type		Description	
Returns:	bool	1 if successful; 0 if an error occurred	
Type		Name	Description
Arguments:	handle	object_handle	Handle of a primitive, module path, timing check, module input port, bit of a module input port, or intermodule path
	double *	rise_delay fall_delay	Rise and fall delay for 2-state primitive, 3-state primitive, module input port, module input port bit, or intermodule path
Conditional	double *	z_delay	Turn-off (to Z)) transition delay for 3-state primitives, module input ports, module input port bits, or intermodule paths
	double *	d1	If accPathDelayCount is set to "1": delay for all transitions for module paths If accPathDelayCount is set to "2" or "3": rise transition delay for module paths If accPathDelayCount is set to "6" or "12": 0→1 transition delay for module paths
Conditional	double *	d2	If accPathDelayCount is set to "2" or "3": fall transition delay for module paths If accPathDelayCount is set to "6" or "12": 1→0 transition delay for module paths
Conditional	double *	d3	If accPathDelayCount is set to "3": turn-off transition delay for module paths If accPathDelayCount is set to "6" or "12": 0→Z transition delay for module paths
Conditional	double *	d4 d5 d6	If accPathDelayCount is set to "6" or "12": d4 is Z→1 transition delay for module paths d5 is 1→Z transition delay for module paths d6 is Z→0 transition delay for module paths
Conditional	double *	d7 d8 d9 d10 d11 d12	If accPathDelayCount is set to "12": d7 is 0→X transition delay for module paths d8 is X→1 transition delay for module paths d9 is 1→X transition delay for module paths d10 is X→0 transition delay for module paths d11 is X→Z transition delay for module paths d12 is Z→X transition delay for module paths
	double *	limit	Limit of timing check

acc_fetch_delays() for min:typ:max delays (accMinTypMaxDelays set to "true")			
Synopsis:	Get existing delay values for primitives, module paths, timing checks, module input ports, or intermodule paths; the delay values are contained in an array.		
Syntax:	acc_append_delays(object_handle, array_ptr)		
Type		Description	
Returns:	bool	1 if successful; 0 if an error is encountered	
Arguments:	Type	Name	Description
	handle	object_handle	Handle of a primitive, module path, timing check, module input port, bit of a module input port, or intermodule path
	double address	array_ptr	Pointer to array of min:typ:max delay values; the size of the array depends on the type of object and the setting of accPathDelayCount (see 18.8)

The ACC routine **acc_fetch_delays()** shall work differently depending on how the configuration parameter **accMinTypMaxDelays** is set. When this parameter is set to "**false**", a single delay per transition shall be assumed, and each delay shall be fetched into variables pointed to as individual arguments. For this *single delay mode*, the first syntax table in this clause shall apply.

When **accMinTypMaxDelays** is set to "**true**", **acc_fetch_delays()** shall fetch one or more sets of minimum:typical:maximum delays into an array, rather than single delays fetched as individual arguments. For this min:typ:max delay mode, the second syntax table in this clause shall apply.

The number delay values that shall be fetched by **acc_fetch_delays()** is determined by the type of object and the setting of configuration parameters. Refer to 18.8 for a description of how the number of delay values are determined.

The ACC routine **acc_fetch_delays()** shall retrieve delays in the timescale of the module that contains the **object_handle**.

The example shown in Figure 19-16 presents a C-language application that uses **acc_fetch_delays()** to retrieve the rise, fall, and turn-off delays of all paths through a module.

```

#include "acc_user.h"

display_path_delays()
{
    handle    mod_handle;
    handle    path_handle;
    double    rise_delay,fall_delay,toz_delay;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*set development version*/
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*set accPathDelayCount to return rise, fall and turn-off delays */
    acc_configure(accPathDelayCount, "3");

    /*get handle to module*/
    mod_handle = acc_handle_tfarg(1);

    /*fetch rise delays for all paths in module "top.m1"*/
    path_handle = null;
    while(path_handle = acc_next_modpath(mod_handle, path_handle) )
    {
        acc_fetch_delays(path_handle,
                        &rise_delay,&fall_delay,&toz_delay);

        /*display rise, fall and turn-off delays for each path*/
        io_printf("For module path %s,delays are:\n",
                acc_fetch_fullname(path_handle) );
        io_printf("rise = %lf, fall = %lf, turn-off = %lf\n",
                rise_delay,fall_delay,toz_delay);
    }
    acc_close();
}

```

Figure 19-16—Using acc_fetch_delays() in single delay mode

The example shown in Figure 19-17 is a C-language code fragment of an application that shows how to fetch min:typ:max delays for the intermodule paths. The example follows these steps:

- a) Declares an array of nine double-precision floating-point values as a buffer for storing three sets of min:typ:max values, one set each for rise, fall, and turn-off delays.
- b) Sets the configuration parameter **accMinTypMaxDelays** to "true" to instruct **acc_fetch_delays()** to retrieve delays in min:typ:max format.
- c) Calls **acc_fetch_delays()** with a valid intermodule path handle and the array pointer.

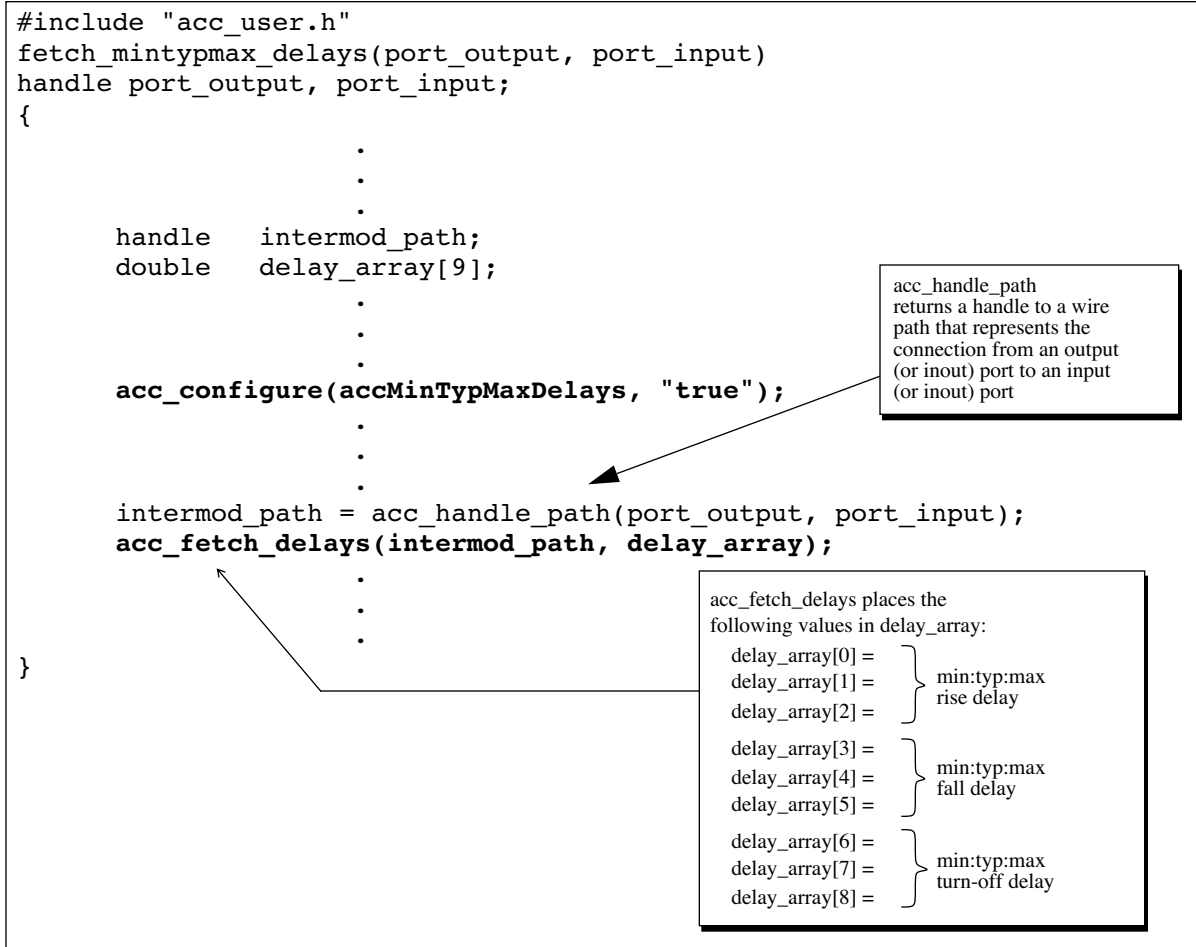


Figure 19-17—Using acc_fetch_delays() in min:typ:max delay mode

19.16 acc_fetch_direction()

acc_fetch_direction()			
Synopsis:	Get the direction of a port or terminal.		
Syntax:	acc_fetch_direction(object_handle)		
Returns:	Type	Description	
	int	A predefined integer constant representing the direction of a port or terminal accInput accOutput accInout accMixedIo	
Arguments:	Type	Name	Description
	handle	object_handle	Handle of a port or terminal

The ACC routine **acc_fetch_direction()** shall return a predefined integer constant indicating the direction of a module port or primitive terminal. The values returned are given in Table 19-17.

Table 19-17—The operation of acc_fetch_direction()

When direction is	acc_fetch_direction() shall return
Input only	accInput
Output only	accOutput
Bidirectional (input and output)	accInout
A concatenation of input ports and output ports	accMixedIo

The example shown in Figure 19-18 presents a C-language application that uses **acc_fetch_direction()** to determine whether or not a port is an input.

```
#include "acc_user.h"

bool    is_port_input(port_handle)
handle  port_handle;
{
    int  direction;

    direction = acc_fetch_direction(port_handle);
    if (direction == accInput || direction == accInout)
        return(true);
    else
        return(false);
}
```

Figure 19-18—Using acc_fetch_direction()

19.17 acc_fetch_edge()

acc_fetch_edge()			
Synopsis:	Get the edge specifier of a module path input terminal.		
Syntax:	acc_fetch_edge(pathio_handle)		
Returns:	Type	Description	
	int	A predefined integer constant representing the edge specifier of a path input or output terminal: <div> accNoedge accEdge01 accEdgex1 accPosedge accEdge10 accEdge1x accNegedge accEdge0x accEdgex0 </div>	
Arguments:	Type	Name	Description
	handle	pathio_handle	Handle to a module path input or output, or handle to a timing check terminal

The ACC routine **acc_fetch_edge()** shall return a value that is a masked integer representing the edge specifier for a module path delay terminal.

Table 19-18 lists the predefined edge specifiers as they are specified in **acc_user.h**.

Table 19-18—Edge specifiers constants

Edge type	Defined constant	Binary value
None	accNoedge	0
Positive edge (0→1,0→x,x→1)	accPosedge	00001011
Negative edge (1→0,1→x,x→0)	accNegedge	01100010
0→1 edge	accEdge01	00000001
1→0 edge	accEdge10	00000010
0→x edge	accEdge0x	00000100
x→1 edge	accEdgex1	00001000
1→x edge	accEdge1x	00010000
x→0 edge	accEdgex0	00100000

The integer mask returned by **acc_fetch_edge()** is usually either **accPosedge** or **accNegedge**. Occasionally, however, the mask is a hybrid mix of specifiers that is equal to neither. The example shown in Figure 19-19 illustrates how to check for these hybrid edge specifiers. The value **accNoEdge** is returned if no edge is found.

The example takes a path input or output and returns the string corresponding to its edge specifier. It provides analogous functionality to that of **acc_fetch_type_str()** in that it returns a string corresponding to an integer value that represents a type.

This example first checks to see whether the returned mask is equal to **accPosedge** or **accNegedge**, which are the most likely cases. If it is not, the application does a bitwise AND with the returned mask and each of the other edge specifiers to find out which types of edges it contains. If an edge type is encoded in the returned mask, the corresponding edge type string suffix is appended to the string "accEdge".

```
char *acc_fetch_edge_str(pathio)
handle pathio;
{
    int edge = acc_fetch_edge(pathio);
    static char edge_str[32];

    if (! acc_error_flag)
    {
        if (edge == accNoEdge)
            strcpy(edge_str, "accNoEdge");

        /* accPosedge == (accEdge01 & accEdge0x & accEdgex1) */
        else if (edge == accPosEdge)
            strcpy(edge_str, "accPosEdge");

        /* accNegedge == (accEdge10 & accEdge 1x & accEdgex0) */
        else if (edge == accNegEdge)
            strcpy(edge_str, "accNegEdge");

        /* edge is neither posedge nor negedge, but some combination
           of other edges */
        else {
            strcpy(edge_str, "accEdge");
            if (edge & accEdge01) strcat(edge_str, "_01");
            if (edge & accEdge10) strcat(edge_str, "_10");
            if (edge & accEdge0x) strcat(edge_str, "_0x");
            if (edge & accEdgex1) strcat(edge_str, "_x1");
            if (edge & accEdge1x) strcat(edge_str, "_1x");
            if (edge & accEdgex0) strcat(edge_str, "_x0");
        }

        return(edge_str);
    }
    else
        return(null);
}
```

Figure 19-19—Using acc_fetch_edge()

19.18 acc_fetch_fullname()

acc_fetch_fullname()			
Synopsis:	Get the full hierarchical name of any named object or module path.		
Syntax:	acc_fetch_fullname(object_handle)		
Returns:	Type	Description	
	char *	Character pointer to a string containing the full hierarchical name of the object	
Arguments:	Type	Name	Description
	handle	object_handle	Handle of the object
Related routines:	Use acc_fetch_name() to find the lowest-level name of the object		
	Use acc_configure(accPathDelimStr...) to set the delimiter string for module path names		

The ACC routine **acc_fetch_fullname()** shall obtain the *full hierarchical name* of an object. The full hierarchical name is the name that uniquely identifies an object. In Figure 19-20, the top-level module, **top1**, contains module instance **mod3**, which contains net **w4**. In this example, the full hierarchical name of the net is **top1.mod3.w4**.

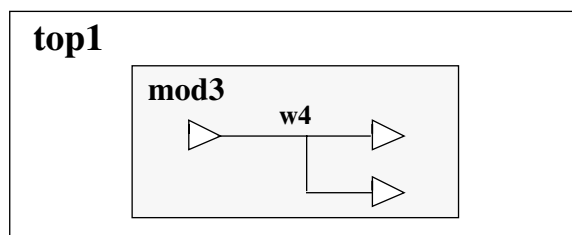


Figure 19-20—A design hierarchy; the fullname of net w4 is "top1.mod.w4"

Table 19-19 lists the objects in a Verilog HDL description for which **acc_fetch_fullname()** shall return a name.

Table 19-19—Named objects supported by acc_fetch_fullname()

Modules	Variables
Module ports	Named events
Module paths	Parameters
Data paths	Specparams
Primitives	Named blocks
Nets	Verilog HDL tasks
Registers	Verilog HDL functions

Module path names shall be derived from their sources and destinations in the following format:

source	path_delimiter	destination
--------	----------------	-------------

By default, the *path_delimiter* shall be the character \$. However, the delimiter can be changed by using the ACC routine **acc_configure()** to set the delimiter parameter **accPathDelimStr** to another character string.

The following examples show names of paths within a top-level module *m3*, as returned by **acc_fetch_fullname()** when the *path_delimiter* is \$. Note that names of module paths with multiple sources or destinations shall be derived from the first source and destination only.

Table 19-20—Module path names returned by acc_fetch_fullname()

For paths in module <i>m3</i>	acc_fetch_fullname() returns a pointer to
(a => q) = 10;	m3.a\$q
(b *> q1,q2) = 8;	m3.b\$q1
(d,e,f *> r,s)= 8;	m3.d\$r

If a Verilog software product creates default names for unnamed instances, **acc_fetch_fullname()** shall return the full hierarchical default name. Otherwise, the routine shall return null for unnamed instances.

Using **acc_fetch_fullname()** with a module port handle shall return the full hierarchical implicit name of the port.

The routine **acc_fetch_fullname()** shall store the returned string in a temporary buffer. To preserve the string for later use in an application, it should be copied to another variable (refer to 18.9).

In the example shown in Figure 19-21, the routine uses **acc_fetch_fullname()** to display the full hierarchical name of an object if the object is a net.

```
#include "acc_user.h"
display_if_net(object_handle)
handle    object_handle;
{
    /*get and display full name if object is a net*/
    if (acc_fetch_type(object_handle) == accNet)
        io_printf("Object is a net: %s\n",
                  acc_fetch_fullname(object_handle) );
    else
        io_printf("Object is not a net: %s\n",
                  acc_fetch_fullname(object_handle) );
}
```

Figure 19-21—Using acc_fetch_fullname()

19.19 acc_fetch_fulltype()

acc_fetch_fulltype()			
Synopsis:	Get the fulltype of an object.		
Syntax:	<code>acc_fetch_fulltype(object_handle)</code>		
Type		Description	
Returns:	int	A predefined integer constant from the list shown in 18.6	
Type		Name	Description
Arguments:	handle	object_handle	Handle of the object
Related routines:	Use <code>acc_fetch_type()</code> to get the general type classification of an object Use <code>acc_fetch_type_str()</code> to get the fulltype as an character string		

The ACC routine **acc_fetch_fulltype()** shall return the *fulltype* of an object. The fulltype is a specific classification of a Verilog HDL object, represented as a predefined constant (defined in `acc_user.h`). Table 18-25 lists all of the fulltype constants that can be returned by **acc_fetch_fulltype()**.

Many Verilog HDL objects have both a *type* and a *fulltype*. The type of an object is its general Verilog HDL type classification. The fulltype is the specific type of the object. The examples in Table 19-21 illustrate the difference between the type of an object and the fulltype of the same object for selected objects.

Table 19-21 — The difference between the type and the fulltype of an object

For a handle to	acc_fetch_type() shall return	acc_fetch_fulltype() shall return
A setup timing check	accTchk	accSetup
An and gate primitive	accPrimitive	accAndGate
A sequential UDP	accPrimitive	accSeqPrim

The example shown in Figure 19-22 uses **acc_fetch_fulltype()** to find and display the fulltypes of timing checks. This application is called by a higher-level application, **display_object_type**, presented as the usage example for **acc_fetch_type()**.

```
#include "acc_user.h"

display_timing_check_type(tchk_handle)
handle    tchk_handle;
{
    /*display timing check type*/
    io_printf("Timing check is");
    switch(acc_fetch_fulltype(tchk_handle) )
    {
        case accHold:
            io_printf(" hold\n");
            break;
        case accNochange:
            io_printf(" nochange\n");
            break;
        case accPeriod:
            io_printf(" period\n");
            break;
        case accRecovery:
            io_printf(" recovery\n");
            break;
        case accSetup:
            io_printf(" setup\n");
            break;
        case accSkew:
            io_printf(" skew\n");
            break;
        case accWidth:
            io_printf(" width\n");
            break;
    }
}
```

Figure 19-22—Using `acc_fetch_fulltype()` to display the fulltypes of timing checks

The example shown in Figure 19-23 uses **acc_fetch_fulltype()** to find and display the fulltypes of primitive objects passed as input arguments. This application is called by a higher-level application, **display_object_type**, presented as the usage example for **acc_fetch_type()**.

```
#include "acc_user.h"

display_primitive_type(primitive_handle)
handle                primitive_handle;
{
    /*display primitive type*/
    io_printf("Primitive is");
    switch(acc_fetch_fulltype(primitive_handle) )
    {
        case accAndGate:
            io_printf(" and gate\n"); break;
        case accBufGate:
            io_printf(" buf gate\n"); break;
        case accBufif0Gate:case accBufif1Gate:
            io_printf(" bufif gate\n"); break;
        case accCmosGate:case accNmosGate:case accPmosGate:
        case accRcmosGate:case accRnmosGate:case accRpmosGate:
            io_printf(" MOS or Cmos gate\n"); break;
        case accCombPrim:
            io_printf(" combinational UDP\n"); break;
        case accSeqPrim:
            io_printf(" sequential UDP\n"); break;
        case accNotif0Gate:case accNotif1Gate:
            io_printf(" notif gate\n"); break;
        case accRtranGate:
            io_printf(" rtran gate\n"); break;
        case accRtranif0Gate:case accRtranif1Gate:
            io_printf(" rtranif gate\n"); break;
        case accNandGate:
            io_printf(" nand gate\n"); break;
        case accNorGate:
            io_printf(" nor gate\n"); break;
        case accNotGate:
            io_printf(" not gate\n"); break;
        case accOrGate:
            io_printf(" or gate\n"); break;
        case accPulldownGate:
            io_printf(" pulldown gate\n"); break;
        case accPullupGate:
            io_printf(" pullup gate\n"); break;
        case accXnorGate:
            io_printf(" xnor gate\n"); break;
        case accXorGate:
            io_printf(" xor gate\n");
    }
}
```

Figure 19-23—Using `acc_fetch_fulltype()` to display the fulltypes of primitives

19.20 acc_fetch_index()

acc_fetch_index()			
Synopsis:	Get the index number for a port or terminal.		
Syntax:	acc_fetch_index(object_handle)		
Returns:	Type		Description
	int	Integer index for a port or terminal, starting with zero	
Arguments:	Type	Name	Description
	handle	object_handle	Handle of the port or terminal

The ACC routine **acc_fetch_index()** shall return the index number for a module port or primitive terminal. Indices are integers that shall start at zero and increase from left to right.

- The index of a *port* shall be its position in a module definition in the Verilog HDL source description.
- The index of a *terminal* shall be its position in a gate, switch, or UDP instance.

Table 19-22 shows how indices shall be derived.

Table 19-22—Deriving indices

For	Indices are
<i>Terminals:</i> <pre>nand g1(out, in1, in2);</pre>	0 for terminal out 1 for terminal in1 2 for terminal in2
<i>Implicit ports:</i> <pre>module A(q, a, b);</pre>	0 for port q 1 for port a 2 for port b
<i>Explicit ports:</i> <pre>module top; reg ra,rb; wire wq; explicit_port_mod epm1(.b(rb), .a(ra), .q(wq)); endmodule module explicit_port_mod(q, a, b); input a, b; output q; nand (q, a, b); endmodule</pre>	0 for explicit port epm1.q 1 for explicit port epm1.a 2 for explicit port epm1.b

The example shown in Figure 19-24 presents a C-language application that uses **acc_fetch_index()** to find and display the input ports of a module.

```
#include "acc_user.h"

display_inputs(module_handle)
handle module_handle;
{
    handle    port_handle;
    int       direction;

    /*get handle for the module and each of its ports*/
    port_handle = null;
    while (port_handle = acc_next_port(module_handle, port_handle) )
    {
        /*determine if port is an input*/
        direction = acc_fetch_direction(port_handle);
        /*give the index of each input port*/
        if (direction == accInput)
            io_printf("Port #%d of %s is an input\n",
                      acc_fetch_index(port_handle),
                      acc_fetch_fullname(module_handle) );
    }
}
```

Figure 19-24—Using acc_fetch_index()

19.21 acc_fetch_location()

acc_fetch_location()			
Synopsis:	Get the location of an object in a Verilog-HDL source file.		
Syntax:	acc_fetch_location(loc_p, object_handle)		
Returns:	Type		Description
	void	None	
Arguments:	Type	Name	Description
	p_location	loc_p	Pointer to a predefined location structure
	handle	object_handle	Handle to an object

The ACC routine **acc_fetch_location()** shall return the *file name* and *line number* in the file for the specified object. The file name and line number shall be returned in an **s_location** data structure. This data structure is defined in **acc_user.h**, and listed in Figure 19-25.

```
typedef struct t_location
{
    int line_no; /* line number in the file */
    char *filename; /* file name */
} s_location, *p_location;
```

Figure 19-25—s_location data structure

filename field is a character pointer.

line_no field is a nonzero positive integer.

Table 19-23 lists the objects that shall be supported by **acc_fetch_location()**.

Table 19-23—Objects supported by acc_fetch_location()

Object type	Location returned
Modules	Module definition
Module ports	Module definition
Module paths	Module path line
Data paths	Module path line
Primitives	Instantiation line
Explicit nets	Definition line
Implicit nets	Line where first used
Registers	Definition line

Table 19-23—Objects supported by `acc_fetch_location()` (continued)

Object type	Location returned
Variables	Definition line
Named events	Definition line
Parameters	Definition line
Specparams	Definition line
Named blocks	Definition line
Verilog HDL tasks	Definition line
Verilog HDL functions	Definition line

The return value for *filename* is placed in the ACC internal string buffer. See 18.9 for an explanation of strings in ACC routines.

The example shown in Figure 19-26 uses **`acc_fetch_location()`** to print the file name and line number for an object.

```
void find_object_location (object)
    handle object;
{
    s_location s_loc;
    p_location loc_p = &s_loc;
    acc_fetch_location(loc_p, object); /*get the filename and line_no*/
    if (! acc_error_flag) /* On success */
        io_printf ("Object located in file %s on line %d \n",
                    loc_p->filename, loc_p->line_no);
}
```

Figure 19-26—Using `acc_fetch_location()`

19.22 acc_fetch_name()

acc_fetch_name()		
Synopsis:	Get the instance name of any named object or module path.	
Syntax:	<code>acc_fetch_name(object_handle)</code>	
Returns:	Type	Description
	char *	Character pointer to a string containing the instance name of the object
Arguments:	Type	Name
	handle	object_handle
Description		
Handle of the named object		
Related routines:	Use <code>acc_fetch_fullname()</code> to get the full hierarchical name of the object	
	Use <code>acc_fetch_defname()</code> to get the definition name of the object	
	Use <code>acc_configure(accPathDelimStr...)</code> to set the naming convention for module paths	

The ACC routine **acc_fetch_name()** shall obtain the *name* of an object. The name of an object is its lowest-level name. In the following example, the top-level module, `top1`, contains module instance `mod3`, which contains net `w4`, as shown in Figure 19-27. In this example, the name of the net is `w4`.

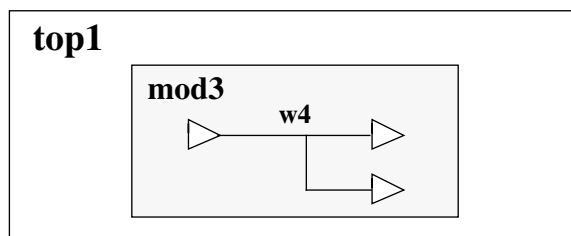


Figure 19-27—A design hierarchy; the name of net `w4` is “`w4`”

The return value for this routine is placed in the ACC internal string buffer. See 18.9 for an explanation of strings in ACC routines.

Table 19-24 lists the objects in a Verilog HDL description for which **acc_fetch_name()** shall return a name.

Table 19-24—Named objects supported by `acc_fetch_name()`

Modules	Variables
Module ports	Named events
Module paths	Parameters
Data paths	Specparams
Primitives	Named blocks
Nets	Verilog HDL tasks
Registers	Verilog HDL functions

Module path names shall be derived from their sources and destinations in the following format:

source	path_delimiter	destination
--------	----------------	-------------

By default, the *path_delimiter* is the character \$. However, the delimiter can be changed by using the ACC routine *acc_configure()* to set the delimiter parameter **accPathDelimStr** to another character string.

Table 19-25 shows names of paths within a top-level module *m3*, as returned by **acc_fetch_name()** when the *path_delimiter* is \$. Note that names of module paths with multiple sources or destinations shall be derived from the first source and destination only.

Table 19-25—Module path names returned by *acc_fetch_name()*

For paths in module <i>m3</i>	<i>acc_fetch_name()</i> returns a pointer to
(a => q) = 10;	a\$q
(b *> q1,q2) = 8;	b\$q1
(d,e,f *> r,s)= 8;	d\$r

If a Verilog software implementation creates default names for unnamed instances, **acc_fetch_name()** shall return the default name. Otherwise, the routine shall return null for unnamed instances.

Using **acc_fetch_name()** with a module port handle shall return the implicit name of the port.

The routine **acc_fetch_name()** shall store the returned string in a temporary buffer. To preserve the string for later use in an application, it should be preserved (refer to 18.9).

The following example uses **acc_fetch_name()** to display the names of top-level modules.

```
#include "acc_user.h"
show_top_mods()
{
    handle    module_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*scan all top-level modules*/
    io_printf("The top-level modules are:\n");
    module_handle = null;
    while (module_handle = acc_next_topmod(module_handle) )
        io_printf(" %s\n",acc_fetch_name(module_handle));
    acc_close();
}
```

Figure 19-28—Using *acc_fetch_name()*

19.23 acc_fetch_paramtype()

acc_fetch_paramtype()			
Synopsis:	Get the data type of a parameter or specparam.		
Syntax:	acc_fetch_paramtype(parameter_handle)		
Returns:	Type		Description
	int	A predefined integer constant representing the data type of a parameter: accIntParam accIntegerParam accRealParam accStringParam	
Arguments:	Type		Description
	handle	parameter_handle	Handle to a parameter or specparam
Related routines:	Use acc_next_parameter() to get all parameters within a module Use acc_next_specparam() to get all specparams within a module		

The ACC routine **acc_fetch_paramtype()** shall return an integer constant that represents the data type of a value that has been assigned to a parameter or specparam.

Figure 19-29 uses **acc_fetch_paramtype()** to display the values of all parameters within a module.

```
#include "acc_user.h"
print_parameter_values()
{
    handle module_handle, param_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    module_handle = acc_handle_tfarg(1);
    param_handle = null;
    while(param_handle = acc_next_parameter(module_handle,param_handle) )
    {
        io_printf("Parameter %s has value: ",acc_fetch_fullname(param_handle));
        switch(acc_fetch_paramtype(param_handle) )
        {
            case accRealParam:
                io_printf("%lf\n", acc_fetch_paramval(param_handle) ); break;
            case accIntegerParam:
                io_printf("%d\n", (int)acc_fetch_paramval(param_handle) ); break;
            case accStringParam:
                io_printf("%s\n", (char*)(int)acc_fetch_paramval(param_handle) ); break;
        }
    }
    acc_close();
}
```

Figure 19-29—Using acc_fetch_paramtype()

19.24 acc_fetch_paramval()

acc_fetch_paramval()			
Synopsis:	Get the value of a parameter or specparam.		
Syntax:	<code>acc_fetch_paramval(parameter_handle)</code>		
Returns:	Type		Description
	double	The value of a parameter or specparam	
Arguments:	Type	Name	Description
	handle	parameter_handle	Handle to a parameter or specparam
Related routines:	Use <code>acc_fetch_paramtype()</code> to retrieve the data type of a parameter Use <code>acc_next_parameter()</code> to scan all parameters within a module Use <code>acc_next_specparam()</code> to scan all specparams within a module		

The ACC routine **acc_fetch_paramtype()** shall return the value stored in a parameter or specparam. The value shall be returned as a double-precision floating-point number.

A parameter value can be stored as one of three data types:

- A double-precision floating-point number
- An integer value
- A string

Therefore, it can be necessary to call **acc_fetch_paramtype()** to determine the data type of the parameter value, as shown in the example in Figure 19-30.

The routine **acc_fetch_paramval()** returns values as type `double`. The values can be converted back to integers or character pointers using the C-language *cast* mechanism, as shown in Table 19-26. Note that some C-language compilers do not allow casting a double-precision value directly to a character pointer; it is therefore necessary to use a two-step cast to first convert the double value to an integer and then convert the integer to a character pointer.

If a character string is returned, it is placed in the ACC internal string buffer. See 18.9 for explanation of strings in ACC routines.

Table 19-26—Casting acc_fetch_paramval() return values

To convert to	Follow these steps
Integer	Cast the return value to the integer data type using the C-language cast operator (int): <pre>int_val=(int) acc_fetch_paramval(...);</pre>
String	Cast the return value to a character pointer using the C-language cast operators (char*)(int): <pre>str_ptr=(char*)(int) acc_fetch_paramval(...);</pre>

The example shown in Figure 19-30 presents a C-language application, `print_parameter_values`, that uses `acc_fetch_paramtype()` to display the values of all parameters within a module.

```
#include "acc_user.h"

print_parameter_values()
{
    handle    module_handle;
    handle    param_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*set development version*/
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for module*/
    module_handle = acc_handle_tfarg(1);

    /*scan all parameters in the module and display their values*/
    /* according to type*/
    param_handle = null;
    while(param_handle = acc_next_parameter(module_handle,param_handle) )
    {
        io_printf("Parameter %s has value: ",acc_fetch_fullname(param_handle));
        switch(acc_fetch_paramtype(param_handle) )
        {
            case accRealParam:
                io_printf("%lf\n", acc_fetch_paramval(param_handle) );
                break;
            case accIntegerParam:
                io_printf("%d\n", (int)acc_fetch_paramval(param_handle) );
                break;
            case accStringParam:
                io_printf("%s\n", (char*)(int)acc_fetch_paramval(param_handle) );
                break;
        }
    }
    acc_close();
}
```



Figure 19-30—Using `acc_fetch_paramval()`

19.25 acc_fetch_polarity()

acc_fetch_polarity()			
Synopsis:	Get the polarity of a path.		
Syntax:	acc_fetch_polarity(path_handle)		
Returns:	Type	Description	
	int	A predefined integer constant representing the polarity of a path: accPositive accNegative accUnknown	
Arguments:	Type	Name	Description
	handle	path_handle	Handle to a module path or data path

The ACC routine **acc_fetch_polarity()** shall return an integer constant that represents the polarity of the specified path. The polarity of a path describes how a signal transition at its source propagates to its destination in the absence of logic simulation events. The return value shall be one of the predefined integer constant polarity types listed in Table 19-27.

Table 19-27—Polarity types returned by acc_fetch_polarity()

Integer constant	Description
accPositive	A rise at the source causes a rise at the destination. A fall at the source causes a fall at the destination.
accNegative	A rise at the source causes a fall at the destination. A fall at the source causes a rise at the destination.
accUnknown	Unpredictable; a rise or fall at the source causes either a rise or fall at the destination.

The example shown in Figure 19-31 takes a path argument and returns the string corresponding to its polarity.

```

char *fetch_polarity_str(path)
{
    switch (acc_fetch_polarity(path)) {
        case accPositive: return("accPositive");
        case accNegative: return("accNegative");
        case accUnknown: return("accUnknown");
        default: return(null);
    }
}

```

Figure 19-31—Using acc_fetch_polarity()

19.26 acc_fetch_precision()

acc_fetch_precision()		
Synopsis:	Get the smallest time precision argument specified in all <code>`timescale</code> compiler directives in a given design.	
Syntax:	<code>acc_fetch_precision()</code>	
	Type	Description
Returns:	int	An integer value that represents a time precision
	Type	Name Description
Arguments:	None	
Related routines:	Use <code>acc_fetch_timescale_info()</code> to get the timescale and precision of a specific object	

The ACC routine **acc_fetch_precision()** shall return the smallest time precision argument specified in all ``timescale` compiler directives for a given design. The value returned shall be the order of magnitude of one second, as shown in Table 19-28.

Table 19-28—Value returned by acc_fetch_precision()

Integer value returned	Simulation time precision represented
2	100 s
1	10 s
0	1 s
-1	100 ms
-2	10 ms
-3	1 ms
-4	100 μ s
-5	10 μ s
-6	1 μ s
-7	100 ns
-8	10 ns
-9	1 ns
-10	100 ps
-11	10 ps
-12	1 ps
-13	100 fs
-14	10 fs
-15	1 fs

If there are no ``timescale` compiler directives specified for a design, **acc_fetch_precision()** shall return a value of 0 (1 s).

19.27 acc_fetch_pulsere()

acc_fetch_pulsere()			
Synopsis:	Get current pulse handling <i>reject_value</i> and <i>e_value</i> for a module path.		
Syntax:	acc_fetch_pulsere(path,r1,e1, r2,e2, r3,e3, r4,e4, r5,e5, r6,e6, r7,e7, r8,e8, r9,e9, r10,e10, r11,e11, r12,e12)		
Type		Description	
Returns:	bool	1 if successful; 0 if an error is encountered	
Arguments:	Type	Name	Description
	handle	path	Handle of module path
	double *	r1...r12	<i>reject_limit</i> values; the number of arguments is determined by accPathDelayCount
	double *	e1...e12	<i>e_limit</i> values; the number of arguments is determined by accPathDelayCount
Related routines:	Use acc_append_pulsere() to add to the existing pulse handling values Use acc_replace_pulsere() to replace existing pulse handling values Use acc_set_pulsere() to set pulse handling values as a percentage of the path delay		

The ACC routine **acc_fetch_pulsere()** shall obtain the current values controlling how pulses are propagated through module paths.

A *pulse* is defined as two transitions on the same path that occur in a shorter period of time than the path delay. Pulse control values determine whether a pulse should be rejected, propagated through to the output, or considered an *error*. The pulse control values consist of a *reject_limit* and an *e_limit* pair of values, where

- The *reject_limit* shall set a threshold for determining when to reject a pulse—any pulse less than the *reject_limit* shall not propagate to the output of the path
- The *e_limit* shall set a threshold for determining when a pulse is an error—any pulse less than the *e_limit* and greater than or equal to the *reject_limit* shall propagate a logic x to the path output
- A pulse that is greater than or equal to the *e_limit* shall propagate through to the path output

Table 19-29 illustrates the relationship between the *reject_limit* and the *e_limit*.

Table 19-29—Path pulse control example

When	The pulse on a module path output shall be
reject_limit = 10.5 e_limit = 22.6	Rejected if < 10.5 An error if ≥ 10.5 and < 22.6 Passed if ≥ 22.6

The number of pulse control values that **acc_fetch_pulsere()** shall retrieve is controlled using the ACC routine **acc_configure()** to set the delay count configuration parameter **accPathDelayCount**, as shown in Table 19-30.

Table 19-30—How the accPathDelayCount affects acc_fetch_pulsere()

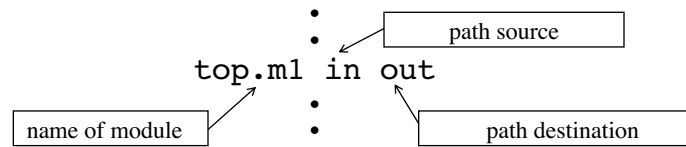
When accPathDelayCount is	acc_fetch_pulsere() shall retrieve
"1"	One pair of reject_limit and e_limit values: one pair for all transitions, r1 and e1
"2"	Two pairs of reject_limit and e_limit values: one pair for rise transitions, r1 and e1 one pair for fall transitions, r2 and e2
"3"	Three pairs of reject_limit and e_limit values: one pair for rise transitions, r1 and e1 one pair for fall transitions, r2 and e2 one pair for turn-off transitions, r3 and e3
"6" (the default)	Six pairs of reject_limit and e_limit values—a different pair for each possible transition among 0, 1, and Z: one pair for 0->1 transitions, r1 and e1 one pair for 1->0 transitions, r2 and e2 one pair for 0->Z transitions, r3 and e3 one pair for Z->1 transitions, r4 and e4 one pair for 1->Z transitions, r5 and e5 one pair for Z->0 transitions, r6 and e6
"12"	Twelve pairs of reject_limit and e_limit values—a different pair for each possible transition among 0, 1, X, and Z: one pair for 0->1 transitions, r1 and e1 one pair for 1->0 transitions, r2 and e2 one pair for 0->Z transitions, r3 and e3 one pair for Z->1 transitions, r4 and e4 one pair for 1->Z transitions, r5 and e5 one pair for Z->0 transitions, r6 and e6 one pair for 0->X transitions, r7 and e7 one pair for X->1 transitions, r8 and e8 one pair for 1->X transitions, r9 and e9 one pair for X->0 transitions, r10 and e10 one pair for X->Z transitions, r11 and e11 one pair for Z->X transitions, r12 and e12

The minimum number of pairs of reject_limit and e_limit arguments to pass to **acc_fetch_pulsere()** shall equal the value of **accPathDelayCount**. Any unused reject_limit and e_limit argument pairs shall be ignored by **acc_fetch_pulsere()** and can be dropped from the argument list.

If **accPathDelayCount** is not set explicitly, it shall default to 6, and therefore six pairs of pulse reject_limit and e_limit arguments have to be used when **acc_fetch_pulsere()** is called. Note that the value assigned to **accPathDelayCount** also affects **acc_append_delays()**, **acc_fetch_delays()**, **acc_replace_delays()**, **acc_append_pulsere()**, and **acc_replace_pulsere()**.

Pulse control values shall be retrieved using the timescale of the module that contains the path.

The example shown in Figure 19-32 shows how an application, `get_pulsevals`, uses `acc_fetch_pulsere()` to retrieve rise and fall pulse handling values of paths listed in a file called `path.dat`. The format of the file is shown in the following diagram.



```

#include <stdio.h>
#include "acc_user.h"

#define NAME_SIZE 256
get_pulsevals()
{
    FILE    *infile;
    char     mod_name[NAME_SIZE];
    char     pathin_name[NAME_SIZE], pathout_name[NAME_SIZE];
    handle   mod, path;
    double   rise_reject_limit, rise_e_limit, fall_reject_limit, fall_e_limit;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*set accPathDelayCount to return two pairs of pulse handling values,*/
    /* one each for rise and fall transitions*/
    acc_configure(accPathDelayCount, "2");

    /*read all module path specifications from file "path.dat"*/
    infile = fopen("path.dat", "r");
    while(fscanf(infile, "%s %s %s"
                mod_name, pathin_name, pathout_name) != EOF)
    {
        mod = acc_handle_object(mod_name);
        path = acc_handle_modpath(mod, pathin_name, pathout_name);
        if(acc_fetch_pulsere(path,
                            &rise_reject_limit, &rise_e_limit,
                            &fall_reject_limit, &fall_e_limit))
        {
            io_printf("rise reject limit = %lf, rise e limit = %lf\n",
                    rise_reject_limit, rise_e_limit);
            io_printf("fall reject limit = %lf, fall e limit = %lf\n",
                    fall_reject_limit, fall_e_limit);
        }
    }
    acc_close();
}

```

Figure 19-32—Using `acc_fetch_pulsere()`

19.28 acc_fetch_range()

acc_fetch_range()			
Synopsis:	Get the most significant bit and least significant bit range values for a vector.		
Syntax:	acc_fetch_range(vector_handle, msb, lsb)		
Type		Description	
Returns:	bool	Zero if successful; nonzero upon error	
Arguments:	Type	Name	Description
	handle	vector_handle	Handle to a vector net or register
	int *	msb	Pointer to an integer variable to hold the most significant bit of vector_handle
	int *	lsb	Pointer to an integer variable to hold the least significant bit of vector_handle
Related routines	Use acc_fetch_size() to get the number of bits in a vector		

The ACC routine **acc_fetch_range()** shall obtain the most significant bit (msb) and least significant bit (lsb) numbers of a vector.

The *msb* shall be the left range element, while the *lsb* shall be the right range element in the Verilog HDL source code.

The example shown in Figure 19-33 takes a handle to a module instance as its input. It then uses **acc_fetch_range()** to display the name and range of each vector net found in the module as: <name>[<msb>:<lsb>].

```
display_vector_nets()
{
    handle mod = acc_handle_tfarg(1);
    handle net;
    int    msb, lsb;

    io_printf ("Vector nets in module %s:\n",
               acc_fetch_fullname (mod));
    net = null;
    while (net = acc_next_net(mod, net))
        if (acc_object_of_type(net, accVector))
        {
            acc_fetch_range(net, &msb, &lsb);
            io_printf("  %s[%d:%d]\n",
                      acc_fetch_name(net), msb, lsb);
        }
}
```

Figure 19-33—Using acc_fetch_range()

19.29 acc_fetch_size()

acc_fetch_size()			
Synopsis:	Get the bit size of a net, register, or port.		
Syntax:	acc_fetch_size(object_handle)		
Returns:	Type	Description	
	int	Number of bits in the net, register, or port	
Arguments:	Type	Name	Description
	handle	object_handle	Handle to a net, register, port or integer, or a bit-select thereof

The ACC routine **acc_fetch_size()** shall return the number of bits of a net, register, or port.

The example shown in Figure 19-34 uses **acc_fetch_size()** to display the size of a vector net.

```
#include "acc_user.h"

void display_vector_size()
{
    handle    net_handle;
    int       size_in_bits;

    /* reset environment for ACC routines */
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get first argument passed to user-defined system task*/
    /* associated with this routine*/
    net_handle = acc_handle_tfarg(1);

    /*if net is a vector, find and display its size in bits*/
    if (acc_object_of_type(net_handle, accVector) )
    {
        size_in_bits = acc_fetch_size(net_handle);
        io_printf("Net %s is a vector of size %d\n",
                  acc_fetch_fullname(net_handle),size_in_bits);
    }
    else
        io_printf("Net %s is not a vector net\n",
                  acc_fetch_fullname(net_handle) );
}
```

Figure 19-34—Using acc_fetch_size()

19.30 acc_fetch_tfarg(), acc_fetch_itfarg()

acc_fetch_tfarg(), acc_fetch_itfarg()			
Synopsis:	Get the value of the specified argument of the system task or function associated with the PLI application; the value is returned as a double-precision number.		
Syntax:	<pre>acc_fetch_tfarg(argument_number) acc_fetch_itfarg(argument_number, tfinst)</pre>		
Type		Description	
Returns:	double	The value of the task/function argument, returned as a double-precision number	
Type		Name	Description
Arguments:	int	argument_number	Integer number that references the system task or function argument by its position in the argument list
	handle	tfinst	Handle to a specific instance of a user-defined system task or function
Related routines:	Use acc_fetch_tfarg_int() or acc_fetch_itfarg_int() to get the task/function argument value as an integer Use acc_fetch_tfarg_str() or acc_fetch_itfarg_str() to get the task/function argument value as a string Use acc_handle_tfinst() to get a handle to a specific instance of a user-defined system task or function		

The ACC routine **acc_fetch_tfarg()** shall return the value of arguments passed to the current instance of a user-defined system task or function. The ACC routine **acc_fetch_itfarg()** shall return the value of arguments passed to a specific instance of a user-defined system task or function, using a handle to the other task or function. The value is returned as a double-precision floating-point number.

Argument numbers shall start at *one* and increase from left to right in the order that they appear in the system task or function call.

If an argument number is passed in that is out of range for the number of arguments in the user-defined system task/function call, **acc_fetch_tfarg()** and **acc_fetch_itfarg()** shall return a value of **0.0** and generate a warning message if warnings are enabled. Note that the **acc_error_flag** is not set for an out-of-range index number.

If a user-defined system task/function argument that does not represent a valued object is referenced, **acc_fetch_tfarg()** and **acc_fetch_itfarg()** shall return a value of **0.0** and generate a warning message if warnings are enabled. Literal numbers, nets, registers, integer variables, and real variables all have values. Objects such as module instance names do not have a value. Note that the **acc_error_flag** is not set when a nonvalued argument is referenced.

The example shown in Figure 19-35 uses **acc_fetch_tfarg()**, **acc_fetch_tfarg_int()**, and **acc_fetch_tfarg_str()** to return the value of the first argument of a user-defined system task or function.

```
#include "acc_user.h"
#include "veriusser.h"

display_arg_value()
{
    int      arg_type;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*set development version*/
    acc_configure(accDevelopmentVersion, "1.5b.3");

    /*check type of argument*/
    io_printf("Argument value is ");

    switch(tf_typeep(1) )
    {
        case tf_readonlyreal:case tf_readwritereal:
            io_printf("%lf\n", acc_fetch_tfarg(1) );
            break;
        case tf_readonly:case tf_readwrite:
            io_printf("%d\n", acc_fetch_tfarg_int(1) );
            break;
        case tf_string:
            io_printf("%s\n", acc_fetch_tfarg_str(1) );
            break;
        default:
            io_printf("Error in argument specification\n");
            break;
    }
    acc_close();
}
```

returns value as a
double-precision
floating-point number

returns value as an
integer number

returns value as a
pointer to a
character string

Figure 19-35—Using `acc_fetch_tfarg()`, `acc_fetch_tfarg_int()`, and `acc_fetch_tfarg_str()`

19.31 acc_fetch_tfarg_int(), acc_fetch_itfarg_int()

acc_fetch_tfarg_int(), acc_fetch_itfarg_int()			
Synopsis:	Get the value of the specified argument of the system task or function associated with the PLI application; the value is returned as an integer number.		
Syntax:	acc_fetch_tfarg_int(argument_number) acc_fetch_itfarg_int(argument_number, tfinst)		
Type		Description	
Returns:	int	The value of the task/function argument, returned as an integer number	
Type		Name	Description
Arguments:	int	argument_number	Integer number that references the system task or function argument by its position in the argument list
	handle	tfinst	Handle to a specific instance of a user-defined system task or function
Related routines:	Use acc_fetch_tfarg() or acc_fetch_itfarg() to get the task/function argument value as a double Use acc_fetch_tfarg_str() or acc_fetch_itfarg_str() to get the task/function argument value as a string Use acc_handle_tfinst() to get a handle to a specific instance of a user-defined system task or function		

The ACC routine **acc_fetch_tfarg_int()** shall return the value of arguments passed to the current user-defined system task or function. The ACC routine **acc_fetch_itfarg_int()** shall return the value of arguments passed to a specific instance of a user-defined system task and function, using a handle to the task or function. The value is returned as an integer number.

Argument numbers shall start at *one* and increase from left to right in the order that they appear in the system task or function call.

If an argument number is passed in that is out of range for the number of arguments in the user-defined system task/function call, **acc_fetch_tfarg_int()** and **acc_fetch_itfarg_int()** shall return a value of **0** and generate a warning message if warnings are enabled. Note that the **acc_error_flag** is not set for an out-of-range index number.

If a user-defined system task/function argument that does not represent a valued object is referenced, **acc_fetch_tfarg_int()** and **acc_fetch_itfarg_int()** shall return a value of **0** and generate a warning message if warnings are enabled. Literal numbers, nets, registers, integer variables, and real variables all have values. Objects such as module instance names do not have a value. Note that the **acc_error_flag** is not set when a nonvalued argument is referenced.

Refer to Figure 19-35 in 19.30 for an example of using **acc_fetch_tfarg_int()**.

19.32 acc_fetch_tfarg_str(), acc_fetch_itfarg_str()

acc_fetch_tfarg_str(), acc_fetch_itfarg_str()			
Synopsis:	Get the value of the specified argument of the system task or function associated with the PLI application; the value is returned as a pointer to a character string.		
Syntax:	acc_fetch_tfarg_str(argument_number) acc_fetch_itfarg_str(argument_number, tfinst)		
Type		Description	
Returns:	char *	The value of the task/function argument, returned as a pointer to a character string	
Type		Name	Description
Arguments:	int	argument_number	Integer number that references the system task or function argument by its position in the argument list
	handle	tfinst	Handle to a specific instance of a user-defined system task or function
Related routines:	Use acc_fetch_tfarg() or acc_fetch_itfarg() to get the task/function argument value as a double Use acc_fetch_tfarg_int() or acc_fetch_itfarg_int() to get the task/function argument value as an integer Use acc_handle_tfinst() to get a handle to a specific instance of a user-defined system task or function		

The ACC routine **acc_fetch_tfarg_str()** shall return the value of arguments passed to the current instance of a user-defined system task or function. The ACC routine **acc_fetch_itfarg_str()** shall return the value of arguments passed to a specific instance or a user-defined system task or function, using a handle to the task or function. The value shall be returned as a pointer to a character string. The return value for this routine is placed in the ACC internal string buffer. See 18.9 for explanation of strings in ACC routines.

Argument numbers shall start at *one* and increase from left to right in the order that they appear in the system task or function call.

If an argument number is passed in that is out of range for the number of arguments in the user-defined system task/function call, **acc_fetch_tfarg_str()** and **acc_fetch_itfarg_str()** shall return a value of `null` and generate a warning message if warnings are enabled. Note that the `acc_error_flag` is not set for an out-of-range index number.

If a user-defined system task/function argument that does not represent a valued object is referenced, **acc_fetch_tfarg_str()** and **acc_fetch_itfarg_str()** shall return a value of `null` and generate a warning message if warnings are enabled. Literal numbers, nets, registers, integer variables, and real variables all have values. Objects such as module instance names do not have a value. Note that the `acc_error_flag` is not set when a nonvalued argument is referenced.

Refer to Figure 19-35 in 19.30 for an example of using **acc_fetch_tfarg_str()**.

19.33 acc_fetch_timescale_info()

acc_fetch_timescale_info()			
Synopsis:	Get timescale information for an object or for an active \$timeformat system task invocation.		
Syntax:	acc_fetch_timescale_info(object_handle, timescale_p)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	handle	object_handle	Handle of a module instance, module definition, PLI user-defined system task/function call, or null
	p_timescale_info	timescale_p	Pointer to a variable defined as a s_timescale_info structure
Related routines:	Use acc_fetch_precision() to fetch the smallest timescale precision in a design		

The ACC routine **acc_fetch_timescale_info()** shall obtain the timescale information for an object or for an active \$timeformat built-in system task invocation. The timescale returned shall be based on the type of object handle, as defined in Table 19-31.

Table 19-31—Return values from acc_fetch_timescale_info()

If the object_handle is	acc_fetch_timescale_info() shall return
A handle to a module instance or module definition	The timescale for the corresponding module definition
A handle to a user-defined system task or function	The timescale for the corresponding module definition that represents the parent module instance of the object
null	The timescale for an active \$timeformat system task invocation

The routine **acc_fetch_timescale_info()** shall return a value to an s_timescale_info structure pointed to by the *timescale_p* argument. This structure is declared in the file **acc_user.h**, as shown in Figure 19-36.

```
typedef struct t_timescale_info {
    short unit;
    short precision;
} s_timescale_info, *p_timescale_info;
```

Figure 19-36—s_timescale_info data structure

- The term *unit* is a short integer that shall represent the timescale unit in all cases of *object*
- The term *precision* is a short integer that shall represent the timescale precision. In the case of a null object handle, precision shall be the number of decimal points specified in the active \$timeformat system task invocation.

The value returned for *unit* and *precision* shall be the order of magnitude of 1 s, as shown in Table 19-32.

Table 19-32—Value returned by `acc_fetch_timescale_info()`

Integer value returned	Time unit r
2	100 s
1	10 s
0	1 s
-1	100 ms
-2	10 ms
-3	1 ms
-4	100 μ s
-5	10 μ s
-6	1 μ s
-7	100 ns
-8	10 ns
-9	1 ns
-10	100 ps
-11	10 ps
-12	1 ps
-13	100 fs
-14	10 fs
-15	1 fs

For example, a call to

```
acc_fetch_timescale_info(obj, timescale_p)
```

Where `obj` is defined in a module that has ``timescale 1us/1ns` specified for its definition, shall return

```
timescale_p->unit: -6
timescale_p->precision: -9
```

19.34 acc_fetch_type()

acc_fetch_type()			
Synopsis:	Get the type of an object.		
Syntax:	<code>acc_fetch_type(object_handle)</code>		
Returns:	Type		Description
	int		A predefined integer constant from the list shown in 18.6
Arguments:	Type	Name	Description
	handle	object_handle	Handle of the object
Related routines:	Use <code>acc_fetch_type()</code> to get the general type classification of an object		
	Use <code>acc_fetch_type_str()</code> to get the type as an character string		

The ACC routine **acc_fetch_type()** shall return the type of an object. The *type* is a general classification of a Verilog HDL object, represented as a predefined constant (defined in `acc_user.h`). Refer to 18.6 for a list of all of the *type* constants that can be returned by **acc_fetch_type()**.

Many Verilog HDL objects can have a *type* and a *fulltype*. The type of an object is its general Verilog HDL type classification. The fulltype is the specific type of the object. Table 19-33 illustrates the difference between the type of an object and the fulltype of the same object.

Table 19-33—The difference between the type and the fulltype of an object

For a handle to	<code>acc_fetch_type()</code> shall return	<code>acc_fetch_fulltype()</code> shall return
A setup timing check	accTchk	accSetup
An and gate primitive	accPrimitive	accAndGate
A sequential UDP	accPrimitive	accSeqPrim

The example shown in Figure 19-37 uses **acc_fetch_type()** to identify the type of an object (the functions `display_primitive_type` and `display_timing_check_type` used in this example are presented in the usage examples in 19.19).


```
#include "acc_user.h"

display_object_type()
{
    handle object_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    object_handle = acc_handle_tfarg(1);

    /*display object type*/
    switch(acc_fetch_type(object_handle) )
    {
        case accModule:
            io_printf("Object is a module\n");
            break;
        case accNet:
            io_printf("Object is a net\n");
            break;
        case accPath:
            io_printf("Object is a module path\n");
            break;
        case accPort:
            io_printf("Object is a module port\n");
            break;
        case accPrimitive:
            display_primitive_type(object_handle);
            break;
        case accTchk:
            display_timing_check_type(object_handle);
            break;
        case accTerminal:
            io_printf("Object is a primitive terminal\n");
            break;
    }
    acc_close();
}
```

Figure 19-37—Using acc_fetch_type()

19.35 acc_fetch_type_str()

acc_fetch_type_str()			
Synopsis:	Get a string that indicates the type of its argument.		
Syntax:	acc_fetch_type_str (type)		
Type		Description	
Returns:	char *	Pointer to a character string	
Type		Name	Description
Arguments:	integer	type	A predefined integer constant that stands for an object type or fulltype
Related routines:	Use acc_fetch_type() to get the type of an object as an integer constant Use acc_fetch_fulltype() to get the fulltype of an object as an integer constant		

The ACC routine **acc_fetch_type_str()** shall return the character string that specifies the type of its argument. The argument passed to **acc_fetch_type_str()** should be an integer value returned from either **acc_fetch_type()** or **acc_fetch_fulltype()**.

The return value for this routine is placed in the ACC internal string buffer. See 18.9 for explanation of strings in ACC routines.

In the example shown in Figure 19-38, a handle to an argument is passed to a C application. The application displays the name of the object and the type of the object.

```
#include "acc_user.h"
void    display_object_type(object)
handle  object;
{
    int type = acc_fetch_type(object);

    io_printf("Object %s is of type %s \n",
        acc_fetch_fullname(object),
        acc_fetch_type_str(type));
}
```

Figure 19-38—Using acc_fetch_type_str()

In this example, if the application is passed a handle to an object named **top.param1**, the application shall produce the following output:

```
Object top.param1 is of type accParameter
```

The output string, **accParameter**, is the name of the integer constant that represents the parameter type.

19.36 acc_fetch_value()

acc_fetch_value()			
Synopsis:	Get the logic or strength value of a net, register, or variable.		
Syntax:	acc_fetch_value(object_handle, format_string, value)		
Returns:	Type	Description	
	char *	Pointer to a character string	
Arguments:	Type	Name	Description
	handle	object_handle	
	quoted string or char *	format_string	A literal string or character string pointer with one of the following specifiers for formatting the return value: "%b" "%d" "%h" "%o" "%v" "%%"
Optional	s_acc_value *	value	Pointer to a structure with the retrieved logic values and strength; used when format string is "%%" (should be set to null when not used)
Related routines:	Use acc_fetch_size() to determine how bits wide the object is Use acc_set_value() to put a logic value on the object		

The ACC routine **acc_fetch_value()** shall return *logic* simulation values for scalar or vector nets, registers, and variables; **acc_fetch_value()** shall return *strength* values for scalar nets and scalar registers only.

The routine **acc_fetch_value()** shall return the logic and strength values in one of two ways:

- The value can be returned as a string
- The value can be returned as an *aval/bval* pair in a predefined structure.

The return method used by **acc_fetch_value()** shall be controlled by the *format_string* argument, as shown in Table 19-34.

Table 19-34—How acc_fetch_value() returns values

format_specifier	Return format	Description
"%b"	binary	Value shall be retrieved as a string, and a character pointer to the string shall returned
"%d"	decimal	
"%h"	hexadecimal	
"%o"	octal	
"%v"	strength	
"%%"	aval/bval pair	Value shall retrieved and placed in a structure variable pointed to by the optional <i>value</i> argument

Note that strings are placed in a temporary buffer, and they should be preserved if not used immediately. Refer to 18.9 for details on preserving strings.

When a format_string of “%%” is specified, **acc_fetch_value()** shall retrieve the logic value and strength to a predefined structure, **s_acc_value**, which is defined in **acc_user.h** and is shown below [note that this structure definition is also used with the **acc_set_value()** routine].

```
typedef struct t_setval_value
{
    int format;
    union
    {
        char *str;
        int scalar;
        int integer;
        double real;
        p_acc_vecval vector;
    } value;
} s_setval_value, *p_setval_value, s_acc_value, *p_acc_value;
```

Figure 19-39—s_acc_value structure

To use the “%%” format_string to retrieve values to a structure requires the following steps:

- A structure variable shall first be declared of type **s_acc_value**.
- The format field of the structure has to be set to a predefined constant. The format controls which fields in the **s_acc_value** structure shall be used when **acc_fetch_value()** returns the value. The predefined constants for the format shall be one of the constants shown in Table 19-35.
- The structure variable has to be passed as the third argument to **acc_fetch_value()**.

Table 19-35—Format constants for the s_acc_value structure

Format constant	acc_fetch_value() shall return the value to the s_acc_value union field
accBinStrVal	str
accOctStrVal	str
accDecStrVal	str
accHexStrVal	str
accScalarVal	scalar
accIntVal	integer
accRealVal	real
accStringVal	str
accVectorVal	vector

For example, calling **acc_fetch_value()** with the following setup would return a string in the **value.str** field. (This is essentially the same as using **acc_fetch_value()** with a **%b** format string.)

```
s_acc_value value;
value.format = accBinStrVal;
```

```
(void)acc_fetch_value(Net,"%%", &value);
```

If the format field for **acc_fetch_value()** is set to **accVectorVal**, then the value shall be placed in the record(s) pointed to by the value field. The value field shall be a pointer to an array of one or more **s_acc_vecval** structures. The **s_acc_vecval** structure is defined in the **acc_user.h** file and is listed in Figure 19-40. The structure shall contain two integers: *aval* and *bval*. Each **s_acc_vecval** record shall represent 32 bits of a vector. The encoding for each bit value is shown in Table 19-36.

```
typedef struct t_acc_vecval
{
    int aval;
    int bval;
} s_acc_vecval, *p_acc_vecval;
```

Figure 19-40—s_acc_vecval structure

Table 19-36—Encoding of bits in the s_acc_vecval structure

aval	bval	Value
0	0	0
1	0	1
0	1	Z
1	1	X

The array of **s_acc_vecval** structures shall contain a record for every 32 bits of the vector, plus a record for any remaining bits. If a vector has *N* bits, then there shall be $(N-1)/32 + 1$ **s_acc_vecval** records. The routine **acc_fetch_size()** can be used to determine the value of *N*. The lsb of the vector shall be represented by the lsb of the first record of **s_acc_vecval** array. The 33rd bit of the vector shall be represented by the lsb of the second record of the array, and so on. See Figure 19-42 for an example of **acc_fetch_value()** used in this way.

Note that when using *aval/bval* pairs, the **s_acc_value** record and the appropriately sized **s_acc_vecval** array shall first be declared. Setting the second parameter to **acc_fetch_value()** to “%%” and the third parameter to **null** shall be an error.

The example application shown in Figure 19-41 uses **acc_fetch_value()** to retrieve the logic values of all nets in a module as strings.

```
#include "acc_user.h"
display_net_values()
{
    handle mod, net;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*get handle for module*/
    mod = acc_handle_tfarg(1);

    /*get all nets in the module and display their values*/
    /* in binary format*/
    net = null;
    while(net = acc_next_net(mod, net))
        io_printf("Net value: %s\n", acc_fetch_value(net, "%b", null));
    acc_close();
}
```

Figure 19-41—Using `acc_fetch_value()` to retrieve the logic values as strings

The example in Figure 19-42 uses **acc_fetch_value()** to retrieve a value into a structure, and then prints the value. The example assumes the application, **my_fetch_value**, is called from the following user-defined system task:

```
$my_fetch_value(R);
```

```
#include "acc_user.h"

int my_fetch_value()
{
    handle reg = acc_handle_tfarg(1);
    int size = ((acc_fetch_size(reg) - 1) / 32) + 1;
    s_acc_value value;
    int index1, min_size;
    static char table[4] = {'0', '1', 'z', 'x'};
    static char outString[33];

    io_printf("The value of %s is ", acc_fetch_name(reg));

    value.format = accVectorVal;
    value.value.vector = (p_acc_vecval)malloc(size * sizeof(s_acc_vecval));

    (void)acc_fetch_value(reg, "%%", &value);

    for (index1 = size - 1; index1 >= 0; index1--)
    {
        register int index2;
        register int abits = value.value.vector[index1].aval;
        register int bbits = value.value.vector[index1].bval;

        if (index1 == size - 1)
        {
            min_size = (acc_fetch_size(reg) % 32);
            if (!min_size)
                min_size = 32;
        }
        else
            min_size = 32;
        outString[min_size] = '\0';
        min_size--;
        outString[min_size] = table[((bbits & 1) << 1) | (abits & 1)];
        abits >>= 1;

        for (index2 = min_size - 1; index2 >= 0; index2--)
        {
            outString[index2] = table[(bbits & 2) | (abits & 1)];
            abits >>= 1;
            bbits >>= 1;
        }
        io_printf("%s", outString);
        {
            io_printf("\n");
            return(0);
        }
    }
}
```

Figure 19-42—Using `acc_fetch_value()` to retrieve values into a data structure

19.37 acc_free()

acc_free()			
Synopsis:	Frees memory allocated by acc_collect() .		
Syntax:	acc_free (handle_array_pointer)		
Type		Description	
Returns:	void	No return	
Type		Name	Description
Arguments:	handle *	Handle_array_pointer	Pointer to the array of handles allocated by acc_collect()
Related routines:	Use acc_collect() to collect handles returned by acc_next_ routines		

The ACC routine **acc_free()** shall deallocate memory that was allocated by the routine **acc_collect()**.

The example shown in Figure 19-43 uses **acc_free()** to deallocate memory allocated by **acc_collect()** to collect handles to all nets in a module.

```
#include "acc_user.h"

display_nets()
{
    handle    *list_of_nets, module_handle;
    int       net_count, i;

    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for module*/
    module_handle = acc_handle_tfarg(1);

    /*collect and display all nets in the module*/
    list_of_nets = acc_collect(acc_next_net, module_handle, &net_count);
    for(i=0; i < net_count; i++)
        io_printf("Net name is: %s\n", acc_fetch_name(list_of_nets[i]));

    /*free memory used by array list_of_nets*/
    acc_free(list_of_nets);

    acc_close();
}
```

Figure 19-43—Using acc_free()

19.38 acc_handle_by_name()

acc_handle_by_name()			
Synopsis:	Get the handle to any named object based on its name and scope.		
Syntax:	<code>acc_handle_by_name(object_name, scope_handle)</code>		
Type		Description	
Returns:	handle	A handle to the specified object	
Type		Name	Description
Arguments:	quoted string or char *	object_name	Literal name of an object or a character string pointer to the object name
	handle	scope_handle	Handle to scope, or null
Related Routines	Use <code>acc_set_scope()</code> to set a local scope in which <code>acc_handle_by_name()</code> shall search Use <code>acc_handle_object()</code> to get a handle based on the local instance name of an object		

The ACC routine **acc_handle_by_name()** shall return the handle to any named object based on its specified name and scope. The routine can be used in two ways, as shown in Table 19-37.

Table 19-37—How acc_handle_by_name() works

When the <i>scope_handle</i> is	acc_handle_by_name() shall
A valid scope handle	Search for the <i>object_name</i> in the scope specified
null	Treat the <i>object_name</i> as a full hierarchical name

The routine **acc_handle_by_name()** combines the functionality of **acc_set_scope()** and **acc_handle_object()**, making it possible to obtain handles for objects that are not in the local scope without having to first change scopes.

Table 19-38 lists the objects in a Verilog HDL description for which **acc_handle_by_name()** shall return a handle.

Table 19-38—Named objects supported by acc_handle_by_name()

Modules	Parameters
Primitives	Specparams
Nets	Named blocks
Registers	Verilog HDL tasks
Variables	Verilog HDL functions
Named events	

The routine **acc_handle_by_name()** does not return handles for module paths, intermodule paths, data paths, or ports. Use an appropriate **acc_next_** or other ACC routines for these objects.

The example shown in Figure 19-44 uses **acc_handle_by_name()** to set the scope and get the handle to an object if the object is in the module.

```
#include "acc_user.h"

is_net_in_module(module_handle, net_name)
handle module_handle;
char *net_name;
{
    handle net_handle;
    handle load_handle, load_net_handle;

    /*set scope to module and get handle for net */
    net_handle = acc_handle_by_name(net_name, module_handle);

    if (net_handle)
        io_printf("Net %s found in module %s\n",
                  net_name,
                  acc_fetch_fullname(module_handle) );
    else
        io_printf("Net %s not found in module %s\n",
                  net_name,
                  acc_fetch_fullname(module_handle) );
}
```

Figure 19-44—Using acc_handle_by_name()

Note that in this example

```
net_handle = acc_handle_by_name(net_name, module_handle);
```

could also have been written as follows:

```
acc_set_scope(module_handle);
net_handle = acc_handle_object(net_name);
```

19.39 **acc_handle_calling_mod_m()**

acc_handle_calling_mod_m()		
Synopsis:	Get a handle to the module containing the instance of the user-defined system task or function that called the PLI application.	
Syntax:	<code>acc_handle_calling_mod_m()</code>	
	Type	Description
Returns:	handle	Handle to a module
	Type	Name
Arguments:	None	

The ACC routine **acc_handle_calling_mod_m()** shall return a handle to the module that contains the instance of the user-defined system task or function that called the PLI application.

19.40 acc_handle_condition()

acc_handle_condition()			
Synopsis:	Get a handle to the conditional expression of a module path, data path, or timing check terminal.		
Syntax:	acc_handle_condition(path_handle)		
Returns:	Type	Description	
	handle	Handle to a conditional expression	
Arguments:	Type	Name	Description
	handle	path_handle	Handle to a module path, data path, or timing check terminal

The ACC routine **acc_handle_condition()** shall return a handle to a conditional expression for the specified module path, data path, or timing check terminal. The routine shall return `null` when

- The module path, data path, or timing check terminal has no condition specified
- The module path has an **ifnone** condition specified

To determine if a module path has an **ifnone** condition specified, use the ACC routine **acc_object_of_type()** to check for the property type of **accModPathHasIfnone**.

The example shown in Figure 19-45 provides functionality to see if a path is conditional, and, if it is, whether it is level-sensitive or edge-sensitive. The application assumes that the input is a valid handle to a module path.

```

bool is_path_conditional(path)
{
    if (acc_handle_condition(path) )
        return(TRUE);
    else
        return(FALSE);
}

bool is_level_sensitive(path)
{
    bool flag;
    handle path_in = acc_next_input(path, null);

    if (is_path_conditional(path) && acc_fetch_edge(path_in))
        flag = FALSE; /* path is edge-sensitive */
    else
        flag = TRUE; /* path is level_sensitive */

    acc_release_object(path_in);

    return (flag);
}

```

Figure 19-45—Using acc_handle_condition()

19.41 acc_handle_conn()

acc_handle_conn()			
Synopsis:	Get the handle to the net connected to a primitive terminal, path terminal, or timing check terminal.		
Syntax:	acc_handle_conn(<i>terminal_handle</i>)		
Type		Description	
Returns:	handle	Handle of a net	
Type		Name	Description
Arguments:	handle	terminal_handle	Handle of the primitive terminal, path terminal, or timing check terminal
Related routines:	Use acc_handle_terminal() or acc_next_terminal() to obtain a terminal_handle		

The ACC routine **acc_handle_conn()** shall return a handle to the net connected to a primitive terminal, path terminal, or timing check terminal. This handle can then be passed to other ACC routines to traverse a design hierarchy or to extract information about the design.

The example shown in Figure 19-46 displays the net connected to the output terminal of a gate.

```
#include "acc_user.h"
display_driven_net()
{
    handle  gate_handle, terminal_handle, net_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for the gate*/
    gate_handle = acc_handle_tfarg(1);

    /*get handle for the gate's output terminal*/
    terminal_handle = acc_handle_terminal(gate_handle, 0);

    /*get handle for the net connected to the output terminal*/
    net_handle = acc_handle_conn(terminal_handle);

    /*display net name*/
    io_printf("Gate %s drives net %s\n",
              acc_fetch_fullname(gate_handle),
              acc_fetch_name(net_handle) );
    acc_close();
}
```

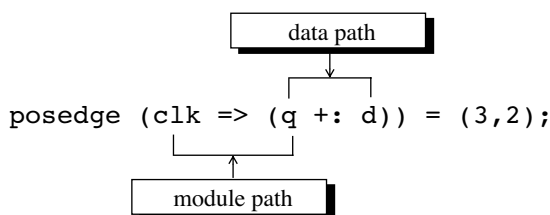
Figure 19-46—Using acc_handle_conn()

19.42 acc_handle_datapath()

acc_handle_datapath()			
Synopsis:	Get a handle to a data path for an edge-sensitive module path.		
Syntax:	acc_handle_datapath(modpath_handle)		
Returns:	Type	Description	
	handle	Handle of a data path	
Arguments:	Type	Name	Description
	handle	modpath_handle	Handle to a module path

The ACC routine **acc_next_datapath()** shall return a handle to the data path associated with an edge-sensitive module path. If there is no data path, `null` shall be returned.

A data path is part of the Verilog HDL description for edge-sensitive module paths, as illustrated below:



The example shown in Figure 19-47 uses **acc_handle_datapath()** to find the data path corresponding to the specified module path and displays the source and destination port names for the data path.

```

display_datapath_terms(modpath)
handle modpath;
{
    handle datapath = acc_handle_datapath(modpath);
    handle pathin  = acc_next_input(datapath, null);
    handle pathout = acc_next_output(datapath, null);

    /* there is only one input and output to a datapath */
    io_printf("DATAPATH INPUT:      %s\n", acc_fetch_fullname(pathin));
    io_printf("DATAPATH OUTPUT:   %s\n", acc_fetch_fullname(pathout));
    acc_release_object(pathin);
    acc_release_object(pathout);
}
  
```

Figure 19-47—Using acc_handle_datapath()

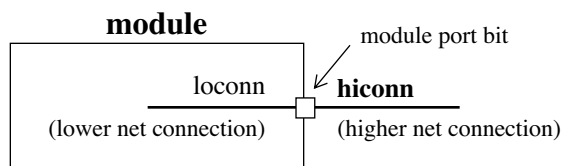
19.43 acc_handle_hiconn()

acc_handle_hiconn()			
Synopsis:	Get the hierarchically higher net connection to a scalar module port or a bit-select of a vector port.		
Syntax:	acc_handle_hiconn(port_ref_handle)		
Type		Description	
Returns:	handle	Handle of a net	
Type		Name	Description
Arguments:	handle	port_ref_handle	Handle to a scalar port or a bit-select of a vector port
Related routines:	Use acc_next_hiconn() to find all nets connected to a scalar port or bit-select of a port Use acc_handle_loconn() to get the hierarchically lower net connection of a port		

The ACC routine **acc_handle_hiconn()** shall return the hierarchically higher net connection for a scalar port or a bit-select of one of the following:

- Vector port
- Part-select of a port
- Concatenation of scalar ports, vector ports, part-selects of ports, or other concatenations

The hiconn is the net connected one level above the hierarchical scope of a module port, as illustrated below:



The example shown in Figure 19-48 uses **acc_handle_hiconn()** and **acc_handle_loconn()** to display the higher and lower connections of a module port.

```
display_port_info(mod, index)
handle mod;
int index;
{
    handle port = acc_handle_port (mod, index);
    handle hiconn, loconn, port_bit;

    if (acc_fetch_size(port) = 1) {
        hiconn = acc_handle_hiconn (port);
        loconn = acc_handle_loconn (port);
        io_printf ("      hi: %s  lo: %s\n",
            acc_fetch_fullname(hiconn), acc_fetch_fullname(loconn));
    }
    else {
        port_bit = null;
        while (port_bit = acc_next_bit (port, port_bit))
        {
            hiconn = acc_handle_hiconn (port_bit);
            loconn = acc_handle_loconn (port_bit);
            io_printf ("      hi: %s  lo: %s\n",
                acc_fetch_fullname(hiconn), acc_fetch_fullname(loconn));
        }
    }
}
```

Figure 19-48—Using `acc_handle_hiconn()` and `acc_handle_loconn()`

19.44 **acc_handle_interactive_scope()**

acc_handle_interactive_scope()			
Synopsis:	Get a handle to the current interactive scope of the software tool.		
Syntax:	<code>acc_handle_interactive_scope()</code>		
Returns:	Type	Description	
	handle	Handle of a Verilog hierarchy scope	
Arguments:	Type	Name	Description
	None		
Related routines:	Use <code>acc_fetch_type()</code> or <code>acc_fetch_fulltype()</code> to determine the scope type returned Use <code>acc_set_interactive_scope()</code> to change the interactive scope		

The ACC routine **acc_handle_interactive_scope()** shall return a handle to the Verilog HDL design scope where the interactive mode of a software product is currently pointing.

A scope shall be

- A top-level module
- A module instance
- A named begin-end block
- A named fork-join block
- A Verilog HDL task
- A Verilog HDL function

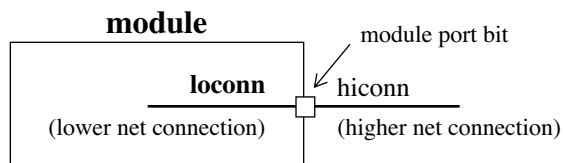
19.45 acc_handle_loconn()

acc_handle_loconn()			
Synopsis:	Gets the hierarchically lower net connection to a scalar module port or a bit-select of a vector port.		
Syntax:	acc_handle_loconn(port_ref_handle)		
Type		Description	
Returns:	handle	Handle of a net	
Type		Name	Description
Arguments:	handle	port_ref_handle	Handle to a scalar port or a bit-select of a vector port
Related routines:	Use acc_next_loconn() to find all nets connected to a scalar port or bit-select of a port Use acc_handle_hiconn() to get the hierarchically higher net connection of a port		

The ACC routine **acc_handle_loconn()** shall return the hierarchically lower net connection for a scalar port or a bit-select of one of the following:

- Vector port
- Part-select of a port
- Concatenation of scalar ports, vector ports, part-selects of ports, or other concatenations

The loconn is the net connected within the hierarchical scope of a module port, as illustrated below:



Refer to the usage example in 19.43 for an example of using **acc_handle_loconn()**.

19.46 acc_handle_modpath()

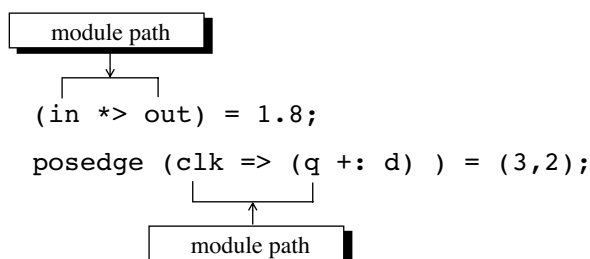
acc_handle_modpath()			
Synopsis:	Gets a handle to a module path.		
Syntax:	acc_handle_modpath(module_handle, source_name, destination_name, source_handle, destination_handle)		
Type		Description	
Returns:	handle	Handle of a module path	
Arguments:	Type	Name	Description
	handle	module_handle	Handle of the module
	quoted string or char *	source_name	Literal string or character string pointer with the name of a net connected to a module path source
	quoted string or char *	destination_name	Literal string or character string pointer with the name of a net connected to a module path destination
Optional	handle	source_handle	Handle of a net connected to a module path source (used when accEnableArgs is set and <i>source_name</i> is null)
Optional	handle	destination_handle	Handle of a net connected to a module path destination (used when accEnableArgs is set and <i>destination_name</i> is null)
Related routines:	Use acc_configure(accEnableArgs, "acc_handle_modpath") to use the source_handle and destination_handle		

The ACC routine **acc_handle_modpath()** shall return a handle to a module path. The routine shall be used in one of two ways, controlled by the configuration of the parameter **accEnableArgs**, as shown in Table 19-39.

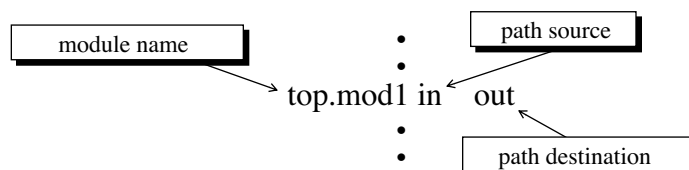
Table 19-39—How acc_handle_modpath() works

Setting of accEnableArgs	acc_handle_modpath() shall
"no_acc_handle_modpath" (the default setting)	Use the name arguments and ignore both handle arguments (the handle arguments can be dropped)
"acc_handle_modpath" and either <i>source_name</i> or <i>destination_name</i> is null	Use the handle argument of the null name argument; if the name argument is not null, the name shall be used and the associated handle argument ignored

A module path is the specify block path for delays in the Verilog HDL description. For example:



The example shown in Figure 19-49 uses **acc_handle_modpath()** to obtain handles for paths that connect the sources and destinations listed in the file **pathconn.dat**. The format of **pathconn.dat** is shown below.



```

#include <stdio.h>
#include "acc_user.h"
#define NAME_SIZE 256
get_paths()
{
    FILE      *infile;
    char      mod_name[NAME_SIZE], src_name[NAME_SIZE], dest_name[NAME_SIZE];
    handle    path_handle, mod_handle;

    /*initialize the environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*set accPathDelimStr to "_"*/
    acc_configure(accPathDelimStr, "_");

    /*read delays from file - "r" means read only*/
    infile = fopen("pathconn.dat","r");
    while (fscanf(infile,"%s %s %s",mod_name,src_name,dest_name) != EOF)
    {
        /*get handle for module mod_name*/
        mod_handle = acc_handle_object(mod_name);
        path_handle = acc_handle_modpath(mod_handle, src_name, dest_name);
        if (!acc_error_flag)
            io_printf("Path %s was found\n",
                      acc_fetch_fullname(path_handle) );
        else
            io_printf("Path %s_%s was not found\n",src_name,dest_name);
    }
    acc_close();
}
  
```

Figure 19-49—Using `acc_handle_modpath()`

19.47 **acc_handle_notifier()**

acc_handle_notifier()			
Synopsis:	Get the notifier register associated with a particular timing check.		
Syntax:	acc_handle_notifier (tchk)		
Returns:	Type	Description	
	handle	Handle to a timing check notifier	
Arguments:	Type	Name	Description
	handle	tchk	Handle of a timing check
Related routines:	Use acc_handle_tchk() to get a handle to a specific timing check Use acc_next_tchk() to get handles to all timing checks in a module		

The ACC routine **acc_handle_notifier()** shall return a handle to the notifier register associated with a timing check.

The example shown in Figure 19-50 uses **acc_handle_notifier()** to display the name of a notifier associated with a timing check.

```
display_notifier(tchk)
handle tchk;
{
    handle ntfy;

    ntfy = acc_handle_notifier(tchk);
    if (ntfy)
        io_printf("Tchk notifier: %s\n", acc_fetch_fullname(ntfy) );
    else
        io_printf("Tchk has no notifier\n");
}
```

Figure 19-50—Using acc_handle_notifier()

19.48 acc_handle_object()

acc_handle_object			
Synopsis:	Get a handle for any named object.		
Syntax:	<code>acc_handle_object(object_name)</code>		
Returns:	Type	Description	
	handle	Handle to an object	
Arguments:	Type	Name	Description
	quoted string or char *	object_name	Literal string or character string pointer with the full or relative hierarchical path name of an object
Related routines:	Use <code>acc_set_scope()</code> to set the scope when using relative path names for an object		

The ACC routine **acc_handle_object()** shall return a handle to a named object. The *object_name* argument shall be a quoted string or pointer to a string. The *object_name* can include a Verilog hierarchy path. The routine shall search for the object using the rules given in Table 19-40.

Table 19-40—How acc_handle_object() works

If <i>object_name</i> contains	acc_handle_object() shall
<i>A full hierarchical path name</i> (a full hierarchical path begins with a top-level module)	Return a handle to the object; no search is performed
<i>No path name</i> or <i>a relative path name</i>	Search for object starting in the current PLI scope, following search rules defined in 12.5

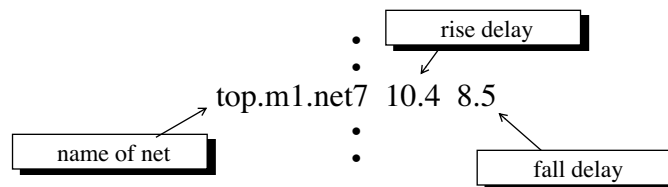
The ACC routine **acc_handle_object()** shall use the current PLI scope as a basis for searching for objects. The PLI scope shall default to the Verilog scope of the system task/function that called the C application of the user, and it can be changed from within the application using **acc_set_scope()**.

Table 19-41 lists the objects in a Verilog HDL description for which **acc_handle_object()** shall return a handle.

Table 19-41—Named objects

Modules	Variables
Module ports	Named events
Module paths	Parameters
Data paths	Specparams
Primitives	Named blocks
Nets	Verilog HDL tasks
Registers	Verilog HDL functions

The example shown in Figure 19-51 uses **acc_handle_object()** to retrieve handles for net names read from a file called **primdelay.dat**. The format of the file is shown below. Note that this example assumes that each net is driven by only one primitive.



```
#include <stdio.h>
#include "acc_user.h"
#define NAME_SIZE 256

write_prim_delays()
{
    FILE      *infile;
    char      full_net_name[NAME_SIZE];
    double    rise,fall;
    handle    net_handle, driver_handle, prim_handle;

    /*initialize the environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*set accPathDelayCount parameter for rise and fall delays only*/
    acc_configure(accPathDelayCount, "2");

    /*read delays from file - "r" means read only*/
    infile = fopen("primdelay.dat","r");
    while (fscanf(infile,"%s %lf %lf",full_net_name,&rise,&fall) != EOF)
    {
        /*get handle for the net*/
        net_handle = acc_handle_object(full_net_name);

        /*get primitive connected to first net driver*/
        driver_handle = acc_next_driver(net_handle, null);
        prim_handle = acc_handle_parent(driver_handle);

        /*replace delays with new values*/
        acc_replace_delays(prim_handle, rise, fall);
    }
    acc_close();
}
```

Figure 19-51—Using acc_handle_object()

19.49 acc_handle_parent()

acc_handle_parent()			
Synopsis:	Get a handle for the parent primitive instance or module instance of an object.		
Syntax:	acc_handle_parent(object_handle)		
Returns:	Type	Description	
	handle	Handle of a primitive or module	
Arguments:	Type	Name	Description
	handle	object_handle	Handle of an object

The ACC routine **acc_handle_parent()** shall return a handle to the parent of any object. A parent is an object that contains another object.

- The parent of a *terminal* shall be the *primitive* that contains the terminal.
- The parent of any other object (except a top-level module) shall be the *module instance* that contains the object.
- Top-level modules do not have parents. When a top-level module handle is passed to **acc_handle_parent()**, it shall return null.

The example shown in Figure 19-52 uses **acc_handle_parent()** to determine which terminals of a primitive drive a net.

```
#include "acc_user.h"

get_primitives(net_handle)
handle    net_handle;
{
    handle    primitive_handle;
    handle    driver_handle;

    /*get primitive that owns each terminal that drives the net*/
    driver_handle = null;
    while (driver_handle = acc_next_driver(net_handle, driver_handle) )
    {
        primitive_handle = acc_handle_parent(driver_handle);
        io_printf("Primitive %s drives net %s\n",
                  acc_fetch_fullname(primitive_handle),
                  acc_fetch_fullname(net_handle) );
    }
}
```

Figure 19-52—Using acc_handle_parent()

19.50 acc_handle_path()

acc_handle_path()			
Synopsis:	Get a handle to an intermodule path that represents the connection from an output or inout port to an input or inout port.		
Syntax:	acc_handle_path(port_output_handle, port_input_handle)		
Type		Description	
Returns:	handle	Handle of the intermodule path	
Arguments:	Type	Name	Description
	handle	port_output_handle	Handle to one of the following: <ul style="list-style-type: none"> • A scalar output port • A scalar bidirectional port • 1 bit of a vector output port • 1 bit of a vector bidirectional port
	handle	port_input_handle	Handle to one of the following: <ul style="list-style-type: none"> • A scalar input port • A scalar bidirectional port • 1 bit of a vector input port • 1 bit of a vector bidirectional port
Related routines:	Use acc_next_port() or acc_handle_port() to retrieve a handle to a scalar port Use acc_next_bit() to retrieve a handle to a bit of a vector port or a bit of a concatenated port Use acc_fetch_direction() to determine whether a port is an input, an output, or bidirectional		

The ACC routine **acc_handle_path()** shall return a handles to an *intermodule path*. An intermodule path shall be a net path that connects an output or inout port of one module to an input or inout port of another module.

The example shown in Figure 19-53 is a C-code fragment that uses **acc_handle_path()** to fetch min:typ:max delays for the intermodule path referenced by *intermod_path*.

```

#include "acc_user.h"
fetch_mintypmax_delays(port_output, port_input)
handle    port_output, port_input;
{
    . . .
    handle    intermod_path;
    double    delay_array[9];
    . . .
    acc_configure(accMinTypMaxDelays, "true");
    . . .
    intermod_path = acc_handle_path(port_output, port_input);
    acc_fetch_delays(intermod_path, delay_array);
    . . .
}

```

acc_handle_path() returns a handle to a net path that represents the connection from an output or inout port to an input (or inout) port

Figure 19-53—Using acc_handle_path()

19.51 acc_handle_pathin()

acc_handle_pathin()				
Synopsis:	Get a handle for the first net connected to a module path source.			
Syntax:	acc_handle_pathin(path_handle)			
Returns:		Type	Description	
		handle	Handle to a net	
Arguments:		Type	Name	Description
		handle	path_handle	Handle of the module path
Related routines:	Use acc_next_modpath() or acc_handle_modpath() to get path_handle			

The ACC routine **acc_handle_pathin()** shall return a handle to the net connected to the first source in a module path. If a module path has more than one input source, only the handle to the net connected to the first source shall be returned. For example:

```

      (posedge clk => (q +: d) ) = (3,2);
      (a,b,c *> d,e,f) = 1.8;

```

The example shown in Figure 19-54 uses **acc_handle_pathin()** to find the net connected to the input of a path.

```

#include "acc_user.h"
get_path_nets(path_handle)
handle path_handle;
{
    handle pathin_handle, pathout_handle;
    pathin_handle = acc_handle_pathin(path_handle);
    pathout_handle = acc_handle_pathout(path_handle);
    io_printf("Net connected to input is: %s\n",
              acc_fetch_name(pathin_handle) );
    io_printf("Net connected to output is: %s\n",
              acc_fetch_name(pathout_handle) );
}

```

Figure 19-54—Using acc_handle_pathin()

19.52 acc_handle_pathout()

acc_handle_pathout()			
Synopsis:	Get a handle for the first net connected to a module path destination.		
Syntax:	acc_handle_pathout(path_handle)		
Returns:	Type	Description	
	handle	Handle to a net	
Arguments:	Type	Name	Description
	handle	path_handle	Handle of the module path
Related routines:	Use acc_next_modpath() or acc_handle_modpath() to get path_handle		

The ACC routine **acc_handle_pathout()** shall return a handle to the net connected to the first destination in a module path. If a module path has more than one output destination, only the handle to the net connected to the first destination shall be returned. For example:

```

(posedge clk => (q += d) ) = (3,2);
(a,b,c *> d,e,f) = 1.8;

```

The example shown in Figure 19-55 uses **acc_handle_pathout()** to find the net connected to the output of a path.

```

#include "acc_user.h"
get_path_nets(path_handle)
handle path_handle;
{
    handle pathin_handle, pathout_handle;
    pathin_handle = acc_handle_pathin(path_handle);
    pathout_handle = acc_handle_pathout(path_handle);
    io_printf("Net connected to input is: %s\n",
              acc_fetch_name(pathin_handle) );
    io_printf("Net connected to output is: %s\n",
              acc_fetch_name(pathout_handle) );
}

```

Figure 19-55—Using acc_handle_pathout()

19.53 acc_handle_port()

acc_handle_port()			
Synopsis:	Get a handle for a module port, based on the position of the port.		
Syntax:	<code>acc_handle_port(module_handle, port_index)</code>		
Type		Description	
Returns:	handle	Handle to a module port	
Type		Name	Description
Arguments:	handle	module_handle	Handle of a module
	int	port_index	An integer index of the desired port
Related routines:	Use <code>acc_next_port()</code> to get handles to all ports of a module		

The ACC routine **acc_handle_port()** shall return a handle to a specific port of a module, based on the position of the port in the module declaration.

The index of a port shall be its position in a module definition in the source description. The indices shall be integers that start at **0** and increase from left to right. Table 19-42 shows how port indices are derived.

Table 19-42—Deriving port indices

For	Indices shall be
Implicit ports: module A(q, a, b);	0 for port q 1 for port a 2 for port b
Explicit ports: module top; reg ra, rb; wire wq; explicit_port_mod epm1(.b(rb), .a(ra), .q(wq)); endmodule module explicit_port_mod(q, a, b); input a, b; output q; nand (q, a, b); endmodule	0 for explicit port epm1.q 1 for explicit port epm1.a 2 for explicit port epm1.b

The example shown in Figure 19-56 uses **acc_handle_port()** to identify whether a particular module port is an output.

```
#include "acc_user.h"

bool is_port_output(module_handle, port_index)
handle    module_handle;
int       port_index;
{
    handle    port_handle;
    int       direction;

    /*check port direction*/
    port_handle = acc_handle_port(module_handle, port_index);
    direction = acc_fetch_direction(port_handle);
    if (direction == accOutput || direction == accInout)
        return(true);
    else
        return(false);
}
```

Figure 19-56—Using acc_handle_port()

19.54 acc_handle_scope()

acc_handle_scope()			
Synopsis:	Get a handle to the scope that contains an object.		
Syntax:	acc_handle_scope(object_handle)		
Returns:	Type		Description
	handle	Handle of a scope	
Arguments:	Type	Name	Description
	handle	object_handle	Handle to an object
Related routines:	Use acc_fetch_type() or acc_fetch_fulltype() to determine the scope type returned		

The ACC routine **acc_handle_scope()** shall return the handle to the scope of an object. A scope shall be

- A top-level module
- A module instance
- A named begin-end block
- A named fork-join block
- A Verilog HDL task
- A Verilog HDL function

The example shown in Figure 19-57 uses **acc_handle_scope()** to display the scope that contains an object.

```

get_scope(obj)
handle obj;
{
    handle scope = acc_handle_scope(obj);

    io_printf ("Scope %s contains object %s\n",
               acc_fetch_fullname(scope), acc_fetch_name(obj));
}

```

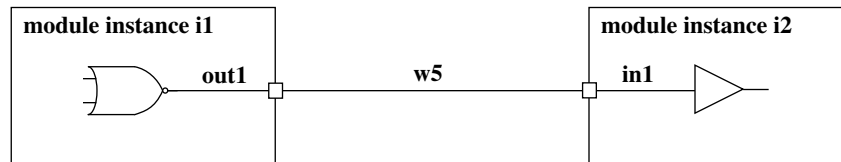
Figure 19-57—Using acc_handle_scope()

19.55 acc_handle_simulated_net()

acc_handle_simulated_net()			
Synopsis:	Get the simulated net associated with the collapsed net passed as an argument.		
Syntax:	acc_handle_simulated_net(collapsed_net_handle)		
Returns:	Type	Description	
	handle	Handle of the simulated net	
Arguments:	Type	Name	Description
	handle	collapsed_net_handle	Handle of a collapsed net
Related routines:	Use acc_object_of_type() to determine if a net has been collapsed		

The ACC routine **acc_handle_simulated_net()** shall return a handle to the simulated net that is associated with a specified collapsed net. If a handle to a net that is not collapsed is passed into the routine, a handle to that same net shall be returned.

When a Verilog HDL source description connects modules together, a chain of nets with different scopes and names are connected, as is illustrated in the following simple diagram:



In this small circuit, nets `out1`, `w5`, and `in1` are all tied together, effectively becoming the same net. Software products can collapse nets that are connected together within the data structure of the product. The resultant net after collapsing is referred to as the *simulated net*; the other nets are referred to as collapsed nets. The ACC routines can obtain a handle to any net, whether it is collapsed or not. The routine **acc_object_of_type()** can be used to determine if a net has been collapsed, and the routine **acc_handle_simulated_net()** can be used to find the resultant net from the net collapsing process.

The example shown in Figure 19-58 uses **acc_handle_simulated_net()** to find all simulated nets within a particular scope. The application then displays each collapsed net, along with the simulated net. The ACC routine **acc_object_of_type()** is used with the property **accCollapsedNet** to determine whether a net has been collapsed onto another net.

```
#include "acc_user.h"

void display_simulated_nets()
{
    handle    mod_handle;
    handle    simulated_net_handle;
    handle    net_handle;

    /*reset environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get scope-first argument passed to user-defined system task*/
    /* associated with this routine*/
    mod_handle = acc_handle_tfarg(1);
    io_printf("In module %s:\n", acc_fetch_fullname(mod_handle) );
    net_handle = null;

    /*display name of each collapsed net and its net of origin*/
    while(net_handle = acc_next_net(mod_handle, net_handle) )
    {
        if (acc_object_of_type(net_handle, accCollapsedNet) )
        {
            simulated_net_handle = acc_handle_simulated_net(net_handle);
            io_printf("    net %s was collapsed onto net %s\n",
                    acc_fetch_name(net_handle),
                    acc_fetch_name(simulated_net_handle) );
        }
    }
}
```

Figure 19-58—Using acc_handle_simulated_net()

19.56 acc_handle_tchk()

acc_handle_tchk()			
Synopsis:	Get a handle for the specified timing check of a module (or cell).		
Syntax:	<pre>acc_handle_tchk(module_handle, timing_check_type, first_arg_conn_name, first_arg_edge_type, second_arg_conn_name, second_arg_edge_type, first_arg_conn_handle, second_arg_conn_handle)</pre>		
Type		Description	
Returns:	handle	Handle to a timing check	
Arguments:	Type	Name	Description
	handle	module_handle	Handle of the module
	integer constant	timing_check_type	One of the following predefined constants: <div style="display: flex; justify-content: space-between;"> accHold accSetup </div> <div style="display: flex; justify-content: space-between;"> accNochange accSkew </div> <div style="display: flex; justify-content: space-between;"> accPeriod accWidth </div> accRecovery
	quoted string or char *	first_arg_conn_name	Name of the net connected to first timing check argument
	integer constant	first_arg_edge_type	Edge of the net connected to first timing check argument One of the following predefined constants: <div style="display: flex; justify-content: space-between;"> accNegedge accNoedge accPosedge </div> or a list of the following constants, separated by +: <div style="display: flex; justify-content: space-between;"> accEdge01 accEdge0x accEdgex1 </div> or a list of the following constants, separated by +: <div style="display: flex; justify-content: space-between;"> accEdge10 accEdge1x accEdgex0 </div>
	Conditional	second_arg_conn_name	Name of the net connected to second timing check argument (depends on type of timing check)
	Conditional	second_arg_edge_type	Edge of the net connected to second timing check argument (depends on type of timing check) Uses same constants as <i>first_arg_edge_type</i>
	Optional	first_arg_conn_handle	Handle of the net connected to first timing check argument (required if accEnableArgs is set and <i>first_arg_conn_name</i> is null)
	Optional	second_arg_conn_handle	Handle of the net connected to second timing check argument (required if accEnableArgs is set and <i>second_arg_conn_name</i> is null)
Related routines:	Use acc_configure(accEnableArgs, "acc_handle_tchk") to enable the optional <i>first_arg_conn_handle</i> and <i>second_arg_conn_handle</i> arguments		

The ACC routine **acc_handle_tchk()** shall return a handle to a timing check based on arguments that describe the type of timing check, signals used, and edge qualifiers for the signals. The signals used to describe the timing check shall be passed as either signal names (passed as either a quoted string or a character string pointer) or signal handles. The number of signal arguments required by **acc_handle_tchk()** shall depend on the type of timing check.

Table 19-43 shows how the number of arguments for **acc_handle_tchk()** is determined.

Table 19-43—How acc_handle_tchk() works

If	acc_handle_tchk() shall
<i>tchk_type</i> is accWidth or accPeriod	ignore arguments: <i>second_arg_conn_name</i> , <i>second_arg_edge_type</i> , and optional <i>second_arg_conn_handle</i>
<i>tchk_type</i> is accHold , accNochange , accRecovery , accSetup , or accSkew	use arguments: <i>second_arg_conn_name</i> , <i>second_arg_edge_type</i> , and optional <i>second_arg_conn_handle</i>
Default mode, or acc_configure(accEnableArgs, "no_acc_handle_tchk") has been called	Use the name arguments and ignore both optional handle arguments
The routine acc_configure(accEnableArgs, "acc_handle_tchk") has been called, and either <i>first_arg_conn_name</i> or <i>second_arg_conn_name</i> is null	Use the associated handle argument of the null of the name argument—if the name argument is not null, the name shall be used and the associated handle argument ignored

NOTE—Unused arguments can be dropped if they do not precede any required arguments; otherwise, the unused arguments should be specified as null.

The routine **acc_handle_tchk()** shall use predefined edge group constants to represent groups of transitions among **0**, **1**, and **X** edge values, as described in Table 19-44. The routine shall treat transitions to or from a logic Z as transitions to or from a logic X.

Table 19-44—Edge group constants

Edge group constant	Description of edge trigger
accPosedge accPosEdge	Any positive transition: 0 to 1 0 to x x to 1
accNegedge accNegEdge	Any negative transition: 1 to 0 1 to x x to 0
accNoedge accNoEdge	Any transition: 0 to 1 1 to 0 0 to x x to 1 1 to x x to 0

The routine **acc_handle_tchk()** shall recognize predefined edge-specific constants that represent individual transitions among **0**, **1**, and **X** edge values that trigger timing checks, as described in Table 19-45.

Table 19-45—Edge specific constants

Edge specific constant	Description of edge trigger
accEdge01	Transition from 0 to 1
accEdge0x	Transition from 0 to x
accEdgex1	Transition from x to 1
accEdge10	Transition from 1 to 0
accEdge1x	Transition from 1 to x
accEdgex0	Transition from x to 0

The Verilog HDL allows multiple edges to be specified for timing checks. The routine **acc_handle_tchk()** shall recognize multiple edges using *edge sums*. Edge sums are lists of edge-specific constants connected by plus (+) signs. They represent the Verilog-HDL edge-control specifiers used by particular timing checks. Figure 19-59 shows a call to **acc_handle_tchk()** that accesses a *\$width* timing check containing edge-control specifiers.

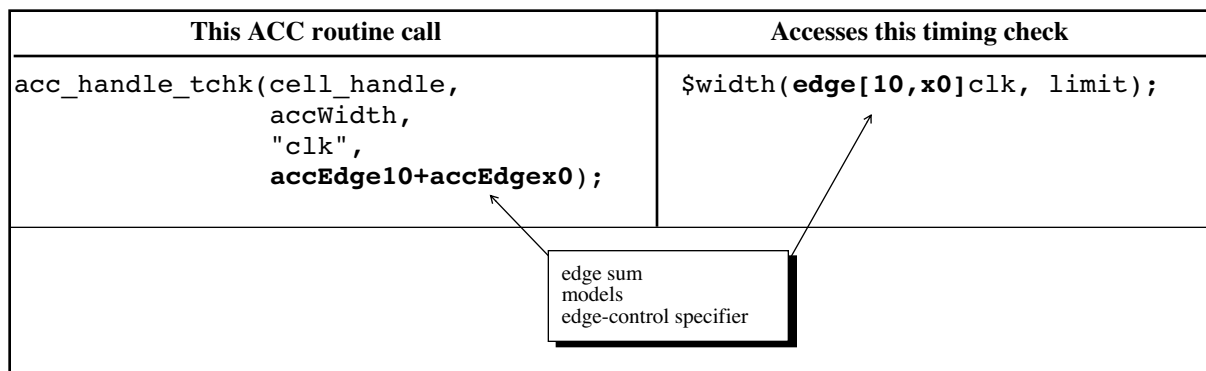


Figure 19-59—Edge sums model edge-control specifiers

The example shown in Figure 19-60 uses **acc_handle_tchk()** to identify all cells in a module that contain either or both of the following timing checks:

- A *\$period* timing check triggered by a positive edge on the clock signal *clk*
- A *\$setup* timing check triggered on signal *d* by any transition and on signal *clk* by either of these clock edge transitions: **1** to **0** or **X** to **0**

Note that in this example:

- a) Both calls to **acc_handle_tchk()** supply *names* for all relevant connections; therefore, the optional handle arguments are not supplied.
- b) For *\$period* timing checks, **acc_handle_tchk()** ignores the *second_arg_conn_name* and *second_arg_edge_type* arguments; therefore, these arguments are not supplied.

```

#include "acc_user.h"

get_ps_tchks()
{
    handle    module_handle, port_handle, net_handle, cell_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*set development version*/
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for module*/
    module_handle = acc_handle_tfarg(1);
    io_printf("Module is %s\n", acc_fetch_name(module_handle) );

    /*scan all cells in module for:                                     */
    /*  period timing checks triggered by a positive clock edge      */
    /*  setup timing checks triggered by 1->0 and x->0 clock edges    */
    cell_handle = null;
    while(cell_handle = acc_next_cell(module_handle, cell_handle) )
    {
        if(acc_handle_tchk(cell_handle, accPeriod, "clk", accPosedge) )
            io_printf("positive clock edge triggers period check in cell %s\n",
                acc_fetch_fullname(cell_handle) );
        if(acc_handle_tchk(cell_handle, accSetup, "d", accNoedge,
            "clk", accEdge10+accEdgex0) )
            io_printf("10 and x0 edges trigger setup check in cell %s\n",
                acc_fetch_fullname(cell_handle) );
    }
    acc_close();
}

```

Figure 19-60—Using acc_handle_tchk()

19.57 acc_handle_tchkarg1()

acc_handle_tchkarg1()			
Synopsis:	Get a handle for the timing check terminal connected to the first argument of a timing check.		
Syntax:	acc_handle_tchkarg1(tchk_handle)		
Returns:	Type		Description
	handle	Handle of a timing check terminal	
Arguments:	Type	Name	Description
	handle	tchk_handle	Handle of a timing check
Related routines:	Use acc_handle_conn() to get the net connected to a timing check terminal		

The ACC routine **acc_handle_tchkarg1()** shall return a handle to the timing check terminal associated with the first argument of a timing check.

In order to trace a timing check terminal in the Verilog HDL description, or to display the name of the terminal, it is first necessary to obtain a handle to the net connected to the terminal. The routine **acc_handle_conn()** with the timing check terminal handle as the argument can be used to obtain the net handle.

The example shown in Figure 19-61 uses **acc_handle_tchkarg1()** and **acc_handle_tchkarg2()** to obtain the nets connected to the first and second arguments of each setup timing check in each cell under a module.

```

#include "acc_user.h"

show_check_nets()
{
    handle    module_handle, cell_handle;
    handle    tchk_handle, tchkarg1_handle, tchkarg2_handle;
    int        tchk_type, counter;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*set development version*/
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for module*/
    module_handle = acc_handle_tfarg(1);
    io_printf("module is %s\n", acc_fetch_fullname(module_handle) );

    /*scan all cells in module for timing checks*/
    cell_handle = null;
    while (cell_handle = acc_next_cell(module_handle, cell_handle) )
    {
        io_printf("cell is: %s\n", acc_fetch_fullname(cell_handle) );
        counter = 0;
        while (tchk_handle = acc_next_tchk(cell_handle, tchk_handle) )
        {
            /*get nets connected to timing check arguments*/
            tchk_type = acc_fetch_type(tchk_handle);
            if (tchk_type == accSetup)
            {
                counter++;
                io_printf("    for setup check #%d:\n", counter);
                tchkarg1_handle = acc_handle_tchkarg1(tchk_handle);
                tchkarg2_handle = acc_handle_tchkarg2(tchk_handle);
                io_printf("        data net is %s\n    reference net is %s\n",
                    acc_fetch_name(acc_handle_conn(tchkarg1_handle) ),
                    acc_fetch_name(acc_handle_conn(tchkarg2_handle) ) );
            }
        }
    }
    acc_close();
}

```

Figure 19-61—Using acc_handle_tchkarg1() and acc_handle_tchkarg2()

19.58 acc_handle_tchkarg2()

acc_handle_tchkarg2()			
Synopsis:	Get a handle for the timing check terminal connected to the second argument of a timing check.		
Syntax:	acc_handle_tchkarg2(tchk_handle)		
Type		Description	
Returns:	handle	Handle to a timing check terminal	
Type		Name	Description
Arguments:	handle	tchk_handle	Handle of a timing check
Related routines:	Use acc_handle_conn() to get the net connected to a timing check terminal		

The ACC routine **acc_handle_tchkarg2()** shall return a handle to the timing check terminal associated with the second argument of a timing check.

In order to trace a timing check terminal in the Verilog HDL description, or to display the name of the terminal, it is first necessary to obtain a handle to the net connected to the terminal. The routine **acc_handle_conn()** with the timing check terminal handle as the argument can be used to obtain the net handle.

Refer to Figure 19-61 for an example of using **acc_handle_tchkarg2()**.

19.59 acc_handle_terminal()

acc_handle_terminal()			
Synopsis:	Get a handle for a primitive terminal based on the position of the primitive terminal.		
Syntax:	<code>acc_handle_terminal(primitive_handle, terminal_index)</code>		
Returns:	Type		Description
	handle	Handle of a primitive terminal	
Arguments:	Type	Name	Description
	handle	primitive_handle	Handle of a primitive
	int	terminal_index	Integer index of the desired terminal
Related routines	Use <code>acc_handle_conn()</code> to get the net connected to a primitive terminal		

The ACC routine **acc_handle_terminal()** shall return a handle of a primitive terminal based on the position of the terminal in the Verilog HDL source description.

The index of a terminal shall be its position in a gate, switch, or UDP declaration. The indices shall be integers that start at zero and increase from left to right. Table 19-46 shows how terminal indices are derived.

Table 19-46—Deriving terminal indices

For	Indices shall be
nand g1(out, in1, in2);	0 for terminal out 1 for terminal in1 2 for terminal in2

The example shown in Figure 19-62 uses **acc_handle_terminal()** to identify the name of a net connected to a primitive terminal.

```
#include "acc_user.h"

print_terminal_net(gate_handle, term_index)
handle    gate_handle;
int       term_index;
{
    handle    term_handle;
    term_handle = acc_handle_terminal(gate_handle, term_index);
    io_printf("%s terminal net #%d is %s\n",
              acc_fetch_name(gate_handle), term_index,
              acc_fetch_name(acc_handle_conn(term_handle) ) );
}
```

Figure 19-62—Using acc_handle_terminal()

19.60 acc_handle_tfarg(), acc_handle_itfarg()

acc_handle_tfarg(), acc_handle_itfarg()			
Synopsis:	Get a handle for the specified argument of a user-defined system task or function.		
Syntax:	<pre>acc_handle_tfarg(argument_number) acc_handle_itfarg(argument_number, instance_handle)</pre>		
Type		Description	
Returns:	handle	Handle to an object	
Arguments:	Type	Name	Description
	int	argument_number	Integer number that references an argument in the system task or function call by its position in the argument list
	handle	instance_handle	Handle to an instance of a system task/function
Related routines:	Use acc_fetch_tfarg() and related routines to get the value of a system task/function argument		

The ACC routine **acc_handle_tfarg()** shall return a handle to an argument in the current instance of a user-defined system task/function. The ACC routine **acc_handle_itfarg()** shall return a handle to an argument in a specific instance of a user-defined system task/function.

Argument numbers shall start at **1** and increase from left to right in the order that they appear in the system task or function call.

The system task/function argument can be an unquoted identifier name of

- A module instance
- A net

The system task/function argument can be a quoted string containing the name of

- A module instance
- A primitive instance
- A net, register, or variable
- A legal bit select of a net, register, or variable

Table 19-47—How acc_handle_tfarg() operates

When	acc_handle_tfarg() shall
The system task or function argument is an unquoted Verilog HDL identifier for a net or module instance	Return a handle to the object
The system task or function argument is a quoted string name of any object	<p>Function similar to acc_handle_object() by searching for an object matching the string and, if found, returning a handle to the object.</p> <p>The object shall be searched for in the following order:</p> <ol style="list-style-type: none"> a) The current PLI scope [as set by acc_set_scope()] b) The scope of the system task/function

The example shown in Figure 19-63 uses **acc_handle_tfarg()** in a C-language application that has the following characteristics:

- It changes the rise and fall delays of a gate.
- It takes three arguments—the first is a Verilog HDL gate and the others are double-precision floating-point constants representing rise and fall delay values.
- It associates through the PLI interface mechanism with a Verilog HDL system task called `$timing_task`.

To invoke the application, the system task `$timing_task` is called from the Verilog HDL source description, as in the following sample call:

```
$timing_task(top.g12, 8.4, 9.2);
```

When Verilog encounters this call, it executes `new_timing`. A handle to the first argument, the gate `top.g12`, is retrieved using **acc_handle_tfarg()**, while the other two arguments—the delay values—are retrieved using **acc_fetch_tfarg()**.

```
#include "acc_user.h"

new_timing()
{
    handle    gate_handle;
    double    new_rise, new_fall;

    /*initialize and configure ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");
    acc_configure(accToHiZDelay, "max");

    /*get handle to gate*/
    gate_handle = acc_handle_tfarg( 1 );

    /* get new delay values */
    new_rise = acc_fetch_tfarg( 2 );
    new_fall = acc_fetch_tfarg( 3 );

    /*place new delays on the gate*/
    acc_replace_delays(gate_handle,new_rise,new_fall);

    /* report action */
    io_printf("Primitive %s has new delays %d %d\n",
              acc_fetch_fullname(gate_handle),
              new_rise, new_fall);

    acc_close();
}
```

Figure 19-63—Using `acc_handle_tfarg()`

19.61 **acc_handle_tfinst()**

acc_handle_tfinst()		
Synopsis:	Get a handle to the current user-defined system task or function call.	
Syntax:	<code>acc_handle_tfinst()</code>	
Returns:	Type	Description
	handle	Handle of a user-defined system task or function
Arguments:	Type	Name Description
	None	
Related routines:	Use <code>acc_fetch_type()</code> or <code>acc_fetch_fulltype()</code> to determine the type of the handle returned	

The ACC routine **acc_handle_tfinst()** is used to obtain a handle of the user-defined system task/function call that invoked the current PLI application.

19.62 acc_initialize()

acc_initialize()		
Synopsis:	Initializes the environment for ACC routines.	
Syntax:	acc_initialize()	
Returns:	Type	Description
	void	
Arguments:	Type	Name
	None	
Related routines:	Use acc_configure() to set configuration parameter after calling acc_initialize()	
	Use acc_close() at the end of a routine that called acc_initialize()	

The ACC routine **acc_initialize()** shall perform the following functions:

- Initialize all configuration parameters to their default values
- Allocate memory for string handling and other internal uses

The routine **acc_initialize()** should be called in a C-language application before invoking any other ACC routines. Potentially, multiple PLI applications running in the same simulation session can interfere with each other because they share the same set of configuration parameters. To guard against application interference, both **acc_initialize()** and **acc_close()** reset any configuration parameters that have changed from their default values.

The example shown in Figure 19-64 uses **acc_initialize()** to initialize the environment for ACC routines.

```
#include "acc_user.h"
append_mintypmax_delays()
{
    handle    prim;
    double    delay_array[9];
    int       i;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*configure ACC routine environment*/
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");
    acc_configure(accMinTypMaxDelays, "true");

    /* append delays for primitive as specified in task/function args*/
    prim = acc_handle_tfarg(1);
    for (i = 0; i < 9; i++)
        delay_array[i] = acc_fetch_tfarg(i+2);
    acc_append_delays(prim, delay_array);
}
```

Figure 19-64—Using acc_initialize()

19.63 acc_next()

acc_next()			
Synopsis:	Get handles to objects of each type specified in an array within the reference scope.		
Syntax:	<code>acc_next(object_type_array, reference_handle, object_handle)</code>		
Type		Description	
Returns:	handle	Handle of the object found	
Arguments:	Type	Name	Description
	static int array	object_type_array	Static integer array containing one or more predefined integer constants that represent the types of objects desired; the last element has to be 0
	handle	reference_handle	Handle of a scope
	handle	object_handle	Handle of the previous object found; initially null

The ACC routine **acc_next()** shall scan for and return handles to one or more types of objects within a scope. This routine performs a more general function than the object-specific *acc_next_* routines, such as **acc_next_net()** and **acc_next_primitive()**, which scan only one type of object within a scope.

The objects for which **acc_next()** is to scan shall be listed as an array of object *types or fulltypes* in a static integer array. The array shall contain any number and combination of the predefined integer constants listed in Table 19-48. The array list shall be terminated by a **0**. The routine **acc_next()** can return objects in an arbitrary order.

The following C-language statement is an example of declaring an array of object types called `net_reg_list`:

```
static int net_reg_list[3] = {accNet, accRegister, 0};
```

When this array is passed to **acc_next()**, the ACC routine shall return handles to nets and registers within the reference object.

Note that a Verilog HDL function contains an object with the same name, size, and type as the function. If the function is scanned for objects of the type of the function, a handle to this object shall be returned.

The objects for which **acc_next()** shall obtain handles are listed in Table 19-48.

Table 19-48—Type and fulltype constants supported by acc_next()

Description		Predefined integer constant
General object <i>types</i>	Integer variable	accIntegerVar
	Module	accModule
	Named event	accNamedEvent
	Net	accNet
	Primitive	accPrimitive
	Real variable	accRealVar
	Register	accRegister
	Time variable	accTimeVar
Module <i>fulltypes</i>	Top-level module	accTopModule
	Module instance	accModuleInstance
	Cell module instance	accCellInstance
Net <i>fulltypes</i>	Wire nets	accWire accTri
	Wired-AND nets	accWand accTriand
	Wired-OR nets	accWor accTrior
	Pulldown, pullup nets	accTri0 accTri1
	Supply nets	accSupply0 accSupply1
	Storage nets	accTtireg

Table 19-48— Type and fulltype constants supported by `acc_next()` (continued)

Description		Predefined integer constant
Primitive <i>fulltypes</i>	N-input, 1-output gates	accAndGate accNandGate accNorGate accOrGate accXnorGate accXorGate
	1-input, N-output gates	accBufGate accNotGate
	Tri-state gates	accBufif0 accBufif1 accNotif0 accNotif1
	MOS gates	accNmosGate accPmosGate accRnmosGate accRpmosGate
	CMOS gates	accCmosGate accRcmosGate
	Bidirectional pass gates	accRtranGate accRtranif0Gate accRtranif1Gate accTranGate accTranif0Gate accTranif1Gate
	Pulldown, pullup gates	accPulldownGate accPullUpGate
	Combinational UDP	accCombPrim
	Sequential UDP	accSeqPrim

The example shown in Figure 19-65 uses **acc_next()** to find all nets and registers in a module. The application then displays the names of these nets and registers.

```
#include "acc_user.h"

void display_nets_and_registers()
{
    static int net_reg_list[3] = {accNet, accRegister, 0};
    handle mod_handle, obj_handle;

    /*reset environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for module-first argument passed to*/
    /* user-defined system task associated with this routine*/
    mod_handle = acc_handle_tfarg(1);
    io_printf("Module %s contains these nets and registers:\n",
              acc_fetch_fullname(mod_handle) );

    /*display names of all nets and registers in the module*/
    obj_handle = null;
    while (obj_handle = acc_next(net_reg_list, mod_handle, obj_handle) )
        io_printf("    %s\n", acc_fetch_name(obj_handle) );
}
```

Figure 19-65—Using acc_next()

19.64 acc_next_bit()

acc_next_bit()			
Synopsis:	Get handles to bits in a port or expanded vector net.		
Syntax:	<code>acc_next_bit(reference_handle, bit_handle)</code>		
Returns:	Type	Description	
	handle	Handle of a port or net bit	
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a port or net
	handle	bit_handle	Handle of the previous bit found; initially null
Related routines:	Use <code>acc_next_port()</code> to return the next port of a module Use <code>acc_handle_port()</code> to return the handle for a module port Use <code>acc_object_of_type()</code> to determine if a vector net is expanded		

The ACC routine **acc_next_bit()** shall obtain handles to the bits of a vector port or an expanded vector net.

An *expanded vector* is vector for which a software product shall permit access to the discrete bits of the vector. The routine **acc_object_of_type()** can be used to determine if a vector reference handle is expanded before calling **acc_next_bit()** with the vector handle. For example:

```

if (acc_object_of_type(net_handle, accExpandedVector) )
    while (bit_handle = acc_next_bit(net_handle, bit_handle) )
        ...

```

When the *reference_handle* object is a vector, the first call to **acc_next_bit()** shall return the handle to the msb (leftmost bit) of the object. Subsequent calls shall return the handles to the remaining bits down to the lsb (rightmost bit). The call after the return of the handle to the lsb returns null. When the *reference_handle* is scalar, **acc_next_bit()** shall treat the object as a 1-bit vector.

The example shown in Figure 19-66 uses **acc_next_bit()** to display the lower connection of each bit of a port.

```
#include "acc_user.h"
display_port_bits(module_handle, port_number)
handle    module_handle;
int       port_number;
{
    handle    port_handle, bit_handle;

    /* get handle for port */
    port_handle = acc_handle_port(module_handle, port_number);

    /* display port number and module instance name */
    io_printf("Port %d of module %s contains the following bits: \n",
              port_number, acc_fetch_fullname(module_handle) );
    /* display lower hierarchical connection of each bit */
    bit_handle = null;
    while (bit_handle = acc_next_bit(port_handle, bit_handle) )
        io_printf("  %s\n", acc_fetch_fullname(bit_handle) );
}
```

Figure 19-66—Using acc_next_bit() with module ports

The example shown in Figure 19-67 uses **acc_next_bit()** to assign a VCL monitor flag to each bit of a vector net.

```
#include "acc_user.h"
void monitor_bits()
{
    handle    bit_handle, net_handle, mod_handle;

    /* reset environment for ACC routines */
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /* get handle for system task argument associated with this routine */
    mod_handle = acc_handle_tfarg(1);

    /* get handles to all nets in the module */
    net_handle = null;
    while (net_handle = acc_next_net(mod_handle, net_handle) )
    {
        /* add VCL monitor each bit of expanded vector nets */
        if (acc_object_of_type(net_handle, accExpandedVector) )
        {
            bit_handle = null;
            while (bit_handle = acc_next_bit(net_handle, bit_handle) )
                acc_vcl_add(bit_handle, net_consumer, null, vcl_verilog_logic);
        }
    }
}
```

Figure 19-67—Using acc_next_bit() with a vector net

19.65 acc_next_cell()

acc_next_cell()			
Synopsis:	Get handles to cell instances within a region that includes the entire hierarchy below a module.		
Syntax:	acc_next_cell(reference_handle, cell_handle)		
Type		Description	
Returns:	handle	Handle of a cell module	
Type		Name	Description
Arguments:	handle	reference_handle	Handle of a module
	handle	cell_handle	Handle of the previous cell found; initially null

The ACC routine **acc_next_cell()** shall return handles to the cell module instances in the reference scope and all module instance scopes below the reference scope. The routine shall not find cells that are instantiated inside other cells.

A cell instance shall be a module instance that has either of these characteristics:

- The module definition appears between the compiler directives ``celldefine` and ``endcelldefine`.
- The module definition is in a model library, where a library is a collection of module definitions in a file or directory that are read by library invocation options.

The example shown in Figure 19-68 uses **acc_next_cell()** to list all cell instances at or below a given hierarchy scope.

```
#include "acc_user.h"
list_cells()
{
    handle    module_handle, cell_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for module*/
    module_handle = acc_handle_tfarg(1);
    io_printf("%s contains the following cells:\n",
              acc_fetch_fullname(module_handle) );

    /*display names of all cells in the module*/
    cell_handle = null;
    while(cell_handle = acc_next_cell(module_handle, cell_handle) )
        io_printf("    %s\n", acc_fetch_fullname(cell_handle) );

    acc_close();
}
```

Figure 19-68—Using acc_next_cell()

19.66 acc_next_cell_load()

acc_next_cell_load()				
Synopsis:	Get handles for cell loads on a net.			
Syntax:	acc_next_cell_load(reference_handle, load_handle)			
Returns:		Type	Description	
		handle	Handle of a primitive input terminal	
Arguments:		Type	Name	Description
		handle	reference_handle	Handle of the net
		handle	load_handle	Handle of the previous load found; initially null
Related routines:		Use acc_next_load() to get a handle to all primitive input terminal loads		

The ACC routine **acc_next_cell_load()** shall return handles to the *cell module instances* that are driven by a net. The handle for a cell load shall be a primitive input terminal connected to an input or inout port of the cell load instance.

The routines **acc_next_load()** and **acc_next_cell_load()** have different functionalities. The routine **acc_next_load()** shall return every primitive input terminal driven by a net, whether it is inside a cell or a module instance. The routine **acc_next_cell_load()** shall return only one primitive input terminal per cell input or inout port driven by a net. Figure 19-69 illustrates the difference, using a circuit in which *net1* drives primitive gates in *cell1*, *cell2*, and *module1*. For this circuit, **acc_next_load()** returns four primitive input terminals as loads on *net1*, while **acc_next_cell_load()** returns two primitive input terminals as loads on *net1*.

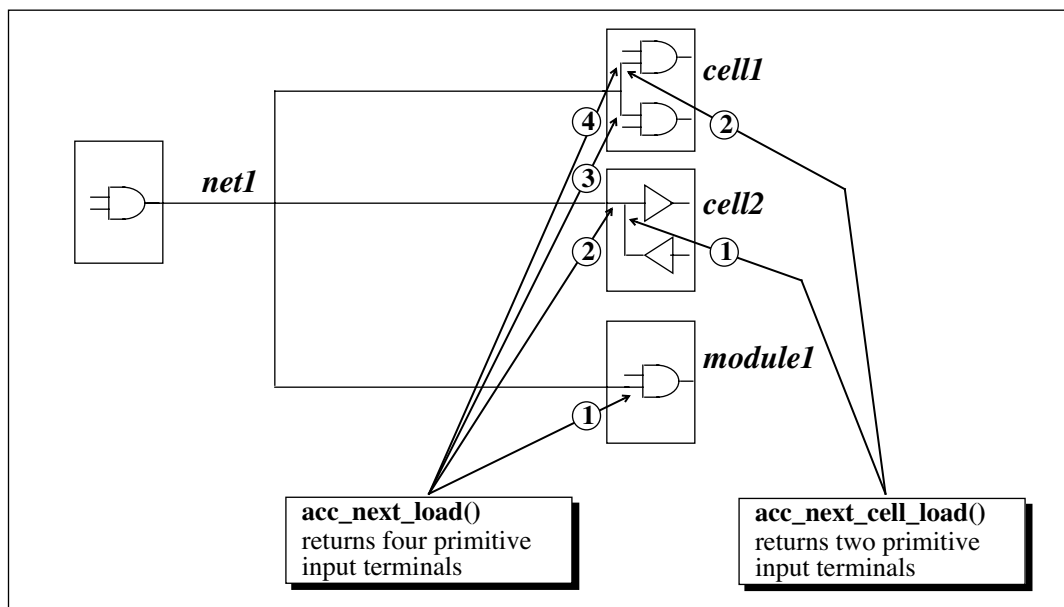


Figure 19-69— The difference between `acc_next_load()` and `acc_next_cell_load()`

The example shown in Figure 19-70 uses **acc_next_cell_load()** to find all cell loads on a net.

```
#include "acc_user.h"

get_cell_loads()
{
    handle    net_handle;
    handle    load_handle,load_net_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for net*/
    net_handle = acc_handle_tfarg(1);

    /*display names of all cell loads on the net*/
    load_handle = null;
    while(load_handle = acc_next_cell_load(net_handle,load_handle) )
    {
        load_net_handle = acc_handle_conn(load_handle);
        io_printf("Cell load is connected to: %s\n",
                  acc_fetch_fullname(load_net_handle) );
    }
    acc_close();
}
```

Figure 19-70—Using acc_next_cell_load()

19.67 acc_next_child()

acc_next_child()			
Synopsis:	Get handles for children of a module.		
Syntax:	<code>acc_next_child(reference_handle, child_handle)</code>		
Returns:	Type		Description
	handle	Handle of a module instance	
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a module
	handle	child_handle	Handle of the previous child found; initially null

The ACC routine **acc_next_child()** shall return handles to the module instances (children) within the reference module. The routine shall also return handles to top-level modules, as shown in Table 19-49.

Table 19-49—How acc_next_child() works

When	acc_next_child() shall
The <i>reference_handle</i> is not null	Scan for modules instantiated inside the module associated with <i>reference_handle</i>
The <i>reference_handle</i> is null	Scan for top-level modules (same as acc_next_topmod())

The ACC routine **acc_next_topmod()** does not work with **acc_collect()** or **acc_count()**, but **acc_next_child()** with a null reference handle argument can be used in place of **acc_next_topmod()**. For example:

```
acc_count(acc_next_child, null); /* counts top-level modules */

acc_collect(acc_next_child, null, &count); /* collect top-level modules */
```

Figure 19-71 shows the use of **acc_next_child()** to display the names of all modules instantiated within a module.

```
#include "acc_user.h"
print_children(module_handle)
handle  module_handle;
{
    handle  child_handle;
    io_printf("Module %s contains the following module instances:\n",
              acc_fetch_fullname(module_handle) );
    child_handle = null;
    while(child_handle = acc_next_child(module_handle, child_handle) )
        io_printf("    %s\n", acc_fetch_name(child_handle) );
}
```

Figure 19-71—Using acc_next_child()

19.68 acc_next_driver()

acc_next_driver()			
Synopsis:	Get handles to primitive terminals that drive a net.		
Syntax:	acc_next_driver(reference_handle, driver_handle)		
Returns:	Type	Description	
	handle	Handle of a primitive terminal	
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a net
	handle	driver_handle	Handle of the previous driver found; initially null

The ACC routine **acc_next_driver()** shall return handles to the primitive output or inout terminals that drive a net.

The example shown in Figure 19-72 uses **acc_next_driver()** to determine which terminals of a primitive drive a net.

```
#include "acc_user.h"

print_drivers(net_handle)
handle    net_handle;
{
    handle    primitive_handle;
    handle    driver_handle;

    io_printf("Net %s is driven by the following primitives:\n",
              acc_fetch_fullname(net_handle) );

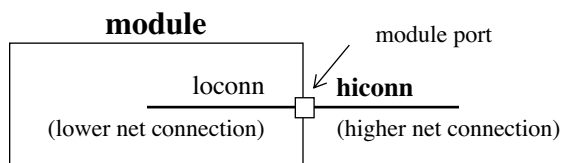
    /*get primitive that owns each terminal that drives the net*/
    driver_handle = null;
    while (driver_handle = acc_next_driver(net_handle, driver_handle) )
    {
        primitive_handle = acc_handle_parent(driver_handle);
        io_printf("    %s\n",
                  acc_fetch_fullname(primitive_handle) );
    }
}
```

Figure 19-72—Using acc_next_driver()

19.69 acc_next_hiconn()

acc_next_hiconn()			
Synopsis:	Get handles for hierarchically higher net connections to a module port.		
Syntax:	acc_next_hiconn(reference_handle, net_handle)		
Returns:	Type		Description
	handle		Handle of a net
Arguments:	Type	Name	Description
	handle	port_handle	Handle of a port
	handle	net_handle	Handle of the previous net found; initially null
Related routines:	Use acc_handle_hiconn() to get a handle to hierarchically higher connection of a specific port bit Use acc_next_loconn() to get handles to the hierarchically lower connection		

The ACC routine **acc_next_hiconn()** shall return handles to the hierarchically higher net connections to a module port. A hierarchically higher connection shall be the part of the net that appears outside the module, as shown in the following diagram:



When the reference handle passed to **acc_next_hiconn()** is a vector port, the routine shall return the hiconn nets bit-by-bit, starting with the msb (leftmost bit) and ending with the lsb (rightmost bit).

The example shown in Figure 19-73 uses **acc_next_hiconn()** and **acc_next_loconn()** to find and display all net connections made externally (hiconn) and internally (loconn) to a module port.


```
#include "acc_user.h"

display_connections(module_handle, port_handle)
handle module_handle, port_handle;
{
    handle    hiconn_net, loconn_net;

    /*get and display low connections*/
    io_printf("For module %s, port #%d internal connections are:\n",
              acc_fetch_fullname(module_handle),
              acc_fetch_index(port_handle) );
    loconn_net = null;
    while (loconn_net = acc_next_loconn(port_handle, loconn_net) )
        io_printf("    %s\n", acc_fetch_fullname(loconn_net) );

    /*get and display high connections*/
    io_printf("For module %s, port #%d external connections are:\n",
              acc_fetch_fullname(module_handle),
              acc_fetch_index(port_handle) );
    hiconn_net = null;
    while (hiconn_net = acc_next_hiconn(port_handle, hiconn_net) )
        io_printf("    %s\n", acc_fetch_fullname(hiconn_net) );
}
```

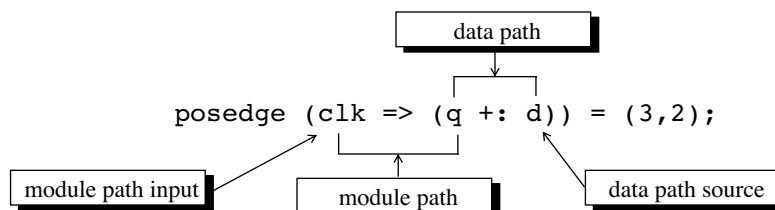
Figure 19-73—Using `acc_next_hiconn()` and `acc_next_loconn()`

19.70 acc_next_input()

acc_next_input()			
Synopsis:	Get handles to input path terminals of a module path or source terminals of a data path.		
Syntax:	<code>acc_next_input (reference_handle, terminal_handle)</code>		
Returns:	Type		Description
	handle		Handle of a module path terminal or a data path terminal
Arguments:	Type	Name	Description
	handle	reference_handle	Handle to a module path or data path
	handle	terminal_handle	Handle of the previous terminal found; initially null
Related routines:	Use <code>acc_handle_conn()</code> to get the net attached to the path terminal Use <code>acc_release_object()</code> to free memory allocated by <code>acc_next_input()</code>		

The ACC routine **acc_next_input()** shall return handles to the input path terminals of a module path or the source terminals of a data path. The routine **acc_handle_conn()** can be passed the input path terminal handle to derive the net connected to the terminal.

A *module path* is the specify block path for delays in the Verilog HDL description. A *data path* is part of the Verilog HDL description for edge-sensitive module paths, as shown in the following diagram:



The example shown in Figure 19-74 uses **acc_next_input()**. It accepts a handle to a scalar net or a net bit-select, and a module path. The application returns `true` if the net is connected to the input of the path.

```

bool is_net_on_path_input(net, path)
handle net; /* scalar net or bit-select of vector net */
handle path;
{
    handle port_in, port_conn, bit;

    /* scan path input terminals */
    port_in = null;
    while (port_in = acc_next_input(path, port_in) )
    {
        /* retrieve net connected to path terminal */
        port_conn = acc_handle_conn (port_in);

        bit = null;
        if (acc_object_of_type (port_conn, accExpandedVector) )
        {
            bit = null;
            while (bit = acc_next_bit (port_conn, bit) )
                if (acc_compare_handles (bit, net) )
                    return (true);
        }
        else
            if (acc_compare_handles(bit, net) )
                return (true);
    }

    return (false);
}

```

Figure 19-74—Using acc_next_input()

19.71 acc_next_load()

acc_next_load()			
Synopsis:	Get handles to primitive terminals driven by a net.		
Syntax:	<code>acc_next_load(reference_handle, load_handle)</code>		
Type		Description	
Returns:	handle	Handle of a primitive terminal	
Type		Name	Description
Arguments:	handle	reference_handle	Handle of a net
	handle	load_handle	Handle of the previous load found; initially null
Related routines:	Use <code>acc_next_cell_load()</code> to get cell module loads		

The ACC routine **acc_next_load()** shall return handles to the primitive loads that are being driven by a net. The handle for a load shall be a primitive input terminal.

The routines **acc_next_load()** and **acc_next_cell_load()** have different functionalities. The routine **acc_next_load()** shall return every primitive input terminal driven by a net, whether it is inside a cell or a module instance. The routine **acc_next_cell_load()** shall return only one primitive input terminal per cell input or inout port driven by a net. Figure 19-75 illustrates the difference, using a circuit in which **net1** drives primitive gates in **cell1**, **cell2**, and **module1**. For this circuit, **acc_next_load()** returns four primitive input terminals as loads on **net1**, while **acc_next_cell_load()** returns two primitive input terminals as loads on **net1**.

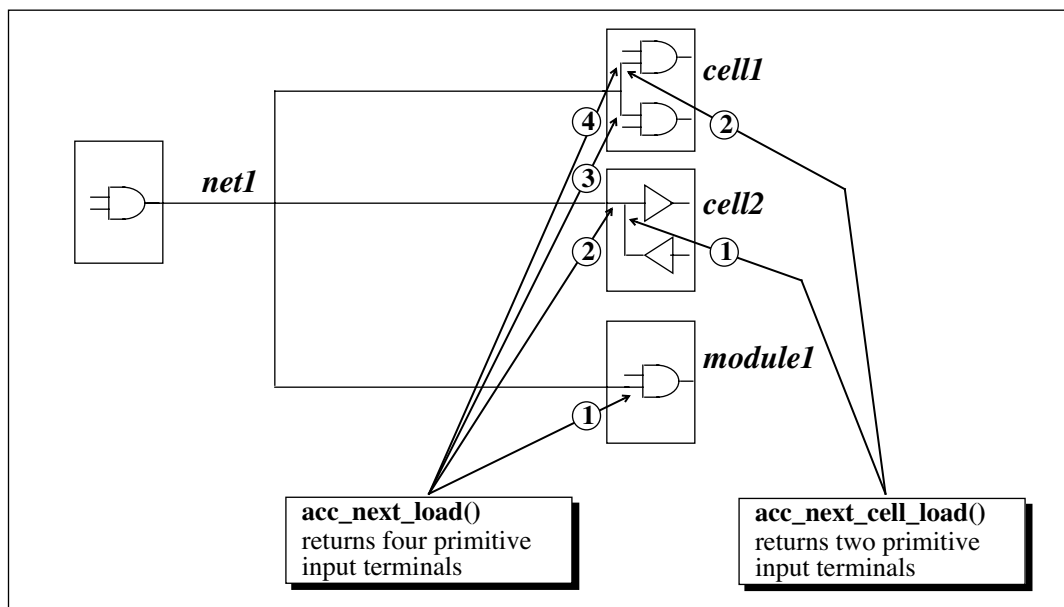


Figure 19-75— The difference between `acc_next_load()` and `acc_next_cell_load()`

The example shown in Figure 19-76 uses **acc_next_load()** to find all terminals driven by a net.

```
#include "acc_user.h"

get_loads()
{
    handle    net_handle, load_handle, load_net_handle;

    /*initialize the environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for net*/
    net_handle = acc_handle_tfarg(1);
    io_printf("Net %s is driven by:\n", acc_fetch_fullname(net_handle) );

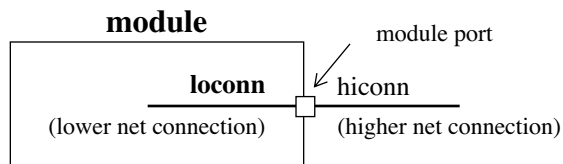
    /*get primitive that owns each terminal driven by the net*/
    load_handle = null;
    while (load_handle = acc_next_load(net_handle, load_handle) )
    {
        load_net_handle = acc_handle_conn(load_handle);
        io_printf("    %s ",
                  acc_fetch_fullname(load_net_handle) );
    }
    acc_close();
}
```

Figure 19-76—Using acc_next_load()

19.72 acc_next_loconn()

acc_next_loconn()			
Synopsis:	Get handles to hierarchically lower net connections to a port of a module.		
Syntax:	<code>acc_next_loconn(reference_handle, net_handle)</code>		
Returns:	Type		Description
	handle		Handle of a net
Arguments:	Type		Description
	handle	reference_handle	Handle of a port
	handle	net_handle	Handle of the previous net found; initially null
Related routines:	Use <code>acc_handle_loconn()</code> to get a handle to hierarchically lower connection of a specific port bit Use <code>acc_next_hiconn()</code> to get handles to the hierarchically higher connection		

The ACC routine **acc_next_loconn()** shall return handles to the hierarchically lower net connections to a module port. A hierarchically lower connection shall be the part of the net that appears inside the module, as shown in the following diagram:



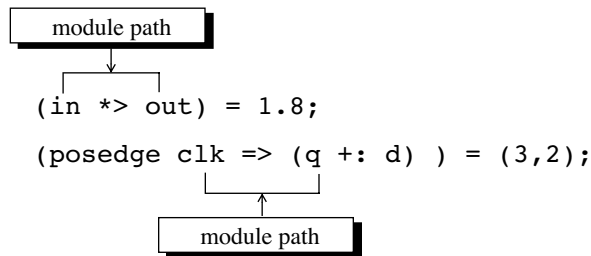
When the reference handle passed to **acc_next_loconn()** is a vector port, the routine shall return the loconn nets bit-by-bit, starting with the msb (leftmost bit) and ending with the lsb (rightmost bit).

Refer to Figure 19-73 for an example of using **acc_next_loconn()**.

19.73 acc_next_modpath()

acc_next_modpath()			
Synopsis:	Get handles to module paths of a module.		
Syntax:	acc_next_modpath(reference_handle, path_handle)		
Returns:	Type		Description
	handle		Handle of a module path
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a module
	handle	path_handle	Handle of the previous path found; initially null

The ACC routine **acc_next_modpath()** shall return handles to the module paths in a module. A module path is the specify block path for delays in the Verilog HDL description. For example:



The example in Figure 19-77 uses **acc_next_modpath()** to list the nets connected to all module paths in a module.

```

#include "acc_user.h"
get_path_nets(module_handle)
handle module_handle;
{
  handle path_handle, pathin_handle, pathout_handle;

  /*scan all paths in the module */
  io_printf("For module %s:\n",acc_fetch_fullname(module_handle) );
  path_handle = null;
  while (path_handle = acc_next_modpath(module_handle, path_handle) )
  {
    io_printf(" path %s connections are:\n",acc_fetch_name(path_handle) );
    pathin_handle = acc_handle_pathin(path_handle);
    pathout_handle = acc_handle_pathout(path_handle);
    io_printf("net %s connected to input\n",acc_fetch_name(pathin_handle) );
    io_printf("net %s connected to output\n",acc_fetch_name(pathout_handle) );
  }
}
  
```

Figure 19-77—Using acc_next_modpath()

19.74 acc_next_net()

acc_next_net()			
Synopsis:	Get handles to nets in a module.		
Syntax:	<code>acc_next_net(reference_handle, net_handle)</code>		
Returns:	Type		Description
	handle		Handle of a net
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a module
	handle	net_handle	Handle of the previous net found; initially null
Related routines:	Use <code>acc_object_of_type()</code> to determine if a net is scalar or vector, expanded or unexpanded Use <code>acc_next_bit()</code> to get handles to all bits of an expanded vector net		

The ACC routine **acc_next_net()** shall return handles to the nets within a module scope. The routine shall return a handle to a vector net as a whole; it does not return a handle to each individual bit of a vector net. The routine **acc_object_of_type()** can be used to determine if a net is vector or scalar and if it is expanded or unexpanded. The routine **acc_next_bit()** can be used to retrieve a handle for each bit of an expanded vector net.

The example shown in Figure 19-78 uses **acc_next_net()** to display the names of all nets in a module.

```
#include "acc_user.h"

display_net_names()
{
    handle    mod_handle, net_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for module*/
    mod_handle = acc_handle_tfarg(1);
    io_printf("Module %s contains the following nets:\n",
              acc_fetch_fullname(mod_handle) );

    /*display names of all nets in the module*/
    net_handle = null;
    while (net_handle = acc_next_net(mod_handle, net_handle) )
        io_printf("    %s\n", acc_fetch_name(net_handle) );

    acc_close();
}
```

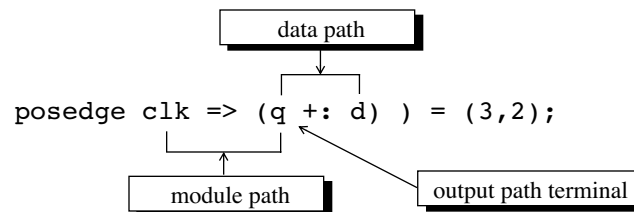
Figure 19-78—Using acc_next_net()

19.75 acc_next_output()

acc_next_output()			
Synopsis:	Get handles to output path terminals of a module path or data path.		
Syntax:	<code>acc_next_output(reference_handle, terminal_handle)</code>		
Returns:	Type		Description
	handle		Handle to a module path terminal or data path terminal
Arguments:	Type	Name	Description
	handle	reference_handle	Handle to a module path or data path
	handle	terminal_handle	Handle of the previous terminal found; initially null
Related routines:	Use <code>acc_handle_conn()</code> to get the net attached to the path terminal Use <code>acc_release_object()</code> to free memory allocated by <code>acc_next_output()</code>		

The ACC routine **acc_next_output()** shall return handles to the output path terminals of a module path or a data path. The routine **acc_handle_conn()** can be passed the output path terminal handle to derive the net connected to the terminal.

A *module path* is the specify block path for delays in the Verilog HDL description. A *data path* is part of the Verilog HDL description for edge-sensitive module paths, as shown in the following illustration:



The example shown in Figure 19-79 uses **acc_next_output()**. It accepts a handle to a scalar net or a net bit-select, and a module path. The application returns `true` if the net is connected to the output of the path.

```
bool is_net_on_path_output(net, path)
handle net; /* scalar net or bit-select of vector net */
handle path;
{
    handle port_out, port_conn, bit;

    /* scan path output terminals */
    port_out = null;
    while (port_out = acc_next_output(path, port_out) )
    {
        /* retrieve net connected to path terminal */
        port_conn = acc_handle_conn (port_out);

        bit = null;
        if (acc_object_of_type (port_conn, accExpandedVector) )
        {
            bit = null;
            while (bit = acc_next_bit (port_conn, bit) )
                if (acc_compare_handles (bit, net) )
                    return (true);
        }
        else
            if (acc_compare_handles (port_conn, net) )
                return (true);
    }

    return (false);
}
```

Figure 19-79—Using acc_next_output()

19.76 acc_next_parameter()

acc_next_parameter()			
Synopsis:	Get handles to parameters within a module.		
Syntax:	acc_next_parameter(reference_handle, parameter_handle)		
Returns:	Type	Description	
	handle	Handle of a parameter	
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a module
	handle	parameter_handle	Handle of the previous parameter found; initially null
Related routines:	Use acc_fetch_paramtype() to determine the parameter data type Use acc_fetch_paramval() to retrieve the parameter value Use acc_next_specparam() to get handles to specify block parameters		

The ACC routine **acc_next_parameter()** shall return handles to the parameters in a module. This handle can be passed to **acc_fetch_paramtype()** and **acc_fetch_paramval()** to retrieve the data type and value of the parameter.

The example shown in Figure 19-80 uses **acc_next_parameter()** to scan for all parameters in a module.

```
#include "acc_user.h"
print_parameter_values(module_handle)
handle  module_handle;
{
    handle  param_handle;
    /*scan all parameters in the module and display values according to type*/
    param_handle = null;
    while (param_handle = acc_next_parameter(module_handle,param_handle) )
    {
        io_printf("Parameter %s = ",acc_fetch_fullname(param_handle) );
        switch (acc_fetch_paramtype(param_handle) )
        {
            case accRealParam:
                io_printf("%lf\n", acc_fetch_paramval(param_handle) );
                break;
            case accIntegerParam:
                io_printf("%d\n", (int)acc_fetch_paramval(param_handle) );
                break;
            case accStringParam:
                io_printf("%s\n", (char*)(int)acc_fetch_paramval(param_handle) );
        }
    }
}
```

Figure 19-80—Using acc_next_parameter()

19.77 acc_next_port()

acc_next_port()			
Synopsis:	Get a handle to input, output, or inout ports of a module or connected to a net.		
Syntax:	<code>acc_next_port(reference, port_handle)</code>		
Returns:	Type	Description	
	handle	Handle of a module port	
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a module or net
	handle	object_handle	Handle of the previous port found; initially null
Related routines:	Use <code>acc_fetch_direction()</code> to determine the direction of a port Use <code>acc_next_portout()</code> to get handles to just output and inout ports		

The ACC routine **acc_next_port()** shall return handles to the input, output, and inout ports of a module. The handles shall be returned in the order specified by the port list in the module declaration, working from left to right.

The routine **acc_next_port()** shall be used two ways, as shown in Table 19-50.

Table 19-50—How acc_next_port() works

If the reference handle is	acc_next_port() shall return
A handle to a module	All ports of the module
A handle to a net	All ports connected to the net within the scope of the net

The example shown in Figure 19-81 uses **acc_next_port()** to find and display the input ports of a module.

```
#include "acc_user.h"

display_inputs(module_handle)
handle    module_handle;
{
    handle    port_handle;
    int        direction;

    /*get handle for each module port*/
    port_handle = null;
    while (port_handle = acc_next_port(module_handle, port_handle) )
    {
        /*give the index of each input port*/
        if (acc_fetch_direction(port_handle) == accInput)
            io_printf("Port #%d of %s is an input\n",
                      acc_fetch_index(port_handle),
                      acc_fetch_fullname(module_handle) );
    }
}
```

Figure 19-81—Using acc_next_port() with a module handle

The example shown in Figure 19-82 uses **acc_next_port()** to find the port that is connected to a net, and then to display information about other nets connected to each bit of the same port.

```
display_port_connections()
{
    handle net = acc_handle_tfarg(1);
    handle port, bit;

    port = bit = null;
    while (port = acc_next_port(net, port) )
        if (acc_object_of_type(port, accVectorPort) )
            while (bit = acc_next_bit(port, bit) )
                io_printf("PORTBIT: %s LOCONN: %s HICONN: %s/n",
                          acc_fetch_fullname(bit),
                          acc_fetch_fullname(acc_handle_loconn(bit) ),
                          acc_fetch_fullname(acc_handle_hiconn(bit) ) );
}
```

Figure 19-82—Using acc_next_port() with a net handle

19.78 acc_next_portout()

acc_next_portout()			
Synopsis:	Get handles to output or inout ports of a module.		
Syntax:	<code>acc_next_portout(reference_handle, port_handle)</code>		
Returns:	Type		Description
	handle		Handle of a module port
Arguments:	Type		Description
	handle	reference_handle	Handle of a module
	handle	port_handle	Handle of the previous port found; initially null
Related routines:	Use <code>acc_fetch_direction()</code> to determine the direction of a port Use <code>acc_next_port()</code> to get handles to input, output, and inout ports		

The ACC routine **acc_next_portout()** shall return handles to the output and inout ports of a module. The handles shall be returned in the order specified by the port list in the module declaration, working from left to right.

The example shown in Figure 19-83 uses **acc_next_portout()** to find the output and inout ports of a module.

```
#include "acc_user.h"

display_outputs(module_handle)
handle    module_handle;
{
    handle    port_handle;

    /*get handle for each module port*/
    port_handle = null;
    while (port_handle = acc_next_portout(module_handle, port_handle) )
    {
        /*give the index of each output or inout port*/
        io_printf("Port #%d of %s is an output or inout\n",
                  acc_fetch_index(port_handle),
                  acc_fetch_fullname(module_handle) );
    }
}
```

Figure 19-83—Using acc_next_portout()

19.79 acc_next_primitive()

acc_next_primitive()			
Synopsis:	Get handles to gates, switches, or user-defined primitives (UDPs) within a module.		
Syntax:	<code>acc_next_primitive(reference_handle, primitive_handle)</code>		
Type		Description	
Returns:	handle	Handle of a primitive	
Type		Name	Description
Arguments:	handle	reference_handle	Handle of a module
	handle	primitive_handle	Handle of the previous primitive found; initially null

The ACC routine **acc_next_primitive()** shall return handles to the built-in and UDPs within a module.

The example shown in Figure 19-84 uses **acc_next_primitive()** to display the definition names of all primitives in a module.

```
#include "acc_user.h"

get_primitive_definitions()
{
    handle    module_handle, prim_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for module*/
    module_handle = acc_handle_tfarg(1);

    io_printf("Module %s contains the following types of primitives:\n",
              acc_fetch_fullname(module_handle) );

    /*get and display defining names of all primitives in the module*/
    prim_handle = null;
    while (prim_handle = acc_next_primitive(module_handle, prim_handle) )
        io_printf("    %s\n",
                  acc_fetch_defname(prim_handle) );
    acc_close();
}
```

Figure 19-84—Using acc_next_primitive()

19.80 acc_next_scope()

acc_next_scope()			
Synopsis:	Get handles to hierarchy scopes within a scope.		
Syntax:	acc_next_scope(reference_handle, scope_handle)		
Returns:	Type	Description	
	handle	Handle to a hierarchy scope	
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a scope
	handle	scope_handle	Handle of the previous scope found; initially null
Related routines:	Use acc_fetch_type() and acc_fetch_fulltype() to determine the type of scope object found Use acc_next_topmod() to get handles to top-module scopes		

The ACC routine **acc_next_scope()** shall return the handles to the internal scopes within a given scope. Internal scopes shall be the immediate children of the *reference_handle*. The reference scope and the internal scopes shall be on of the following:

- A top-level module
- A module instance
- A named begin-end block
- A named fork-join block
- A Verilog HDL task
- A Verilog HDL function

19.81 acc_next_specparam()

acc_next_specparam()			
Synopsis:	Get handles to specify block parameters within a module.		
Syntax:	acc_next_specparam(reference_handle, specparam_handle)		
Returns:	Type		Description
	handle		Handle of a specparam
Arguments:	Type		Description
	handle	module_handle	Handle of a module
	handle	specparam_handle	Handle of the previous specparam found; initially null
Related routines:	Use acc_fetch_paramtype() to determine the parameter data type Use acc_fetch_paramval() to retrieve the parameter value Use acc_next_parameter() to get handles to module parameters		

The ACC routine **acc_next_specparam()** shall return handles to the specify block parameters in a module. This handle can be passed to **acc_fetch_paramtype()** and **acc_fetch_paramval()** to retrieve the data type and value.

The example shown in Figure 19-85 uses **acc_next_specparam()** to scan for all specparams in a module.

```
#include "acc_user.h"
print_specparam_values(module_handle)
handle    module_handle;
{
    handle    sparam_handle;
    /*scan all parameters in the module and display values according to type*/
    sparam_handle = null;
    while (sparam_handle = acc_next_specparam(module_handle,sparam_handle) )
    {
        io_printf("Specparam %s = ", acc_fetch_fullname(sparam_handle) );
        switch (acc_fetch_paramtype(sparam_handle) )
        {
            case accRealParam:
                io_printf("%lf\n", acc_fetch_paramval(sparam_handle) );
                break;
            case accIntegerParam:
                io_printf("%d\n", (int)acc_fetch_paramval(sparam_handle) );
                break;
            case accStringParam:
                io_printf("%s\n", (char*)(int)acc_fetch_paramval(sparam_handle));
        }
    }
}
```

Figure 19-85—Using acc_next_specparam()

19.82 acc_next_tchk()

acc_next_tchk()			
Synopsis:	Get handles to timing checks within a module.		
Syntax:	acc_next_tchk(reference_handle, timing_check_handle)		
Returns:	Type	Description	
	handle	Handle of a timing check	
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a module
	handle	timing_check_handle	Handle of the previous timing check found; initially null
Related routines:	Use acc_handle_tchk() to get a timing check handle using the timing check description Use acc_handle_tchkarg1() and acc_handle_tchkarg2() to get handles of the timing check arguments Use acc_handle_notifier() to get a handle to the timing check notifier register Use acc_fetch_delays(), acc_append_delays(), and acc_replace_delays() to read or modify timing check values		

The ACC routine **acc_next_tchk()** shall return handles to the timing checks within a module. The handles can be passed to other ACC routines to get the nets or notifier in the time check, and to read or modify timing check values.

The example shown in Figure 19-86 uses **acc_next_tchk()** to display information about setup timing checks.

```
#include "acc_user.h"
show_setup_check_nets()
{
    handle          mod_handle,cell_handle;
    handle          tchk_handle,tchkarg1_handle,tchkarg2_handle;
    int             tchk_type,counter;

    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for module*/
    mod_handle = acc_handle_tfarg(1);

    /*scan all cells in module for timing checks*/
    cell_handle = null;
    while (cell_handle = acc_next_cell(mod_handle, cell_handle) )
    {
        io_printf("cell is: %s\n", acc_fetch_name(cell_handle) );
        counter = 0;
        tchk_handle = null;
        while (tchk_handle = acc_next_tchk(cell_handle, tchk_handle) )
        {
            /*get nets connected to timing check arguments*/
            tchk_type = acc_fetch_fulltype(tchk_handle);
            if (tchk_type == accSetup)
            {
                counter++;
                io_printf("    for setup check #%d:\n",counter);
                tchkarg1_handle = acc_handle_tchkarg1(tchk_handle,mod_handle);
                tchkarg2_handle = acc_handle_tchkarg2(tchk_handle,mod_handle);
                io_printf("        1st net is %s\n        2nd net is %s\n",
                    acc_fetch_name(acc_handle_conn(tchkarg1_handle) ),
                    acc_fetch_name(acc_handle_conn(tchkarg2_handle) ) );
            }
        }
    }
    acc_close();
}
```

Figure 19-86—Using acc_next_tchk()

19.83 acc_next_terminal()

acc_next_terminal()			
Synopsis:	Get handles to terminals of a gate, switch, or user-defined primitive (UDP).		
Syntax:	acc_next_terminal(reference_handle, terminal_handle)		
Returns:	Type	Description	
	handle	Handle of a primitive terminal	
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a gate, switch or UDP
	handle	terminal_handle	Handle of the previous terminal found; initially null

The ACC routine **acc_next_terminal()** shall return handles to the terminals on a primitive. The handles shall be returned in the order of the primitive instance statement, starting at terminal 0 (the leftmost terminal).

The example shown in Figure 19-87 uses **acc_next_terminal()** together with **acc_handle_conn()** to retrieve all nets connected to a primitive.

```
#include "acc_user.h"

display_terminals()
{
    handle    prim_handle,term_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*set development version*/
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for primitive*/
    prim_handle = acc_handle_tfarg(1);

    io_printf("Connections to primitive %s:\n",
              acc_fetch_fullname(prim_handle) );
    /*scan all terminals of the primitive
    /* and display their nets*/
    term_handle = null;
    while (term_handle = acc_next_terminal(prim_handle,term_handle) )
        io_printf("    %s\n",
                  acc_fetch_name(acc_handle_conn(term_handle) ) );
    acc_close();
}
```

Figure 19-87—Using acc_next_terminal()

19.84 acc_next_topmod()

acc_next_topmod()			
Synopsis:	Get handles to top-level modules.		
Syntax:	acc_next_topmod(module_handle)		
Type		Description	
Returns:	handle	Handle of a top-level module	
Type		Name	Description
Arguments:	handle	module_handle	Handle of the previous top-level module found; initially null
Related routines:	Use acc_next_child() with a null reference_handle to collect or count top-level modules with acc_collect() and acc_count()		

The ACC routine **acc_next_topmod()** shall return handles to the top-level modules in a design.

The ACC routine **acc_next_topmod()** does not work with **acc_collect()** or **acc_count()**, but **acc_next_child()** with a null reference handle argument can be used in place of **acc_next_topmod()**. For example:

```
acc_count(acc_next_child, null); /* counts top-level modules */
acc_collect(acc_next_child, null, &count); /* collect top-level modules */
```

The example shown in Figure 19-88 uses **acc_next_topmod()** to display the names of all top-level modules.

```
#include "acc_user.h"

show_top_modules()
{
    handle      module_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*scan all top-level modules*/
    io_printf("The top-level modules are:\n");
    module_handle = null;
    while (module_handle = acc_next_topmod(module_handle) )
        /*display the instance name of each module*/
        io_printf("    %s\n", acc_fetch_name(module_handle) );
    acc_close();
}
```

Figure 19-88—Using acc_next_topmod()

19.85 acc_object_in_typelist()

acc_object_in_typelist()			
Synopsis:	Determine whether an object fits a type or fulltype, or special property, as specified in an input array.		
Syntax:	<code>acc_object_in_typelist(object_handle, object_type_array)</code>		
Returns:	Type	Description	
	bool	true if the type, fulltype, or property of an object matches one specified in the array; false if there is no match	
Arguments:	Type	Name	Description
	handle	object_handle	Handle of an object
	static integer array	object_type_array	Static integer array containing one or more predefined integer constants that represent the types and properties of objects desired; the last element shall be 0
Related routines:	Use <code>acc_object_of_type()</code> to check for a match to a single predefined constant		

The ACC routine **acc_object_in_typelist()** shall determine whether an object fits one of a list of types, fulltypes, or special properties. The properties for which **acc_object_in_typelist()** is to check shall be listed as an array of constants in a static integer array. The array can contain any number and combination of the predefined integer constants, and it shall be terminated by a **0**.

The following C-language statement shows an example of how to declare an array of object types called `wired_nets`:

```
static int wired_nets[5]={accWand,accWor,accTriand,accTrior,0};
```

When this array is passed to **acc_object_in_typelist()**, the ACC routine shall return `true` if its `object_handle` argument is a wired net.

All type and fulltype constants shall be supported by **acc_object_in_typelist()**. These constants are listed in Table 18-25.

The special property constants supported by **acc_object_in_typelist()** are listed in Table 19-51.

The example shown in Figure 19-89 uses **acc_object_in_typelist()** to determine if a net is a wired net. The application then displays the name of each wired net found.

```
#include "acc_user.h"

display_wired_nets()
{
    static int wired_nets[5]={accWand,accWor,accTriand,accTrior,0};
    handle net_handle;

    /*reset environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get handle for net*/
    net_handle = acc_handle_tfarg(1);

    /*if a wired logic net, display its name*/
    if (acc_object_in_typelist(net_handle,wired_nets) )
        io_printf("Net %s is a wired net\n",acc_fetch_name(net_handle) );
    else
        io_printf("Net %s is not a wired net\n",acc_fetch_name(net_handle) );
}
```

Figure 19-89—Using acc_object_in_typelist()

19.86 acc_object_of_type()

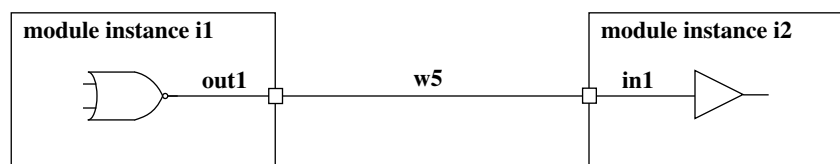
acc_object_of_type()			
Synopsis:	Determine whether an object fits a specified type or fulltype, or special property.		
Syntax:	<code>acc_object_of_type(object_handle, object_type)</code>		
Returns:	Type		Description
	bool		true if the type, fulltype, or property of an object matches the object_type argument false if there is no match
Arguments:	Type	Name	Description
	handle	object_handle	Handle of an object
	int	object_type	An integer constant that represents a type, fulltype, or special property
Related routines:	Use acc_object_in_typelist() to check for a match to any of several predefined constants		

The ACC routine **acc_object_of_type()** shall determine whether an object fits a specified type, fulltype, or special property. The type, fulltype, or property is an integer constant, defined in `acc_user.h`. All type and fulltype constants shall be supported by **acc_object_of_type()**. These constants are listed in Table 18-25. The special property constants supported by **acc_object_of_type()** are listed in Table 19-51.

Table 19-51 — Special object properties

Property of object	Predefined integer constant
Scalar	accScalar
Vector	accVector
Collapsed net	accCollapsedNet
Expanded vector	accExpandedVector
Unexpanded vector	accUnexpandedVector
Hierarchy scope	accScope
Module path with ifnone condition	accModPathHasIfnone

Simulated nets and *collapsed* nets are defined as follows. When a Verilog HDL source description connects modules together, a chain of nets with different scopes and names are connected, as is illustrated in the following simple diagram:



In this small circuit, nets `out1`, `w5`, and `in1` are all tied together, effectively becoming the same net. Software products can collapse nets that are connected together within the data structure of the product. The resultant net after collapsing is referred to as a simulated net; the other nets are referred to as collapsed nets. The ACC routines can obtain a handle to any net, whether it is collapsed or not. The routine `acc_object_of_type()` can be used to determine if a net has been collapsed. The routine `acc_handle_simulated_net()` can be used to find the resultant net from the net collapsing process.

Expanded and *unexpanded* vectors determine if ACC routines can access a vector as a whole or access the bits within a vector. If a vector has the property `accExpandedVector`, then access to the discrete bits of the vector shall be permitted. This property has to be `true` in order for certain ACC routines, such as `acc_next_bit()`, to access each bit of a vector. If a vector has the property `accUnexpandedVector`, then access to the vector as a whole shall be permitted. This property has to be `true` in order for certain ACC routines to access the complete vector. A vector object can have just one of these properties `true`, or both can be `true`.

The example shown in Figure 19-90 uses `acc_object_of_type()` to determine whether nets are collapsed nets. The application then displays each collapsed net, along with the simulated net.

```
#include "acc_user.h"

void display_collapsed_nets()
{
    handle    mod_handle;
    handle    net_handle;
    handle    simulated_net_handle;

    /*reset environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*get scope-first argument passed to user-defined system task*/
    /* associated with this routine*/
    mod_handle = acc_handle_tfarg(1);
    io_printf("In module %s:\n", acc_fetch_fullname(mod_handle) );
    net_handle = null;

    /*display name of each collapsed net and its net of origin*/
    while (net_handle = acc_next_net(mod_handle, net_handle) )
    {
        if (acc_object_of_type(net_handle, accCollapsedNet) )
        {
            simulated_net_handle = acc_handle_simulated_net(net_handle);
            io_printf("    net %s was collapsed onto net %s\n",
                      acc_fetch_name(net_handle),
                      acc_fetch_name(simulated_net_handle) );
        }
    }
}
```

Figure 19-90—Using `acc_object_of_type()`

19.87 acc_product_type()

acc_product_type()		
Synopsis:	Get the software product type that is calling the PLI application.	
Syntax:	<code>acc_product_type()</code>	
Returns:	Type	Description
	int	A predefined integer constant representing the software product type
Arguments:	Type	Name Description
	None	

The ACC routine **acc_product_type()** shall return a predefined integer constant that identifies the class of software product that is calling the PLI application. This information can be useful when a PLI application needs to customize the routine to specific types of software implementations. For example, a delay calculator might use typical delays for logic simulation and min:typ:max delays for timing analysis.

The integer constant values returned by **acc_product_type()** are listed in Table 19-52.

Table 19-52—Product types returned by acc_product_type()

If the product is	acc_product_type() returns
A logic simulator	accSimulator
A timing analyzer	accTimingAnalyzer
A fault simulator	accFaultSimulator
Some other product	accOther

NOTE—Software product vendors can define additional integer constants specific to their products.

The example shown in Figure 19-91 uses **acc_product_type()** to identify and display the product type being used.

```
#include "acc_user.h"
show_application()
{
    /* reset environment for ACC routines */
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /* show application type and ACC routine version */
    switch (acc_product_type() )
    {
        case accSimulator:
            io_printf("Running logic simulation with PLI version %s\n",acc_version());
            break;
        case accTimingAnalyzer:
            io_printf("Running timing analysis with PLI version %s\n",acc_version());
            break;
        case accFaultSimulator:
            io_printf("Running fault simulation with PLI version %s\n",acc_version());
            break;
        default:
            io_printf("Running other product with PLI version %s\n",acc_version());
    }
    acc_close();
}
```

Figure 19-91 — Using acc_product_type()

19.88 acc_product_version()

acc_product_version()		
Synopsis:	Get the version of the software product that is linked to the ACC routines.	
Syntax:	acc_product_version()	
	Type	Description
Returns:	char *	Pointer to a character string
	Type	Name Description
Arguments:	None	
Related routines:	Use acc_product_type() to get the type of software product Use acc_version() to get the version of PLI ACC routines	

The ACC routine **acc_product_version()** shall return a pointer to a character string that indicates the version of the software product that called the PLI application. The return value for this routine is placed in the ACC internal string buffer. See 18.9 for explanation of strings in ACC routines.

The character string shall be in the following format:

<product_name> Version <version_number>

For example:

"Verilog Simulator Version OVIsim 1.0"

The string returned by **acc_product_version()** shall be defined by the software tool vendor.

The example shown in Figure 19-92 uses **acc_product_version()** to identify the version of the software product that is linked to ACC routines.

```
#include "acc_user.h"

show_versions()
{
    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*show version of ACC routines*/
    /* and version of Verilog that is linked to ACC routines*/
    io_printf("Running %s with %s\n",acc_version(),acc_product_version() );
    acc_close();
}
```

Figure 19-92—Using acc_product_version()

19.89 acc_release_object()

acc_release_object()			
Synopsis:	Deallocate memory allocated by calls to acc_next_input() and acc_next_output().		
Syntax:	acc_release_object(object_handle)		
Returns:	Type		Description
	void		
Arguments:	Type	Name	Description
	handle	object_handle	Handle to an input or output terminal path
Related routines:	Use acc_next_input() to get handles to module path inputs and data path inputs		
	Use acc_next_output() to get handles to module path outputs and data path outputs		

The ACC routine **acc_release_object()** shall deallocate memory that was allocated by a call to **acc_next_input()** or **acc_next_output()**. The routine should be called after using these ACC routines under the following circumstances:

- Not all inputs or outputs were scanned.
- The input or output path had only one terminal.
- An error was returned.

The example shown in Figure 19-93 finds the data path corresponding to an input module path, and it displays the source and destination port names for the data path. The example calls **acc_next_input()** and **acc_next_output()** to get the first input and output, respectively, for a given path. Since these routines are only called once, **acc_release_object()** is called to free the memory allocated for the input and output handles.

```
void display_datapath_terms(modpath)
handle modpath;
{
    handle datapath = acc_handle_datapath(modpath);
    handle pathin   = acc_next_input(datapath, null);
    handle pathout  = acc_next_output(datapath, null);
    /* there is only one input and output to a data path */
    io_printf("DATAPATH INPUT:   %s\n", acc_fetch_fullname(pathin) );
    io_printf("DATAPATH OUTPUT:  %s\n", acc_fetch_fullname(pathout) );
    acc_release_object(pathin);
    acc_release_object(pathout);
}
```

Figure 19-93—Using acc_release_object()

19.90 acc_replace_delays()

acc_replace_delays() for single delay values (accMinTypMaxDelays set to "false")			
Synopsis:	Replace existing delays for primitives, module paths, timing checks, module input ports, and inter-module paths.		
Syntax: Primitives Ports Port bits Intermodule paths	acc_replace_delays(object_handle, rise_delay, fall_delay, z_delay)		
Module paths	acc_replace_delays(object_handle, d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12)		
Timing checks	acc_replace_delays(object_check_handle, limit)		
Type		Description	
Returns:	bool	1 if successful; 0 if an error occurred	
Type		Name	Description
Arguments:	handle	object_handle	Handle of a primitive, module path, timing check, module input port, bit of a module input port, or intermodule path
	double	rise_delay fall_delay	Rise and fall delay for 2-state primitive, 3-state primitive, module input port, module input port bit, or intermodule path
Conditional	double	z_delay	If accToHiZDelay is set to "from_user": turn-off (to Z) transition delay for 3-state primitives, module input ports, module input port bits, or intermodule paths
	double	d1	If accPathDelayCount is set to "1": delay for all transitions for module paths If accPathDelayCount is set to "2" or "3": rise transition delay for module paths If accPathDelayCount is set to "6" or "12": 0→1 transition delay for module paths
Conditional	double	d2	If accPathDelayCount is set to "2" or "3": fall transition delay for module paths If accPathDelayCount is set to "6" or "12": 1→0 transition delay for module paths
Conditional	double	d3	If accPathDelayCount is set to "3": turn-off transition delay for module paths If accPathDelayCount is set to "6" or "12": 0→Z transition delay for module paths
Conditional	double	d4 d5 d6	If accPathDelayCount is set to "6" or "12": d4 is Z→1 transition delay for module paths d5 is 1→Z transition delay for module paths d6 is Z→0 transition delay for module paths
Conditional	double	d7 d8 d9 d10 d11 d12	If accPathDelayCount is set to "12": d7 is 0→X transition delay for module paths d8 is X→1 transition delay for module paths d9 is 1→X transition delay for module paths d10 is X→0 transition delay for module paths d11 is X→Z transition delay for module paths d12 is Z→X transition delay for module paths
	double	limit	Limit of timing check

acc_replace_delays() for min:typ:max delays (accMinTypMaxDelays set to "true")			
Synopsis:	Replace min:typ:max delay values for primitives, module paths, timing checks, module input ports, or intermodule paths; the delay values are contained in an array .		
Syntax:	<code>acc_append_delays(object_handle, array_ptr)</code>		
Type		Description	
Returns:	bool	1 if successful; 0 if an error is encountered	
Type		Name	Description
Arguments:	handle	object_handle	Handle of a primitive, module path, timing check, module input port, bit of a module input port, or intermodule path
	double address	array_ptr	Pointer to array of min:typ:max delay values; the size of the array depends on the type of object and the setting of accPathDelayCount (see 18.8)

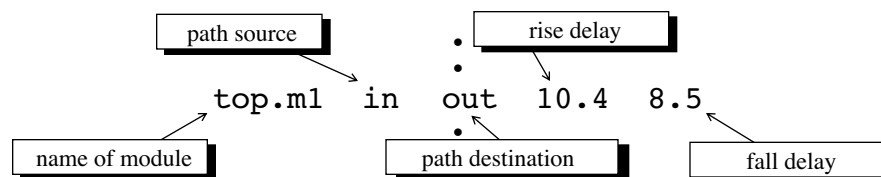
The ACC routine **acc_replace_delays()** shall work differently depending on how the configuration parameter **accMinTypMaxDelays** is set. When this parameter is set to **false**, a single delay per transition shall be assumed, and delays shall be passed as individual arguments. For this single delay mode, the first syntax table in this clause shall apply.

When **accMinTypMaxDelays** is set to **true**, **acc_replace_delays()** shall pass one or more sets of minimum:typical:maximum delays contained in an array, rather than single delays passed as individual arguments. For this min:typ:max delay mode, the second syntax table in this clause shall apply.

The number of delay values replaced by **acc_replace_delays()** shall be determined by the type of object and the setting of configuration parameters. Refer to 18.8 for a description of how the number of delay values are determined.

The routine **acc_replace_delays()** shall write delays in the timescale of the module that contains the object_handle.

The example shown in Figure 19-94 uses **acc_replace_delays()** to replace the current delays on a path with new delay values read from a file called **pathdelay.dat**. The format of the file is shown in the following diagram:



```

#include <stdio.h>
#include "acc_user.h"

write_path_delays()
{
    FILE      *infile;
    char      full_module_name[NAME_SIZE];
    char      pathin_name[NAME_SIZE], pathout_name[NAME_SIZE];
    double    rise,fall;
    handle    mod_handle, path_handle;

    /*initialize the environment for ACC routines*/
    acc_initialize();

    /*set development version*/
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*set accPathDelayCount parameter to return rise and fall delays only*/
    acc_configure(accPathDelayCount, "2");

    /*read delays from file - "r" means read only*/
    infile = fopen("pathdelay.dat","r");
    fscanf(infile, "%s %s %s %lf %lf",
           full_module_name,pathin_name,pathout_name,&rise,&fall);

    /*get handle for the module and the path*/
    mod_handle = acc_handle_object(full_module_name);
    path_handle = acc_handle_modpath(mod_handle,pathin_name,pathout_name);

    /*replace delays with new values*/
    acc_replace_delays(path_handle, rise, fall);

    acc_close();
}

```

Figure 19-94—Using `acc_replace_delays()` in single delay mode

The example shown in Figure 19-95 uses `acc_replace_delays()` to scale the min:typ:max delays on all primitive delays inside cells within a given scope. The application fetches the existing delays for an object, multiplies the delays by a scale factor, and replaces the delays with the new, scaled values. This example assumes that the user application is associated through the PLI interface mechanism with a user-defined system task called `$scaleprimdelays`. The scope and scale factors are passed as arguments as follows:

```

$scaleprimdelays( mychip, 0.4, 1.0, 1.6 );

```

Diagram illustrating the arguments of the `$scaleprimdelays` function call:

- `mychip`: scope
- `0.4`: scale factor for minimum delay
- `1.0`: scale factor for typical delay
- `1.6`: scale factor for maximum delay


```

#include "acc_user.h"
#include "veriususer.h"

void scale_prim_delays()
{
    handle top, cell, prim;
    int i;
    double da[9]; ← array has to hold three sets
                    ← of min:typ:max values for
                    ← rise, fall, and turn-off delays

    double min_scale_factor, typ_scale_factor, max_scale_factor;

    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");
    acc_configure(accMinTypMaxDelays, "true");

    top = acc_handle_tfarg(1); ← argument #1: Scope
    min_scale_factor = acc_fetch_tfarg(2); ← argument #2: Scale factor for minimum delay
    typ_scale_factor = acc_fetch_tfarg(3); ← argument #3: Scale factor for typical delay
    max_scale_factor = acc_fetch_tfarg(4); ← argument #4: Scale factor for maximum delay

    io_printf("Scale min:typ:max delays for primitives in cells below %s\n",
              acc_fetch_fullname(top) );
    io_printf("Scaling factors-min:typ:max-%4.2f:%4.2f:%4.2f\n",
              min_scale_factor, typ_scale_factor, max_scale_factor);
    cell = null;
    while (cell = acc_next_cell(top, cell) )
    {
        prim = null;
        while (prim = acc_next_primitive(cell, prim) )
        {
            acc_fetch_delays(prim, da); ← fetch min:typ:max
                                         ← delays and store
                                         ← in array da as follows:
                                         da[0] } typical
                                         da[1] } rise
                                         da[2] } delay
                                         da[3] } typical
                                         da[4] } fall
                                         da[5] } delay
                                         da[6] } typical
                                         da[7] } turn-off
                                         da[8] } delay

            for (i=0; i<9; i+=3)
                da[i] = da[i]*min_scale_factor;
            for (i=1; i<9; i+=3)
                da[i] = da[i]*typ_scale_factor;
            for (i=2; i<9; i+=3)
                da[i] = da[i]*max_scale_factor;
            ← scale
            ← delays
            acc_replace_delays(prim, da); ← replace min:typ:max
                                         ← delays with scaled values
        }
    }
}

```

Figure 19-95—Using acc_replace_delays() in min:typ:max delays mode

19.91 acc_replace_pulsere()

acc_replace_pulsere()			
Synopsis:	Replace existing pulse handling reject_value and e_value for a module path.		
Syntax:	acc_replace_pulsere(path,r1,e1, r2,e2, r3,e3, r4,e4, r5,e5, r6,e6, r7,e7, r8,e8, r9,e9, r10,e10, r11,e11, r12,e12)		
Type		Description	
Returns:	bool	1 if successful; 0 if an error is encountered	
Arguments:	Type	Name	Description
	handle	path	Handle of module path
	double	r1...r12	reject_limit values; the number of arguments is determined by accPathDelayCount
	double	e1...e12	e_limit values; the number of arguments is determined by accPathDelayCount
Related routines:	Use acc_fetch_pulsere() to get current pulse handling values Use acc_append_pulsere() to append existing pulse handling values Use acc_set_pulsere() to set pulse handling values as a percentage of the path delay		

The ACC routine **acc_replace_pulsere()** shall replace existing pulse handling *reject_limit* and *e_limit* values for a module path. The reject_limit and e_limit values are used to control how *pulses* are propagated through module paths.

A pulse is defined as two transitions on the same path that occur in a shorter period of time than the path delay. Pulse control values determine whether a pulse should be rejected, propagated through to the output, or considered an error. The pulse control values consist of a *reject_limit* and an *e_limit* pair of values, where:

- The reject_limit shall set a threshold for determining when to reject a pulse—any pulse less than the reject_limit shall not propagate to the output of the path
- The e_limit shall set a threshold for determining when a pulse is considered to be an error—any pulse less than the e_limit and greater than or equal to the reject_limit shall propagate a logic x to the path output
- A pulse that is greater than or equal to the e_limit shall propagate through to the path output

The example in Table 19-53 illustrates the relationship between the reject_limit and the e_limit.

Table 19-53—Path pulse control example

When	The pulse on a module path output shall be
<i>reject_limit</i> = 10.5 <i>e_limit</i> = 22.6	Rejected if < 10.5 An error if ≥ 10.5 and < 22.6 Passed if ≥ 22.6

The following rules shall apply when specifying pulse handling values:

- a) The value of reject_limit shall be less than or equal to the value of e_limit.
- b) The reject_limit and e_limit shall not be greater than the path delay.

The number of pulse control values that **acc_replace_pulsere()** sets shall be controlled using the ACC routine **acc_configure()** to set the delay count configuration parameter **accPathDelayCount**, as shown in Table 19-54.

Table 19-54—How the value of `accPathDelayCount` affects `acc_append_pulsere()`

When <code>accPathDelayCount</code> is	<code>acc_replace_pulsere()</code> shall write
"1"	One pair of <code>reject_limit</code> and <code>e_limit</code> values: one pair for all transitions, <code>r1</code> and <code>e1</code>
"2"	Two pairs of <code>reject_limit</code> and <code>e_limit</code> values: one pair for rise transitions, <code>r1</code> and <code>e1</code> one pair for fall transitions, <code>r2</code> and <code>e2</code>
"3"	Three pairs of <code>reject_limit</code> and <code>e_limit</code> values: one pair for rise transitions, <code>r1</code> and <code>e1</code> one pair for fall transitions, <code>r2</code> and <code>e2</code> one pair for turn-off transitions, <code>r3</code> and <code>e3</code>
"6" (the default)	Six pairs of <code>reject_limit</code> and <code>e_limit</code> values—a different pair for each possible transition among 0, 1, and Z: one pair for 0->1 transitions, <code>r1</code> and <code>e1</code> one pair for 1->0 transitions, <code>r2</code> and <code>e2</code> one pair for 0->Z transitions, <code>r3</code> and <code>e3</code> one pair for Z->1 transitions, <code>r4</code> and <code>e4</code> one pair for 1->Z transitions, <code>r5</code> and <code>e5</code> one pair for Z->0 transitions, <code>r6</code> and <code>e6</code>
"12"	Twelve pairs of <code>reject_limit</code> and <code>e_limit</code> values—a different pair for each possible transition among 0, 1, X and Z: one pair for 0->1 transitions, <code>r1</code> and <code>e1</code> one pair for 1->0 transitions, <code>r2</code> and <code>e2</code> one pair for 0->Z transitions, <code>r3</code> and <code>e3</code> one pair for Z->1 transitions, <code>r4</code> and <code>e4</code> one pair for 1->Z transitions, <code>r5</code> and <code>e5</code> one pair for Z->0 transitions, <code>r6</code> and <code>e6</code> one pair for 0->X transitions, <code>r7</code> and <code>e7</code> one pair for X->1 transitions, <code>r8</code> and <code>e8</code> one pair for 1->X transitions, <code>r9</code> and <code>e9</code> one pair for X->0 transitions, <code>r10</code> and <code>e10</code> one pair for X->Z transitions, <code>r11</code> and <code>e11</code> one pair for Z->X transitions, <code>r12</code> and <code>e12</code>

The minimum number of pairs of `reject_limit` and `e_limit` arguments to pass to **acc_replace_pulsere()** shall equal the value of **accPathDelayCount**. Any unused `reject_limit` and `e_limit` argument pairs shall be ignored by **acc_replace_pulsere()** and can be dropped from the argument list.

If **accPathDelayCount** is not set explicitly, it shall default to 6, and therefore six pairs of pulse `reject_limit` and `e_limit` arguments have to be passed when **acc_replace_pulsere()** is called. Note that the value assigned to **accPathDelayCount** also affects **acc_append_delays()**, **acc_fetch_delays()**, **acc_replace_delays()**, **acc_append_pulsere()**, and **acc_fetch_pulsere()**.

Pulse control values shall be replaced using the timescale of the module that contains the path.

The example shown in Figure 19-96 uses **acc_replace_pulsere()** to replace rise and fall pulse handling values of paths listed in a file `path.dat`.

```
#include <stdio.h>
#include "acc_user.h"

#define NAME_SIZE 256

replace_halfpulsevals()
{
    FILE      *infile;
    char      mod_name[NAME_SIZE];
    char      pathin_name[NAME_SIZE], pathout_name[NAME_SIZE];
    handle    mod, path;
    double    rise_reject_limit, rise_e_limit, fall_reject_limit, fall_e_limit;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*set accPathDelayCount to return two pairs of pulse handling values;*/
    /* one each for rise and fall transitions*/
    acc_configure(accPathDelayCount, "2");

    /*read all module path specifications from file "path.dat"*/
    infile = fopen("path.dat", "r");
    while(fscanf(infile, "%s %s %s", mod_name, pathin_name, pathout_name) != EOF)
    {
        mod = acc_handle_object(mod_name);
        path = acc_handle_modpath(mod, pathin_name, pathout_name);
        rise_reject_limit = .05;
        if(acc_replace_pulsere(path, &rise_reject_limit, &rise_e_limit,
                                &fall_reject_limit, &fall_e_limit) )
        {
            io_printf("rise reject limit = %lf, rise e limit = %lf\n",
                      rise_reject_limit, rise_e_limit);
            io_printf("fall reject limit = %lf, fall e limit = %lf\n",
                      fall_reject_limit, fall_e_limit);
        }
    }
    acc_close();
}
```

Figure 19-96—Using **acc_replace_pulsere()**

19.92 acc_reset_buffer()

acc_reset_buffer()			
Synopsis:	Reset the string buffer to the beginning.		
Syntax:	acc_reset_buffer()		
Returns:	Type	Description	
	void		
Arguments	Type	Name	Description
	None		
Related routines:	All ACC routines that return a pointer to a character string		

The ACC routine **acc_reset_buffer()** shall reset the string buffer to its beginning. The string buffer shall be used as temporary storage by other ACC routines that return a pointer to a character string. Refer to 18.9 for more information on the character string buffer.

19.93 acc_set_interactive_scope()

acc_set_interactive_scope()			
Synopsis:	Set the interactive scope of a software tool.		
Syntax:	acc_handle_interactive_scope()		
Returns:	Type		Description
	handle	Handle of a Verilog hierarchy scope	
Arguments:	Type	Name	Description
	None		
Related routines:	Use acc_handle_interactive_scope() to get a handle for the current interactive scope		

The ACC routine **acc_set_interactive_scope()** shall set the Verilog HDL design scope where the interactive mode of the software product is operating.

A scope shall be

- A top-level module
- A module instance
- A named begin-end block
- A named fork-join block
- A Verilog HDL task
- A Verilog HDL function

19.94 acc_set_pulsere()

acc_set_pulsere()			
Synopsis:	Set the pulse handling values for a module path as a percentage of the path delay.		
Syntax:	acc_set_pulsere(path_handle, reject_percentage, e_percentage)		
Type		Description	
Returns:	void	No return	
Arguments:	Type	Name	Description
	handle	path_handle	Handle of a module path
	double	reject_percentage	Multiplier of the delay value that forms the upper limit for rejecting a path output pulse
	double	e_percentage	Multiplier of the delay value that forms the upper limit for setting a path output pulse to x .
Related routines:	Use acc_fetch_pulsere() to get current pulse handling values Use acc_append_pulsere() to append existing pulse handling values Use acc_replace_pulsere() to replace existing pulse handling values		

The ACC routine **acc_set_pulsere()** shall set the pulse handling values *reject_percentage* and *e_percentage* for a module path, specified as a percentage multiplier of the path delay.

A pulse is defined as two transitions on the same path that occur in a shorter period of time than the path delay. Pulse control values determine whether a pulse should be rejected, propagated through to the output, or considered an error. The pulse control values consist of a *reject_percentage* and an *e_percentage* pair of values, where

- The *reject_percentage* shall set a threshold for determining when to reject a pulse—any pulse less than the *reject_percentage* shall not propagate to the output of the path
- The *e_percentage* shall set a threshold for determining when a pulse is considered to be an error—any pulse less than the *e_percentage* and greater than or equal to the *reject_percentage* shall propagate a logic **x** to the path output
- A pulse that is greater than or equal to the *e_percentage* shall propagate through to the path output

The example in Table 19-55 illustrates the relationship between the *reject_percentage* and the *e_percentage*.

Table 19-55—Path pulse control example

Given a path with a delay of 5.0	
When	A pulse on a module path output shall be
<i>reject_percentage</i> = 0.5 <i>e_percentage</i> = 1.0	Rejected if < 2.5 (50% of path delay) An error if ≥ 2.5 and < 5.0 (between 50% and 100% of path delay) Passed if ≥ 5.0 (greater than or equal to 100% of path delay)

The following rules shall apply when specifying pulse handling values:

- a) The *reject_percentage* and *e_percentage* shall be greater than or equal to 0.0 and less than or equal to 1.0.

- b) The value of reject_percentage shall be less than or equal to the value of e_percentage.

The example shown in Figure 19-97 uses **acc_set_pulsere()** to set pulse control values for each path in a module such that all pulses between 0 and the path delay generate an X at the path output.

```
#include "acc_user.h"

set_pulse_control_e(module)
    handle    module;
{
    handle    path;

    /*set pulse control values for all paths in the module*/
    path = null;
    while (path = acc_next_modpath(module, path) )
        acc_set_pulsere(path, 0.0, 1.0);
}
```

Figure 19-97—Using acc_set_pulsere()

19.95 acc_set_scope()

acc_set_scope()			
Synopsis:	Set a scope for acc_handle_object() to use when searching in the design hierarchy.		
Syntax:	acc_set_scope(module_handle, module_name)		
Type		Description	
Returns:	char *	Pointer to a character string containing the full hierarchical name of the scope set; <i>null</i> if an error occurred	
Type		Name	Description
Arguments:	handle	module_handle	A handle to a module
Optional	quoted string or char *	module_name	Quoted string or pointer to a character string with the name of a module instance (optional: used when accEnableArgs is set and module_handle is null)
Related routines:	Use acc_handle_object() to get a handle to any named object Use acc_configure(accEnableArgs,"acc_set_scope") to use the module_name argument Use acc_set_interactive_scope() to set the interactive scope		

The ACC routine **acc_set_scope()** shall set the scope and search rules for the routine **acc_handle_object()**. The way that **acc_set_scope()** functions shall be dependent on the setting of configuration parameters as shown in Table 19-56.

Table 19-56—How acc_set_scope() works

If	acc_set_scope() shall
Default mode, or acc_configure(accEnableArgs, "no_acc_set_scope") is called, and <i>module_handle</i> is a valid handle	Set the scope to the level of <i>module_handle</i> in the design hierarchy and ignore the optional <i>module_name</i> argument
Default mode, or acc_configure(accEnableArgs, "no_acc_set_scope") is called, and <i>module_handle</i> is null	Set the scope to the top-level module that appears first in the source description
The routine acc_configure(accEnableArgs, "acc_set_scope") has been called, and <i>module_handle</i> is a null	Set scope to the level of <i>module_name</i> in the design hierarchy
The routine acc_configure(accEnableArgs, "acc_set_scope") has been called, and <i>module_handle</i> is a valid handle	Set scope to the level of <i>module_handle</i> in the design hierarchy and ignore the optional <i>module_name</i> argument
The routine acc_configure(accEnableArgs, "acc_set_scope") has been called, and <i>module_handle</i> and <i>module_name</i> are both null	Set scope to the top-level module that appears first in the source description

To use the optional `module_name` argument, the configuration parameter **accEnableArgs** first has to be set by calling **acc_configure()** as follows:

```
acc_configure(accEnableArgs, "acc_set_scope");
```

If **accEnableArgs** is not set for **acc_set_scope()**, the routine shall ignore its optional argument. When the optional argument is not required for a call to **acc_set_scope()**, the argument can be dropped.

The example shown in Figure 19-98 uses **acc_set_scope()** to set a scope for the ACC routine **acc_handle_object()** to determine if a net is in a module.

```
#include "acc_user.h"

is_net_in_module(module_handle, net_name)
handle    module_handle;
char      *net_name;
{
    handle    net_handle;
    handle    load_handle, load_net_handle;

    /*set scope to module*/
    acc_set_scope(module_handle);

    /*get handle for net*/
    net_handle = acc_handle_object(net_name);

    if (net_handle)
        io_printf("Net %s found in module %s\n",
                  net_name,
                  acc_fetch_fullname(module_handle) );
    else
        io_printf("Net %s not found in module %s\n",
                  net_name,
                  acc_fetch_fullname(module_handle) );
}
```

Figure 19-98—Using acc_set_scope()

19.96 acc_set_value()

acc_set_value()			
Synopsis:	Set and propagate a value on a register, variable, or a sequential UDP; procedurally assign a register or variable; force a register, variable, or net.		
Syntax:	<code>acc_set_value(object_handle, value_p, delay_p)</code>		
Type		Description	
Returns:	int	Zero if no errors; nonzero if an error occurred	
Type		Name	Description
Arguments:	handle	object_handle	Handle to a register or sequential UDP
	p_setval_value	value_p	Pointer to a structure containing value to be set
	p_setval_delay	delay_p	Pointer to a structure containing delay before value is set
Related routines:	Use <code>acc_fetch_value()</code> to retrieve a logic value Use <code>acc_fetch_size()</code> to get the number of bits in a vector		

The ACC routine **acc_set_value()** shall set and propagate a value onto a register, variable, or a sequential UDP. The routine shall also perform procedural assign/deassign or a procedural force/release functions.

The logic value and propagation delay information shall be placed in separate structures. To use **acc_set_value()** to propagate a value, follow these basic steps:

- Allocate memory for the structures `s_setval_value`, `s_setval_delay`, and if using vectors, `s_acc_vecval`.
- Set the appropriate fields in each structure to the desired values.
- Call **acc_set_value()** with an object handle and pointers to the `s_setval_value` and `s_setval_delay` structures.

The structure `s_setval_value` shall contain the value to be written. A value can be entered into this structure as a string, scalar, integer, real, or as an *aval/bval* pair. The `s_setval_value` structure is defined in `acc_user.h` and listed in Figure 19-99 (note that this structure is also used with **acc_fetch_value()** routine).

The *format* field in the `s_setval_value` structure shall indicate the value type. The format shall be a predefined integer constant, listed in Table 19-57.

The *value* union in the `s_setval_value` structure shall be the value to be written. The value is placed in the appropriate field within the union for the format selected.

```
typedef struct t_setval_value
{
    int format;
    union
    {
        char *str;
        int scalar;
        int integer;
        double real;
        p_acc_vecval vector;
    } value;
} s_setval_value, *p_setval_value, s_acc_value, *p_acc_value;
```

Figure 19-99—The s_setval_value structure used by acc_set_value()**Table 19-57—Predefined constants for the format field of s_setval_value**

Value format	Definition
accScalarVal	One of: acc0 , acc1 , accZ , accX
accVectorVal	<i>aval</i> and <i>bval</i> bit groups, with each group being an integer quantity
accIntVal	An integer quantity
accRealVal	A real-valued quantity
accStringVal	For integers and appropriately sized registers, any ASCII string; for real-valued objects, any string that represents a real number
accBinStrVal	A base 2-bit representation as a string
accOctStrVal	A base 8-bit representation as a string
accDecStrVal	A base 10-bit representation as a string
accHexStrVal	A base 16-bit representation as a string

When the *format* field of the **s_acc_vecval** structure is set to **accVectorVal**, the *value* union field used shall be *vector*. The *vector* field is set to a pointer or an array of **s_acc_vecval** structures that contain *aval/bval* pairs for each bit of the vector. The **s_acc_vecval** structure is listed in Figure 19-100.

```
typedef struct t_acc_vecval
{
    int aval;
    int bval;
} s_acc_vecval, *p_acc_vecval;
```

Figure 19-100—s_acc_vecval structure

The array of `s_acc_vecval` structures shall contain a record for every 32 bits of the vector, plus a record for any remaining bits. Memory has to be allocated by the user for array of `s_acc_vecval` structures. If a vector has N bits, the size of the array shall be $((N-1)/32)+1$ `s_acc_vecval` records. The routine `acc_fetch_size()` can be used to determine the value of N .

The lsb of the vector shall be represented by the lsb of the first record of `s_acc_vecval` array. The 33rd bit of the vector shall be represented by the lsb of the second record of the array, and so on. Each bit of the vector shall be encoded as an *aval/bval* pair. The encoding for each bit is shown in Table 19-36.

Table 19-58—Encoding of bits in the `s_acc_vecval` structure

aval	bval	Value
0	0	0
1	0	1
0	1	Z
1	1	X

The structure `s_setval_delay` shall control how values are to be propagated into the Verilog HDL data structure. The structure is defined in `acc_user.h` and is listed in Figure 19-101.

The *time* field in the `s_setval_delay` structure shall indicate the delay that shall take place before a register value assignment. The time field shall be of type `s_acc_time` structure, as shown in Figure 19-101.

The *model* field in the `s_setval_delay` structure shall determine how the delay shall be applied, and how other simulation events scheduled for the same object shall be affected. The delay *model* shall be specified using predefined integer constants, listed in Table 19-59. Note that the constants listed in Table 19-59 can also be used.

```
typedef struct t_setval_delay
{
    s_acc_time time;
    int model;
} s_setval_delay, *p_setval_delay;
```

Figure 19-101—The `s_setval_delay` structure for `acc_set_value()`

Table 19-59—Predefined delay constants for the model field of s_setval_delay

Integer constant	Delay model	Description
accNoDelay	No delay	Sets a register or sequential UDP to the indicated value with no delay; other events scheduled for the object are not affected
accInertialDelay	Inertial delay	Sets a register to the indicated value after the specified delay; all scheduled events on the object are removed before this event is scheduled
accTransportDelay	Modified transport delay	Sets a register to the indicated value after the specified delay; all scheduled events on the object for times later than this event are removed
accPureTransportDelay	Pure transport delay	Sets a register to the indicated value after the specified delay; no scheduled events on the object are removed

When setting the value of a sequential UDP, the *model* field shall be **accNoDelay**, and the new value shall be assigned with no delay even if the UDP instance has a delay.

The **s_acc_time** structure shall hold the delay value that shall be used by **acc_set_value()**. The **s_acc_time** structure is defined in **acc_user.h** and is listed in Figure 19-101.

The *type* field in the **s_acc_time** structure shall indicate the data type of the delay that shall be stored in the structure. The type shall be specified using predefined integer constants, listed in Table 19-60.

The *low* field shall be an integer that represents the lower 32 bits of a 64-bit delay value.

The *high* field shall be an integer that represents the upper 32 bits of a 64-bit delay value.

The *real* field shall be a double that represents the delay as a real number value.

```
typedef struct t_acc_time
{
    int type;
    int low,
        high;
    double real;
} s_acc_time, *p_acc_time;
```

Figure 19-102—The s_acc_time structure for acc_set_value()

Table 19-60—Predefined time constants for the type field of `s_acc_time`

Integer constant	Description
accTime	Delay is a 64-bit integer; time shall be scaled to the timescale in effect for the module containing the object.
accSimTime	Delay is a 64-bit integer; time shall be scaled to the time units being used by the simulator
accRealTime	Delay is a real number; time shall be scaled to the timescale in effect for the module containing the object.

The routine **acc_set_value()** shall be used to perform a procedural continuous assignment of a value to a register or to deassign the register. This shall be the same functionality as the procedural **assign** and **deassign** keywords in the Verilog HDL.

The routine **acc_set_value()** shall also be used to perform a procedural force of value onto a register or net, or to release the register or net. This shall be the same functionality as the procedural **force** and **release** keywords in the Verilog HDL.

When an object is deassigned or released using **acc_set_value()**, the current value of the object shall be returned to the `s_setval_value` structure.

To assign, deassign, force, or release an object using **acc_set_value()**, the `s_setval_value` and `s_setval_delay` structures shall be allocated and the fields shall be set to the appropriate values. For the *model* field of the `s_setval_delay` structure, one of the predefined constants listed in Table 19-61 shall be used.

Table 19-61—Predefined assign/force constants for the model field of `s_setval_delay`

Integer constant	Description
accAssignFlag	Continuously assigns a register to the indicated value with no delay; other events scheduled for the object are overridden. Same functionality as the Verilog HDL procedural assign keyword.
accDeassignFlag	Deassigns a continuously assigned register; other events scheduled for the object are no longer overridden. Same functionality as the Verilog HDL procedural deassign keyword.
accForceFlag	Forces a value onto a register or net; other events scheduled for the object are overridden. Same functionality as the Verilog HDL procedural force keyword.
accReleaseFlag	Releases a forced register or net; other events scheduled for the object are no longer overridden, and nets immediately return to the current driven value. Same functionality as the Verilog HDL procedural release keyword.

The example shown in Figure 19-103 uses **acc_set_value()** to set and propagate a value onto a register. This example assumes the application is linked to a user-defined system task (using the PLI interface mechanism) called `$my_set_value()`, which has the following usage for a four bit register, `r1`:

```
$my_set_value(r1, "x011", 2.4);
```

```
int my_set_value()
{
    static s_setval_delay delay_s = {{accRealTime},accInertialDelay};

    static s_setval_value value_s = {accBinStrVal};

    handle reg = acc_handle_tfarg(1);

    value_s.value.str = acc_fetch_tfarg_str(2);

    delay_s.time.real= acc_fetch_tfarg(3);

    acc_set_value(reg, &value_s, &delay_s);
}
```

Figure 19-103—Using acc_set_value()

19.97 acc_vcl_add()

acc_vcl_add()			
Synopsis:	Set a callback to a consumer routine with value change information whenever an object changes value.		
Syntax:	acc_vcl_add(object_handle, consumer_routine, user_data, vcl_flag)		
Type		Description	
Returns:	void		
Arguments:	Type	Name	Description
	handle	object_handle	Handle to an object to be monitored (such as a register or net)
	C routine pointer	consumer_routine	Unquoted name of the C routine to be called when the object changes value
	char *	user_data	User-defined data that is passed back to the consumer routine when the object changes value
	int	vcl_flag	Predefined integer constant that selects the type of change information reported to the consumer routine
Related routines:	Use acc_vcl_delete() to remove a VCL callback monitor		

The ACC routine **acc_vcl_add()** shall set up a callback monitor on an object that shall call a user-defined consumer routine when the object changes value. The consumer routine shall be passed logic value information or logic value and strength information about the object.

The **acc_vcl_add()** routine requires four arguments, as described in the following paragraphs.

The *object_handle* argument is a handle to the object to be monitored by an application. The VCL shall monitor value changes for the following objects:

- Scalar, vector, and bit-selects of registers
- Scalar, vector, and bit-selects of nets
- Integer, real and time variables
- Module ports
- Primitive output or inout terminals
- Events

The *object_handle* passed to **acc_vcl_add()** is not returned when the consumer routine is called. However, the handle can be passed using the *user_data* argument.

The *consumer_routine* argument is a pointer to a C application. This application shall be called whenever the object changes value. When a value change callback occurs, the *consumer_routine* shall be passed the *user_data* argument and a pointer to a **vc_record** structure, which shall contain information about the change.

Refer to 18.10 for a full description of consumer routines and the **vc_record** structure.

The *user_data* argument is user-defined data, such as the object name, the object handle, the object value, or a pointer to a data structure. The value of the *user_data* argument shall be passed to the consumer routine each time a callback occurs. Note that the *user_data* argument is defined as character string pointer, and therefore any other type should be cast to a **char***.

The *vcl_flag* argument shall set the type of information the callback mechanism shall report. There are two types of flags, as shown in Table 19-62.

Table 19-62—vcl_flag constants used in acc_vcl_add()

vcl_flag	What it does
vcl_verilog_logic	Indicates the VCL callback mechanism shall report information on logic value changes
vcl_verilog_strength	Indicates the VCL callback mechanism shall report information on logic value and strength changes

If an application calls **acc_vcl_add()** with the same arguments more than once, the VCL callback mechanism shall only call the consumer routine once when the object changes value. If any of the VCL arguments, including the *user_data*, are different, the VCL callback mechanism shall call the consumer routine multiple times, once for each unique **acc_vcl_add()**.

If multiple PLI applications monitor the same object at the same time, each application shall receive a separate call whenever that object changes value. Typically, multiple applications have distinct consumer routines and *user_data* pointers. These different consumer routines allow the value change information to be processed in different ways.

Refer to 18.10 for an example of using **acc_vcl_add()**.

19.98 acc_vcl_delete()

acc_vcl_delete()			
Synopsis:	Removes a VCL callback monitor.		
Syntax:	<code>acc_vcl_delete(object_handle, consumer_routine, user_data, vcl_flag)</code>		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	handle	object_handle	Handle to the object to be monitored specified in the call to <code>acc_vcl_add()</code>
	C routine pointer	consumer_routine	Unquoted name of the C routine specified in the call to <code>acc_vcl_add()</code>
	char *	user_data	User-defined data specified in the call to <code>acc_vcl_add()</code>
	int	vcl_flag	Predefined integer constant; vcl_verilog
Related routines:	Use <code>acc_vcl_add()</code> to place a VCL callback monitor on an object		

The ACC routine **acc_vcl_delete()** shall remove a VCL callback monitor previously requested with a call to **acc_vcl_add()**. The **acc_vcl_delete()** routine requires four arguments, as described in the following paragraphs. When multiple PLI applications are monitoring the same object, **acc_vcl_delete()** shall stop monitoring the object only for the application associated with a specific **acc_vcl_add()** call.

The *object_handle* argument is a handle to the object for which the VCL callback monitor is to be removed. This has to be a handle to the same object that was used when **acc_vcl_add()** was called.

The *consumer_routine* argument is the unquoted name of the C application called by the VCL callback monitor. This has to be the same C application that was specified when **acc_vcl_add()** was called.

The *user_data* argument is user-defined data that is passed to the consumer routine each time the object changes value. This has to be the same value that was specified when **acc_vcl_add()** was called.

The *vcl_flag* argument is a predefined integer constant and has to be **vcl_verilog**. This constant shall be used in place of the *vcl_flag* values used with **acc_vcl_add()**.

Refer to 18.10 for an example of using **acc_vcl_delete()**.

19.99 acc_version()

acc_version()		
Synopsis:	Get a pointer to a character string that indicates version number of the ACC routine software.	
Syntax:	acc_version()	
Returns:	Type	Description
	char *	Character string pointer
Arguments:	Type	Name
	None	
Related routines:	Use acc_product_version() to get the version of the software product in use Use acc_product_type() to get the type of software product in use	

The ACC routine **acc_version()** shall return a pointer to a character string that indicates the version of the ACC routines used in the software product that called the PLI application. The return value for this routine is placed in the ACC internal string buffer. See 18.9 for explanation of strings in ACC routines.

The character string shall be in the following format:

Access routines Version <version_number>

For example, if the software product is using the IEEE Std 1364-1995 PLI version of ACC routines, **acc_version()** would return a pointer to the following string:

"Access routines Version IEEE 1364 PLI"

NOTE—The string returned by **acc_version()** shall be defined by the software product vendor.

The example shown in Figure 19-104 uses **acc_version()** to identify the version of ACC routines linked to the application.

```
#include "acc_user.h"

show_versions()
{
    /*initialize environment for ACC routines*/
    acc_initialize();
    acc_configure(accDevelopmentVersion, "IEEE 1364 PLI");

    /*show version of ACC routines*/
    /* and version of Verilog that is linked to ACC routines*/
    io_printf("Running %s with %s\n",acc_version(),acc_product_version() );
    acc_close();
}
```

Figure 19-104—Using acc_version()

Section 20

Using TF routines

This section provides an overview of the types of operations that are done with the PLI task/function (TF) routines. Detailed descriptions of the routines are provided in the next section.

20.1 TF routine definition

The PLI TF routines, sometimes referred to as *utility routines*, provide a mechanism to manipulate the arguments of user-defined system tasks and functions and to synchronize interaction between a task and the simulator. Appropriate applications include stimulus generation, error checking, and interfaces to C models.

20.2 TF routine parameters

In the context of TF routines, the term *parameter* shall refer to the arguments of user-defined system tasks and functions—it does *not* refer to Verilog HDL parameters.

The number of parameters passed to a system task shall be returned by **tf_nump()**. A type for each parameter shall be returned by **tf_typep()** and is primarily used to determine if a parameter is writable.

A parameter shall be considered *read-only* if, in the Verilog HDL source description, the parameter cannot be used on the left-hand side of a procedural assignment statement. Signals declared as one of the net data types or the event data type, or bit-selects, part-selects, or concatenations of net data types, shall be read-only. A module instance name or a primitive instance name shall also be read-only.

Parameters shall be considered *writable* from the PLI if the parameters can be used on the left-hand side of procedural assignment in the Verilog HDL source description. Signals declared as reg, integer, time, or real shall be writable, as well as bit-selects, part-selects, and concatenations of these data types.

20.3 Reading and writing parameter values

User-defined system task and function parameter values can be determined and altered in a number of ways with the TF routines, depending on factors such as value type, data size, and desired format.

20.3.1 Reading and writing 2-state parameter values

To access the 2-state (logic 0 and 1) value of a parameter of size less than or equal to 32 bits, the routine **tf_getp()** can be used. To set the 2-state value of a parameter of size less than or equal to 32 bits, **tf_putp()** can be used. If the parameter is 33–64 bits, **tf_getlongp()** and **tf_putlongp()** can be used. For parameters of type real, **tf_getrealp()** and **tf_putrealp()** can be used.

20.3.2 Reading and writing 4-state values

If 4-states (logic 0, 1, X, and Z) are required and a string representation of the value is appropriate, **tf_strgetp()** can be used to access the value. The routines **tf_strdelpuip()**, **tf_strlongdelpuip()**, and **tf_strrealdelpuip()** can be used to write 4-state values to writable parameters. For applications with a high frequency of PLI calls, the overhead of these string-based routines can be excessive. The following paragraph describes an alternative.

4-state values can also be accessed with the routine **tf_exprinfo()**. This routine shall create a persistent structure that contains the 4-state value of a parameter encoded in a **s_vecval** structure. After **tf_exprinfo()** has been called once for a parameter, the pointer to the **s_vecval** structure can be saved. The parameter value can be changed using that structure along with routines **tf_propagatep()** to send the value in the structure into a simulation and **tf_evaluatep()** to update the value in the structure to the current simulation value.

20.3.3 Reading and writing strength values

Strength values on scalar net parameters can be accessed with the routine **tf_nodeinfo()**.

20.3.4 Reading and writing to memories

Memory array values can be accessed with the routine **tf_nodeinfo()**. This routine returns pointer to a **memval** structure that represents the array in the Verilog HDL software product. Setting a value in the **memval** structure shall make it available for the software tool access, but this does not automatically cause the value to be propagated to any right-hand-side memory references.

20.3.5 Reading and writing string values

The routine **tf_getcstringp()** shall return the string representation of a string constant or a vector parameter. If the parameter is not a string value or vector, the results of **tf_getcstringp()** are not predictable. There is no direct method to write string values using TF routines, but it can be accomplished by writing character values to 8-bit register elements in a vector register using the **tf_exprinfo()** value structure.

20.3.6 Writing return values of user-defined functions

2-state values can be set as the return value of a user-defined function using **tf_putp()** and **tf_putrealp()** with a parameter value of 0.

20.3.7 Writing the correct C data types

It is important to ensure that the data type of the argument to any of the **tf_put** routines is consistent with the data type required by the routine and specified parameter. There is no inherent data type checking in the C language.

The following examples illustrate what cautions should be taken.

If the second parameter of a system task/function instance is of type **tf_readwritereal**, meaning the parameter is declared as a real register in the Verilog HDL source description, the following **tf_put** routines shall produce valid results:

```
int i = 5;
tf_putp(2, i); /* write an integer value to 2nd parameter */
```

This example sets the second task/function parameter to 5.0—assigning an integer value to a real variable is legal in the Verilog HDL.

```
double d = 5.7;
tf_putrealp(2, d); /* write a real value to 2nd parameter */
```

This example sets the second task/function parameter to 5.7.

The following routines, however, shall produce invalid results for the following reasons:

```
int i = 5;
tf_putrealp(2, i); /* invalid result */
```

The statement `int i = 5` passes a 32-bit integer to **tf_putrealp()**, which expects a 64-bit double value type. Since there is no data type checking, **tf_putrealp()** shall read 32 bits of undefined data and try to use it as if it were valid data. The result is unpredictable.

```
float f = 5;
tf_putrealp(2, f); /* invalid result */
```

The float statement passes a 32-bit float to **tf_putrealp()**, which is expecting a 64-bit double value type. The result is unpredictable.

```
double d = 5.7;
tf_putp(2, d); /* invalid result */
```

The **tf_putp()** routine shall take only the lower 32 bits of the 64-bit double passed to it by the statement `double d = 5.7`.

20.4 Value change detection

Value changes on parameters can be detected by enabling asynchronous callbacks with **tf_asynchon()**. The callbacks can be disabled with **tf_asynchoff()**. When parameter change callbacks are enabled with **tf_asynchon()**, whenever a parameter changes value, the misc tf application associated with the user-defined system task/function shall be called back with three integer arguments: *data*, *reason*, and *paramvc*. Argument *reason* shall be **reason_paramvc** if the parameter changed value, or **reason_paramdrc** if a driver of the parameter changed value but the parameter did not change. The value change can be examined immediately, or a second callback can be requested later in the same time step (as described in 20.6). By setting a second callback at the end of the time step, an application can process all parameter value changes within in a time step at once. The routines **tf_copypvc_flag()**, **tf_movepvc_flag()**, **tf_testpvc_flag()**, and **tf_getpchange()** can be used to determine all the parameters that changed in a time step.

20.5 Simulation time

TF routines are provided to read simulation time and to scale delays to simulation time scales.

The routines **tf_gettime()** and **tf_getlongtime()** shall return the current simulation time in unsigned format. These times shall be scaled to the timescale of the module where the system task or function is invoked. The routine **tf_str_gettime()** shall return unscaled simulation time in a string format.

PLI TF routines that involve time shall automatically scale delay values to the timescale of the module containing the instance of the user-defined task or function.

The routines **tf_gettimeunit()** and **tf_gettimeprecision()** can be used to obtain the timescale unit and precision of a module. These routines can also be used to obtain the internal simulation time unit, which is the smallest precision of all modules within a simulation. The routines **tf_scale_delay()**, **tf_scale_longdelay()**, **tf_scale_realdelay()**, **tf_unscale_delay()**, **tf_unscale_longdelay()**, and **tf_unscale_realdelay()** can be used to convert between scaled delays and internal simulation time.

20.6 Simulation synchronization

There are TF routines that allow synchronized calling of the misc tf application associated with a user-defined system task or function. The misc tf application can be called at the end of the current time step or at some future time step.

The routines **tf_synchronize()** and **tf_rosynchronize()** shall cause the misctf application associated with a user-defined system task to be called back in the current simulation time step.

The **tf_synchronize()** routine shall place the callback at the end of the inactive event queue for the current time step. The misctf application shall be called with **reason_synch**. It is possible for subsequent events to be added to the current time step after the **tf_synchronize()** callback (for this reason, when the callback occurs, the next scheduled time step cannot be determined). The PLI application can propagate new values from the misctf call in **reason_synch** mode.

The **tf_rosynchronize()** callback shall occur after all active, inactive, and nonblocking assign events for a time step have been processed. The misctf application shall be called with **reason_rosynch**. With **reason_rosynch**, it is possible to determine the time of the next scheduled time step using **tf_getnextlongtime()**. Values cannot be written to parameters during a **reason_rosynch** callback (the 'ro' indicates read-only).

The routine **tf_setdelay()** and its variations shall schedule the misctf application to be called back at a specified time with reason argument **reason_reactivate**. The routine **tf_clearalldelays()** shall remove any previously scheduled callbacks of this type.

20.7 Instances of user-defined task or functions

The routine **tf_getinstance()** shall return a unique identifier for each instance of a user-defined system task or function in the Verilog HDL source description. This value can then be used as the *instance_p* argument to all the *tf_i** routines so that the parameters of one instance can be manipulated from another task or function instance.

20.8 Module and scope instance names

The full hierarchical path name of the module that contains an instance shall be returned by the routine **tf_mipname()**. The full name of the containing scope, which can be an Verilog HDL task or function, a named block, or a module instance, shall be returned by **tf_spname()**.

20.9 Saving information from one system TF call to the next

The PLI TF routines **tf_setworkarea()** and **tf_getworkarea()** shall provide a special storage *work area* that can be used for:

- Saving data during one call to a PLI application that can be retrieved during a subsequent call to the application.
- Passing data from one type of PLI application to another, such as from a checktf application to a calltf application.

20.10 Displaying output messages

The routine **io_printf()** can be used in place of the C **printf()** statement. This routine has essentially the same syntax and semantics as **printf()**, but it displays the output message to both the standard output of the software product and to the log file of the software product.

The routine **io_mcdprintf()** is also similar to the C **printf()**, but permits writing information to files that were opened within the Verilog HDL source description using the **\$fopen()** built-in system function.

The routines **tf_warning()**, **tf_error()**, **tf_message()**, and **tf_text()** can be used to display warning and error messages that are automatically formatted to a similar format as the warning and error messages for the software product. The routines **tf_error()** and **tf_message()** shall also provide control for aborting the software product execution when an error is detected.

20.11 Stopping and finishing

The routines **tf_dostop()** and **tf_dofinish()** are the PLI equivalents to the built-in system tasks **\$stop()** and **\$finish()**.

Section 21

TF routine definitions

This section defines the PLI TF routines, explaining their function, syntax, and usage. The routines are listed in alphabetical order. See Section 19 for conventions that are used in the definitions of the PLI routines.

21.1 io_mcdprintf()

io_mcdprintf()			
Synopsis:	Write a formatted message to one or more files.		
Syntax:	io_mcdprintf(mcd, format, arg1,...arg12)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	int	mcd	An integer multichannel descriptor value representing one or more open files
	quoted string or char *	format	A quoted character string or pointer to a character string that controls the message to be written
		arg1...arg12	1 to 12 arguments used in the format control string; the type of each argument should be consistent with how it is used in the format string
Related routines:	Use io_printf() to write messages to standard output and to a Verilog product log file		

The TF routine **io_mcdprintf()** shall write a formatted message to one or more open files, pointed to by the mcd (multichannel descriptor). The functionality is similar to the C fprintf() function.

The *mcd* value uses the file descriptors created by the **\$fopen** built-in Verilog HDL system task. The value of the mcd generated by one or more calls to **\$fopen** shall be passed to the PLI application using a task/function argument.

The *format* control string uses the same formatting controls as the C printf() function (for example, %d).

The maximum number of arguments that can be used in the format control string is 12.

21.2 io_printf()

io_printf()			
Synopsis:	Print a formatted message to the standard output of a product and to the log file of a product.		
Syntax:	io_printf(format, arg1,...arg12)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	quoted string or char *	format	A quoted character string or pointer to a character string that controls the message to be written
		arg1...arg12	1 to 12 arguments used in the format control string; the type of each argument should be consistent with how it is used in the format string
Related routines:	Use io_mcdprintf() to write a formatted message to one or more open files Use tf_message(), tf_error(), or tf_warning() to write error or warning messages		

The TF routine **io_printf()** shall write a formatted message as text output. The functionality is similar to the C printf() function. However, **io_printf()** differs from printf() because it ensures the message is written to both the standard output text window of the product and the output log file of the product.

The *format* control string uses the same formatting controls as the C printf() function (for example, %d).

The maximum number of arguments that can be used in the format control string is 12.

21.3 mc_scan_plusargs()

mc_scan_plusargs()			
Synopsis:	Scan software product invocation command line for plus (+) options.		
Syntax:	mc_scan_plusargs(startarg)		
Returns:	Type	Description	
	char *	Pointer to a string with the result of the search	
Arguments:	Type	Name	Description
	quoted string or char *	startarg	A quoted string or pointer to a character string with the first part of the invocation option to search for

The TF routine **mc_scan_plusargs()** shall scan all software product invocation command options and match a given string to a plus argument. The match is case sensitive.

The routine **mc_scan_plusargs()** shall

- Return `null` if *startarg* is not found
- Return the remaining part of command argument if *startarg* is found (e.g., if the invocation option string is `"+siz64"`, and *startarg* is `"siz"`, then `"64"` is returned)
- Return a pointer to a C string with a `null` terminator if there is no remaining part of a found plus argument

21.4 tf_add_long()

tf_add_long()			
Synopsis:	Add two 64-bit integers.		
Syntax:	tf_add_long(&aof_low1, &aof_high1, low2, high2)		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	int *	aof_low1	Pointer to least significant 32 bits of first operand
	int *	aof_high1	Pointer to most significant 32 bits of first operand
	int	low2	Least significant 32 bits of second operand
	int	high2	Most significant 32 bits of second operand
Related routines:	Use tf_subtract_long() to subtract two 64-bit integers Use tf_multiply_long() to multiply two 64-bit integers Use tf_divide_long() to divide two 64-bit integers Use tf_compare_long() to compare two 64-bit integers		

The TF routine **tf_add_long()** shall add two 64-bit values. After calling **tf_add_long()**, the variables used to pass the first operand shall contain the results of the addition. Figure 21-1 shows the high and low 32 bits of two long integers and how **tf_add_long()** shall add them.

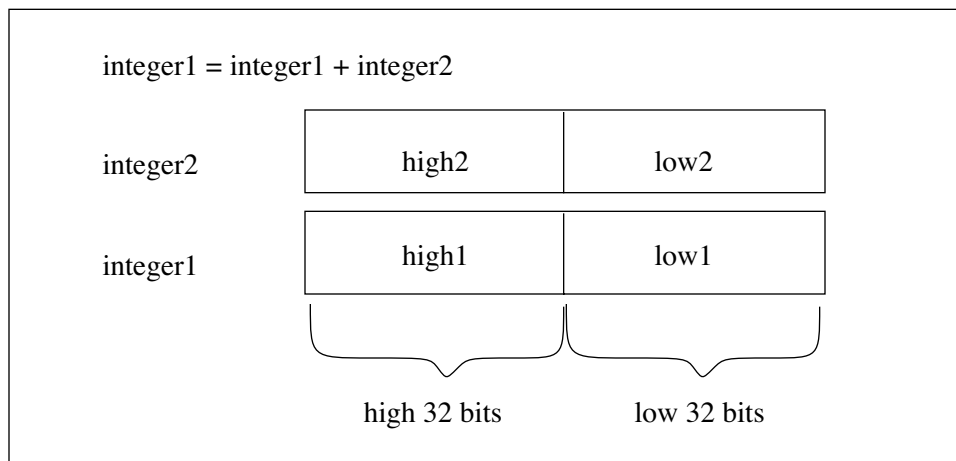


Figure 21-1 — Adding with tf_add_long()

21.5 **tf_asynchoff()**, **tf_iasynchoff()**

tf_asynchoff(), tf_iasynchoff()			
Synopsis:	Disable asynchronous calling of the misctf application.		
Syntax:	<pre>tf_asynchoff() tf_iasynchoff(instance_p)</pre>		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_asyncchon() or tf_iasynchon() to enable asynchronous calling of the misctf application Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_asynchoff()** and **tf_iasynchoff()** shall disable further calling of the misctf application for **reason_paramvc** for the current instance or a specific instance of a user-defined system task or function.

Asynchronous calling is first enabled by the routines **tf_asyncchon()** or **tf_iasynchon()**.

21.6 tf_asynchon(), tf_iasynchon()

tf_asynchon(), tf_iasynchon()			
Synopsis:	Enable asynchronous calling of the misctf application for parameter value changes.		
Syntax:	<pre>tf_asynchon() tf_iasynchon(instance_p)</pre>		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_asynchoff() or tf_iasynchoff() to disable asynchronous calling of the misctf application Use tf_getpchange() or tf_igetpchange() to get the index number of the parameter that changed Use tf_copypvc_flag() or tf_icopypvc_flag() to copy pvc flags Use tf_movepvc_flag() or tf_imovepvc_flag() to move a pvc flag to the saved pvc flag Use tf_testpvc_flag() or tf_itestpvc_flag() to get the value of a saved pvc flag Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_asynchon()** and **tf_iasynchon()** shall enable a misctf user application to be called asynchronously whenever a parameter value changes in the current instance or in a specific instance of a user-defined system task or function. After enabling, the routine specified by misctf in the PLI interface mechanism shall be called with a reason of **reason_paramvc** each time any task/function parameter changes value, or **reason_paramdrc** each time any task/function parameter changes strength. The parameter index number of the parameter that changed value is passed to the misctf application as a third C argument, **paramvc**.

The value change can be examined immediately, or a second callback can be requested later in the same time step (as described in 20.6). By setting a second callback at the end of the time step, an application can process all parameter value changes within in a time step at once. The routines **tf_copypvc_flag()**, **tf_movepvc_flag()**, **tf_testpvc_flag()**, and **tf_getpchange()** can be used to determine all the parameters that changed in a time step.

Task/function parameter index numbering shall proceed from left to right, and the leftmost parameter shall be number 1.

21.7 **tf_clearalldelays()**, **tf_iclearalldelays()**

tf_clearalldelays(), tf_iclearalldelays()			
Synopsis:	Clear all scheduled reactivations by <code>tf_setdelay()</code> or <code>tf_isetdelay()</code> .		
Syntax:	<pre>tf_clearalldelays() tf_iclearalldelays(instance_p)</pre>		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use <code>tf_setdelay()</code> or <code>tf_isetdelay()</code> to schedule a reactivation Use <code>tf_getinstance()</code> to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_clearalldelays()** and **tf_iclearalldelays()** shall clear all reactivation delays, which shall remove the effect of all previous **tf_setdelay()** or **tf_isetdelay()** calls for the current instance or specific instance of a user-defined system task or function.

21.8 tf_compare_long()

tf_compare_long()			
Synopsis:	Compare two 64-bit integer values.		
Syntax:	tf_compare_long(low1, high1, low2, high2)		
Returns:	Type		Description
	int		An integer flag indicating the result of the comparison
Arguments:	Type	Name	Description
	int	low1	Least significant 32 bits of first operand
	int	high1	Most significant 32 bits of first operand
	int	low2	Least significant 32 bits of second operand
	int	high2	Most significant 32 bits of second operand
Related routines:	Use tf_add_long() to add two 64-bit integers Use tf_subtract_long() to subtract two 64-bit integers Use tf_multiply_long() to multiply two 64-bit integers Use tf_divide_long() to divide two 64-bit integers		

The TF routine **tf_compare_long()** shall compare two 64-bit integers and return one of the values given in Table 21-1.

Table 21-1—Return values for tf_compare_long()

When	tf_compare_long() shall return
operand1 < operand2	-1
operand1 = operand2	0
operand1 > operand 2	1

21.9 tf_copypvc_flag(), tf_icopypvc_flag()

tf_copypvc_flag(), tf_icopypvc_flag()			
Synopsis:	Copy parameter value change flags.		
Syntax:	<pre>tf_copypvc_flag(nparam) tf_icopypvc_flag(nparam, instance_p)</pre>		
Type		Description	
Returns:	int	The value of the pvc flag	
Type		Name	Description
Arguments:	int	nparam	Index number of the user-defined system task or function parameter, or -1
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_asynchon() or tf_iasynchon() to enable pvc flags Use tf_getpchange() or tf_igetpchange() to get the index number of the parameter that changed Use tf_movepvc_flag() or tf_imovepvc_flag() to move a pvc flag to the saved pvc flag Use tf_testpvc_flag() or tf_itestpvc_flag() to get the value of a saved pvc flag Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_copypvc_flag()** and **tf_icopypvc_flag()** shall copy the current pvc flag to the saved pvc flag and return the value of the flag that was copied. The argument *nparam* is the index number of a parameter in the current instance or a specific instance of a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1. If *nparam* is **-1**, then all parameter pvc flags shall be copied and the logical OR of all saved flags returned.

Parameter Value Change (pvc) flags shall be used to indicate whether a particular user-defined system task or function parameter has changed value. Each parameter shall have two pvc flags: a current pvc flag, which shall be set by a software product when the change occurs, and a saved pvc flag, which shall be controlled by the user.

NOTE—PVC flags shall not be set by the software product until **tf_asynchon()** or **tf_iasynchon()** has been called.

21.10 tf_divide_long()

tf_divide_long()			
Synopsis:	Divide two 64-bit integers.		
Syntax:	tf_divide_long(&aof_low1, &aof_high1, low2, high2)		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	int *	aof_low1	Pointer to least significant 32 bits of first operand
	int *	aof_high1	Pointer to most significant 32 bits of first operand
	int	low2	Least significant 32 bits of second operand
	int	high2	Most significant 32 bits of second operand
Related routines:	Use tf_add_long() to add two 64-bit integers Use tf_subtract_long() to subtract two 64-bit integers Use tf_multiply_long() to multiply two 64-bit integers Use tf_compare_long() to compare two 64-bit integers		

The TF routine **tf_divide_long()** shall divide two 64-bit values. After calling **tf_divide_long()**, the variables used to pass the first operand shall contain the result of the division.

The operands shall be assumed to be in two's complement form. Figure 21-2 shows the high and low 32 bits of two long integers and how **tf_divide_long()** shall divide them.

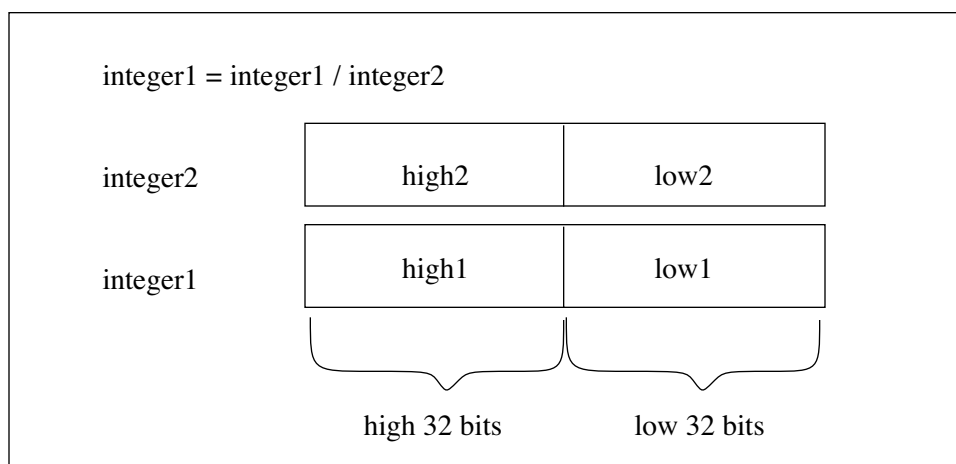


Figure 21-2—Dividing with tf_divide_long()

21.11 tf_dofinish()

tf_dofinish()			
Synopsis:	Exit software product execution.		
Syntax:	tf_dofinish()		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	None		
Related routines:	Use tf_dostop() to cause a product to enter interactive mode		

The TF routine **tf_dofinish()** shall finish the software product execution the same as if a **\$finish()** built-in system task had been executed in the Verilog HDL source description.

21.12 tf_dostop()

tf_dostop()			
Synopsis:	Cause software product to enter interactive mode.		
Syntax:	tf_dostop()		
Returns:	Type		Description
	void		
Arguments:	Type	Name	Description
	None		
Related routines:	Use tf_dofinish() exit software product execution		

The TF routine **tf_dostop()** shall cause a software product to enter into its interactive mode as if a **\$stop()** built-in system task had been executed in the Verilog HDL source description.

21.13 tf_error()

tf_error()			
Synopsis:	Report an error message.		
Syntax:	tf_error(format, arg1,...arg5)		
Returns:	Type		Description
	void		
Arguments:	Type	Name	Description
	quoted string or char *	format	A quoted character string or pointer to a character string that controls the message to be written
		arg1...arg5	One to five arguments used in the format control string; the type of each argument should be consistent with how it is used in the format string
Related routines:	Use tf_message() to write error messages with additional format control Use tf_warning() to write a warning message Use io_printf() or io_mcdprintf() to write a formatted message		

The TF routine **tf_error()** shall provide an error reporting mechanism compatible with error messages generated by the software product.

- The *format* control string uses the same formatting controls as the C `printf()` function (for example, %d).
- The maximum number of arguments that can be used in the format control string is five.
- The location information (file name and line number) of the current instance of the user-defined system task or function is appended to the message using a format compatible with error messages generated by the software product.
- The *message* is written to both the standard output text window and the output log file of the product.

If **tf_error()** is called by the checktf application associated with the user-defined system task or function, the following rules shall apply:

- If the checktf application is called when the Verilog HDL source code was being parsed or compiled, parsing or compilation shall be aborted after the error is reported.
- If the checktf application is called when the user-defined task or function was invoked on the interactive command line, the interactive command shall be aborted.

21.14 tf_evaluatep(), tf_ievaluatep()

tf_evaluatep(), tf_ievaluatep()			
Synopsis:	Evaluate a parameter expression.		
Syntax:	tf_evaluatep(nparam) tf_ievaluatep(nparam, instance_p)		
Type		Description	
Returns:	void		
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_exprinfo() or tf_iexprinfo() to get a pointer to the s_tfexprinfo structure Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_evaluatep()** and **tf_ievaluatep()** shall evaluate the current value of the specified parameter in the current instance or a specific instance of a user-defined system task or function. The current value shall be returned to the value cell in the **tf_exprinfo** structure returned from a previous call to the routine **tf_exprinfo()** or **tf_iexprinfo()**. This can be a more efficient way to obtain the current value of an expression than to call **tf_exprinfo()** or **tf_iexprinfo()** repeatedly.

The argument *nparam* shall be the index number of a parameter in a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1.

21.15 tf_exprinfo(), tf_iexprinfo()

tf_exprinfo(), tf_iexprinfo()			
Synopsis:	Get parameter expression information.		
Syntax:	tf_exprinfo(nparam, exprinfo_p) tf_iexprinfo(nparam, exprinfo_p, instance_p)		
Type		Description	
Returns:	struct t_tfexprinfo *	Pointer to a structure containing the value of the second argument if successful; 0 if an error occurred	
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter
	struct t_tfexprinfo *	exprinfo_p	Pointer to a variable declared as a t_tfexprinfo structure type
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_nodeinfo() or tf_inodeinfo() for additional information on writable parameters Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_exprinfo()** and **tf_iexprinfo()** shall return a pointer to a structure containing general information about the specified parameter in the current instance or a specific instance of a user-defined system task or function. The information shall be stored in the C structure **s_tfexprinfo**.

Memory space shall first be allocated to hold the information before calling **tf_exprinfo()** or **tf_iexprinfo()**. For example:

```

{
  s_tfexprinfo info;          /* declare a variable of the structure type */
  tf_exprinfo(n, &info);      /* pass tf_exprinfo a pointer to the variable */
  ...
}
```

This routine shall return the second argument, which is the pointer to the information structure. If *nparam* is out of range, or if some other error is found, then **0** shall be returned. The argument *nparam* shall be the index number of a parameter in a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1.

The `s_tfexprinfo` structure is defined in `veriusers.h` and is listed in Figure 21-3.

```
typedef struct t_tfexprinfo
{
    short    expr_type;
    short    padding;
    struct    t_vecval *expr_value_p;
    double   real_value;
    char     *expr_string;
    int      expr_ngroups;
    int      expr_vec_size;
    int      expr_sign;
    int      expr_lhs_select;
    int      expr_rhs_select;
} s_tfexprinfo, *p_tfexprinfo;
```

Figure 21-3—The `s_tfexprinfo` structure definition

The *expr_type* of the `s_tfexprinfo` structure shall indicate the Verilog HDL data type of the parameter, and it shall be one of the predefined constants as given in Table 21-2 and defined in `veriusers.h`.

Table 21-2—Predefined constants used with `tf_exprinfo()`

Predefined constant	Description
tf_nullparam	For null or non-existent parameters
tf_string	For string parameters
tf_readonly	For integer parameters
tf_readonlyreal	For real parameters
tf_readwrite	For integer parameters
tf_readwritereal	For real parameters
tf_rwbselect	Bit-select parameters
tf_rwpselect	Part-select parameters
tf_rwmselect	Memory-select parameters

If the expression type is **tf_readonly**, **tf_readwrite**, **tf_rwbselect**, **tf_rwpselect**, or **tf_rwmselect**, the *expr_value_p* of the `s_tfexprinfo` structure shall be a pointer to an array of `s_vecval` structures that shall contain the resultant value of the expression. The `s_vecval` structure for representing vector values is defined in `veriusers.h` and is listed in Figure 21-4.

```
typedef struct t_vecval
{
    int      avalbits;
    int      bvalbits;
} s_vecval, *p_vecval;
```

Figure 21-4—The s_vecval structure definition

If the number of bits in the vector (defined by the *expr_vec_size* field of the *s_tfexprinfo* structure) is less than or equal to 32, then there shall only be one *s_vecval* group in the *expr_value_p* array. For 33 bits to 64 bits, there shall be two groups in the array, and so on. The number of groups shall also be given by the value of the *expr_ngroups* field of the *s_tfexprinfo* structure. The components *avalbits* and *bvalbits* of the *s_vecval* structure shall hold the bit patterns making up the value of the parameter. The lsb in the value shall be represented by the lsb's in the *avalbits* and *bvalbits* components, and so on. The bit coding shall be as given in Table 21-3.

Table 21-3—avalbits/bvalbits encoding

aval / bval	Logic value
00	0
10	1
01	High impedance
11	Unknown

If the expression type is **tf_readonlyreal** or **tf_readwritereal**, the *real_value* field of the *s_tfexprinfo* structure shall contain the value.

If the expression is of type **tf_string**, the *expr_string* field of the *s_tfexprinfo* structure shall point to the string.

If the expression type is **tf_readonly**, **tf_readwrite**, **tf_rwbselect**, **tf_rwpartselect**, or **tf_rwmemselect**, the *expr_ngroups* of the *s_tfexprinfo* structure shall indicate the number of groups for the parameter expression value and determine the array size of the *expr_value_p* value structure pointer. If the expression type is **tf_readonlyreal** or **tf_readwritereal**, *expr_ngroups* shall be 0.

If the expression type is **tf_readonly**, **tf_readwrite**, **tf_rwbselect**, **tf_rwpartselect**, or **tf_rwmemselect**, the *expr_vec_size* field of the *s_tfexprinfo* structure shall indicate the total number of bits in the array of *expr_value_p* value structures. If the expression type is **tf_readonlyreal** or **tf_readwritereal**, *expr_vec_size* shall be 0.

The *expr_sign* field of the *s_tfexprinfo* structure shall indicate the sign type of the expression. It shall be 0 for unsigned or nonzero for signed.

21.16 tf_getcstringp(), tf_igetcstringp()

tf_getcstringp(), tf_igetcstringp()			
Synopsis:	Get parameter value as a string.		
Syntax:	<pre>tf_getcstringp(nparam) tf_igetcstringp(nparam, instance_p)</pre>		
Type		Description	
Returns:	char *	Pointer to a character string	
Type		Name	Description
Arguments:	int	nparam	Index number of the user-defined system task or function parameter
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getp() or tf_igetp() to get a parameter value as a 32-bit integer Use tf_getlongp() or tf_igetlongp() to get a parameter value as a 64-bit integer Use tf_getrealp() or tf_igetrealp() to get a parameter value as a double Use tf_strgetp() or tf_istrgetp() to get a parameter value as a formatted string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getcstringp()** and **tf_igetcstringp()** shall return a character string representing the value of the specified parameter in the current instance or a specific instance of a user-defined system task or function. If the parameter identified by *nparam* is a literal string, a variable, or an expression, then **tf_getcstringp()** or **tf_igetcstringp()** shall convert its value to a C-language ASCII string by

- Eliminating leading zeros
- Converting each group of 8 bits to an ASCII character
- Adding a “\0” string termination character to the end.

If the parameter identified by *nparam* is null or if *nparam* is out of range, then a null shall be returned.

The argument *nparam* shall be the index number of a parameter in a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1.

21.17 **tf_getinstance()**

tf_getinstance()		
Synopsis:	Get a pointer to the current instance of a user-defined system task or function.	
Syntax:	<code>tf_getinstance()</code>	
Returns:	Type	Description
	char *	Pointer to a system task or function instance
Arguments:	Type	Name Description
	None	

The TF routine **tf_getinstance()** shall return a pointer that identifies the current instance of the user-defined task or function in the Verilog HDL source code. The pointer returned by **tf_getinstance()** can be used later in other TF routine calls to refer to this instance of the task or function. Many of the TF routines are in two forms. One deals with the current task or function instance. The other deals with some other instance of the task or function, where the instance pointer for the other instance was previously obtained using **tf_getinstance()** during a call to a user routine initiated by that instance.

21.18 tf_getlongp(), tf_igetlongp()

tf_getlongp(), tf_igetlongp()			
Synopsis:	Get parameter value as a 64-bit integer.		
Syntax:	tf_getlongp(aof_highvalue, nparam) tf_igetlongp(aof_highvalue, nparam, instance_p)		
Type		Description	
Returns:	int	Least significant (rightmost) 32 bits of the parameter value	
Arguments:	Type	Name	Description
	int *	aof_highvalue	Pointer to most significant (leftmost) 32 bits of the parameter value
	int	nparam	Index number of the user-defined system task or function parameter
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getp() or tf_igetp() to get a parameter value as a 32-bit integer Use tf_getrealp() or tf_igetrealp() to get a parameter value as a double Use tf_getcstringp() or tf_igetcstringp() to get a parameter value as a string Use tf_strgetp() or tf_istrgetp() to get a parameter value as a formatted string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getlongp()** and **tf_igetlongp()** shall return a 64-bit integer value for the parameter specified by *nparam* in the current instance or a specific instance of a user-defined system task or function. If *nparam* is out of range or the parameter is null, then 0 shall be returned.

The argument *nparam* shall be the index number of a parameter in a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1.

21.19 tf_getlongtime(), tf_igetlongtime()

tf_getlongtime(), tf_igetlongtime()			
Synopsis:	Get current simulation time as a 64-bit integer.		
Syntax:	<pre>tf_getlongtime(aof_hightime) tf_igetlongtime(aof_hightime, instance_p)</pre>		
Type		Description	
Returns:	int	Least significant (rightmost) 32 bits of simulation time	
Type		Name	Description
Arguments:	int *	aof_hightime	Pointer to most significant (leftmost) 32 bits of simulation time
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_gettime() to get the simulation time as a 32-bit integer Use tf_str_gettime() to get the simulation time as a character string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getlongtime()** and **tf_igetlongtime()** shall return the simulation time as a 64-bit integer. The high 32 bits of simulation time shall be assigned to the *aof_hightime* argument, and the low 32 bits of time shall be returned.

Time shall be expressed in the timescale unit of the module containing the current instance or a specific instance of the user-defined system task or function.

21.20 tf_getnextlongtime()

tf_getnextlongtime()			
Synopsis:	Get next time at which a simulation event is scheduled.		
Syntax:	tf_getnextlongtime(aof_lowtime, aof_hightime)		
Returns:	Type		Description
	int		Integer value representing the meaning of the next event time obtained
Arguments:	Type	Name	Description
	int *	aof_lowtime	Pointer to least significant (rightmost) 32 bits of simulation time
	int *	aof_hightime	Pointer to most significant (leftmost) 32 bits of simulation time

The TF routine **tf_getnextlongtime()** shall assign the 64-bit time of the next simulation event to *aof_lowtime* and *aof_hightime*, and it shall return an integer value that indicates the meaning of the time assigned. The time shall be expressed in the timescale units of the module containing the current user-defined system task or function instance.

The **tf_getnextlongtime()** routine shall only return the time for the next simulation event when it is called in a *read-only synchronize mode*. A read-only synchronize mode occurs when the misctf user application has been called with **reason_rosynch**. If **tf_getnextlongtime()** is not called in read-only synchronize mode, then the current simulation time shall be assigned.

Table 21-4 summarizes the functions of **tf_getnextlongtime()**.

Table 21-4—Return values for tf_getnextlongtime()

When	tf_getnextlongtime() shall return	tf_getnextlongtime() shall assign to aof_lowtime and aof_hightime
tf_getnextlongtime() was called from a misctf application that was called with reason_rosynch	0	The next simulation time for which an event is scheduled
There are no more future events scheduled	1	0
tf_getnextlongtime() was not called from a misctf application that was called with reason_rosynch	2	The current simulation time

NOTE—Case 2 shall take precedence over case 1.

21.21 tf_getp(), tf_igetp()

tf_getp(), tf_igetp()			
Synopsis:	Get a parameter value as an integer or character string pointer.		
Syntax:	<pre>tf_getp(nparam) tf_igetp(nparam, instance_p)</pre>		
Type		Description	
Returns:	int	Integer value of a parameter or character string pointer of parameter string value	
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getlongp() or tf_igetlongp() to Get a parameter value as a 64-bit integer Use tf_getrealp() or tf_igetrealp() to get a parameter value as a double Use tf_getcstringp() or tf_igetcstringp() to get a parameter value as a string Use tf_strgetp() or tf_istrgetp() to get a parameter value as a formatted string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getp()** and **tf_igetp()** shall return a value of the parameter specified by *nparam* in the current instance or a specific instance of a user-defined system task or function. If the value of the parameter is an integer or a real number, the routines shall return an integer value. If the parameter is a literal string, then the routines shall return a pointer to a “C” type string (a string terminated by a “\0” character). If *nparam* is out of range or the parameter is null, then 0 shall be returned.

The argument *nparam* shall be the index number of a parameter in a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1.

The routines **tf_getp()** and **tf_getrealp()** differ in the value returned, as shown by the following example.

If the fourth parameter in the user-defined system task or function has a value of 9.6 (a real value), then

```
int ivalue = tf_getp(4)
```

would set *ivalue* to 10, whereas

```
double dvalue = tf_getrealp(4)
```

would set *dvalue* to 9.6.

In the first example, note that the int conversion rounds off the value of 9.6 to 10 (rather than truncating it to 9). In the second example, note that the real value has to be declared as a “double” (not as a “float”). Rounding is performed following the Verilog HDL rules.

21.22 tf_getpchange(), tf_igetpchange()

tf_getpchange(), tf_igetpchange()			
Synopsis:	Get the index number of the next parameter that changed value.		
Syntax:	<pre>tf_getpchange(nparam) tf_igetpchange(nparam, instance_p)</pre>		
Type		Description	
Returns:	int	Index number of the parameter that changed	
Type		Name	Description
Arguments:	int	nparam	Index number of the user-defined system task or function parameter
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_asynchon() or tf_iasynchon() to enable pvc flags Use tf_movepvc_flag(-1) to save pvc flags before calling tf_getpchange() Use tf_copypvc_flag() or tf_icopypvc_flag() to copy pvc flags Use tf_testpvc_flag() or tf_itestpvc_flag() to get the value of a saved pvc flag Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getpchange()** and **tf_igetpchange()** shall return the number of the next parameter with a number greater than *nparam* that changed value for the current instance or for a specific instance of a user-defined system task or function. The *nparam* argument shall be 0 the first time this routine is called within a given user routine invocation. The routines shall return the parameter number if there is a change in a parameter with a number greater than *nparam*, and they shall return 0 if there are no changes in parameters greater than *nparam* or if an error is detected. The routine shall use the saved pvc flags, so it is necessary to execute **tf_movepvc_flag(-1)** prior to calling the routine.

The argument *nparam* shall be the index number of a parameter in a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1.

PVC flags shall indicate whether a particular user-defined system task or function parameter has changed value. Each parameter shall have two pvc flags: a current pvc flag, which shall be set by a software product when the change occurs, and a saved pvc flag, which shall be controlled by the user.

NOTE—PVC flags shall not be set by the software product until **tf_asynchon()** or **tf_iasynchon()** has been called.

21.23 tf_getrealp(), tf_igetrealp()

tf_getrealp(), tf_igetrealp()			
Synopsis:	Get a parameter value as a double-precision value.		
Syntax:	<pre>tf_getrealp(nparam) tf_igetrealp(nparam, instance_p)</pre>		
Type		Description	
Returns:	double	Double-precision value of a parameter	
Type		Name	Description
Arguments:	int	nparam	Index number of the user-defined system task or function parameter
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getp() or tf_igetp() to get a parameter value as a 32-bit integer Use tf_getlongp() or tf_igetlongp() to get a parameter value as a 64-bit integer Use tf_getcstringp() or tf_igetcstringp() to get a parameter value as a string Use tf_strgetp() or tf_istrgetp() to get a parameter value as a formatted string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getrealp()** and **tf_igetrealp()** shall return a double-precision value of the parameter specified by *nparam* in the current instance or a specific instance of a user-defined system task or function. If *nparam* is out of range or the parameter is null, then 0 shall be returned.

The routines **tf_getrealp()** and **tf_igetrealp()** cannot handle literal strings. Therefore, before calling these routines, **tf_typep()** or **tf_itypep()** should be called to check the type of the parameter.

The argument *nparam* shall be the index number of a parameter in a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1.

21.24 tf_getrealtime(), tf_igetrealtime()

tf_getrealtime(), tf_igetrealtime()			
Synopsis:	Get the current simulation time in double-precision format.		
Syntax:	<pre>tf_getrealtime() tf_igetrealtime(instance_p)</pre>		
Type		Description	
Returns:	double	Current simulation time	
Type		Name	Description
Arguments:	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_gettime() to get the lower 32-bits of simulation time as an integer Use tf_gettime() to get the full 64-bits of simulation time as an integer Use tf_str_gettime() to get simulation time as a character string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getrealtime()** and **tf_igetrealtime()** shall return the simulation time as a real number in double-precision format.

Time shall be expressed in the timescale unit of the module containing the current instance or a specific instance of a user-defined system task or function.

21.25 tf_gettime(), tf_igettime()

tf_gettime(), tf_igettime()			
Synopsis:	Get the current simulation time as a 32-bit integer.		
Syntax:	<pre>tf_gettime() tf_igettime(instance_p)</pre>		
Type		Description	
Returns:	int	Least significant 32 bits of simulation time	
Type		Name	Description
Arguments:	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getlongtime() to get the full 64 bits of simulation time Use tf_getrealttime() to get the simulation time as a double-precision real number Use tf_str_gettime() to get simulation time as a character string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_gettime()** and **tf_igettime()** shall return the lower 32 bits of simulation time as an integer.

Time shall be expressed in the timescale unit of the module containing the current instance or a specific instance of a user-defined system task or function.

21.26 tf_gettimeprecision(), tf_igettimeprecision()

tf_gettimeprecision(), tf_igettimeprecision()			
Synopsis:	Get the timescale precision of a module or a simulation.		
Syntax:	<pre>tf_gettimeprecision() tf_igettimeprecision(instance_p)</pre>		
Type		Description	
Returns:	int	An integer value that represents a time precision	
Type		Name	Description
Arguments:	char *	instance_p	Pointer to a specific instance of a user-defined system task or function or <i>null</i> to represent the simulation
Related routines:	Use tf_gettimeunit() or tf_igettimeunit() to get the timescale time units Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_gettimeprecision()** and **tf_igettimeprecision()** shall return the timescale precision for the module that contains the current instance or a specific instance of a user-defined system task or function. The time precision is set by the ``timescale` Verilog HDL compiler directive in effect when the module was compiled. The routines shall return an integer code representing the time precision, as shown in Table 21-5.

Table 21-5—Code returned by tf_gettimeprecision() and tf_igettimeprecision()

Integer code returned	Simulation time precision
2	100 s
1	10 s
0	1 s
-1	100 ms
-2	10 ms
-3	1 ms
-4	100 μ s
-5	10 μ s
-6	1 μ s
-7	100 ns
-8	10 ns
-9	1 ns
-10	100 ps
-11	10 ps
-12	1 ps
-13	100 fs
-14	10 fs
-15	1 fs

When **tf_igettimeprecision()** is called with a *null* instance pointer, the routine shall return the simulation time unit, which is the smallest time precision used by all modules in a design.

21.27 tf_gettimeunit(), tf_igettimeunit()

tf_gettimeunit(), tf_igettimeunit()			
Synopsis:	Get the timescale unit of a module or a simulation.		
Syntax:	<pre>tf_gettimeunit() tf_igettimeunit(instance_p)</pre>		
Type		Description	
Returns:	int	An integer value that represents a time unit	
Type		Name	Description
Arguments:	char *	instance_p	Pointer to a specific instance of a user-defined system task or function or <i>null</i> to represent the simulation
Related routines:	Use tf_gettimeprecision() or tf_igettimeprecision() to get the timescale time precision Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_gettimeunit()** and **tf_igettimeunit()** shall return the timescale time units for the module that contains the current instance or a specific instance of a user-defined system task or function. The time unit for a module is set by the ``timescale` Verilog HDL compiler directive in effect when the module was compiled. The routines shall return an integer code representing the time unit, as shown in Table 21-6.

Table 21-6—Code returned by tf_gettimeunit() and tf_igettimeunit()

Integer code returned	Simulation time unit
2	100 s
1	10 s
0	1 s
-1	100 ms
-2	10 ms
-3	1 ms
-4	100 μ s
-5	10 μ s
-6	1 μ s
-7	100 ns
-8	10 ns
-9	1 ns
-10	100 ps
-11	10 ps
-12	1 ps
-13	100 fs
-14	10 fs
-15	1 fs

When **tf_igettimeunit()** is called with a *null* instance pointer, the routines shall return the simulation time unit, which is the smallest time precision used by all modules in a design.

21.28 tf_getworkarea(), tf_igetworkarea()

tf_getworkarea(), tf_igetworkarea()			
Synopsis:	Get work area pointer.		
Syntax:	<pre>tf_getworkarea() tf_igetworkarea(instance_p)</pre>		
Type		Description	
Returns:	char *	Pointer to a work area shared by all routines for a specific task/function instance	
Type		Name	Description
Arguments:	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_setworkarea() or tf_isetworkarea() to put a value into the work area pointer Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getworkarea()** and **tf_igetworkarea()** shall return the work area pointer value of the current instance or a specific instance of a user-defined system task or function. The value of the work area pointer shall be placed there by a previous call to the routine **tf_setworkarea()** or **tf_isetworkarea()**. These routines can be used as a means for two user applications to share information. For example, a checktf user application might open a file and then place the file pointer into the workarea using **tf_setworkarea()**. Later, the calltf user application can retrieve the file pointer using **tf_getworkarea()**.

21.29 `tf_long_to_real()`

<code>tf_long_to_real()</code>			
Synopsis:	Convert a long integer to a real number.		
Syntax:	<code>tf_long_to_real(low, high, aof_real)</code>		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	int	low	Least significant (rightmost) 32 bits of a 64-bit integer
	int	high	Most significant (leftmost) 32 bits of a 64-bit integer
	double *	aof_real	Pointer to a double-precision variable
Related routines:	Use <code>tf_real_to_long()</code> to convert a real number to a 64-bit integer Use <code>tf_longtime_tostr()</code> to convert a long integer to a character string		

The TF routine **`tf_long_to_real()`** shall convert a 64-bit long integer to a real (double-precision floating-point) number. The variable pointed to by *aof_real* shall contain the converted number upon return from this routine.

21.30 tf_longtime_tostr()

tf_longtime_tostr()			
Synopsis:	Convert 64-bit integer time value to a character string.		
Syntax:	tf_longtime_tostr(lowtime, hightime)		
Returns:	Type		Description
	char *	Pointer to a character string representing the simulation time value	
Arguments:	Type	Name	Description
	int	lowtime	Least significant (rightmost) 32 bits of simulation time
	int	hightime	Most significant (leftmost) 32 bits of simulation time
Related routines:	Use tf_getlongtime() to get the current simulation time as a 64-bit integer		

The TF routine **tf_longtime_tostr()** shall convert a 64-bit integer time value to a character string. The time value shall be unsigned.

21.31 tf_message()

tf_message()			
Synopsis:	Report an error or warning message with software product interruption control.		
Syntax:	tf_message(level, facility, code, message, arg1,...arg5)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	int	level	A predefined constant indicating the severity level of the error
	quoted string or char *	facility	A quoted character string or pointer to a character string used in the output message
	quoted string or char *	code	A quoted character string or pointer to a character string used in the output message
	quoted string or char *	message	A quoted character string or pointer to a character string that controls the message to be written
		arg1...arg5	One to five arguments used in the format control string; the type of each argument should be consistent with how it is used in the message string
Related routines:	Use tf_text() to store error information prior to calling tf_message Use tf_error() to report error messages Use tf_warning() to report warning messages		

The TF routine **tf_message()** shall display warning or error message information using the warning and error message format for a software product. The location information (file name and line number) of the current instance of the user-defined system task or function shall be appended to the message using a format compatible with warning and error messages generated by the software product, and the message shall be written to both the standard output text window and the output log file of the product.

The *level* field shall indicate the severity level of the error, specified as a predefined constant. There shall be five levels: *ERR_ERROR*, *ERR_SYSTEM*, *ERR_INTERNAL*, *ERR_MESSAGE*, and *ERR_WARNING*. If **tf_message()** is called by the checktf application associated with the user-defined system task or function, the following rules shall apply:

- If the checktf application is called when the Verilog HDL source code was being parsed or compiled, and the *level* is *ERR_ERROR*, *ERR_SYSTEM*, or *ERR_INTERNAL*, then parsing or compilation shall be aborted after an error message is reported.
- If the checktf application is called when the Verilog HDL source code was being parsed or compiled, and the *level* is *ERR_WARNING* or *ERR_MESSAGE*, then parsing or compilation shall continue after a warning message is reported.
- If the checktf application is called when the user-defined task or function was invoked on the interactive command line, the interactive command shall be aborted after a warning message or error message is reported.

The *facility* and *code* fields shall be string arguments that can be used in the Verilog software product message syntax. These strings shall be less than 10 characters in length.

The *message* argument shall be a user-defined control string containing the message to be displayed. The control string shall use the same formatting controls as the C `printf()` function (for example, `%d`). The message shall use up to a maximum of five variable arguments. There shall be no limit to the length of a variable argument. Formatting characters, such as `\n`, `\t`, `\b`, `\f`, or `\r`, do not need to be included in the message—the software product shall automatically format each message.

An example of a **tf_message()** call and the output generated are shown below. Note that the format of the output shall be defined by the software product.

Calling **tf_message()** with the arguments:

```
tf_message(ERR_ERROR, "User", TFARG",  
           "Argument number %d is illegal in task %s", argnum, taskname);
```

Might produce the output:

```
ERROR!    Argument number 2 is illegal in task      [User-TFARG]  
          $usertask
```

The routine **tf_message()** provides more control over the format and severity of error or warning messages than the routines **tf_error()** and **tf_warning()** can provide. In addition, the routine **tf_message()** can be used in conjunction with **tf_text()**, which shall allow an error or warning message to be stored while a PLI application executes additional code before the message is printed and parsing or compilation of Verilog HDL source possibly aborted.

21.32 tf_mipname(), tf_imipname()

tf_mipname(), tf_imipname()			
Synopsis:	Get the hierarchical module instance path name as a string.		
Syntax:	<code>tf_mipname()</code> <code>tf_imipname(instance_p)</code>		
Type		Description	
Returns:	char *	Pointer to a string containing the hierarchical path name	
Type		Name	Description
Arguments:	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use <code>tf_spname()</code> or <code>tf_ispname()</code> to get the scope path name Use <code>tf_getinstance()</code> to get a pointer to an instance of a user-defined system task or function		

The TF routine **tf_mipname()** shall return the Verilog HDL hierarchical path name to the module instance containing the call to the current instance or a specific instance of a user-defined system task or function.

The string obtained shall be stored in a temporary buffer. If the string is needed across multiple calls to the PLI application, the string should be preserved.

21.33 tf_movepvc_flag(), tf_imovepvc_flag()

tf_movepvc_flag(), tf_imovepvc_flag()			
Synopsis:	Move parameter value change flags.		
Syntax:	<pre>tf_movepvc_flag(nparam) tf_imovepvc_flag(nparam, instance_p)</pre>		
Type		Description	
Returns:	int	The value of the pvc flag	
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter, or -1
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_asynchon() or tf_iasynchon() to enable pvc flags Use tf_getpchange() or tf_igetpchange() to get the index number of the parameter that changed Use tf_copypvc_flag() or tf_icopypvc_flag() to copy a pvc flag to the saved pvc flag Use tf_testpvc_flag() or tf_itestpvc_flag() to get the value of a saved pvc flag Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_movepvc_flag()** and **tf_imovepvc_flag()** shall move the current pvc flag to the saved pvc flag and clear the current flag for the current instance or a specific instance of a user-defined system task or function. The routine shall return the value of the flag that was moved.

The argument *nparam* shall be the index number of a parameter in a specific instance of a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1. If *nparam* is **-1**, then all parameter pvc flags shall be moved and the logical OR of all saved flags returned.

PVC flags shall be used to indicate whether a particular user-defined system task or function parameter has changed value. Each parameter shall have two pvc flags: a current pvc flag, which shall be set by a software product when the change occurs, and a saved pvc flag, which shall be controlled by the user.

NOTE—PVC flags shall not be set by the software product until **tf_asynchon()** or **tf_iasynchon()** has been called.

21.34 tf_multiply_long()

tf_multiply_long()			
Synopsis:	Multiply two 64 bit integers.		
Syntax:	tf_multiply_long(&aof_low1, &aof_high1, low2, high2)		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	int *	aof_low1	Pointer to least significant 32 bits of first operand
	int *	aof_high1	Pointer to most significant 32 bits of first operand
	int	low2	Least significant 32 bits of second operand
	int	high2	Most significant 32 bits of second operand
Related routines:	Use tf_add_long() to add two 64-bit integers Use tf_subtract_long() to subtract two 64-bit integers Use tf_divide_long() to divide two 64-bit integers Use tf_compare_long() to compare two 64-bit integers		

The TF routine **tf_multiply_long()** shall add two 64-bit values. After calling **tf_multiply_long()**, the variables used to pass the first operand shall contain the results of the multiplication. Figure 21-5 shows the high and low 32 bits of two long integers and how **tf_multiply_long()** shall multiply them.

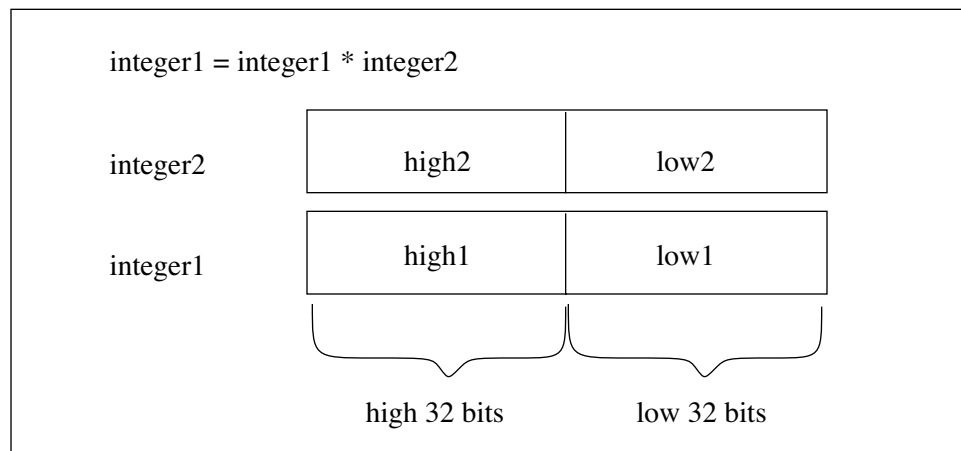


Figure 21-5—Multiplying with tf_multiply_long()

21.35 tf_nodeinfo(), tf_inodeinfo()

tf_nodeinfo(), tf_inodeinfo()			
Synopsis:	Get parameter node information.		
Syntax:	<pre>tf_nodeinfo(nparam, nodeinfo_p) tf_inodeinfo(nparam, nodeinfo_p, instance_p)</pre>		
Type		Description	
Returns:	struct t_tfnodeinfo *	The value of the second argument if successful; 0 if an error occurred	
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter
	struct t_tfnodeinfo *	nodeinfo_p	Pointer to a variable declared as the t_tfnodeinfo structure type
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_exprinfo() or tf_iexprinfo() for general information on parameters Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_nodeinfo()** and **tf_inodeinfo()** shall obtain information about the specified writable parameter in the current instance or a specific instance of a user-defined system task or function.

The information shall be stored in the C structure **s_tfnodeinfo** as defined in the file **veriusers.h**. The routine shall only be called for parameters that are writable (e.g., Verilog HDL register data types). For parameters that are read only, the TF routines **tf_exprinfo()** or **tf_iexprinfo()** can be used.

Memory space shall first be allocated to hold the information before calling **tf_nodeinfo()** or **tf_inodeinfo()**. For example:

```
{
  s_tfnodeinfo info;      /* declare a variable of the structure type */
  tf_nodeinfo(n, &info); /* pass tf_nodeinfo a pointer to the variable */
  ...
}
```

The routines shall return the second argument, which is the pointer to the information structure. If *nparam* is out of range, or if some other error is found, then **0** shall be returned.

The argument *nparam* shall be the index number of a parameter in a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1.

The **s_tfnodeinfo** structure is defined in **veriusers.h** and is listed in Figure 21-6.


```

typedef struct t_tfnodeinfo
{
    short    node_type;
    short    padding;
    union
    {
        struct t_vecval *vecval_p;
        struct t_strengthval *strengthval_p;
        char    *memoryval_p;
        double *real_val_p;
    } node_value;
    char    *node_symbol;
    int     node_ngroups;
    int     node_vec_size;
    int     node_sign;
    int     node_ms_index;
    int     node_ls_index;
    int     node_mem_size;
    int     node_lhs_element;
    int     node_rhs_element;
    int     *node_handle;
} s_tfnodeinfo, *p_tfnodeinfo;

```

Figure 21-6—The s_tfnodeinfo structure definition

The following paragraphs define the field of the s_tfnodeinfo structure.

The *node_type* field of the s_tfnodeinfo structure shall indicate the Verilog HDL data type of the parameter, and is one of the predefined constants as given in Table 21-7 and defined in `veriusers.h`.

Table 21-7—Predefined constants for node_type

Predefined constant	Description
tf_null_node	Not a writable parameter
tf_reg_node	Parameter references a register variable
tf_integer_node	Parameter references an integer variable
tf_real_node	Parameter references a real variable
tf_time_node	Parameter references a time variable
tf_netvector_node	Parameter references a vector net
tf_netscalar_node	Parameter references a scalar net
tf_memory_node	Parameter references a memory

The *node_value* field of the *s_tfnodeinfo* structure shall be a union of pointers to value structures defining the current value on the node referenced by the parameter. The union member accessed shall depends on the *node_type*. The union members are given in Table 21-8.

Table 21-8—How the *node_value* union is used

When the <i>node_type</i> is	The union member used is
tf_reg_node , tf_integer_node , tf_time_node , or tf_netvector_node	<i>vecval_p</i>
tf_real_node	<i>real_value_p</i>
tf_netscalar_node	<i>strengthval_p</i>
tf_memory_node	<i>memoryval_p</i>

If the *node_type* is **tf_reg_node**, **tf_integer_node**, **tf_time_node**, or **tf_netvector_node**, then *node_value* shall be a pointer to an array of *s_vecval* structures that gives the resultant value of the node. The *s_vecval* structure for representing vector values is defined in *veriusers.h* and is listed in Figure 21-7.

```
typedef struct t_vecval
{
    int      avalbits;
    int      bvalbits;
} s_vecval, *p_vecval;
```

Figure 21-7—The *s_vecval* structure definition

If the number of bits in the vector (defined by the *node_vec_size* field of the *s_tfnodeinfo* structure) is less than or equal to 32, then there shall only be one *s_vecval* group in the *node_value_p* array. For 33 bits to 64 bits, two groups shall be in the array, and so on. The number of groups shall also given by the value of *node_ngroups*. The fields for *avalbits* and *bvalbits* of the *s_vecval* structure shall hold the bit patterns making up the value of the parameter. The lsb in the value shall be represented by the lsb's in the *avalbits* and *bvalbits* components, and so on. The bit coding shall be as given in Table 21-9.

Table 21-9—avalbits/bvalbits encoding

aval / bval	Logic value
00	0
10	1
01	High impedance
11	Unknown

If the *node_type* field of the *s_tfnodeinfo* structure is **tf_netscalar_node**, then the *node_value.strengthval_p* field of the *s_tfnodeinfo* structure shall point to an *s_strengthval* structure of the form given in Figure 21-8.

```
typedef struct t_strengthval
{
    int strength0;
    int strength1;
} s_strengthval, *p_strengthval;
```

Figure 21-8—The s_strengthval structure definition

In the `s_strengthval` structure, *strength0* shall give the 0-strength bit pattern for the value, and *strength1* shall give the 1-strength bit pattern. Refer to 7.10 for details about these bit patterns.

If the *node_type* field of the `s_tfnodeinfo` structure is **tf_memory_node**, then *node_value.memoryval_p* shall point to a `memval` structure giving the total contents of the memory. The structure is organized as shown in Figure 21-9.

```
struct
{
    char avalbits[node_ngroups];
    char bvalbits[node_ngroups];
} memval[node_mem_size];
```

Figure 21-9—The memval structure definition

Note that a pointer to the `memval` structure data structure cannot be represented in C, so the *node_value.memoryval_p* field of the `s_tfnodeinfo` structure is declared as a pointer to a `char` type. The memory element addressed by the left-hand-side index given in the memory declaration shall be located in the first group of bytes, which is the byte group represented by `memval[0]`.

The *node_symbol* field of the `s_tfnodeinfo` structure shall be a string pointer to the identifier of the parameter.

If the *node_type* field of the `s_tfnodeinfo` structure is **tf_reg_node**, **tf_integer_node**, **tf_time_node**, or **tf_netvector_node**, then the *node_ngroups* field of the `s_tfnodeinfo` structure shall indicate the number of groups for the parameter *nodevalue* and shall determine the array size of the *node_value.vecval_p* value structure. If the *node_type* is **tf_real_node**, then *node_ngroups* shall be 0.

If the *node_type* field of the `s_tfnodeinfo` structure is **tf_reg_node**, **tf_integer_node**, **tf_time_node**, or **tf_netvector_node**, then the *node_vec_size* field of the `s_tfnodeinfo` structure shall indicate the total number of bits in the array of the *node_value.vecval_p* structure. If the *node_type* is **tf_real_node**, then *node_vec_size* shall be 0.

The *node_sign* field of the `s_tfnodeinfo` structure shall indicate the sign type of the node as follows: 0 for unsigned, nonzero for signed.

If the *node_type* is **tf_memory_node**, then *node_mem_size* shall indicate the number of elements in the *node_value.memoryval_p* structure.

21.36 tf_nump(), tf_inump()

tf_nump(), tf_inump()			
Synopsis:	Get number of task or function parameters.		
Syntax:	<pre>tf_nump () tf_inump (instance_p)</pre>		
Type		Description	
Returns:	int	The number of parameters	
Type		Name	Description
Arguments:	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_nump()** and **tf_inump()** shall return the number of parameters (task/function arguments) specified in the current instance or a specific instance of a user-defined task or function statement in the Verilog source description. The number returned shall be greater than or equal to zero.

21.37 tf_propagatep(), tf_ipropagatep()

tf_propagatep(), tf_ipropagatep()			
Synopsis:	Propagate a parameter value.		
Syntax:	tf_propagatep(nparam) tf_ipropagatep(nparam, instance_p)		
Type		Description	
Returns:	void		
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_exprinfo() or tf_iexprinfo() to get a parameter expression value Use tf_nodeinfo() or tf_inodeinfo() to get a parameter node value Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_ipropagatep()** and **tf_ipropagatep()** shall write a value to a parameter node of the current instance or a specific instance of a user-defined system task or function, and then propagate the value to any continuous assignments that read the value of the node.

In order to write values back into a Verilog software product data structure using **tf_ipropagatep()** and **tf_ipropagatep()**, the value shall first be placed into the value structure pointed to by the component **expr_value_p** as allocated by calling **tf_exprinfo()** or **tf_iexprinfo()**. The structure for **tf_exprinfo()** and **tf_iexprinfo()** shall be used for all parameter types except memories. For memories, the value to be propagated shall be placed in the structure used with **tf_nodeinfo()** or **tf_inodeinfo()**.

21.38 tf_putlongp(), tf_iputlongp()

tf_putlongp(), tf_iputlongp()			
Synopsis:	Write a 64-bit integer value to a parameter or function return.		
Syntax:	tf_putlongp(nparam, lowvalue, highvalue) tf_iputlongp(nparam, lowvalue, highvalue, instance_p)		
Type		Description	
Returns:	void		
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter or 0 to return a function value
	int	lowvalue	Least significant (rightmost) 32 bits of value
	int	highvalue	Most significant (leftmost) 32 bits of value
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_putp() or tf_iputp() to put a parameter value as a 32-bit integer Use tf_putrealp() or tf_iputrealp() to get a parameter value as a double Use tf_strdelputp() to put a value as a formatted string with delay Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_putlongp()** and **tf_iputlongp()** shall write a 64-bit integer value to the parameter specified by *nparam* of the current instance or a specific instance of a user-defined system task. If *nparam* is 0, **tf_putlongp()** and **tf_iputlongp()** shall write the value as the return of a user-defined system function. If *nparam* is out of range or the parameter cannot be written to, then the routines shall do nothing.

The argument *nparam* shall be the index number of a parameter in a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1.

The data type of *value* should be consistent with the type of put routine and the type of the parameter to which the value shall be written. Refer to 20.3 for more details on proper data type selection with put routines.

21.39 tf_putp(), tf_iputp()

tf_putp(), tf_iputp()			
Synopsis:	Put an integer value to a parameter or function return.		
Syntax:	<pre>tf_putp(nparam, value) tf_iputp(nparam, value, instance_p)</pre>		
Type		Description	
Returns:	void		
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter or 0 to return a function value
	int	value	An integer value
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_putlongp() or tf_iputlongp() to put a parameter value as a 64-bit integer Use tf_putrealp() or tf_iputrealp() to get a parameter value as a double Use tf_strdelputp() to put a value as a formatted string with delay Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routine **tf_putp()** and **tf_iputp()** shall write an integer value to the parameter specified by *nparam* of the current instance or a specific instance of a user-defined system task. If *nparam* is 0, **tf_putp()** or **tf_iputp()** shall write the value as the return of a user-defined system function. If *nparam* is out of range or the parameter cannot be written to, then the routines shall do nothing.

The argument *nparam* shall be the index number of a parameter in a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1.

The data type of *value* should be consistent with the type of put routine and the type of the parameter to which the value shall be written. Refer to 20.3 for more details on proper data type selection with put routines.

21.40 tf_putrealp(), tf_iputrealp()

tf_putrealp(), tf_iputrealp()			
Synopsis:	Write a real value to a parameter or function return.		
Syntax:	<pre>tf_putrealp(nparam, value) tf_iputrealp(nparam, value, instance_p)</pre>		
Type		Description	
Returns:	void		
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter or 0 to return a function value
	double	value	A double-precision value
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_putp() or tf_iputp() to put a parameter value as a 32-bit integer Use tf_putlongp() or tf_iputlongp() to put a parameter value as a 64-bit integer Use tf_strdelputp() to put a value as a formatted string with delay Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_putrealp()** and **tf_iputrealp()** shall write a double-precision real value to the parameter specified by *nparam* of the current instance or a specific instance of a user-defined system task. If *nparam* is 0, **tf_putrealp()** and **tf_iputrealp()** shall write the value as the return of a user-defined system function. If *nparam* is out of range or the parameter cannot be written to, then the routines shall do nothing.

The argument *nparam* shall be the index number of a parameter in a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1.

The data type of *value* should be consistent with the type of put routine and the type of the parameter to which the value shall be written. Refer to 20.3 for more details on proper data type selection with put routines.

21.41 tf_read_restart()

tf_read_restart()			
Synopsis:	Get a block of data from a previously written save file.		
Syntax:	tf_read_restart(blockptr, blocklen)		
Returns:	Type	Description	
	int	Nonzero if successful; zero if an error occurred	
Arguments:	Type	Name	Description
	char *	blockptr	Pointer to block of saved data
	int	blocklen	Length of block
Related routines:	Use tf_write_save() to save a block of data		

The TF routine **tf_read_restart()** shall read back a block of memory that was saved with **tf_write_save()**. This routine shall only be called from the miscf application when the miscf routine is invoked with **reason_restart**.

The argument *blockptr* shall be a pointer to an allocated block of memory to which the saved data shall be restored.

The argument *blocklen* shall be the length in bytes of the allocated block of memory. Exactly as many bytes have to be restored as were written with **tf_write_save()**.

If any user task instance pointers have been saved (for use with *tf_i** calls), **tf_getinstance()** has to be used to get new instance pointer values after the restart. If pointers to user data were saved, the application of the user has to implement a scheme to reconnect them properly.

21.42 tf_real_to_long()

tf_real_to_long()			
Synopsis:	Convert a real number to a 64-bit integer.		
Syntax:	tf_real_to_long(realvalue, aof_int_lo, aof_int_hi)		
Returns:	Type		Description
	void		
Arguments:	Type	Name	Description
	double	realvalue	Value to be converted
	int *	aof_low	Pointer to an integer variables for storing the least significant (rightmost) 32 bits of the converted value
	int *	aof_high	Pointer to an integer variables for storing the most significant (leftmost) 32 bits of the converted value
Related routines:	Use tf_long_to_real() to convert a 64-bit integer to a real number		

The TF routine **tf_real_to_long()** shall convert a double-precision floating-point number to a long (64-bit) integer. The converted value shall be returned in the variables pointed to by *aof_low* and *aof_high*.

21.43 tf_rosynchronize(), tf_irosynchronize()

tf_rosynchronize(), tf_irosynchronize()			
Synopsis:	Synchronize to end of simulation time step.		
Syntax:	<pre>tf_rosynchronize() tf_irosynchronize(instance_p)</pre>		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function Use tf_synchronize() to synchronize to end of simulation time step Use tf_getnextlongtime() to get next time at which a simulation event is scheduled		

The TF routine **tf_rosynchronize()** and **tf_irosynchronize()** shall schedule a callback to the misctf application associated with the current instance or a specific instance of a user-defined system task or function. The misctf application shall be called with a reason of **reason_rosynch** at the end of the current simulation time step.

The routines **tf_synchronize()** and **tf_rosynchronize()** have different functionality. The routine **tf_synchronize()** shall call the associated misctf application at the end of the current simulation time step with **reason_synch**, and the misctf application shall be allowed to schedule additional simulation events using routines such as **tf_strdelputp()**.

The routine **tf_rosynchronize()** shall call the associated misctf application at the end of the current simulation time step with **reason_rosynch**, and the PLI shall not be allowed to schedule any new events. This guarantees that all simulation events for the current time are completed. Calls to routines such as **tf_strdelputp()** and **tf_setdelay()** are illegal during processing of the misctf application with reason **reason_rosynch**.

The routine **tf_getnextlongtime()** shall only return the next simulation time for which an event is scheduled when used in conjunction with the routines **tf_rosynchronize()** and **tf_irosynchronize()**.

21.44 tf_scale_longdelay()

tf_scale_longdelay()			
Synopsis:	Convert a 64-bit integer delay to internal simulation time units.		
Syntax:	<pre>tf_scale_longdelay(instance_p, delay_lo, delay_hi, &aof_delay_lo, &aof_delay_hi)</pre>		
Type		Description	
Returns:	void		
Arguments:	Type	Name	Description
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
	int	delay_lo	Least significant (rightmost) 32 bits of the delay to be converted
	int	delay_hi	Most significant (leftmost) 32 bits of the delay to be converted
	int *	aof_delay_lo	Pointer to a variable to store the least significant (rightmost) 32 bits of the conversion result
	int *	aof_delay_hi	Pointer to a variable to store the most significant (leftmost) 32 bits of the conversion result
Related routines:	Use tf_scale_realdelay() to scale real number delays Use tf_unscale_longdelay() to convert a delay to the time unit of a module Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routine **tf_scale_longdelay()** shall convert a long (64-bit) integer delay into the timescale of the module containing the instance of the user-defined system task or function pointed to by *instance_p*. The arguments *aof_delay_lo* and *aof_delay_hi* shall contain the address of the converted delay returned by the routine.

21.45 tf_scale_realdelay()

tf_scale_realdelay()			
Synopsis:	Convert a double-precision floating-point delay to internal simulation time units.		
Syntax:	tf_scale_realdelay(instance_p, realdelay, &aof_realdelay)		
Returns:	Type		Description
	void		
Arguments:	Type	Name	Description
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
	double	realdelay	Value of the delay to be converted
	double *	aof_realdelay	Pointer to a variable to store the conversion result
Related routines:	Use tf_scale_longdelay() to scale 64-bit integer delays Use tf_unscale_realdelay() to convert a delay to the time unit of a module Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routine **tf_scale_realdelay()** shall convert a double-precision floating-point delay into the timescale of the module containing the instance of the user-defined system task or function pointed to by *instance_p*. The argument *aof_realdelay* shall contain the address of the converted delay returned by the routine.

21.46 tf_setdelay(), tf_isetdelay()

tf_setdelay(), tf_isetdelay()			
Synopsis:	Activate the miscf application at a particular simulation time.		
Syntax:	<pre>tf_setdelay(delay) tf_isetdelay(delay, instance_p)</pre>		
	Type	Description	
Returns:	int	1 if successful; 0 if an error occurred	
	Type	Name	Description
Arguments:	int	delay	32-bit integer delay time
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_setlongdelay() or tf_isetlongdelay() for 64-bit integer reactivation delays Use tf_setrealdelay() or tf_isetrealdelay() for real number reactivation delays Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_setdelay()** and **tf_isetdelay()** shall schedule a callback to the miscf application associated with the current instance or a specific instance of a user-defined system task or function. The miscf application shall be called at a future *reactivation time*. The reactivation time shall be the current simulation time plus the specified delay. The miscf application shall be called at the reactivation time with a reason of **reason_reactivate**. The **tf_setdelay()** and **tf_isetdelay()** routines can be called several times with different delays, and several reactivations shall be scheduled.

The *delay* argument shall be a 32-bit integer and shall be greater than or equal to 0. The delay shall assume the timescale units specified for the module containing the specific system task call.

21.47 tf_setlongdelay(), tf_isetlongdelay()

tf_setlongdelay(), tf_isetlongdelay()			
Synopsis:	Activate the miscf application at a particular simulation time.		
Syntax:	tf_setlongdelay(lowdelay, highdelay) tf_isetlongdelay(lowdelay, highdelay, instance_p)		
	Type	Description	
Returns:	int	1 if successful; 0 if an error occurred	
	Type	Name	Description
Arguments:	int	lowdelay	Least significant (rightmost) 32 bits of the delay time to reactivation
	int	highdelay	Most significant (leftmost) 32 bits of the delay time to reactivation
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_setdelay() or tf_isetdelay() for 32-bit integer reactivation delays Use tf_setrealdelay() or tf_isetrealdelay() for real number reactivation delays Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_setlongdelay()** and **tf_isetlongdelay()** shall schedule a callback to the miscf application associated with the current instance or a specific instance of a user-defined system task or function. The miscf application shall be called at a future *reactivation time*. The reactivation time shall be the current simulation time plus the specified delay. The miscf routine shall be called at the reactivation time with a reason of **reason_reactivate**. The **tf_setlongdelay()** and **tf_isetlongdelay()** routines can be called several times with different delays, and several reactivations shall be scheduled.

The *delay* argument shall be a 64-bit integer and shall be greater than or equal to 0. The delay shall assume the timescale units specified for the module containing the specific system task call.

21.48 tf_setrealdelay(), tf_isetrealdelay()

tf_setrealdelay(), tf_isetrealdelay()			
Synopsis:	Activate the <i>misctf</i> application at a particular simulation time.		
Syntax:	<pre>tf_setrealdelay(realdelay) tf_isetrealdelay(realdelay, instance_p)</pre>		
	Type	Description	
Returns:	int	1 if successful; 0 if an error occurred	
	Type	Name	Description
Arguments:	double	realdelay	Double-precision delay time to reactivation
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_setdelay() or tf_isetdelay() for 32-bit integer reactivation delays Use tf_setlongdelay() or tf_isetlongdelay() for 64-bit integer reactivation delays Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_setrealdelay()** and **tf_isetrealdelay()** shall schedule a callback to the *misctf* application associated with the current instance or a specific instance of a user-defined system task or function. The *misctf* application shall be called at a future *reactivation time*. The reactivation time shall be the current simulation time plus the specified delay. The *misctf* application shall be called at the reactivation time with a reason of **reason_reactivate**. The **tf_setrealdelay()** and **tf_isetrealdelay()** routines can be called several times with different delays, and several reactivations shall be scheduled.

The *delay* argument shall be a double-precision value and shall be greater than or equal to 0.0. The delay shall assume the timescale units specified for the module containing the specific system task call.

21.49 tf_setworkarea(), tf_isetworkarea()

tf_setworkarea(), tf_isetworkarea()			
Synopsis:	Store user data pointer in work area.		
Syntax:	<pre>tf_setworkarea(workarea) tf_isetworkarea(workarea, instance_p)</pre>		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	char *	workarea	Pointer to user data
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getworkarea() or tf_igetworkarea() to retrieve the user data pointer Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_setworkarea()** and **tf_isetworkarea()** shall store a pointer to user data in the work area of the current instance or a specific instance of a user-defined system task or function. The pointer that is stored can be retrieved by calling **tf_getworkarea()** or **tf_igetworkarea()**.

The work area can be used for

- Saving information during one call to a PLI routine, which can be retrieved upon a subsequent invocation of the routine
- Passing information from type of PLI application to another, such as from a checktf application to a calltf application

Note that the workarea pointer is a *char ** type. If the memory allocated for the user data is of some other type, it should be cast to *char **.

21.50 tf_sizep(), tf_isizep()

tf_sizep(), tf_isizep()			
Synopsis:	Get the bit length of a parameter.		
Syntax:	tf_sizep(nparam) tf_isizep(nparam, instance_p)		
Type		Description	
Returns:	int	The number of bits of the parameter	
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_sizep()** and **tf_isizep()** shall return the value size in bits of the specified parameter in the current instance or a specific instance of a user-defined system task or function.

If the specified parameter is a literal string, **tf_sizep()** and **tf_isizep()** shall return the string length.

If the specified parameter is real or if an error is detected, **tf_sizep()** and **tf_isizep()** shall return 0.

The argument *nparam* shall be the index number of a parameter in a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1.

21.51 tf_spname(), tf_ispname()

tf_spname(), tf_ispname()			
Synopsis:	Get scope hierarchical path name as a string.		
Syntax:	tf_spname() tf_ispname(instance_p)		
Type		Description	
Returns:	char *	Pointer to a character string with the hierarchical path name	
Type		Name	Description
Arguments:	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_spname()** and **tf_ispname()** shall return a pointer to the Verilog HDL hierarchical path name to the scope containing the call of a specific instance of a user-defined system task or function.

A scope shall be

- A top-level module
- A module instance
- A named begin-end block
- A named fork-join block
- A Verilog HDL task
- A Verilog HDL function

The string obtained shall be stored in a temporary buffer. If the string is needed across multiple calls to the PLI application, the string should be preserved.

21.52 tf_strdelputp(), tf_istrdelputp()

tf_strdelputp(), tf_istrdelputp()			
Synopsis:	Write a value to a parameter from string value specification, using a 32-bit integer delay.		
Syntax:	<pre>tf_strdelputp(nparam, bitlength, format, value_p, delay, delaytype) tf_istrdelputp(nparam, bitlength, format, value_p, delay, delaytype, instance_p)</pre>		
Type		Description	
Returns:	int	1 if successful; 0 if an error is detected	
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter
	int	bitlength	Number of bits the value represents
	int	format	A character in single quotes representing the radix (base) of the value
	quoted string or char *	value_p	Quoted character string or pointer to a character string with the value to be written
	int	delay	Integer value representing the time delay before the value should be written to the parameter
	int	delaytype	Integer code representing the delay mode for applying the value
Related routines:	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
	Use tf_strlongdelputp() or tf_istrlongdelputp() for 64-bit integer delays Use tf_strrealdelputp() or tf_istrrealdelputp() for real number delays Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_strdelputp()** and **tf_istrdelputp()** shall write a string value to the specified parameter of the current instance or a specific instance of a user-defined system task. The actual change to the parameter shall be scheduled as an event on the parameter in the Verilog model at a future simulation time.

The *bitlength* argument shall define the value size in bits.

The *format* shall define the format of the value specified by *value_p* and shall be one of the characters given in Table 21-10.

Table 21-10—Format characters

Format character	Description
'b' or 'B'	Value is in binary
'o' or 'O'	Value is in octal
'd' or 'D'	Value is in decimal
'h' or 'H'	Value is in hexadecimal

The *delay* argument shall represent the amount of time before the value shall be applied to the parameter, and it shall be greater than or equal to 0. The delay shall assume the timescale units of the module containing the instance of the user-defined system task or function.

The *delaytype* argument shall determine how the value shall be scheduled in relation to other simulation events on the same register. The *delaytype* shall be one of integer values shown in Table 21-11.

Table 21-11—delaytype codes

delaytype code	Definition	Description
0	Inertial delay	All scheduled events on the output parameter in the Verilog model are removed before scheduling a new event
1	Modified transport delay	All events that are scheduled for times later than the new event on the output parameter in the Verilog model are removed before scheduling a new event
2	Pure transport delay	No scheduled events on the output parameter in the Verilog model are removed before scheduling a new event—the last event to be scheduled is not necessarily the last one to occur

21.53 tf_strgetp(), tf_istrgetp()

tf_strgetp(), tf_istrgetp()			
Synopsis:	Get formatted parameter values.		
Syntax:	<pre>tf_strgetp(nparam, format) tf_istrgetp(nparam, format, instance_p)</pre>		
Type		Description	
Returns:	char *	Pointer to a character string with the parameter value	
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter
	char	format	Character in single quotes controlling the return value format
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getp() or tf_igetp() to get a parameter value as a 32-bit integer Use tf_getlongp() or tf_igetlongp() to get a parameter value as a 64-bit integer Use tf_getrealp() or tf_igetrealp() to get a parameter value as a double Use tf_getcstringp() or tf_igetcstringp() to get a parameter value as a string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_strgetp()** and **tf_istrgetp()** shall return a pointer to a string that contains the value of the parameter expression of the current instance or a specific instance of a user-defined system task or function.

The string format is specified by *format*, and shall be one of the following characters shown in Table 21-12.

Table 21-12—Format characters

Format character	Description
'b' or 'B'	Value is in binary
'o' or 'O'	Value is in octal
'd' or 'D'	Value is in decimal
'h' or 'H'	Value is in hexadecimal

The string value returned shall have the same form as output from the formatted built-in system task **\$display()** in terms of value lengths and value characters used. The length shall be of arbitrary size (not limited to 32 bits as with the **tf_getp()** routine), and unknown and high-impedance values shall be obtained.

The referenced parameter can be a string, in which case a pointer to the string shall be returned (the *format* shall be ignored in this case). The string obtained shall be stored in a temporary buffer. If the string is needed across multiple calls to the PLI application, the string should be preserved.

A null pointer shall be returned for errors.

21.54 tf_strgettime()

tf_strgettime()			
Synopsis:	Get the current simulation time as a string.		
Syntax:	tf_strgettime()		
Returns:	Type	Description	
	char *	Pointer to a character string with the simulation time	
Arguments:	Type	Name	Description
			No arguments
Related routines:	Use tf_gettime() to get simulation time as a 32-bit integer value Use tf_getlongtime() to get simulation time as a 64-bit integer value Use tf_getrealtime() to get simulation time as a real value		

The TF routine **tf_strgettime()** shall return a pointer to a string, which shall be the ASCII representation of the current simulation time. The string obtained shall be stored in a temporary buffer. If the string is needed across multiple calls to the PLI application, the string should be preserved.

21.55 `tf_strlongdelputp()`, `tf_istrlongdelputp()`

tf_strlongdelpu ^t (), tf_istrlongdelpu ^t ()			
Synopsis:	Write a value to a parameter from string value specification, using a 64-bit integer delay.		
Syntax:	tf_strlongdelpu ^t (nparam, bitlength, format, value_p, lowdelay, highdelay, delaytype) tf_istrlongdelpu ^t (nparam, bitlength, format, value_p, lowdelay, highdelay, delaytype, instance_p)		
Type		Description	
Returns:	int	1 if successful; 0 if an error is detected	
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter
	int	bitlength	Number of bits the value represents
	int	format	A character in single quotes representing the radix (base) of the value
	quoted string or char *	value_p	Quoted character string or pointer to a character string with the value to be written
	int	lowdelay	Least significant (rightmost) 32 bits of delay before the value is be written to the parameter
	int	highdelay	Most significant (leftmost) 32 bits of delay before the value is be written to the parameter
	int	delaytype	Integer code representing the delay mode for applying the value
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_strdelpu ^t () or tf_istrdelpu ^t () for 32-bit integer delays Use tf_strrealdelpu ^t () or tf_istrrealdelpu ^t () for real number delays Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_strlongdelputp()** and **tf_istrlongdelputp()** shall write a string value to the specified parameter of the current instance or a specific instance of a user-defined system task. The actual change to the parameter shall be scheduled as an event on the parameter in the Verilog model at a future simulation time.

The *bitlength* argument shall define the value size in bits.

The *format* shall define the format of the value specified by *value_p* and shall be one of the characters shown in Table 21-13.

Table 21-13—Format characters

Format character	Description
'b' or 'B'	Value is in binary
'o' or 'O'	Value is in octal
'd' or 'D'	Value is in decimal
'h' or 'H'	Value is in hexadecimal

The *delay* argument shall represent the amount of time before the value shall be applied to the parameter, and it shall be greater than or equal to 0. The delay shall assume the timescale units of the module containing the instance of the user-defined system task or function.

The *delaytype* argument shall determine how the value shall be scheduled in relation to other simulation events on the same register. The *delaytype* shall be one of integer values shown in Table 21-14.

Table 21-14—delaytype codes

delaytype code	Definition	Description
0	Inertial delay	All scheduled events on the output parameter in the Verilog model are removed before scheduling a new event
1	Modified transport delay	All events that are scheduled for times later than the new event on the output parameter in the Verilog model are removed before scheduling a new event
2	Pure transport delay	No scheduled events on the output parameter in the Verilog model are removed before scheduling a new event—the last event to be scheduled is not necessarily the last one to occur

21.56 tf_strrealdelputp(), tf_istrrealdelputp()

tf_strrealdelputp(), tf_istrrealdelputp()			
Synopsis:	Write a value to a parameter from string value specification, using a real number delay.		
Syntax:	tf_strrealdelputp(nparam, bitlength, format, value_p, realdelay, delaytype) tf_istrrealdelputp(nparam, bitlength, format, value_p, realdelay, delaytype, instance_p)		
Returns:		Type	Description
		int	1 if successful; 0 if an error is detected
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter
	int	bitlength	Number of bits the value represents
	int	format	A character in single quotes representing the radix (base) of the value
	quoted string or char *	value_p	Quoted character string or pointer to a character string with the value to be written
	double	realdelay	Double-precision value representing the time delay before the value shall be written to the parameter
	int	delaytype	Integer code representing the delay mode for applying the value
Related routines:		char *	instance_p Pointer to a specific instance of a user-defined system task or function
Use tf_strdelputp() or tf_istrdelputp() for 32-bit integer delays Use tf_strlongdelputp() or tf_istrlongdelputp() for 64-bit integer delays Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function			

The TF routines **tf_strrealdelputp()** and **tf_istrrealdelputp()** shall write a string value to the specified parameter of the current instance or a specific instance of a user-defined system task. The actual change to the parameter shall be scheduled as an event on the parameter in the Verilog model at a future simulation time.

The *bitlength* argument shall define the value size in bits.

The *format* shall define the format of the value specified by *value_p* and shall be one of the characters given in Table 21-15.

Table 21-15—Format characters

Format character	Description
'b' or 'B'	Value is in binary
'o' or 'O'	Value is in octal
'd' or 'D'	Value is in decimal
'h' or 'H'	Value is in hexadecimal

The *delay* argument shall represent the amount of time before the value shall be applied to the parameter, and it shall be greater than or equal to 0. The delay shall assume the timescale units of the module containing the instance of the user-defined system task or function.

The *delaytype* argument shall determine how the value shall be scheduled in relation to other simulation events on the same register. The *delaytype* shall be one of integer values shown in Table 21-16.

Table 21-16—delaytype codes

delaytype code	Definition	Description
0	Inertial delay	All scheduled events on the output parameter in the Verilog model are removed before scheduling a new event
1	Modified transport delay	All events that are scheduled for times later than the new event on the output parameter in the Verilog model are removed before scheduling a new event
2	Pure transport delay	No scheduled events on the output parameter in the Verilog model are removed before scheduling a new event—the last event to be scheduled is not necessarily the last one to occur

21.57 tf_subtract_long()

tf_subtract_long()			
Synopsis:	Subtract two 64-bit integers.		
Syntax:	tf_subtract_long(&aof_low1, &aof_high1, low2, high2)		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	int *	aof_low1	Pointer to least significant 32 bits of first operand
	int *	aof_high1	Pointer to most significant 32 bits of first operand
	int	low2	Least significant 32 bits of second operand
	int	high2	Most significant 32 bits of second operand
Related routines:	Use tf_add_long() to add two 64-bit integers Use tf_multiply_long() to multiply two 64-bit integers Use tf_divide_long() to divide two 64-bit integers Use tf_compare_long() to compare two 64-bit integers		

The TF routine **tf_subtract_long()** shall subtract two 64-bit values. After calling **tf_subtract_long()**, the variables used to pass the first operand shall contain the results of the subtraction. The operands shall be assumed to be in two's complement form. Figure 21-10 shows the high and low 32 bits of two long integers and how **tf_subtract_long()** shall subtract them.

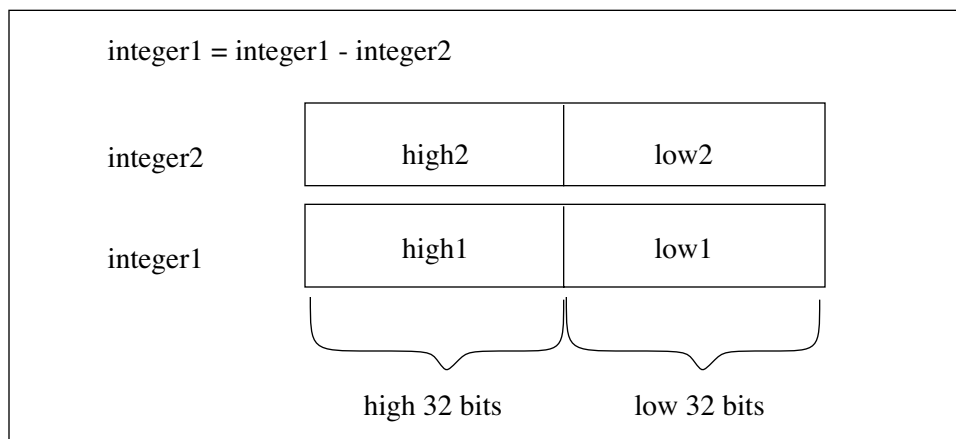


Figure 21-10—Subtracting with tf_subtract_long()

The example program fragment shown in Figure 21-11 uses **tf_subtract_long()** to calculate the relative time from the current time to the next event time (this example assumes that the code is executed during a misctf application call with reason of **reason_rosynch**).

The text message generated by this example is split off the **io_printf()** calls. If done in a single **io_printf()**, the second call to **tf_longtime_tostr()** would overwrite the string from the first call, since the string is placed in a temporary buffer.

```
{
    int currlow, currhigh;
    int relalow, relahigh;

    currlow = tf_getlongtime(&currhigh);
    io_printf("At time %s: ", tf_longtime_tostr(currlow, currhigh));
    if(tf_getnextlongtime(&relalow, &relahigh) == 0)
    {
        tf_subtract_long(&relalow, &relahigh, currlow, currhigh);
        io_printf("relative time to next event is %s",
            tf_longtime_tostr(relalow, relahigh));
    }
    else
        printf("there are no future events");
}
```

Figure 21-11—Using `tf_subtract_long()`

21.58 tf_synchronize(), tf_isynchronize()

tf_synchronize(), tf_isynchronize()			
Synopsis:	Synchronize to end of simulation time step.		
Syntax:	<pre>tf_synchronize() tf_isynchronize(instance_p)</pre>		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_rosynchronize() for read-only synchronization Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function Use tf_getnextlongtime() to get next time at which a simulation event is scheduled		

The TF routines **tf_synchronize()** and **tf_isynchronize()** shall schedule a callback to the misctf application associated with the current instance or a specific instance of a user-defined system task or function. The misctf application shall be called with a reason of **reason_synch** at the end of the current simulation time step.

The routines **tf_synchronize()** and **tf_rosynchronize()** have different functionality. The routine **tf_synchronize()** shall call the associated misctf application at the end of the current simulation time step with **reason_synch**, and the misctf application shall be allowed to schedule additional simulation events using routines such as **tf_strdelputp()**.

The routine **tf_rosynchronize()** shall call the associated misctf application at the end of the current simulation time step with **reason_rosynch**, and the PLI shall not be allowed to schedule any new events. This guarantees that all simulation events for the current time are completed. Calls to routines such as **tf_strdelputp()** and **tf_setdelay()** are illegal during processing of the misctf application with reason **reason_rosynch**.

The routine **tf_getnextlongtime()** shall only return the next simulation time for which an event is scheduled when used in conjunction with the routines **tf_rosynchronize()** and **tf_irosynchronize()**.

21.59 tf_testpvc_flag(), tf_itestpvc_flag()

tf_testpvc_flag(), tf_itestpvc_flag()			
Synopsis:	Test parameter value change flags.		
Syntax:	<pre>tf_testpvc_flag(nparam) tf_itestpvc_flag(nparam, instance_p)</pre>		
Type		Description	
Returns:	int	The value of the saved pvc flag	
Type		Name	Description
Arguments:	int	nparam	Index number of the user-defined system task or function parameter, or -1
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_asynchon() or tf_iasynchon() to enable pvc flags Use tf_getpchange() or tf_igetpchange() to get the index number of the parameter that changed Use tf_copypvc_flag() or tf_icopypvc_flag() to copy a pvc flag to the saved pvc flag Use tf_movepvc_flag() or tf_imovepvc_flag() to move a pvc flag to the saved pvc flag Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_testpvc_flag()** and **tf_itestpvc_flag()** shall return value of the saved pvc flag. The argument *nparam* shall be the index number of a parameter in a specific instance of a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1. If *nparam* is **-1**, then all parameter pvc flags shall be tested and the logical OR of all saved flags returned.

PVC flags shall be used to indicate whether a particular user-defined system task or function parameter has changed value. Each parameter shall have two pvc flags: a current pvc flag, which shall be set by a software product when the change occurs, and a saved pvc flag, which shall be controlled by the user.

NOTE—PVC flags shall not be set by the software product until **tf_asynchon()** or **tf_iasynchon()** has been called.

21.60 tf_text()

tf_text()			
Synopsis:	Store error message information.		
Syntax:	tf_text(message, arg1,...arg5)		
Type		Description	
Returns:	void		
Arguments:	Type	Name	Description
	quoted string or char *	message	A quoted character string or pointer to a character string with a message to be stored
		arg1...arg5	One to five arguments used in the format control string; the type of each argument should be consistent with how it is used in the message string
Related routines:	Use tf_message() to display the stored error message		

The TF routine **tf_text()** shall store text messages about an error in a buffer, which will be printed when the routine **tf_message()** is called. The routine shall provide a method for a PLI application to store information about one or more errors before it calls the **tf_message()** TF routine. This allows an application to process all of a routine, such as syntax checking, before calling **tf_message()**, which can be set to abort processing after printing messages. An application shall be able to call **tf_text()** any number of times before it calls **tf_message()**.

When the application calls **tf_message()**, the information stored by **tf_text()** shall be displayed before the information in the call to **tf_message()**. Each call to **tf_message()** shall clear the buffer where **tf_text()** stores its information.

The *message* argument is a user-defined control string containing the message to be displayed. The control string uses the same formatting controls as the C `printf()` function (for example, %d). The message shall use up to a maximum of five variable arguments. There shall be no limit to the length of a variable argument. Formatting characters, such as \n, \t, \b, \f, or \r, do not need to be included in the message—the software product shall automatically format each message.

An example of using **tf_text()** and **tf_message()** calls and the output generated follow. Note that the format of the output shall be defined by the software product.

Calling **tf_text()** and **tf_message()** with the arguments:

```
tf_text ("Argument number %d", argnum);
...
tf_message(ERR_ERROR, "User", TFARG",
           " is illegal in task %s", taskname);
```

Might produce the output:

```
ERROR!   Argument number 2 is illegal in task      [User-TFARG]
          $usertask
```


21.61 tf_typep(), tf_itypep()

tf_typep(). tf_itypep()			
Synopsis:	Get a parameter type.		
Syntax:	tf_typep(nparam) tf_itypep(nparam, instance_p)		
Type		Description	
Returns:	int	A predefined integer constant representing the Verilog HDL data type for the parameter	
Arguments:	Type	Name	Description
	int	nparam	Index number of the user-defined system task or function parameter
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_typep()** and **tf_itypep()** shall return an integer constant indicating the type of a parameter for the current instance or a specific instance of a user-defined system task or function. The integer constants shall be as shown in Table 21-17.

Table 21-17 — Predefined tf_typep() constants

Predefined constant	Description
tf_nullparam	The parameter is a <i>null</i> expression (where no text has been given as the parameter), or <i>nparam</i> is out of range
tf_string	The parameter is a literal string
tf_readonly	The parameter is a vector expression with a value that can be read but not written
tf_readwrite	The parameter is a vector expression with a value that can be read and written
tf_readonlyreal	The parameter is a real number expression with a value that can be read but not written
tf_readwritereal	The parameter is a real number expression with a value that can be read and written

- A *read only* expression shall be any expression that would be illegal as a left-hand-side construct in a Verilog HDL procedural assignment (e.g., an expression using *net* data types or *event* data types)
- A *read/write* expression shall be any expression that would be legal as a left-hand-side construct in a Verilog HDL procedural assignments (e.g., an expression using *reg*, *integer*, *time*, or *real* data types)

The argument *nparam* shall be the index number of a parameter in a user-defined system task or function. Task/function parameter index numbering shall proceed from left to right, with the leftmost parameter being number 1.

21.62 tf_unscale_longdelay()

tf_unscale_longdelay()			
Synopsis:	Convert a delay from internal simulation time units to the timescale of a particular module.		
Syntax:	<pre>tf_unscale_longdelay(instance_p, delay_lo, delay_hi, &aof_delay_lo, &aof_delay_hi)</pre>		
Type		Description	
Returns:	void		
Arguments:	Type	Name	Description
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
	int	delay_lo	Least significant (rightmost) 32 bits of the delay to be converted
	int	delay_hi	Most significant (leftmost) 32 bits of the delay to be converted
	int *	aof_delay_lo	Pointer to a variable to store the least significant (rightmost) 32 bits of the conversion result
	int *	aof_delay_hi	Pointer to a variable to store the most significant (leftmost) 32 bits of the conversion result
Related routines:	Use tf_unscale_realdelay() to unscale real number delays Use tf_scale_longdelay() to convert a delay to the simulation time unit Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routine **tf_unscale_longdelay()** shall convert a long (64-bit) integer delay expressed in internal simulation time into the time units of the module containing the user-defined system task or function referenced by the *instance_p* pointer. The argument *aof_delay_lo* and *aof_delay_hi* shall contain the address of the converted delay returned by the routine.

21.63 tf_unscale_realdelay()

tf_unscale_realdelay()			
Synopsis:	Convert a delay expressed in internal simulation time units to the timescale of a particular module.		
Syntax:	tf_unscale_realdelay(instance_p, realdelay, &aof_realdelay)		
Returns:	Type		Description
	void		
Arguments:	Type	Name	Description
	char *	instance_p	Pointer to a specific instance of a user-defined system task or function
	double	delay	Value of the delay to be converted
	double *	aof_realdelay	Pointer to a variable to store the conversion result
Related routines:	Use tf_unscale_longdelay() to unscale 64-bit integer delays Use tf_scale_realdelay() to convert a delay to the simulation time unit Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routine **tf_unscale_realdelay()** shall convert a double-precision delay expressed in internal simulation time into the time units of the module containing the user-defined system task or function referenced by the *instance_p* pointer. The argument *aof_realdelay* shall contain the address of the converted delay returned by the routine.

21.64 tf_warning()

tf_warning()			
Synopsis:	Report a warning message.		
Syntax:	tf_warning(format, arg1,...arg5)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	quoted string or char *	format	A quoted character string or pointer to a character string that controls the message to be written
		arg1...arg5	One to five arguments used in the format control string; the type of each argument should be consistent with how it is used in the format string
Related routines:	Use tf_message() to write warning messages with additional format control Use tf_error() to write a warning message Use io_printf() or io_mcdprintf() to write a formatted message		

The TF routine **tf_warning()** shall provide a warning reporting mechanism compatible with warning messages generated by the software product.

- The *format* control string shall use the same formatting controls as the C *printf()* function (for example, %d).
- The maximum number of arguments that shall be used in the format control string is 5.
- The location information (file name and line number) of the current instance of the user-defined system task or function shall be appended to the message using a format compatible with error messages generated by the software product.
- The message shall be written to both the standard output text window and the output log file of the product.

The **tf_warning()** routine shall not abort parsing or compilation of Verilog HDL source code.

21.65 **tf_write_save()**

tf_write_save()			
Synopsis:	Append a block of data to a save file.		
Syntax:	tf_write_save(blockptr, blocklen)		
Returns:	Type	Description	
	int	Nonzero value if successful, zero if an error is encountered	
Arguments:	Type	Name	Description
	char *	blockptr	Pointer to the first byte of the block of data to be saved
	int	blocklen	Number of bytes are to be saved
Related routines:	Use tf_read_restore() to retrieve the data saved		

The TF routine **tf_write_save()** shall write user-defined data to the end of a save file being written by the **\$save** built-in system task. This routine shall be called from the misc tf application when misc tf is invoked with **reason_save**.

The argument *blockptr* shall be a pointer to an allocated block of memory containing the data to be saved.

The argument *blocklen* shall be the length in bytes of the allocated block of memory. Note that exactly as many bytes shall be restored using **tf_read_restore()** as were written with **tf_write_save()**.

Section 22

Using VPI routines

Sections 22 and 23 specify the Verilog Procedural Interface (VPI) for the Verilog HDL. This section describes how the VPI routines are used, and Section 23 defines each of the routines in alphabetical order.

22.1 The VPI interface

The VPI interface provides routines that allow Verilog product users to access information contained in a Verilog design, and that allow facilities to interact dynamically with a software product. Applications of the VPI interface can include delay calculators and annotators, connecting a Verilog simulator with other simulation and CAE systems, and customized debugging tasks.

The functions of the VPI interface can be grouped into two main areas:

- Dynamic software product interaction using VPI callbacks
- Access to Verilog HDL objects and simulation specific objects

22.1.1 VPI callbacks

Dynamic software product interaction shall be accomplished with a registered callback mechanism. VPI callbacks shall allow a user to request that a Verilog HDL software product, such as a logic simulator, call a user-defined application when a specific activity occurs. For example, the user can request that the user application `my_monitor()` be called when a particular net changes value, or that `my_cleanup()` be called when the software product execution has completed.

The VPI callback facility shall provide the user with the means to interact dynamically with a software product, detecting the occurrence of value changes, advancement of time, end of simulation, etc. This feature allows applications such as integration with other simulation systems, specialized timing checks, complex debugging features, etc.

The reasons for which callbacks shall be provided can be separated into four categories:

- *Simulation event* (e.g., a value change on a net or a behavioral statement execution)
- *Simulation time* (e.g., the end of a time queue or after certain amount of time)
- *Simulator action/feature* (e.g., the end of compile, end of simulation, restart, or enter interactive mode)
- *User-defined system task or function execution*

VPI callbacks shall be registered by the user with the functions `vpi_register_cb()` and `vpi_register_systf()`. These routines indicate the specific reason for the callback, the application to be called, and what system and user data shall be passed to the callback application when the callback occurs. A facility is also provided to call the callback functions when a Verilog HDL product is first invoked. A primary use of this facility shall be for registration of user-defined system tasks and functions.

22.1.2 VPI access to Verilog HDL objects and simulation objects

Accessible Verilog HDL objects and simulation objects and their relationships and properties are described using data model diagrams. These diagrams are presented in 22.5. The data diagrams indicate the routines and constants that are required to access and manipulate objects within an application environment. An associated set of routines to access these objects is defined in Section 23.

The VPI interface also includes a set of utility routines for functions such as handle comparison, file handling, and redirected printing, which are described in 23.12.

VPI routines provide access to objects in an *instantiated* Verilog design. An instantiated design is one where each instance of an object is uniquely accessible. For instance, if a module *m* contains wire *w* and is instantiated twice as *m1* and *m2*, then *m1.w* and *m2.w* are two distinct objects, each with its own set of related objects and properties.

The VPI interface is designed as a *simulation* interface, with access to both Verilog HDL objects and specific simulation objects. This simulation interface is different from a hierarchical language interface, which would provide access to HDL information but would not provide information about simulation objects.

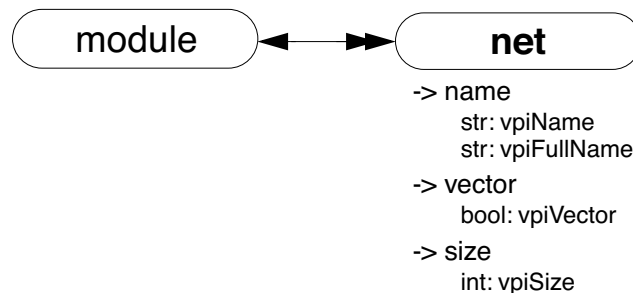
22.1.3 Error handling

To determine if an error occurred, the routine **vpi_chk_error()** shall be provided. The **vpi_chk_error()** routine shall return a nonzero value if an error occurred in the previously called VPI routine. Callbacks can be set up for when an error occurs as well. The **vpi_chk_error()** routine can provide detailed information about the error.

22.2 VPI object classifications

VPI objects are classified with data model diagrams. These diagrams provide a graphical representation of those objects within a Verilog design to which the VPI routines shall provide access. The diagrams shall show the relationships between objects and the properties of each object. Objects with sufficient commonality are placed in groups. Group relationships and properties apply to all the objects in the group.

As an example, this simplified diagram shows that there is a *one-to-many relationships* from objects of type **module** to objects of type **net**, and a *one-to-one relationship* from objects of type **net** to objects of type **module**. Objects of type **net** have properties **vpiName**, **vpiVector**, and **vpiSize**, with C data types string, Boolean, and integer respectively.



The VPI object data diagrams are presented in 22.5.

22.2.1 Accessing object relationships and properties

The VPI interface defines the C data type of **vpiHandle**. All objects are manipulated via a **vpiHandle** variable. Object handles can be accessed from a relationship with another object, or from a hierarchical name, as the following example demonstrates:

```

vpiHandle net;
net = vpi_handle_by_name("top.m1.w1", NULL);

```

This example call retrieves a handle to wire `top.m1.w1` and assigns it to the **vpiHandle** variable `net`. The `NULL` second argument directs the routine to search for the name from the top level of the design.

The VPI interface provides generic functions for tasks, such as traversing relationships and determining property values. One-to-one relationships are traversed with routine **vpi_handle()**. In the following example, the module that contains `net` is derived from a handle to that net:

```

vpiHandle net, mod;
net = vpi_handle_by_name("top.m1.w1", NULL);
mod = vpi_handle(vpiModule, net);

```

The call to **vpi_handle()** in the above example shall return a handle to module `top.m1`.

Properties of objects shall be derived with routines in the `vpi_get` family. The routine **vpi_get()** returns integer and Boolean properties. The routine **vpi_get_str()** accesses string properties. To retrieve a pointer to the full hierarchical name of the object referenced by handle `mod`, the following call would be made:

```

char *name = vpi_get_str(vpiFullName, mod);

```

In the above example, character pointer `name` shall now point to the string `"top.m1"`.

One-to-many relationships are traversed with an iteration mechanism. The routine **vpi_iterate()** creates an object of type **vpi_iterator**, which is then passed to the routine **vpi_scan()** to traverse the desired objects. In the following example, each net in module `top.m1` is displayed:

```

vpiHandle itr;
itr = vpi_iterate(vpiNet, mod);
while (net = vpi_scan(itr) )
    vpi_printf("\t%s\n", vpi_get_str(vpiFullName, net) );

```

As the above examples illustrate, the routine naming convention is a *'vpi'* prefix with *'_'* word delimiters (with the exception of callback-related defined values, which use the *'cb'* prefix). Macro-defined types and properties have the *'vpi'* prefix, and they use capitalization for word delimiters.

The routines for traversing Verilog HDL structures and accessing objects are described in Section 23.

22.2.2 Delays and values

Most properties are of type integer, Boolean, or string. Delay and logic value properties, however, are more complex and require specialized routines and associated structures. The routines **vpi_get_delays()** and **vpi_put_delays()** use structure pointers, where the structure contains the pertinent information about delays. Similarly, simulation values are also handled with the routines **vpi_get_value()** and **vpi_put_value()**, along with an associated set of structures.

The routines and C structures for handling delays and logic values are presented in Section 23.

22.3 List of VPI routines by functional category

The VPI routines can be divided into groups based on primary functionality.

- VPI routines for simulation-related callbacks
- VPI routines for system task/function callbacks
- VPI routines for traversing Verilog HDL hierarchy
- VPI routines for accessing properties of objects
- VPI routines for accessing objects from properties
- VPI routines for delay processing

- VPI routines for logic and strength value processing
- VPI routines for simulation time processing
- VPI routines for miscellaneous utilities

Tables 22-1 through 22-9 list the VPI routines by major category. Section 23 defines each of the VPI routines, listed in alphabetical order.

Table 22-1—VPI routines for simulation related callbacks

To	Use
Register a simulation-related callback	vpi_register_cb()
Remove a simulation-related callback	vpi_remove_cb()
Get information about a simulation-related callback	vpi_get_cb_info()

Table 22-2—VPI routines for system task/function callbacks

To	Use
Register a system task/function callback	vpi_register_systf()
Get information about a system task/function callback	vpi_get_systf_info()

Table 22-3—VPI routines for traversing Verilog HDL hierarchy

To	Use
Obtain a handle for an object with a one-to-one relationship	vpi_handle()
Obtain handles for objects in a one-to-many relationship	vpi_iterate() vpi_scan()
Obtain a handles for an object in a many-to-one relationship	vpi_handle_multi()

Table 22-4—VPI routines for accessing properties of objects

To	Use
Get the value of objects with types of <code>int</code> or <code>bool</code>	vpi_get()
Get the value of objects with types of <code>string</code>	vpi_get_str()

Table 22-5—VPI routines for accessing objects from properties

To	Use
Obtain a handle for a named object	vpi_handle_by_name()
Obtain a handle for an indexed object	vpi_handle_by_index()

Table 22-6—VPI routines for delay processing

To	Use
Retrieve delays or timing limits of an object	vpi_get_delays()
Write delays or timing limits to an object	vpi_put_delays()

Table 22-7—VPI routines for logic and strength value processing

To	Use
Retrieve logic value or strength value of an object	vpi_get_value()
Write logic value or strength value to an object	vpi_put_value()

Table 22-8—VPI routines for simulation time processing

To	Use
Find the current simulation time or the scheduled time of future events	vpi_get_time()

Table 22-9—VPI routines for miscellaneous utilities

To	Use
Write to <code>stdout</code> and the current log file	vpi_printf()
Open a file for writing	vpi_mcd_open()
Close one or more files	vpi_mcd_close()
Write to one or more files	vpi_mcd_printf()
Retrieve the name of an open file	vpi_mcd_name()
Retrieve data about product invocation options	vpi_get_vlog_info()


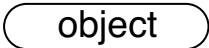
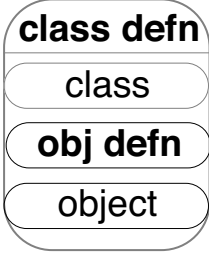
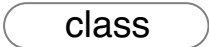
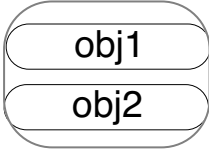
Table 22-9—VPI routines for miscellaneous utilities (continued)

To	Use
See if two handles refer to the same object	vpi_compare_objects()
Obtain error status and error information about the previous call to a VPI routine	vpi_chk_error()
Free memory allocated by VPI routines	vpi_free_object()

22.4 Key to object model diagrams

This clause contains the keys to the symbols used in the object model diagrams. Keys are provided for objects and classes, traversing relationships, and accessing properties.

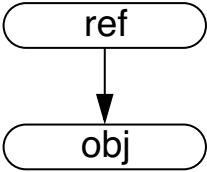
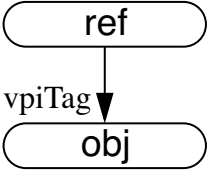
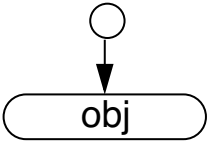
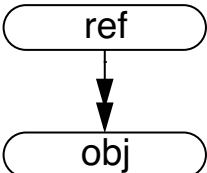
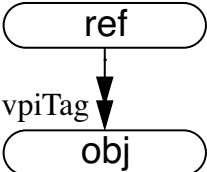
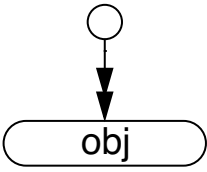
22.4.1 Diagram key for objects and classes

	Object Definition: Bold letters in a solid enclosure indicate an object definition. The properties of the object are defined in this location.
	Object Reference: Normal letters in a solid enclosure indicate an object reference.
	Class Definition: <i>Bold italic</i> letters in a dotted enclosure indicate a class definition, where the class groups other objects and classes. Properties of the class are defined in this location. The class definition can contain an object definition.
	Class Reference: <i>Italic</i> letters in a dotted enclosure indicate a class reference.
	Unnamed Class: A dotted enclosure with no name is an unnamed class. It is sometimes convenient to group objects although they shall not be referenced as a group elsewhere, so a name is not indicated.

22.4.2 Diagram key for accessing properties

<div>obj</div> <div>-> vector bool: vpiVector</div> <div>-> size int: vpiSize</div>	<p>Integer and Boolean properties are accessed with the routine vpi_get().</p> <p>Example: Given a vpiHandle <code>obj_h</code> to an object of type vpiObj, get the size of the object.</p> <pre>bool vect_flag = vpi_get(vpiVector, obj_h); int size = vpi_get_size(vpiSize, obj_h);</pre>
<div>obj</div> <div>-> name str: vpiName str: vpiFullName</div>	<p>String properties are accessed with routine vpi_get_str().</p> <p>Example:</p> <pre>char name[nameSize]; vpi_get_str(vpiName, obj_h);</pre>
<div>object</div> <div>-> complex func1() func2()</div>	<p>Complex properties for time and logic value are accessed with the indicated routines. See the descriptions of the routines for usage.</p>

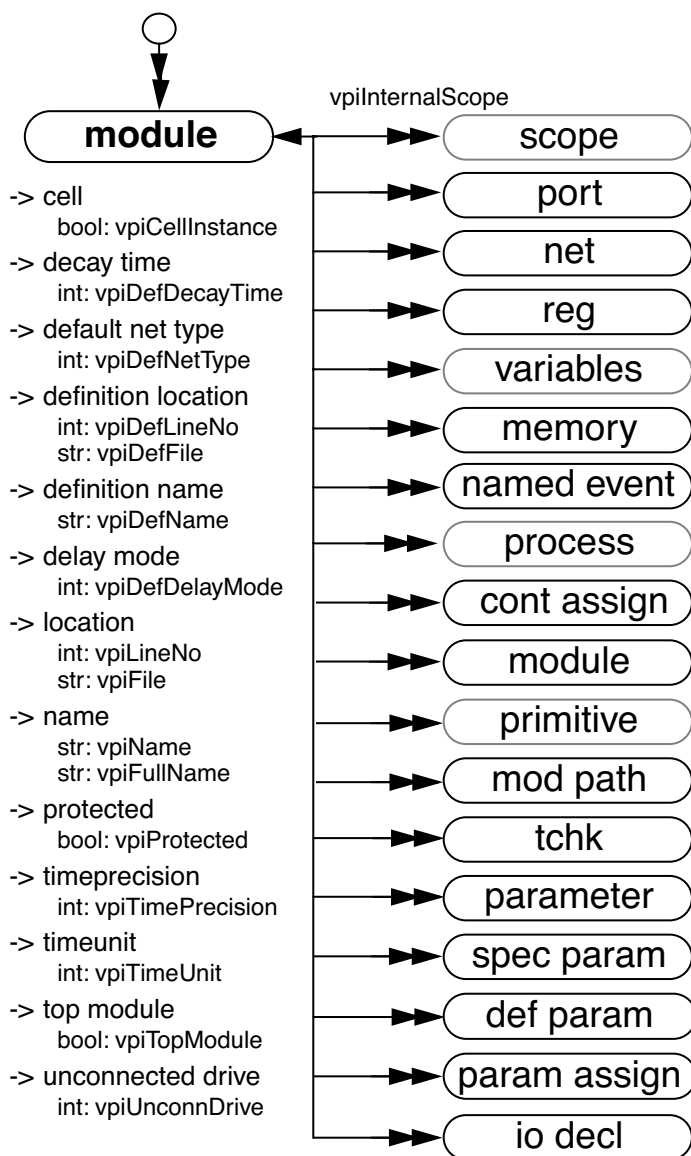
22.4.3 Diagram key for traversing relationships

	<p>A single arrow indicates a <i>one-to-one</i> relationship accessed with the routine vpi_handle().</p> <p>Example: Given vpiHandle variable <code>ref_h</code> of type <code>ref</code>, access <code>obj_h</code> of type vpiObj:</p> <pre>obj_h = vpi_handle(vpiObj, ref_h);</pre>
	<p>A tagged <i>one-to-one</i> relationship is traversed similarly, using vpiTag instead of vpiObj:</p> <p>Example:</p> <pre>obj_h = vpi_handle(vpiTag, ref_h);</pre>
	<p>A top-level <i>one-to-one</i> relationship is traversed similarly, using NULL instead of <code>ref_h</code>:</p> <p>Example:</p> <pre>obj_h = vpi_handle(vpiObj, NULL);</pre>
	<p>A double arrow indicates a <i>one-to-many</i> relationship accessed with the routine vpi_scan().</p> <p>Example: Given vpiHandle variable <code>ref_h</code> of type <code>ref</code>, scan objects of type vpiObj:</p> <pre>itr = vpi_iterate(vpiObj, ref_h); while (obj_h = vpi_scan(itr)) /* process 'obj_h' */</pre>
	<p>A tagged <i>one-to-many</i> relationship is traversed similarly, using vpiTag instead of vpiObj:</p> <p>Example:</p> <pre>itr = vpi_iterate(vpiTag, ref_h); while (obj_h = vpi_scan(itr)) /* process 'obj_h' */</pre>
	<p>A top-level <i>one-to-many</i> relationship is traversed similarly, using NULL instead of <code>ref_h</code>:</p> <p>Example:</p> <pre>itr = vpi_iterate(vpiObj, NULL); while (obj_h = vpi_scan(itr)) /* process 'obj_h' */</pre>

22.5 Object data model diagrams

Subclauses 22.5.1 through 22.5.21 contain the data model diagrams that define the accessible objects and groups of objects, along with their relationships and properties.

22.5.1 Module

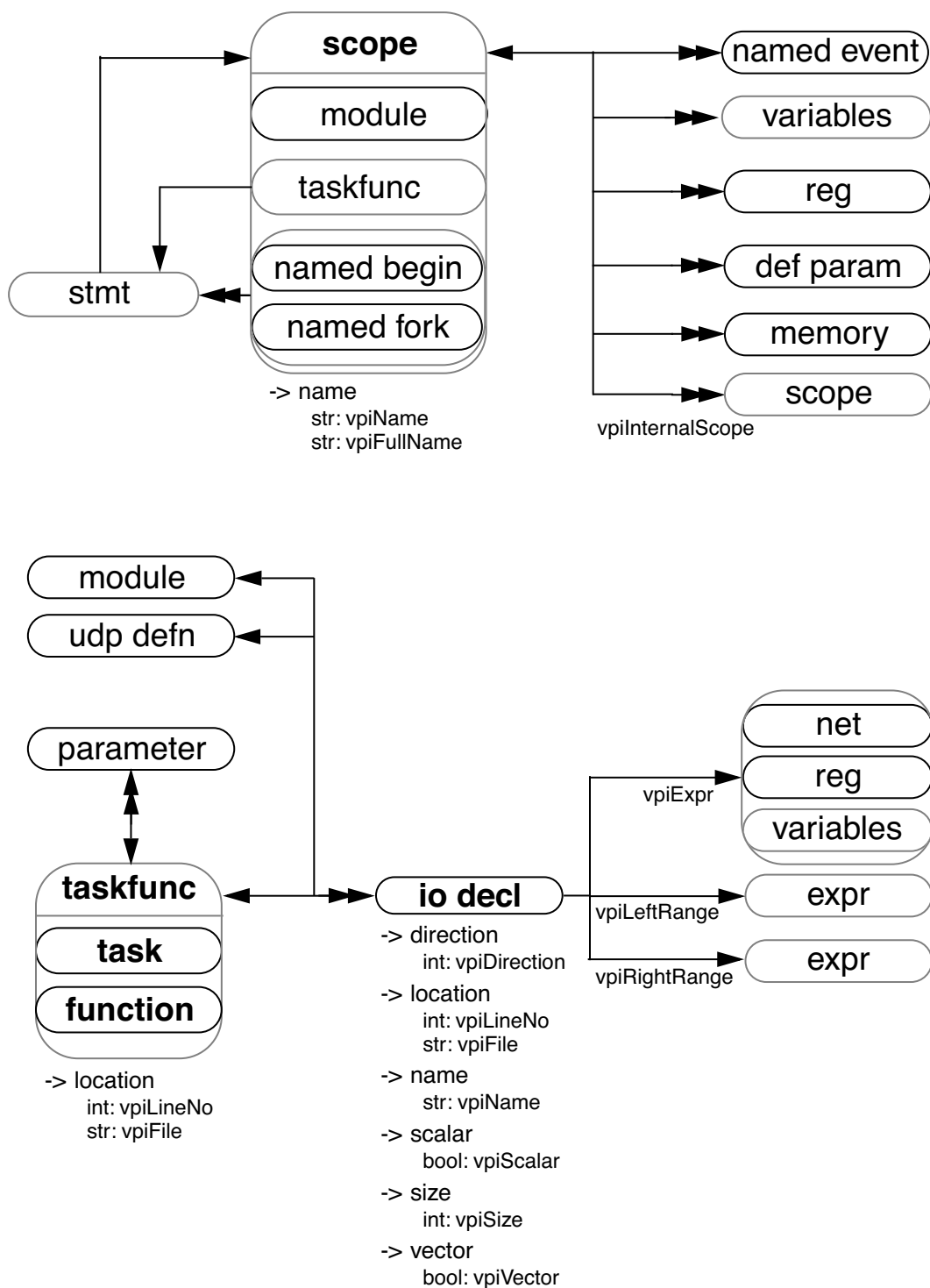


NOTES

1— Top-level modules shall be accessed using **vpi_iterate()** with a NULL reference object.

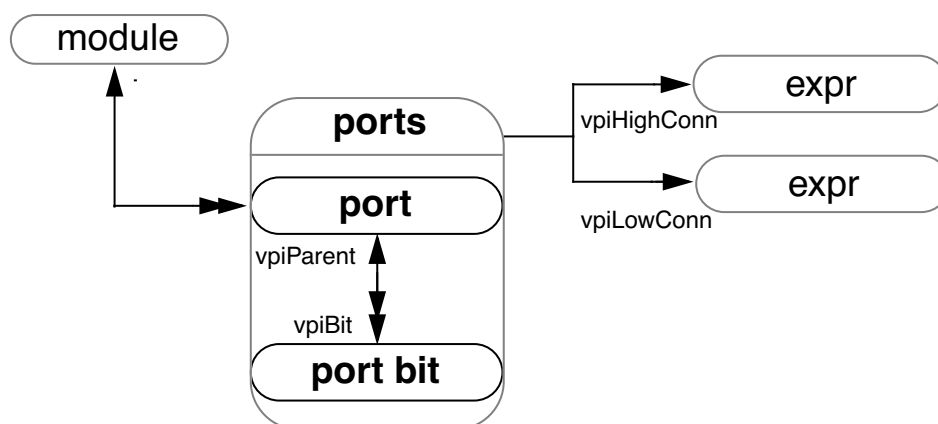
2— Passing a NULL handle to **vpi_get()** with types **vpiTimePrecision** or **vpiTimeUnit** shall return the smallest time precision of all modules in the instantiated design.

22.5.2 Scope, task, function, IO declaration



NOTE—A Verilog HDL function shall contain an object with the same name, size, and type as the function.

22.5.3 Ports

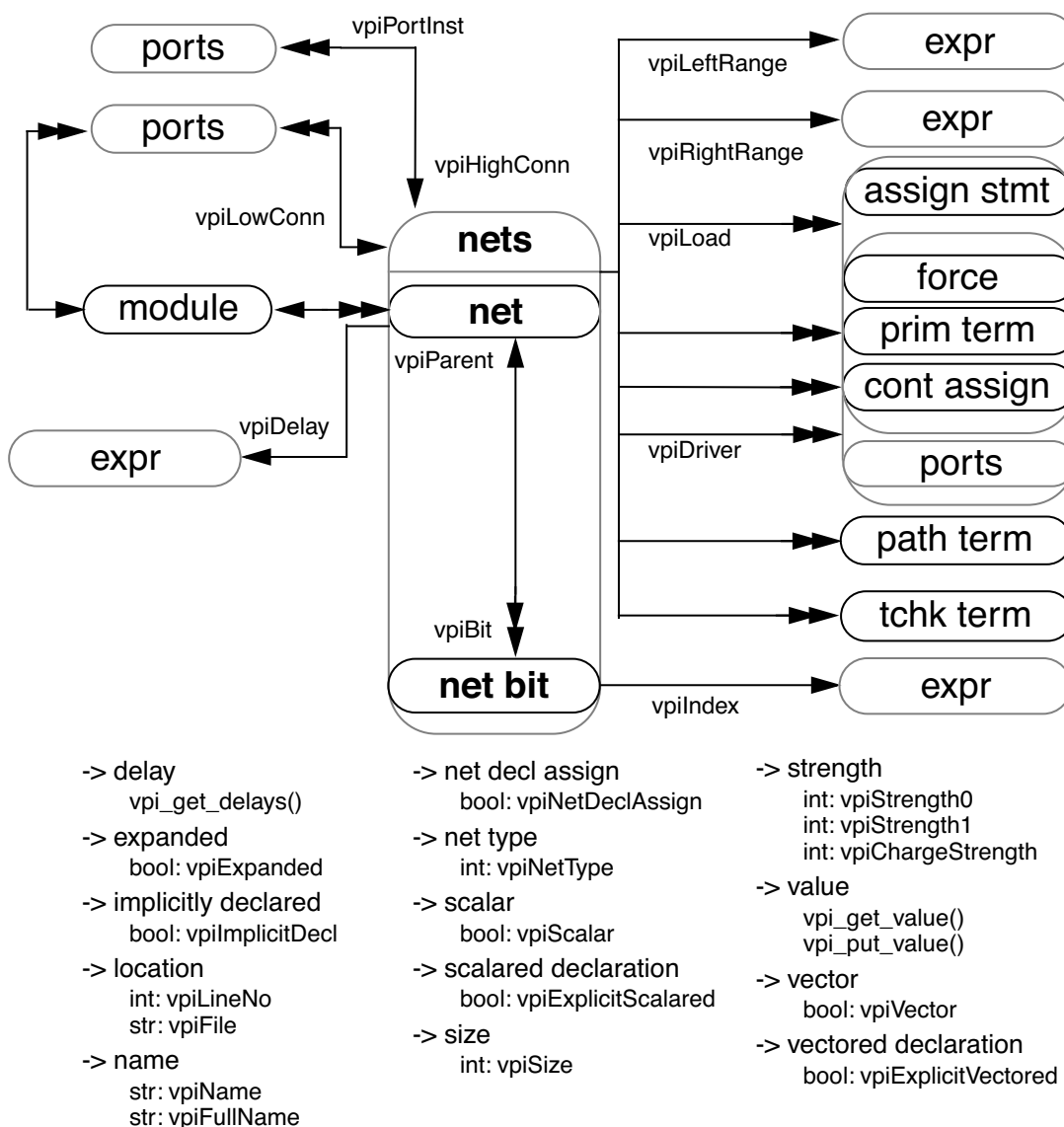


- > connected by name
bool: vpiConnByName
- > delay (mipd)
vpi_get_delays()
vpi_put_delays()
- > direction
int: vpiDirection
- > explicitly named
bool: vpiExplicitName
- > index
int: vpiPortIndex
- > location
int: vpiLineNo
str: vpiFile
- > name
str: vpiName
str: vpiFullName
- > scalar
bool: vpiScalar
- > size
int: vpiSize
- > vector
bool: vpiVector

NOTES

- 1— **vpiHighConn** shall indicate the hierarchically higher (closer to the top module) port connection.
- 2— **vpiLowConn** shall indicate the lower (further from the top module) port connection.
- 3— Properties *scalar* and *vector* shall indicate if the port is 1 bit or more than 1 bit. They shall not indicate anything about what is connected to the port.
- 4— Properties *index* and *name* shall not apply for port bits.
- 5— If a port is explicitly named, then the explicit name shall be returned. If not, and a name exists, then that name shall be returned. Otherwise, **NULL** shall be returned.
- 6— **vpiPortIndex** can be used to determine the port order.

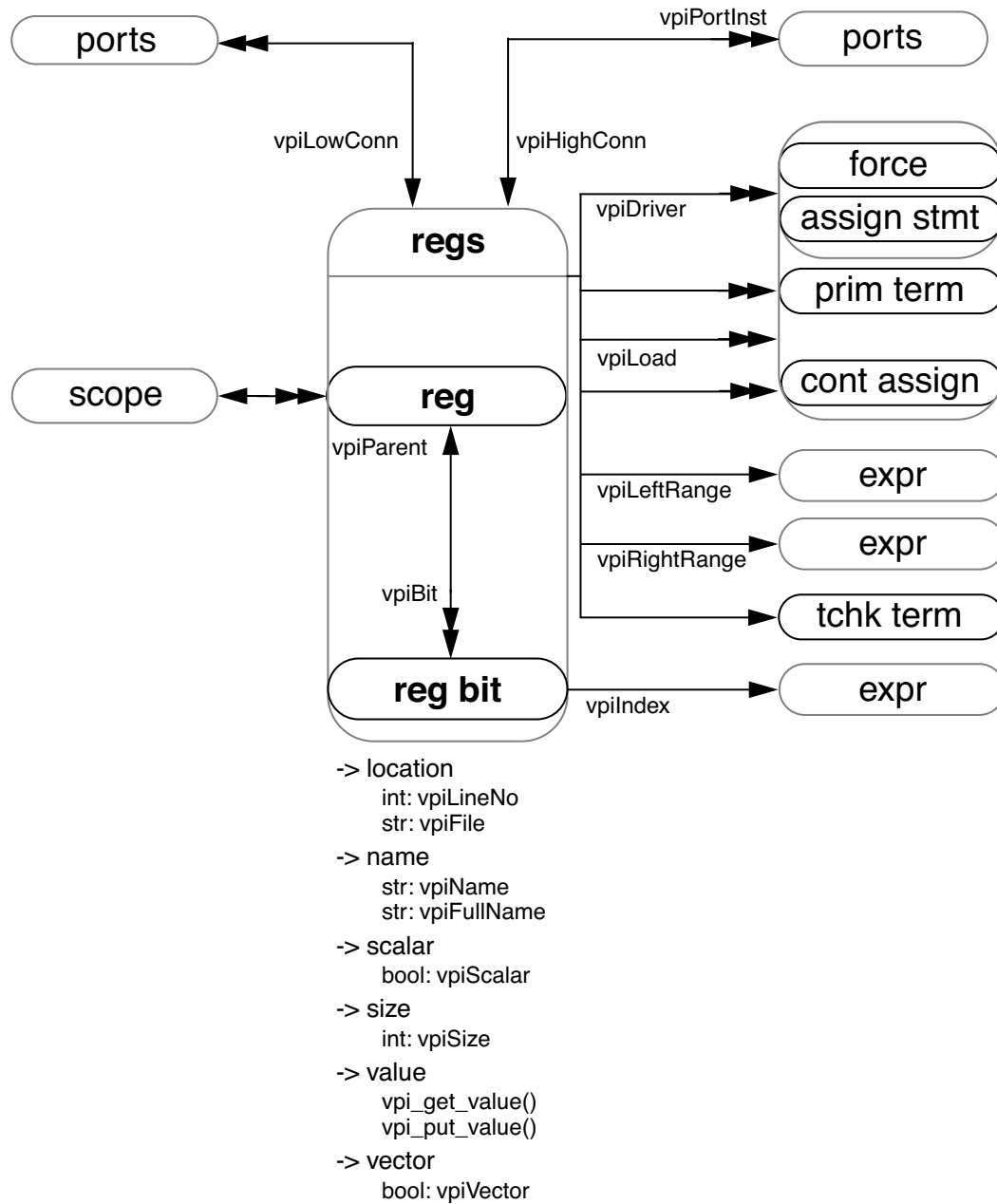
22.5.4 Nets



NOTES

- 1 — For vectors, net bits shall be available regardless of vector expansion.
- 2 — Continuous assignments and primitive terminals shall be accessed regardless of hierarchical boundaries.
- 3 — Continuous assignments and primitive terminals shall only be accessed from scalar nets or bit selects.
- 4 — For **vpiPortInst** and **vpiPort**, if the reference handle is a bit or the entire vector, the relationships shall return a handle to either a port bit or the entire port, respectively.
- 5 — For implicit nets, **vpiLineNo** shall return 0, and **vpiFile** shall return the filename where the implicit net is first referenced.
- 6 — Only active forces and assign statements shall be returned for **vpiLoad**.
- 7 — Only active forces shall be returned for **vpiDriver**.
- 8 — **vpiDriver** shall also return ports that are driven by objects other than nets and net bits.

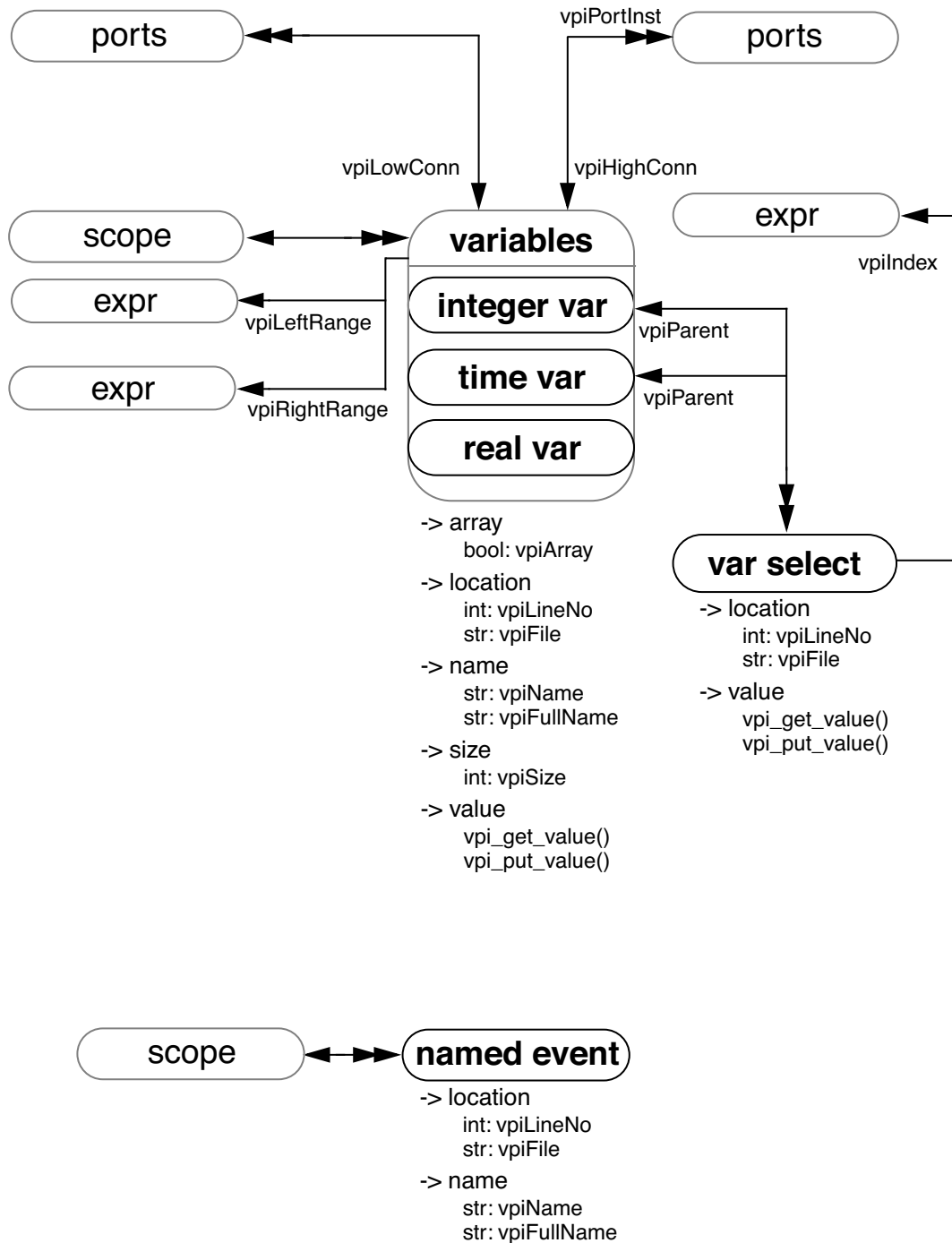
22.5.5 Regs



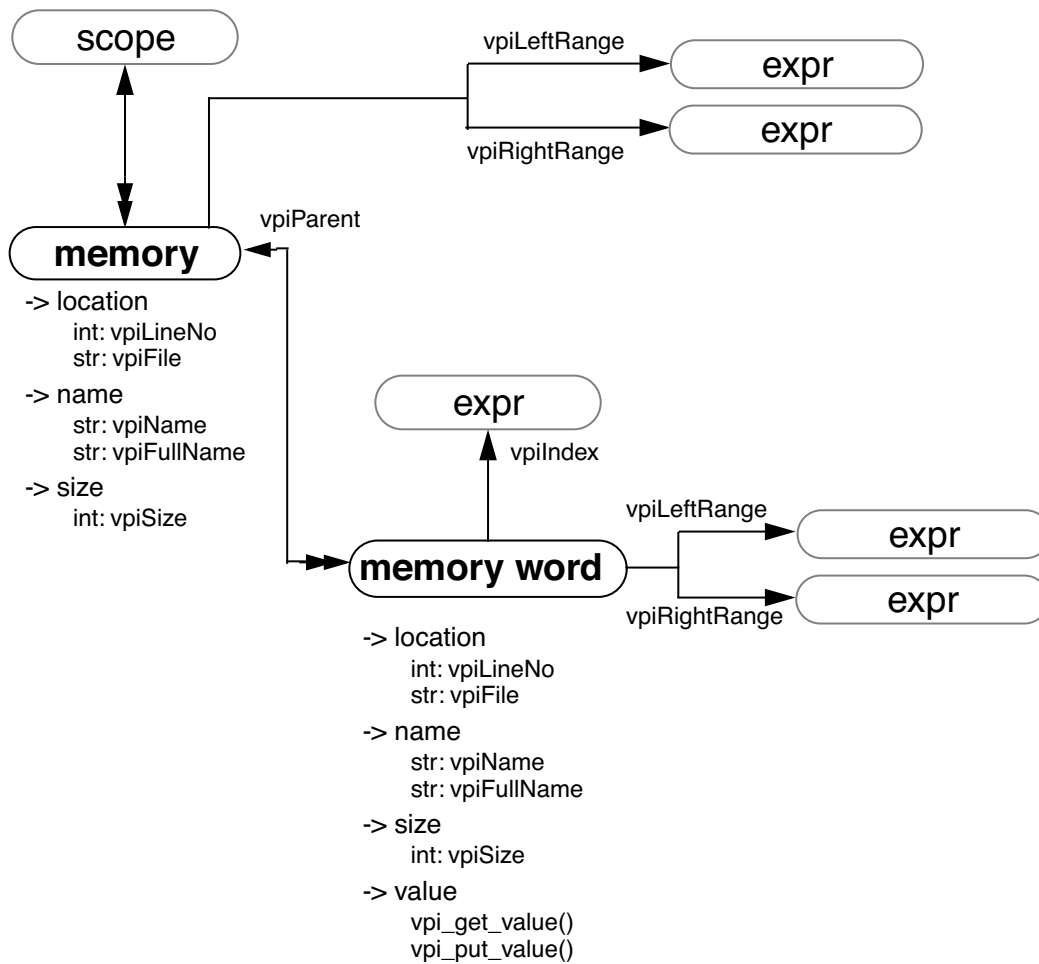
NOTES

- 1— Continuous assignments and primitive terminals shall be accessed regardless of hierarchical boundaries.
- 2— Continuous assignments and primitive terminals shall only be accessed from scalar regs and bit selects.
- 3— Only active forces and assign statements shall be returned for **vpiLoad** and **vpiDriver**.

22.5.6 Variables, named event



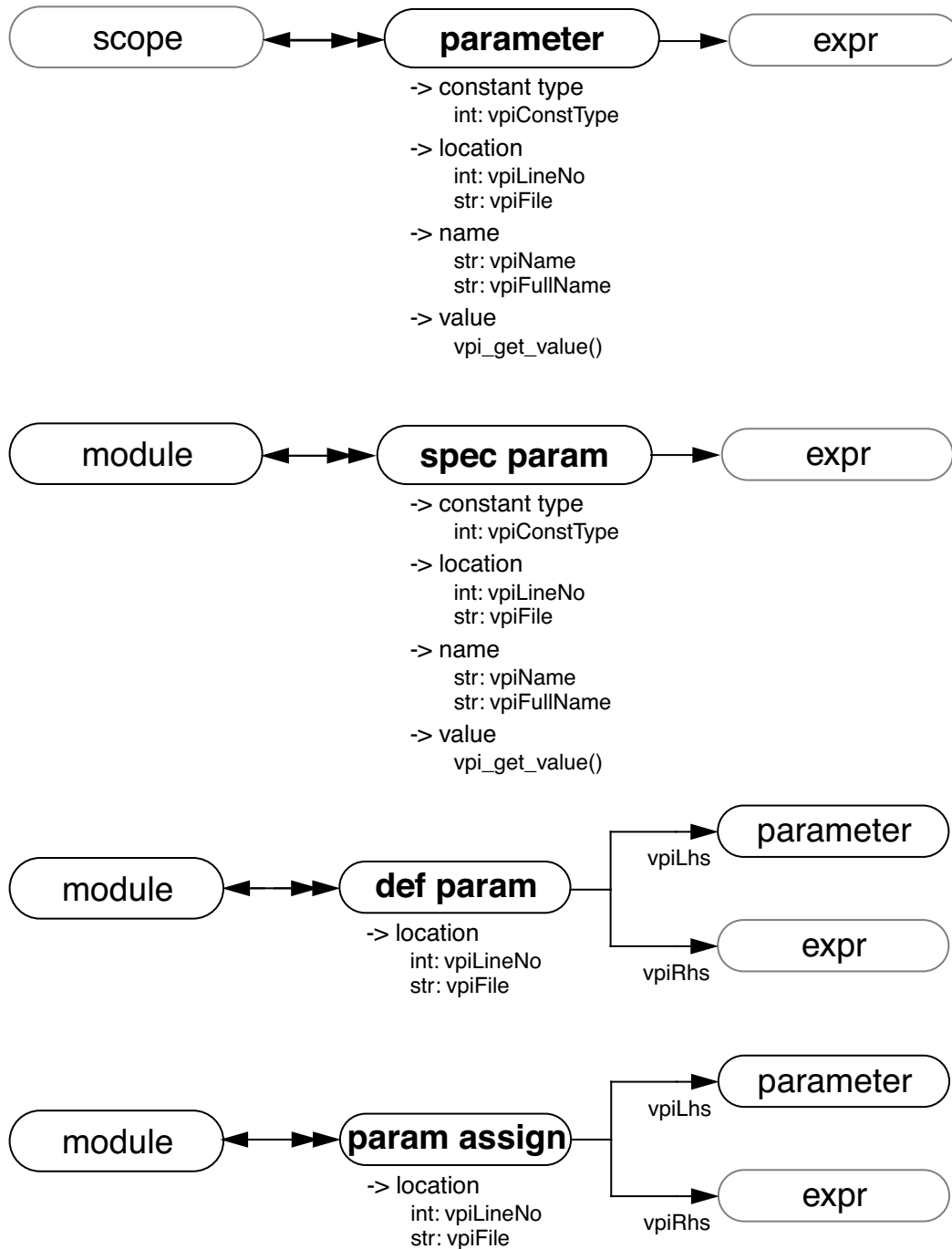
NOTE—**vpiLeftRange** and **vpiRightRange** shall be invalid for reals, since there cannot be arrays of reals.

22.5.7 Memory**NOTES**

1 — **vpiSize** for a memory shall return the number of words in the memory.

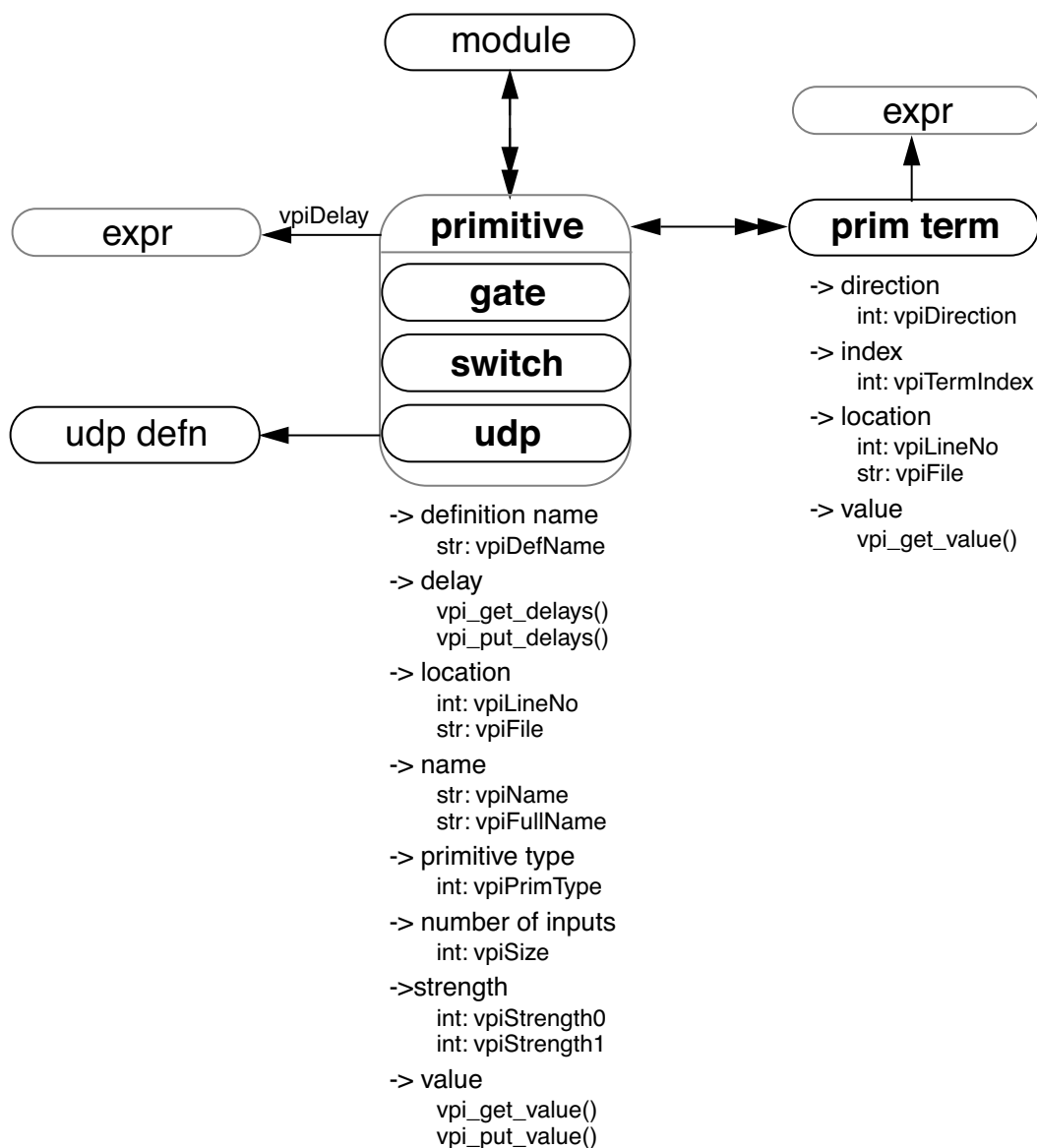
2 — **vpiSize** for a memory word shall return the number of bits in the word.

22.5.8 Parameter, specparam



NOTES

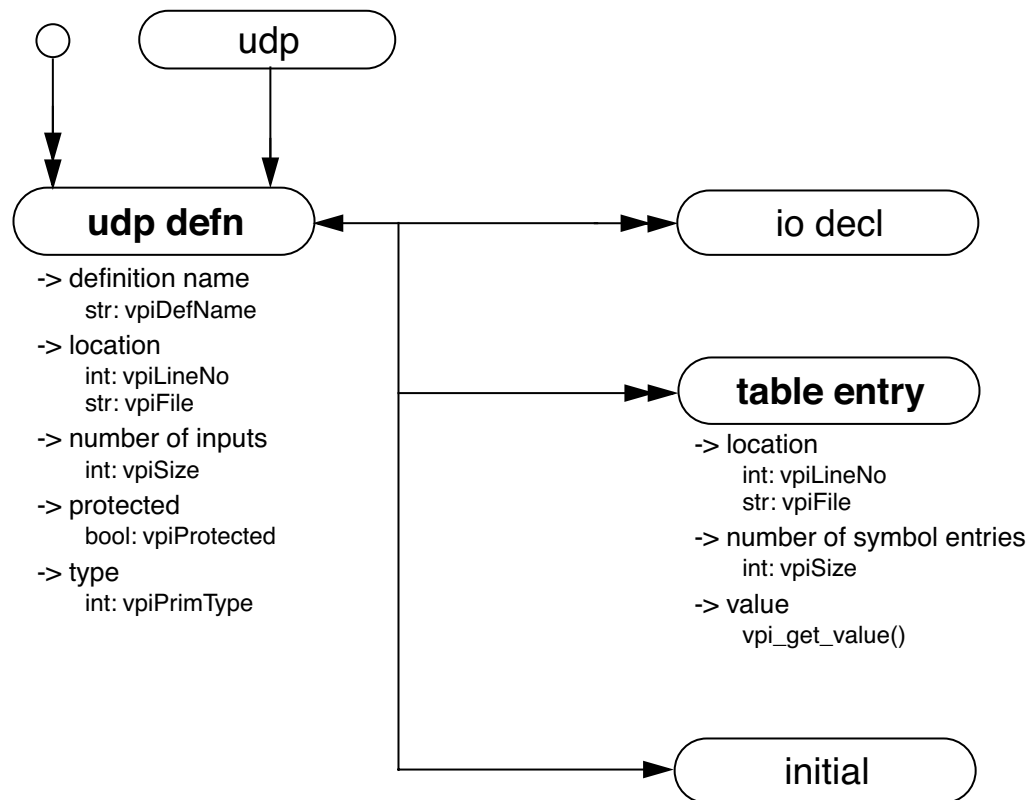
- 1— Obtaining the value from the object **parameter** shall return the final value of the parameter after all module instantiation overrides and defparams have been resolved.
- 2— **vpiLhs** from a param assign object shall return a handle to the overridden parameter.

22.5.9 Primitive, prim term**NOTES**

1 — **vpiSize** shall return the number of inputs.

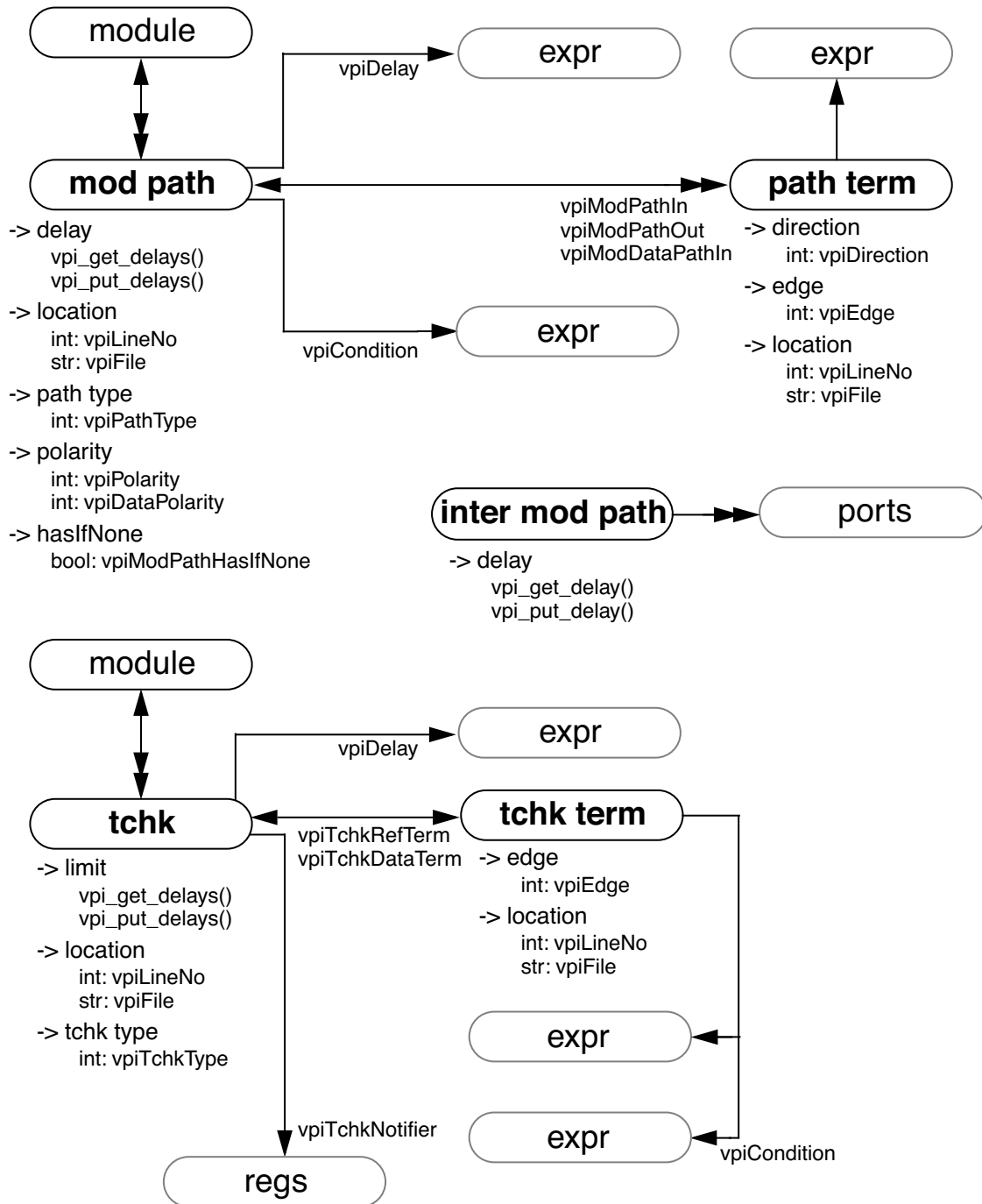
2 — For primitives, **vpi_put_value()** shall only be used with sequential UDP primitives.

22.5.10 UDP



NOTE—Only string (decompilation) and vector (ASCII values) shall be obtained for table entry objects using **vpi_get_value()**. Refer to the definition of **vpi_get_value()** for additional details.

22.5.11 Module path, timing check, intermodule path

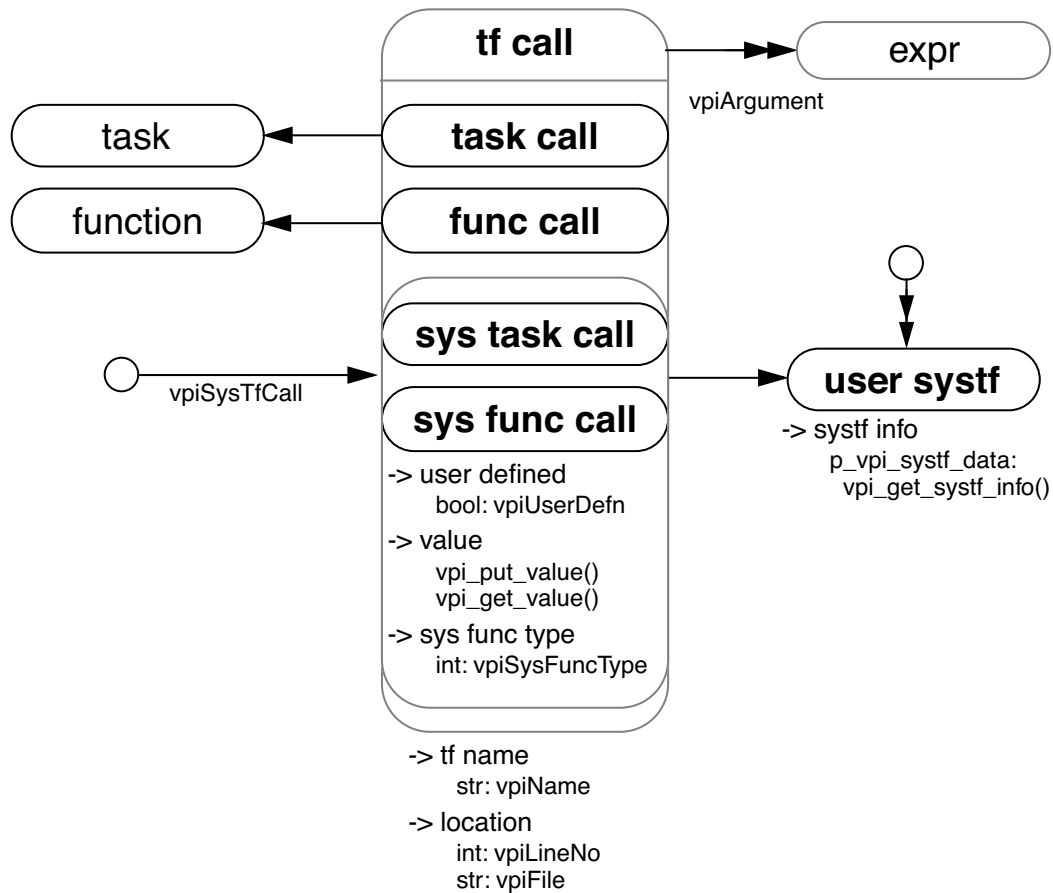


NOTES

1— The **vpiTchkRefTerm** is the first terminal for all tchks except \$setup, where **vpiTchkDataTerm** is the first terminal and **vpiTchkRefTerm** is the second terminal.

2— To get to an intermodule path, **vpi_handle_multi(vpiInterModPath, port1, port2)** can be used.

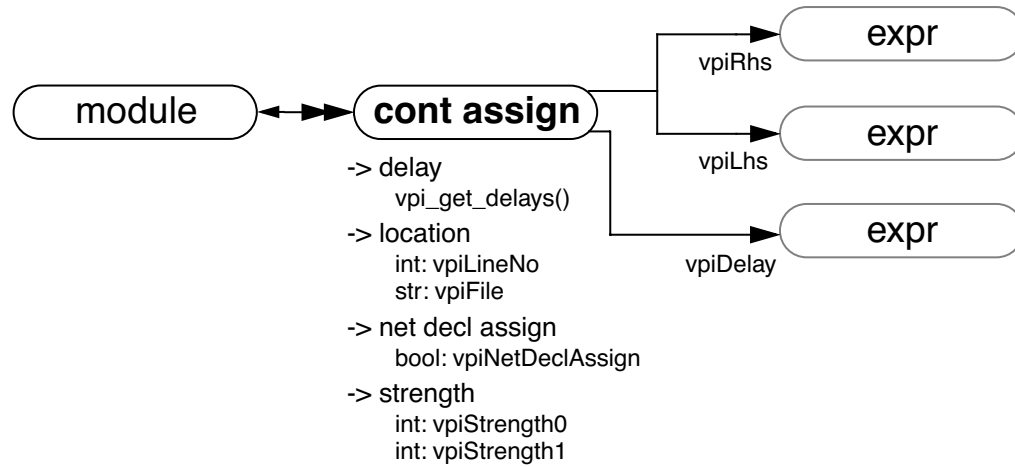
22.5.12 Task and function call



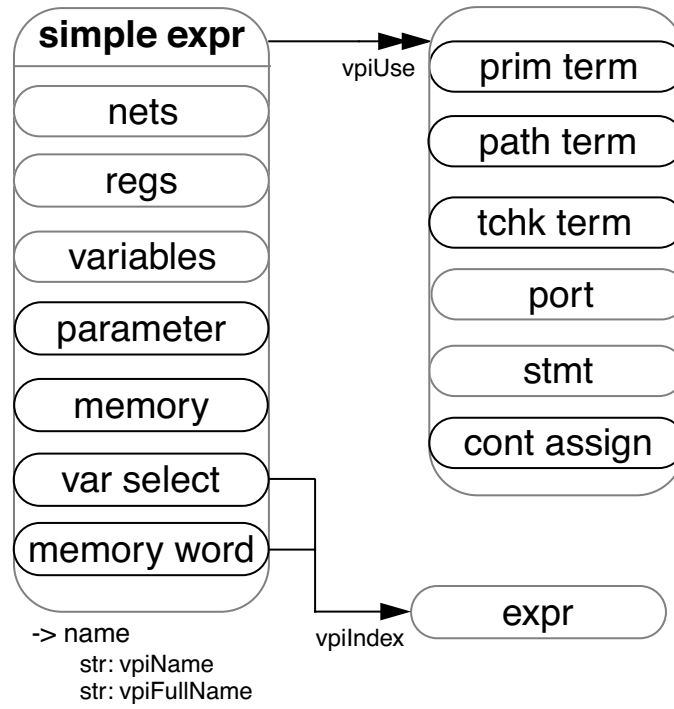
NOTES

- 1— The system task or function that invoked an application shall be accessed with **vpi_handle(vpiSysTfCall, NULL)**.
- 2— **vpi_get_value()** shall return the current value of the system function.
- 3— If the **vpiUserDefn** property of a system task or function call is true, then the properties of the corresponding systf object shall be obtained via **vpi_get_systf_info()**.
- 4— All user-defined system tasks or functions shall be retrieved using **vpi_iterate()**, with **vpiUserSystf** as the type argument, and a **NULL** reference argument.

22.5.13 Continuous assignment

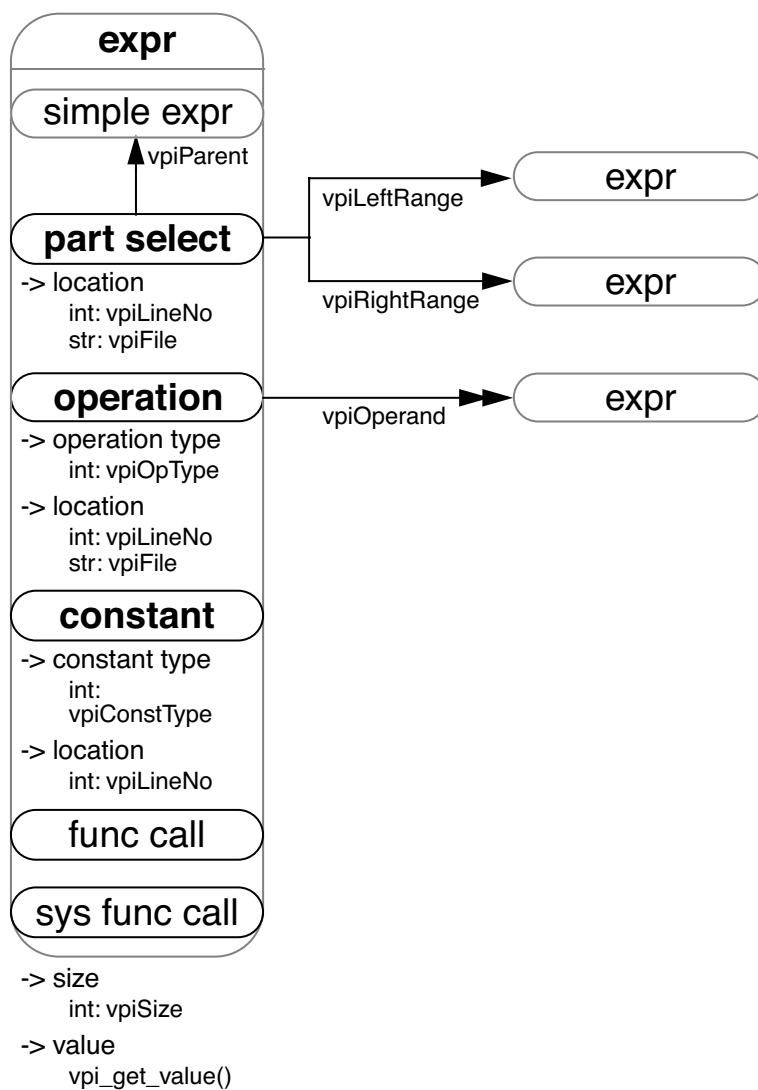


22.5.14 Simple expressions



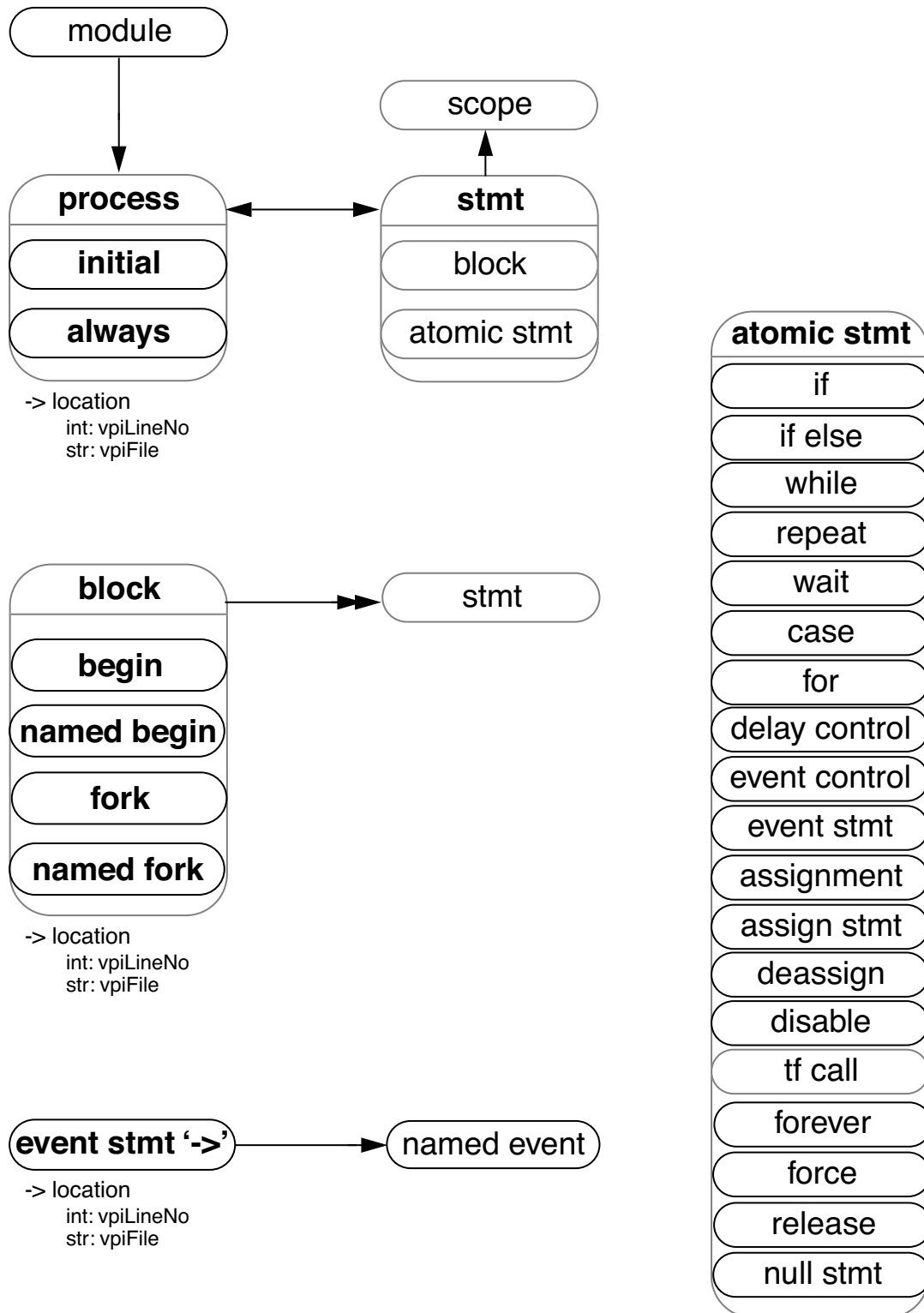
NOTES

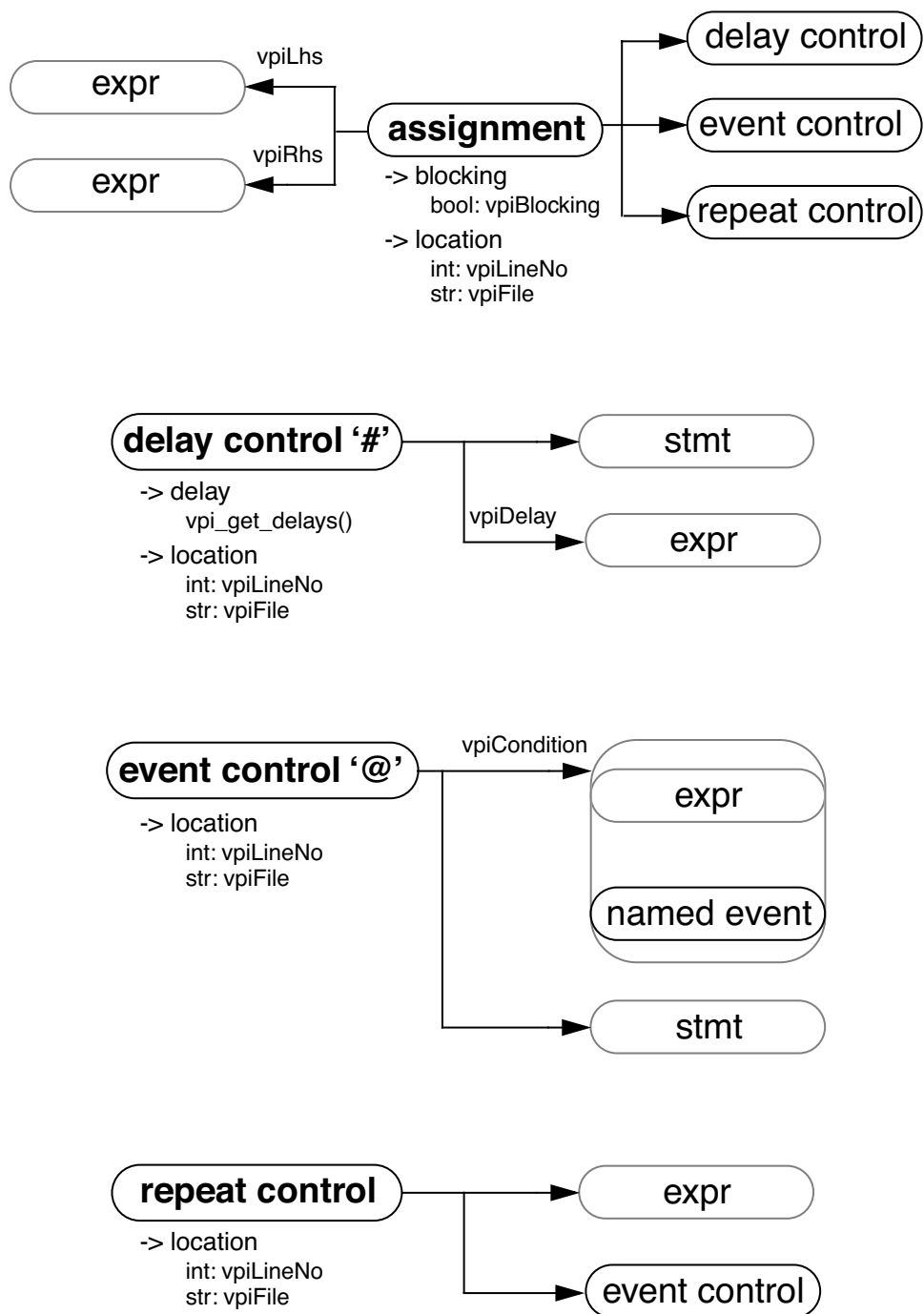
- 1— For vectors, the **vpiUse** relationship shall access any use of the vector or part-selects or bit-selects thereof.
- 2— For bit-selects, the **vpiUse** relationship shall access any specific use of that bit, any use of the parent vector, and any part-select that contains that bit.

22.5.15 Expressions

NOTE—For an operator whose type is **vpiMultiConcat**, the first operand shall be the multiplier expression.

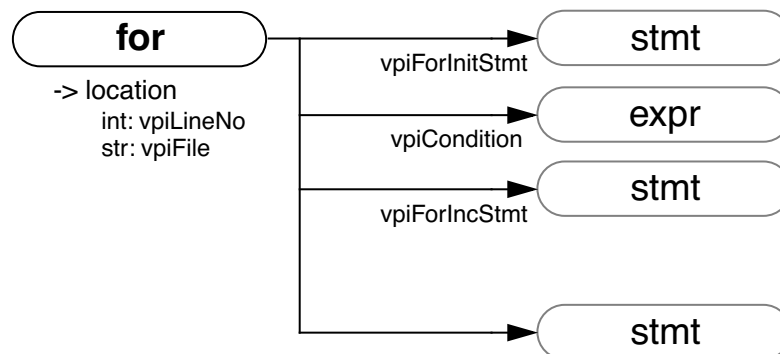
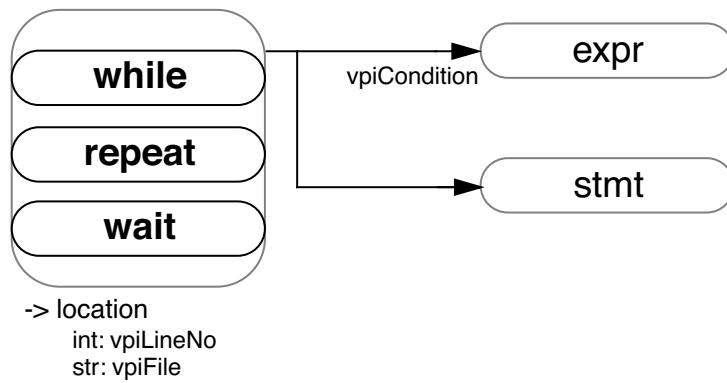
22.5.16 Process, block, statement, event statement

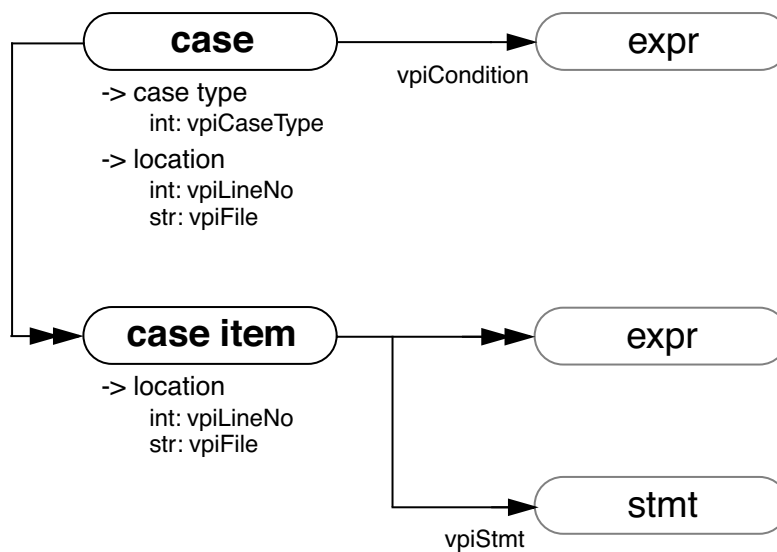
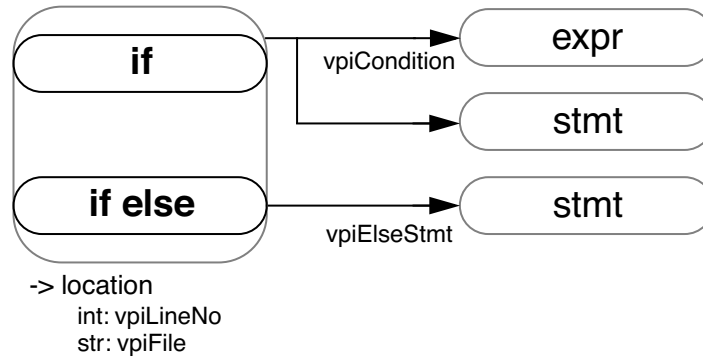


22.5.17 Assignment, delay control, event control, repeat control

NOTE—For delay control and event control associated with assignment, the statement shall always be NULL.

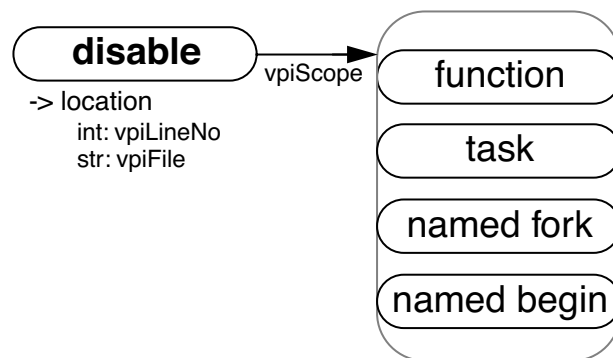
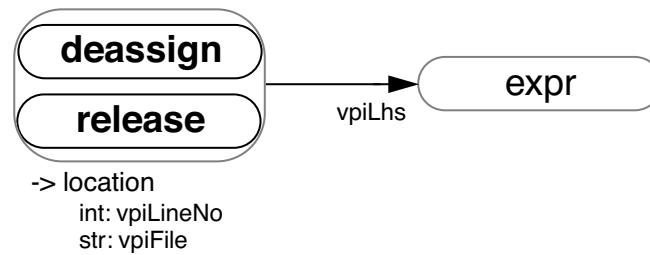
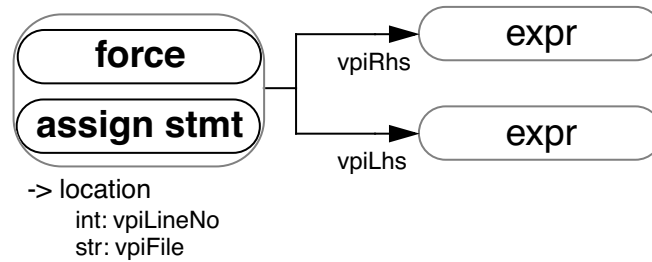
22.5.18 While, repeat, wait, for, forever

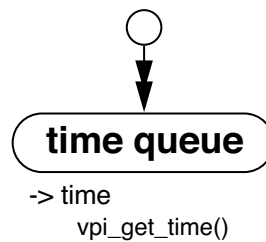
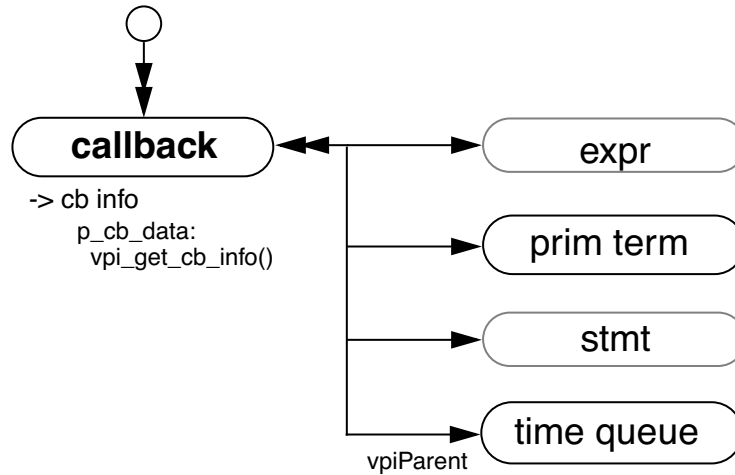


22.5.19 If, if-else, case**NOTES**

- 1— The *case item* shall group all case conditions that branch to the same statement.
- 2— **vpi_iterate()** shall return NULL for the default case item since there is no expression with the default case.

22.5.20 Assign statement, deassign, force, release, disable



22.5.21 Callback, time queue**NOTES**

- 1— To get information about the callback object, the routine **vpi_get_cb_info()** can be used.
- 2— To get callback objects not related to the above objects, the second argument to **vpi_iterate()** shall be **NULL**.
- 3— The time queue objects shall be returned in increasing order of simulation time.
- 4— **vpi_iterate()** shall return **NULL** if there is nothing left in the simulation queue.
- 5— If any events after read only sync remain in the current queue, then it shall not be returned as part of the iteration.

Section 23

VPI routine definitions

This section describes the Verilog Procedural Interface (VPI) routines, explaining their function, syntax, and usage. The routines are listed in alphabetical order. See Section 19 for the conventions used in the definitions of the PLI routines.

23.1 vpi_chk_error()

vpi_chk_error()			
Synopsis:	Retrieve information about VPI routine errors.		
Syntax:	vpi_chk_error(error_info_p)		
Type		Description	
Returns:	bool	true on success and false on failure	
Type		Name	Description
Arguments:	p_vpi_error_info	error_info_p	Pointer to a structure containing error information

The VPI routine **vpi_chk_error()** shall return **true** if the previous call to a VPI routine resulted in an error. If an error occurred, the **s_vpi_error_info** structure shall contain information about the error. If the error information is not needed, a NULL can be passed to the routine.

The **s_vpi_error_info** structure used by **vpi_chk_error()** is defined in **vpi_user.h** and is listed in Figure 23-1.

```
typedef struct t_vpi_error_info {
    int state; /* vpi[Compile,PLI,Run] */
    int level; /* vpi[Notice, Warning, Error, System, Internal] */
    char *message;
    char *product;
    char *code;
    char *file;
    int line;
} s_vpi_error_info, *p_vpi_error_info;
```

Figure 23-1—The s_vpi_error_info structure definition

23.2 vpi_compare_objects()

vpi_compare_objects()			
Synopsis:	Compare two handles to determine if they reference the same object.		
Syntax:	<code>vpi_compare_objects(obj1, obj2)</code>		
Returns:	Type	Description	
	bool	true if the two handles refer to the same object. Otherwise, false	
Arguments:	Type	Name	Description
	vpiHandle	obj1	Handle to an object
	vpiHandle	obj2	Handle to an object

The VPI routine **vpi_compare_objects()** shall return `true` if the two handles refer to the same object. Otherwise, `false` shall be returned. Handle equivalence cannot be determined with a C '==' comparison.

23.3 vpi_free_object()

vpi_free_object()			
Synopsis:	Free memory allocated by VPI routines.		
Syntax:	<code>vpi_free_object(obj)</code>		
Returns:	Type	Description	
	bool	true on success and false on failure	
Arguments:	Type	Name	Description
	vpiHandle	obj	Handle of an object

The VPI routine **vpi_free_object()** shall free memory allocated for objects. It shall generally be used to free memory created for iterator objects. The iterator object shall automatically be freed when **vpi_scan()** returns NULL either because it has completed an object traversal or encountered an error condition. If neither of these conditions occur (which can happen if the code breaks out of an iteration loop before it has scanned every object), **vpi_free_object()** should be called to free any memory allocated for the iterator. This routine can also optionally be used for implementations that have to allocate memory for objects. The routine shall return **true** on success and **false** on failure.

23.4 vpi_get()

vpi_get()			
Synopsis:	Get the value of an integer or Boolean property of an object.		
Syntax:	vpi_get(prop, obj)		
Returns:	Type	Description	
	int	Value of an integer or Boolean property	
Arguments:	Type	Name	Description
	int	prop	An integer constant representing the property of an object for which to obtain a value
	vpiHandle	obj	Handle to an object
Related routines:	Use vpi_get_str() to get string properties		

The VPI routine **vpi_get()** shall return the value of object properties, for properties of type *int* and *bool* (*bool* shall be defined to *int*). Object properties of type *bool* shall return **1** for true and **0** for false. For object properties of type *int* such as **vpiSize**, any integer shall be returned. For object properties of type *int* that return a defined value, refer to Annex E for the value that shall be returned. Note for object property **vpiTimeUnit**, if the object is **NULL**, then the simulation time unit shall be returned.

23.5 vpi_get_cb_info()

vpi_get_cb_info()			
Synopsis:	Retrieve information about a simulation-related callback.		
Syntax:	<code>vpi_get_cb_info(obj, cb_data_p)</code>		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	vpiHandle	obj	Handle to a simulation-related callback
	p_cb_data	cb_data_p	Pointer to a structure containing callback information
Related routines:	Use <code>vpi_get_systf_info()</code> to retrieve information about a system task/function callback		

The VPI routine **vpi_get_cb_info()** shall return information about a simulation-related callback in an `s_cb_data` structure. The memory for this structure shall be allocated by the user.

The `s_cb_data` structure used by **vpi_get_cb_info()** is defined in `vpi_user.h` and is listed in Figure 23-2.

```
typedef struct t_cb_data {
    int reason;
    int (*cb_rtn)();
    vpiHandle obj;
    p_vpi_time time; /* structure with simulation time info */
    p_vpi_value value; /* structure with simulation value info */
    char *user_data; /* user data to be passed to callback function */
} s_cb_data, *p_cb_data;
```

Figure 23-2—The s_cb_data structure definition

23.6 vpi_get_delays()

vpi_get_delays()			
Synopsis:	Retrieve the delays or pulse limits of an object.		
Syntax:	<code>vpi_get_delays(obj, delay_p)</code>		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	vpiHandle	obj	Handle to an object
	p_vpi_delay	delay_p	Pointer to a structure containing delay information
Related routines:	Use <code>vpi_put_delays()</code> to set the delays or timing limits of an object		

The VPI routine **vpi_get_delays()** shall retrieve the delays or pulse limits of an object and place them in an `s_vpi_delay` structure that has been allocated by the user. The format of the delay information shall be controlled by the *time_type* flag in the `s_vpi_delay` structure. This routine shall ignore the value of the *type* flag in the `s_vpi_time` structure.

The `s_vpi_delay` and `s_vpi_time` structures used by both **vpi_get_delays()** and **vpi_put_delays()** are defined in `vpi_user.h` and are listed in Figures 23-3 and 23-4.

```
typedef struct t_vpi_delay {
    struct t_vpi_time *da; /* ptr to user allocated array of delay
                           values */
    int no_of_delays;      /* number of delays */
    int time_type;         /* [vpiScaledRealTime, vpiSimTime] */
    bool mtm_flag;         /* true for mtm */
    bool append_flag;      /* true for append, false for replace */
    bool pulser_flag;      /* true for pulser values */
} s_vpi_delay, *p_vpi_delay;
```

Figure 23-3—The s_vpi_delay structure definition

```
typedef struct t_vpi_time
{
    int type;              /* [vpiScaledRealTime, vpiSimTime] */
    unsigned int high, low; /* for vpiSimTime */
    double real;           /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 23-4—The s_vpi_time structure definition

The *da* field of the `s_vpi_delay` structure shall be a user-allocated array of `s_vpi_time` structures. This array shall store delay values returned by **vpi_get_delays()**. The number of elements in this array shall be determined by

- The number of delays to be retrieved

- The **mtm_flag** setting
- The **pulsere_flag** setting

The number of delays to be retrieved shall be set in the *no_of_delays* field of the **s_vpi_delay** structure. Legal values for the number of delays shall be determined by the type of object.

- For primitive objects, the *no_of_delays* value shall be 2 or 3.
- For path delay objects, the *no_of_delays* value shall be 1, 2, 3, 6, or 12.
- For timing check objects, the *no_of_delays* value shall match the number of limits existing in the timing check.

The user allocated **s_vpi_delay** array shall contain delays in the same order in which they occur in the Verilog HDL description. The number of elements for each delay shall be determined by the flags **mtm_flag** and **pulsere_flag**, as shown in Table 23-1.

Table 23-1 — Size of the s_vpi_delay->da array

Flag values	Number of s_vpi_time array elements required for s_vpi_delay->da	Order in which delay elements shall be filled
mtm_flag = false pulsere_flag = false	<i>no_of_delays</i>	1st delay: da[0] -> 1st delay 2nd delay: da[1] -> 2nd delay ...
mtm_flag = true pulsere_flag = false	3 * <i>no_of_delays</i>	1st delay: da[0] -> min delay da[1] -> typ delay da[2] -> max delay 2nd delay: ...
mtm_flag = false pulsere_flag = true	3 * <i>no_of_delays</i>	1st delay: da[0] -> delay da[1] -> reject limit da[2] -> error limit 2nd delay element: ...
mtm_flag = true pulsere_flag = true	9 * <i>no_of_delays</i>	1st delay: da[0] -> min delay da[1] -> typ delay da[2] -> max delay da[3] -> min reject da[4] -> typ reject da[5] -> max reject da[6] -> min error da[7] -> typ error da[8] -> max error 2nd delay: ...

The delay structure has to be allocated before passing a pointer to **vpi_get_delays()**. In the following example, a static structure, **prim_da**, is allocated for use by each call to the **vpi_get_delays()** function.

```
display_prim_delays(prim)
vpiHandle prim;t2

{
    static s_vpi_time prim_da[3];
    static s_vpi_delay delay_s = {NULL, 3, vpiScaledRealTime};
    static p_vpi_delay delay_p = &delay_s;

    delay_s.da = &prim_da;
    vpi_get_delays(prim, delay_p);
    vpi_printf("Delays for primitive %s: %6.2f %6.2f %6.2f\n",
```

```
    vpi_get_str(vpiFullName, prim)
    delay_p->da[0].real, delay_p->da[1].real, delay_p->da[2].real);
}
```

23.7 vpi_get_str()

vpi_get_str()			
Synopsis:	Get the value of a string property of an object.		
Syntax:	<code>vpi_get_str(prop, obj)</code>		
Returns:	Type	Description	
	char *	Pointer to a character string containing the property value	
Arguments:	Type	Name	Description
	int	prop	An integer constant representing the property of an object for which to obtain a value
	vpiHandle	obj	Handle to an object
Related routines:	Use vpi_get() to get integer and Boolean properties		

The VPI routine **vpi_get_str()** shall return string property values. The string shall be placed in a temporary buffer that shall be used by every call to this routine. If the string is to be used after a subsequent call, the string should be copied to another location. Note that a different string buffer shall be used for string values returned through the `s_vpi_value` structure.

The following example illustrates the usage of **vpi_get_str()**.

```
char *str;
vpiHandle mod = vpi_handle_by_name("top.mod1", NULL);
vpi_printf ("Module top.mod1 is an instance of %s\n",
            vpi_get_str(vpiDefName, mod));
```

23.8 vpi_get_systf_info()

vpi_get_systf_info()			
Synopsis:	Retrieve information about a user-defined system task/function-related callback.		
Syntax:	<code>vpi_get_systf_info(obj, systf_data_p)</code>		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	vpiHandle	obj	Handle to a system task/function-related callback
	p_vpi_systf_data	systf_data_p	Pointer to a structure containing callback information
Related routines:	Use <code>vpi_get_cb_info()</code> to retrieve information about a simulation-related callback		

The VPI routine **vpi_get_systf_info()** shall return information about a user-defined system task or function callback in an `s_vpi_systf_data` structure. The memory for this structure shall be allocated by the user.

The `s_vpi_systf_data` structure used by **vpi_get_systf_info()** is defined in `vpi_user.h` and is listed in Figure 23-5.

```
typedef struct t_vpi_systf_data {
    int type;           /* vpiSys[Task,Function] */
    int sysfunctype;    /* vpiSysFunc[Int,Real,Time,Sized] */
    char *tfname;       /* first character must be "$" */
    int (*calltf)();
    int (*compiletf)();
    int (*sizetf)();    /* for vpiSysFuncSized system functions only */
    char *user_data;
} s_vpi_systf_data, *p_vpi_systf_data;
```

Figure 23-5—The s_vpi_systf_data structure definition

23.9 vpi_get_time()

vpi_get_time()			
Synopsis:	Retrieve the current simulation.		
Syntax:	vpi_get_time(obj, time_p)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	vpiHandle	obj	Handle to an object
	p_vpi_time	time_p	Pointer to a structure containing time information
Related routines:			

The VPI routine **vpi_get_time()** shall retrieve the current simulation time, using the time scale of the object. If *obj* is NULL, the simulation time is retrieved using the simulation time unit. The *time_p->type* field shall be set to indicate if scaled real or simulation time is desired. The memory for the *time_p* structure shall be allocated by the user.

The *s_vpi_time* structure used by **vpi_get_time()** is defined in *vpi_user.h* and is listed in Figure 23-6 [this is the same time structure as used by **vpi_put_value()**].

```
typedef struct t_vpi_time {
    int type;                /* for vpiScaledRealTime, vpiSimTime */
    unsigned int high, low; /* for vpiSimTime */
    double real;             /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 23-6—The s_vpi_time structure definition

23.10 vpi_get_value()

vpi_get_value()			
Synopsis:	Retrieve the simulation value of an object.		
Syntax:	vpi_get_value(obj, value_p)		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	vpiHandle	obj	Handle to an expression
	p_vpi_value	value_p	Pointer to a structure containing value information
Related routines:	Use vpi_put_value() to set the value of an object		

The VPI routine **vpi_get_value()** shall retrieve the simulation value of VPI objects. The value shall be placed in an `s_vpi_value` structure, which has been allocated by the user. The format of the value shall be set by the *format* field of the structure.

When the *format* field is **vpiObjTypeVal**, the routine shall fill in the value and change the *format* field based on the object type, as follows:

- For an integer, **vpiIntVal**
- For a real, **vpiRealVal**
- For a scalar, either **vpiScalar** or **vpiStrength**
- For a time variable, **vpiTimeVal** with **vpiSimTime**
- For a vector, **vpiVectorVal**

The buffer this routine uses for string values shall be different from the buffer that **vpi_get_str()** shall use.

The `s_vpi_value`, `s_vpi_vecval` and `s_vpi_strengthval` structures used by **vpi_get_value()** are defined in `vpi_user.h` and are listed in Figures 23-7, 23-8, and 23-9.

```
typedef struct t_vpi_value {
    int format; /* vpi[Bin,Oct,Dec,Hex]Str,Scalar,Int,Real,String,
                  Time,Vector,Strength,ObjType]Val*/
    union {
        char *str;
        int scalar; /* vpi[0,1,X,Z] */
        int integer;
        double real;
        struct t_vpi_time *time;
        struct t_vpi_vecval *vector;
        struct t_vpi_strengthval *strength;
        char *misc;
    } value;
} s_vpi_value, *p_vpi_value;
```

Figure 23-7— The s_vpi_value structure definition

```
typedef struct t_vpi_vecval {
    int aval, bval; /* bit encoding: ab: 00=0, 10=1, 11=X, 01=Z */
} s_vpi_vecval, *p_vpi_vecval;
```

Figure 23-8—The s_vpi_vecval structure definition

```
typedef struct t_vpi_strengthval {
    int logic; /* vpi[0,1,X,Z] */
    int s0, s1; /* refer to strength coding in the LRM */
} s_vpi_strengthval, *p_vpi_strengthval;
```

Figure 23-9—The s_vpi_strengthval structure definition

For vectors, the *p_vpi_vecval* field shall point to an array of *s_vpi_vecval* structures. The size of this array shall be determined by the size of the vector, where $array_size = ((vector_size-1)/32 + 1)$. The lsb of the vector shall be represented by the lsb of the 0-indexed element of *s_vpi_vecval* array. The 33rd bit of the vector shall be represented by the lsb of the 1-indexed element of the array, and so on. The memory for the union members *str*, *time*, *vector*, *strength*, and *misc* of the value union in the *s_vpi_value* structure shall be provided by the routine **vpi_get_value()**. This memory shall only be valid until the next call to **vpi_get_value()**. [Note that the user must provide the memory for these members when calling **vpi_put_value()**]. When a value change callback occurs for a value type of **vpiVectorVal**, the system shall create the associated memory (an array of *s_vpi_vecval* structures) and free the memory upon the return of the callback.

Table 23-2—Return value field of the s_vpi_value structure union

Format	Union member	Return description
vpiBinStrVal	str	String of binary char(s) [1, 0, x, z]
vpiOctStrVal	str	String of octal char(s) [0–7, x, X, z, Z] x When all the bits are x X When some of the bits are x z When all the bits are z Z When some of the bits are z
vpiDecStrVal	str	String of decimal char(s) [0–9]
vpiHexStrVal	str	String of hex char(s) [0–f, x, X, z, Z] x When all the bits are x X When some of the bits are x z When all the bits are z Z When some of the bits are z
vpiScalarVal	scalar	vpi1, vpi0, vpiX, vpiZ, vpiH, vpiL
vpiIntVal	integer	Integer value of the handle. Any bits x or z in the value of the object are mapped to a 0
vpiRealVal	real	Value of the handle as a double
vpiStringVal	str	A string where each 8-bit group of the value of the object is assumed to represent an ASCII character

Table 23-2—Return value field of the `s_vpi_value` structure union (continued)

Format	Union member	Return description
vpiTimeVal	time	Integer value of the handle using two integers
vpiVectorVal	vector	<i>aval/bval</i> representation of the value of the object
vpiStrengthVal	strength	Value plus strength information of a scalar object only
vpiObjectVal	—	Return a value in the closest format of the object

NOTE—If the object has a real value, it shall be converted to an integer using the rounding defined by the Verilog HDL before being returned in a format other than **vpiRealVal**.

To get the ASCII values of UDP table entries (as explained in Table 8-1), the *p_vpi_vecval* field shall point to an array of `s_vpi_vecval` structures. The size of this array shall be determined by the size of the table entry (no. of symbols per table entry), where $array_size = ((table_entry_size - 1) / 4 + 1)$. Each symbol shall require a nibble; the ordering of the symbols within `s_vpi_vecval` shall be the most significant nibble of *abit* first, then the least significant nibble of *abit*, then the most significant nibble of *bbit* and then the least significant nibble of *bbit*. Each symbol can be either one or two characters; when it is a single character, the second half of the nibble shall be an ASCII “0”.

The *misc* field in the `s_vpi_value` structure shall provide for alternative value types, which can be implementation specific. If this field is utilized, one or more corresponding format types shall also be provided.

In the following example, the binary value of each net that is contained in a particular module and whose name begins with a particular string is displayed. [This function makes use of the `strcmp()` facility normally declared in a `string.h` C library.]

```
void display_certain_net_values(mod, target)
vpiHandle mod;
char *target;
{
    static s_vpi_value value_s = {vpiBinStrVal};
    static p_vpi_value value_p = &value_s;
    vpiHandle net, itr;

    itr = vpi_iterate(vpiNet, mod);
    while (net = vpi_scan(itr))
    {
        char *net_name = vpi_get_str(vpiName, net);
        if (strcmp(target, net_name) == 0)
        {
            vpi_get_value(net, value_p);
            vpi_printf("Value of net %s: %s\n",
                vpi_get_str(vpiFullName, net), value_p->value.str);
        }
    }
}
```

23.11 vpi_get_vlog_info()

vpi_get_vlog_info()			
Synopsis:	Retrieve information about Verilog simulation execution.		
Syntax:	vpi_get_vlog_info(vlog_info_p)		
Type		Description	
Returns:	bool	true on success and false on failure	
Type		Name	Description
Arguments:	p_vpi_vlog_info	vlog_info_p	Pointer to a structure containing simulation information

The VPI routine **vpi_get_vlog_info()** shall obtain the following information about Verilog product execution:

- The number of invocation options (*argc*)
- Invocation option values (*argv*)
- Product and version strings

The information shall be contained in an `s_vpi_vlog_info` structure. The routine shall return `true` on success and `false` on failure.

The `s_vpi_vlog_info` structure used by **vpi_get_vlog_info()** is defined in `vpi_user.h` and is listed in Figure 23-10.

```
typedef struct t_vpi_vlog_info {
    int argc;
    char **argv;
    char *product;
    char *version;
} s_vpi_vlog_info, *p_vpi_vlog_info;
```

Figure 23-10—The `s_vpi_vlog_info` structure definition

23.12 vpi_handle()

vpi_handle()			
Synopsis:	Obtain a handle to an object with a one-to-one relationship.		
Syntax:	<code>vpi_handle(type, ref)</code>		
Returns:	Type	Description	
	vpiHandle	Handle to an object	
Arguments:	Type	Name	Description
	int	type	An integer constant representing the type of object for which to obtain a handle
	vpiHandle	ref	Handle to a reference object
Related routines:	Use vpi_iterate() and vpi_scan() to obtain handles to objects with a one-to-many relationship Use vpi_handle_multi() to obtain a handle to an object with a many-to-one relationship		

The VPI routine **vpi_handle()** shall return the object of type *type* associated with object *ref*. The one-to-one relationships that are traversed with this routine are indicated as single arrows in the object diagrams.

The following example application displays each primitive that an input net drives.

```

void display_driven_primitives(net)
vpiHandle net;
{
    vpiHandle load, prim, itr;
    vpi_printf("Net %s drives terminals of the primitives: \n",
        vpi_get_str(vpiFullName, net));
    itr = vpi_iterate(vpiLoad, net);
    if (!itr)
        return;
    while (load = vpi_scan(itr))
    {
        switch(vpi_get(vpiType, load))
        {
            case vpiGate:
            case vpiSwitch:
            case vpiUdp:
                prim = vpi_handle(vpiPrimitive, load);
                vpi_printf("\t%s\n", vpi_get_str(vpiFullName, prim));
            }
        }
    }
}

```

23.13 vpi_handle_by_index()

vpi_handle_by_index()			
Synopsis:	Get a handle to an object using its index number within a parent object.		
Syntax:	<code>vpi_handle_by_index(obj, index)</code>		
Returns:	Type	Description	
	vpiHandle	Handle to an object	
Arguments:	Type	Name	Description
	vpiHandle	obj	Handle to an object
	int	index	Index number of the object for which to obtain a handle

The VPI routine **vpi_handle_by_index()** shall return a handle to an object based on the index number of the object within a parent object. This function can be used to access all objects that can access an expression using **vpiIndex**. Argument *obj* shall represent the parent of the indexed object. For example, to access a net-bit, *obj* would be the associated net, while for a memory word, *obj* would be the associated memory.

23.14 vpi_handle_by_name()

vpi_handle_by_name()			
Synopsis:	Get a handle to an object with a specific name.		
Syntax:	vpi_handle_by_name(name, scope)		
Returns:	Type	Description	
	vpiHandle	Handle to an object	
Arguments:	Type	Name	Description
	char *	name	A character string or pointer to a string containing the name of an object
	vpiHandle	scope	Handle to a Verilog HDL scope

The VPI routine **vpi_handle_by_name()** shall return a handle to an object with a specific name. This function can be applied to all objects with a *fullname* property. The *name* can be hierarchical or simple. If *scope* is **NULL**, then *name* shall be searched for from the top level of hierarchy. Otherwise, *name* shall be searched for from *scope* using the scope search rules defined by the Verilog HDL.

23.15 vpi_handle_multi()

vpi_handle_multi()			
Synopsis:	Obtain a handle to intermodule paths with a many-to-one relationship.		
Syntax:	<code>vpi_handle_multi(type, ref1, ref2, ...)</code>		
Returns:	Type	Description	
	vpiHandle	Handle to an object	
Arguments:	Type	Name	Description
	int	type	An integer constant representing the type of object for which to obtain a handle
	vpiHandle	ref1, ref2, ...	Handles to two or more reference objects
Related routines:	Use vpi_iterate() and vpi_scan() to obtain handles to objects with a one-to-many relationship Use vpi_handle() to obtain handles to objects with a one-to-one relationship		

The VPI routine **vpi_handle_multi()** shall return a handle to objects of type **vpiInterModPath** associated with a list of *output port* and *input port* reference objects. The ports shall be of the same size and can be at different levels of the hierarchy. This routine performs a *many-to-one* operation instead of the usual one-to-one or one-to-many.

23.16 vpi_iterate()

vpi_iterate()			
Synopsis:	Obtain an iterator handle to objects with a one-to-many relationship.		
Syntax:	<code>vpi_iterate(type, ref)</code>		
Returns:	Type		Description
	vpiHandle		Handle to an iterator for an object
Arguments:	Type	Name	Description
	int	type	An integer constant representing the type of object for which to obtain iterator handles
	vpiHandle	ref	Handle to a reference object
Related routines:	Use <code>vpi_scan()</code> to traverse the HDL hierarchy using the iterator handle returned from <code>vpi_iterate()</code> Use <code>vpi_handle()</code> to obtain handles to object with a one-to-one relationship Use <code>vpi_handle_multi()</code> to obtain a handle to an object with a many-to-one relationship		

The VPI routine **vpi_iterate()** shall be used to traverse one-to-many relationships, which are indicated as double arrows in the object diagrams. The **vpi_iterate()** routine shall return a handle to an iterator, whose type shall be **vpi_iterator**, which can be used by **vpi_scan()** to traverse all objects of type *type* associated with object *ref*. To get the reference object from the iterator object use **vpi_handle(vpiUse, iterator_handle)**. If there are no objects of type *type* associated with the reference handle *ref*, then the **vpi_iterate()** routine shall return NULL.

The following example application uses **vpi_iterate()** and **vpi_scan()** to display each net (including the size for vectors) declared in the module. The example assumes it shall be passed a valid module handle.

```

void display_nets(mod)
vpiHandle mod;
{
    vpiHandle net;
    vpiHandle itr;

    vpi_printf("Nets declared in module %s\n",
        vpi_get_str(vpiFullName, mod));

    itr = vpi_iterate(vpiNet, mod);
    while (net = vpi_scan(itr))
    {
        vpi_printf("\t%s", vpi_get_str(vpiName, net));
        if (vpi_get(vpiVector, net))
        {
            vpi_printf(" of size %d\n", vpi_get(vpiSize, net));
        }
        else vpi_printf("\n");
    }
}

```

23.17 vpi_mcd_close()

vpi_mcd_close()			
Synopsis:	Close one or more files opened by vpi_mcd_open().		
Syntax:	vpi_mcd_close(<i>mcd</i>)		
Returns:	Type	Description	
	unsigned int	0 if successful, the <i>mcd</i> of unclosed channels if unsuccessful	
Arguments:	Type	Name	Description
	unsigned int	<i>mcd</i>	A multichannel descriptor representing the files to close
Related routines:	Use vpi_mcd_open() to open a file Use vpi_mcd_printf() to write to an opened file Use vpi_mcd_name() to get the name of a file represented by a channel descriptor		

The VPI routine **vpi_mcd_close()** shall close the file(s) specified by a multichannel descriptor, *mcd*. Several channels can be closed simultaneously, since channels are represented by discrete bits in the integer *mcd*. On success this routine returns a 0; on error it returns the *mcd* value of the unclosed channels.

23.18 vpi_mcd_name()

vpi_mcd_name()			
Synopsis:	Get the name of a file represented by a channel descriptor.		
Syntax:	<code>vpi_mcd_name(cd)</code>		
Returns:	Type		Description
	char *	Pointer to a character string containing the name of a file	
Arguments:	Type	Name	Description
	unsigned int	cd	A single-channel descriptor representing a file
Related routines:	Use <code>vpi_mcd_open()</code> to open a file Use <code>vpi_mcd_close()</code> to close files Use <code>vpi_mcd_printf()</code> to write to an opened file		

The VPI routine **vpi_mcd_name()** shall return the name of a file represented by a single-channel descriptor, *cd*. On error, the routine shall return NULL.

23.19 vpi_mcd_open()

vpi_mcd_open()			
Synopsis:	Open a file for writing.		
Syntax:	<code>vpi_mcd_open(file)</code>		
Returns:	Type		Description
	unsigned int	A multichannel descriptor representing the file that was opened	
Arguments:	Type	Name	Description
	char *	file	A character string or pointer to a string containing the file name to be opened
Related routines:	Use <code>vpi_mcd_close()</code> to close a file Use <code>vpi_mcd_printf()</code> to write to an opened file Use <code>vpi_mcd_name()</code> to get the name of a file represented by a channel descriptor		

The VPI routine **vpi_mcd_open()** shall open a file for writing and return a corresponding multichannel descriptor number (*mcd*). The following channel descriptors are predefined and shall be automatically opened by the system:

- Descriptor 1 is *stdout*
- Descriptor 2 is *stderr*
- Descriptor 3 is the current log file

The **vpi_mcd_open()** routine shall return a **0** on error. If the file is already opened, **vpi_mcd_open()** shall return the descriptor number.

23.20 vpi_mcd_printf()

vpi_mcd_printf()			
Synopsis:	Write to one or more files opened with vpi_mcd_open().		
Syntax:	vpi_mcd_printf(mcd, format, format, ...)		
Type		Description	
Returns:	int	The number of characters written	
Type		Name	Description
Arguments:	unsigned int	mcd	A multichannel descriptor representing the files to which to write
	char *	format	A format string using the C fprintf() format
Related routines:	Use vpi_mcd_open() to open a file Use vpi_mcd_close() to close a file Use vpi_mcd_name() to get the name of a file represented by a channel descriptor		

The VPI routine **vpi_mcd_printf()** shall write to one or more channels (up to 32) determined by the *mcd*. An *mcd* of 1 (bit 0 set) corresponds to Channel 1, a *mcd* of 2 (bit 1 set) corresponds to Channel 2, a *mcd* of 4 (bit 2 set) corresponds to Channel 3, and so on. Channel 1 is *stdout*, channel 2 is *stderr*, and channel 3 is the current log file. Several channels can be written to simultaneously, since channels are represented by discrete bits in the integer *mcd*. The text to be written shall be controlled by one or more format strings. The format strings shall use the same format as the C `fprintf()` routine. The routine shall return the number of characters printed.

23.21 vpi_printf()

vpi_printf()			
Synopsis:	Write to stdout and the current product log file.		
Syntax:	<code>vpi_printf(format, format,...)</code>		
	Type	Description	
Returns:	int	The number of characters written	
	Type	Name	Description
Arguments:	char *	format	A format string using the C printf() format
Related routines:	Use vpi_mcd_printf() to write to an opened file		

The VPI routine **vpi_printf()** shall write to both *stdout* and the current product log file. The format strings shall use the same format as the C `printf()` routine. The routine shall return the number of characters printed, and the routine shall return EOF if an error occurred.

23.22 vpi_put_delays()

vpi_put_delays()			
Synopsis:	Set the delays or timing limits of an object.		
Syntax:	vpi_put_delays(obj, delay_p)		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	vpiHandle	obj	Handle to an object
	p_vpi_delay	delay_p	Pointer to a structure containing delay information
Related routines:	Use vpi_get_delays() to retrieve delays or timing limits of an object		

The VPI routine **vpi_put_delays()** shall set the delays or timing limits of an object as indicated in the *delay_p* structure. The same ordering of delays shall be used as described in the **vpi_get_delays()** function. If only the delay changes, and not the pulse limits, the pulse limits shall retain the values they had before the delays were altered.

The *s_vpi_delay* and *s_vpi_time* structures used by both **vpi_get_delays()** and **vpi_put_delays()** are defined in *vpi_user.h* and are listed in Figures 23-11 and 23-12.

```
typedef struct t_vpi_delay {
    struct t_vpi_time *da; /* ptr to user allocated array of delay
                           values */
    int no_of_delays;      /* number of delays */
    int time_type;         /* [vpiScaledRealTime, vpiSimTime] */
    bool mtm_flag;         /* true for mtm */
    bool append_flag;      /* true for append, false for replace */
    bool pulser_flag;      /* true for pulser values */
} s_vpi_delay, *p_vpi_delay;
```

Figure 23-11 – The s_vpi_delay structure definition

```
typedef struct t_vpi_time
{
    int type;              /* [vpiScaledRealTime, vpiSimTime] */
    unsigned int high, low; /* for vpiSimTime */
    double real;           /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 23-12 – The s_vpi_time structure definition

The *da* field of the *s_vpi_delay* structure shall be a user-allocated array of *s_vpi_time* structures. This array shall store the delay values to be written by **vpi_put_delays()**. The number of elements in this array shall be the same as described in 23.6.

The following example application accepts a module path handle, rise and fall delays, and replaces the delays of the indicated path.

```
void set_path_rise_fall_delays(path, rise, fall)
vpiHandle path;
double rise, fall;
{
    static s_vpi_time path_da[2];
    static s_vpi_delay delay_s = {NULL, 2, vpiScaledRealTime};
    static p_vpi_delay delay_p = &delay_s;

    delay_s.da = &path_da;
    path_da[0].real = rise;
    path_da[1].real = fall;

    vpi_put_delays(path, delay_p);
}
```

23.23 vpi_put_value()

vpi_put_value()			
Synopsis:	Set a value on an object.		
Syntax:	<code>vpi_put_value(obj, value_p, time_p, flags)</code>		
Type		Description	
Returns:	vpiHandle	Handle to the scheduled event caused by vpi_put_value()	
Arguments:	Type	Name	Description
	vpiHandle	obj	Handle to an object
	p_vpi_value	value_p	Pointer to a structure with value information
	p_vpi_time	time_p	Pointer to a structure with delay information
	int	flags	Integer constants that set the delay mode
Related routines:	Use vpi_get_value() to retrieve the value of an expression		

The VPI routine **vpi_put_value()** shall set simulation logic values on an object. The value to be set shall be stored in an `s_vpi_value` structure that has been allocated. The delay time before the value is set shall be stored in an `s_vpi_time` structure that has been allocated. The routine can be applied to nets, regs, variables, memory words, system function calls, sequential UDPs, and schedule events. The *flags* argument shall be used to direct the routine to use one of the following delay modes:

vpiInertialDelay	All scheduled events on the object shall be removed before this event is scheduled.
vpiTransportDelay	All events on the object scheduled for times later than this event shall be removed (modified transport delay).
vpiPureTransportDelay	No events on the object shall be removed (transport delay).
vpiNoDelay	The object shall be set to the passed value with no delay. Argument <i>time_p</i> shall be ignored and can be set to NULL.
vpiForceFlag	The object shall be forced to the passed value with no delay (same as the Verilog HDL procedural force). Argument <i>time_p</i> shall be ignored and can be set to NULL.
vpiReleaseFlag	The object shall be released from a forced value (same as the Verilog HDL procedural release). Argument <i>time_p</i> shall be ignored and can be set to NULL. The <i>value_p</i> shall contain the current value of the object.
vpiCancelEvent	A previously scheduled event shall be cancelled. The object passed to vpi_put_value() shall be a handle to an object of type vpiSchedEvent .

If the *flags* argument also has the bit mask **vpiReturnEvent**, **vpi_put_value()** shall return a handle of type **vpiSchedEvent** to the newly scheduled event, provided there is some form of a delay and an event is scheduled. If the bit mask is not used, or if no delay is used, or if an event is not scheduled, the return value shall be NULL.

The handle to the event can be cancelled by calling **vpi_put_value()** with the flag set to **vpiCancelEvent**. It shall not be an error to cancel an event that has already occurred. The scheduled event can be tested by calling **vpi_get()** with the flag **vpiScheduled**. If an event is cancelled, it shall simply be removed from the event queue. Any effects that were caused by scheduling the event shall remain in effect (e.g., events that were cancelled due to inertial delay).

Calling **vpi_free_object()** on the handle shall free the handle but shall not effect the event.

Sequential UDPs shall be set to the indicated value with no delay regardless of any delay on the primitive instance.

The **s_vpi_value** and **s_vpi_time** structures used by **vpi_put_value()** are defined in **vpi_user.h** and are listed in Figures 23-13 and 23-14.

```
typedef struct t_vpi_value {
    int format; /* vpi[[Bin,Oct,Dec,Hex]Str,Scalar,Int,Real,String,
                  Time,Vector,Strength,ObjType]Val*/
    union {
        char *str;
        int scalar; /* vpi[0,1,X,Z] */
        int integer;
        double real;
        struct t_vpi_time *time;
        struct t_vpi_vecval *vector;
        struct t_vpi_strengthval *strength;
        char *misc;
    } value;
} s_vpi_value, *p_vpi_value;
```

Figure 23-13—The s_vpi_value structure definition

```
typedef struct t_vpi_time {
    int type; /* for vpiScaledRealTime, vpiSimTime */
    unsigned int high, low; /* for vpiSimTime */
    double real; /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 23-14—The s_vpi_time structure definition

For **vpiScaledRealTime**, the indicated time shall be in the timescale associated with the object.

23.24 vpi_register_cb()

vpi_register_cb()			
Synopsis:	Register simulation-related callbacks.		
Syntax:	<code>vpi_register_cb(cb_data_p)</code>		
Returns:	Type	Description	
	vpiHandle	Handle to the callback object	
Arguments:	Type	Name	Description
	p_cb_data	cb_data_p	Pointer to a structure with data about when callbacks should occur and the data to be passed
Related routines:	Use <code>vpi_register_systf()</code> to register callbacks for user-defined system tasks and functions Use <code>vpi_remove_cb()</code> to remove callbacks registered with <code>vpi_register_cb()</code>		

The VPI routine **vpi_register_cb()** is used for registration of simulation-related callbacks to a user-provided application for a variety of reasons during a simulation. The reasons for which a callback can occur are divided into three categories:

- Simulation event
- Simulation time
- Simulation action or feature

How callbacks are registered for each of these categories is explained in the following paragraphs.

The *cb_data_p* argument shall point to a `s_cb_data` structure, which is defined in `vpi_user.h` and given in Figure 23-15.

```
typedef struct t_cb_data {
    int reason;
    int (*cb_rtn)();
    vpiHandle obj;
    p_vpi_time time;      /* structure defined in vpi_user.h */
    p_vpi_value value;    /* structure defined in vpi_user.h */
    int index; /* index of memory word or var select which changed */
    char *user_data;      /* user data to be passed to callback function */
} s_cb_data, *p_cb_data;
```

Figure 23-15—The s_cb_data structure definition

For all callbacks, the *reason* field of the `s_cb_data` structure shall be set to a predefined constant, such as **cbValueChange**, **cbAtStartOfSimTime**, **cbEndOfCompile**, etc. The reason constant shall determine when the user application shall be called back. Refer to the `vpi_user.h` file listing in Annex E for a list of all callback reason constants.

The *cb_rtn* field of the `s_cb_data` structure shall be set to the application name, which shall be invoked when the simulator executes the callback. The use of the remaining fields are detailed in the following subclauses.

23.24.1 Simulation-event-related callbacks

The **vpi_register_cb()** callback mechanism can be registered for callbacks to occur for simulation events, such as value changes on an expression or terminal, or the execution of a behavioral statement. When the *cb_data_p->reason* field is set to one of the following, the callback shall occur as described below:

cbValueChange	After value change on an expression or terminal
cbStmt	Before execution of a behavioral statement
cbForce/cbRelease	After a force or release has occurred on a simple expression

The following fields shall need to be initialized before passing the *s_cb_data* structure to **vpi_register_cb()**:

<i>cb_data_p->obj</i>	This field shall be assigned a handle to an expression, terminal, or statement for which the callback shall occur. For force and release callbacks, if this is set to NULL, every force and release shall generate a callback.
<i>cb_data_p->time->type</i>	This field shall be set to either vpiScaledRealTime or vpiSimTime , depending on what time information the user application requires during the callback. If simulation time information is not needed during the callback, this field can be set to vpiSuppressTime .
<i>cb_data_p->value->format</i>	This field shall be set to one of the value formats indicated in Table 23-3. If value information is not needed during the callback, this field can be set to vpiSuppressVal . For cbStmt callbacks, value information is not passed to the callback routine, so this field shall be ignored.

Table 23-3—Value format field of *cb_data_p->value->format*

Format	Registers a callback to return
vpiBinStrVal	String of binary char(s) [1, 0, x, z]
vpiOctStrVal	String of octal char(s) [0–7, x, X, z, Z]
vpiDecStrVal	String of decimal char(s) [0–9]
vpiHexStrVal	String of hex char(s) [0–f, x, X, z, Z]
vpiScalarVal	vpi1, vpi0, vpiX, vpiZ, vpiH, vpiL
vpiIntVal	Integer value of the handle
vpiRealVal	Value of the handle as a double
vpiStringVal	An ASCII string
vpiTimeVal	Integer value of the handle using two integers
vpiVectorVal	<i>aval/bval</i> representation of the value of the object
vpiStrengthVal	Value plus strength information of a scalar object only
vpiObjectVal	Return a value in the closest format of the object

When a simulation event callback occurs, the user application shall be passed a single argument, which is a pointer to an *s_cb_data* structure [this is not a pointer to the same structure that was passed to **vpi_register_cb()**]. The *time* and *value* information shall be set as directed by the *time type* and *value* format fields in the call to **vpi_register_cb()**. The *user_data* field shall be equivalent to the *user_data* field passed to **vpi_register_cb()**. The user application can

use the information in the passed structure and information retrieved from other VPI interface routines to perform the desired callback processing.

For a **cbValueChange** callback, if the *obj* is a memory word or a variable array, the *value* in the **s_cb_data** structure shall be the value of the memory word or variable select that changed value. The *index* field shall contain the index of the memory word or variable select that changed value.

For **cbForce** and **cbRelease** callbacks, *obj* shall be a handle to the forced or released object, while *value* shall contain the new value for a force, or the current value for a release, in the format specified by the registration call.

The following example shows an implementation of a simple monitor functionality for scalar nets, using a simulation-event-related callback.

```

setup_monitor(net)
vpiHandle net;
{
    static s_vpi_time time_s = {vpiScaledRealTime};
    static s_vpi_value value_s = {vpiBinStrVal};
    static s_cb_data cb_data_s =
        {cbValueChange, my_monitor, NULL, &time_s, &value_s};
    char *net_name = vpi_get_str(vpiFullName, net);
    cb_data_s.obj = net;
    cb_data_s.user_data = malloc(strlen(net_name)+1);
    strcpy(cb_data_s.user_data, net_name);
    vpi_register_cb(&cb_data_s);
}

my_monitor(cb_data_p)
p_cb_data cb_data_p; {
    vpi_printf("%d %d: %s = %s\n",
        cb_data_p->time->high, cb_data_p->time->low,
        cb_data_p->user_data,
        cb_data_p->value->value.str);
}

```

23.24.2 Simulation-time-related callbacks

The **vpi_register_cb()** can register callbacks to occur for simulation time reasons, include callbacks at the beginning or end of the execution of a particular time queue. The following time-related callback reasons are defined:

cbAtStartOfSimTime	Callback shall occur before execution of events in a specified time queue. A callback can be set for any time, even if no event is present.
cbReadWriteSynch	Callback shall occur after execution of events for a specified time.
cbReadOnlySynch	Same as cbReadWriteSynch , except that writing values or scheduling events before the next scheduled event is not allowed.
cbNextSimTime	Callback shall occur before execution of events in the next event queue.
cbAfterDelay	Callback shall occur after a specified amount of time, before execution of events in a specified time queue. A callback can be set for anytime, even if no event is present.

The following fields shall need to be set before passing the **s_cb_data** structure to **vpi_register_cb()**:

<i>cb_data_p->time->type</i>	This field shall be set to either vpiScaledRealTime or vpiSimTime , depending on what time information the user application requires during the callback.
------------------------------------	---

cb_data_p->[time->low,time->high,time->real]

These fields shall contain the requested time of the callback or the delay before the callback.

The *value* fields are ignored for all reasons with simulation-time-related callbacks.

The *obj* field is ignored for all reasons except when the *reason* field equals **cbAfterDelay** and *cb_data_p->time->type* equals **vpiScaledRealTime**, in which case the object determines the time scaling.

For reason **vpiNextSimTime**, the time structure is ignored.

When a simulation-time-related callback occurs, the user callback application shall be passed a single argument, which is a pointer to an **s_cb_data** structure [this is not a pointer to the same structure that was passed to **vpi_register_cb()**]. The *time* structure shall contain the current simulation time. The *user_data* field shall be equivalent to the *user_data* field passed to **vpi_register_cb()**.

The callback application can use the information in the passed structure and information retrieved from other interface routines to perform the desired callback processing.

23.24.3 Simulator action and feature related callbacks

The **vpi_register_cb()** can register callbacks to occur for simulator action reasons or simulator feature reasons. *Simulator action reasons* are callbacks such as the end of compilation or end of simulation. *Simulator feature reasons* are software-product-specific features, such as restarting from a saved simulation state or entering an interactive mode. Actions are differentiated from features in that actions shall occur in all VPI-compliant products, whereas features might not exist in all VPI-compliant products.

The following action-related callbacks shall be defined:

cbEndOfCompile	End of simulation data structure compilation or build
cbStartOfSimulation	Start of simulation (beginning of time 0 simulation cycle)
cbEndOfSimulation	End of simulation (e.g., \$finish system task executed)
cbError	Simulation run-time error occurred
cbTchkViolation	Timing check error occurred

Examples of possible feature related callbacks are

cbStartOfSave	Simulation save state command invoked
cbEndOfSave	Simulation save state command completed
cbStartOfRestart	Simulation restart from saved state command invoked
cbEndOfRestart	Simulation restart command completed
cbEnterInteractive	Simulation entering interactive debug mode (e.g., \$stop system task executed)
cbExitInteractive	Simulation exiting interactive mode
cbInteractiveScopeChange	Simulation command to change interactive scope executed
cbUnresolvedSystf	Unknown user-defined system task or function encountered

The only fields in the **s_cb_data** structure that shall need to be setup for simulation action/feature callbacks are the *reason*, *cb_rtn*, and *user_data* (if desired) fields.

When a simulation action/feature callback occurs, the user routine shall be passed a pointer to an **s_cb_data** structure. The *reason* field shall contain the reason for the callback. For **cbTchkViolation** callbacks, the *obj* field shall be a handle to the timing check. For **cbInteractiveScopeChange**, *obj* shall be a handle to the new scope. For

cbUnresolvedSystf, *user_data* shall point to the name of the unresolved task or function. On a **cbError** callback, the routine **vpi_chk_error()** can be called to retrieve error information.

The following example shows a callback application that reports cpu usage at the end of a simulation. If the user routine **setup_report_cpu()** is placed in the **vlog_startup_routines** list, it shall be called just after the simulator is invoked.

```
static int initial_cputime_g;

void report_cpu()
{
    int total = get_current_cputime() - initial_cputime_g;
    vpi_printf("Simulation complete. CPU time used: %d\n", total);
}

void setup_report_cpu()
{
    static s_cb_data cb_data_s = {cbEndOfSimulation, report_cpu};
    initial_cputime_g = get_current_cputime();
    vpi_register_cb(&cb_data_s);
}
```

23.25 vpi_register_systf()

vpi_register_systf()			
Synopsis:	Register user-defined system task/function-related callbacks.		
Syntax:	<code>vpi_register_systf(systf_data_p)</code>		
Type		Description	
Returns:	<code>vpiHandle</code>	Handle to the callback object	
Type		Name	Description
Arguments:	<code>p_vpi_systf_data</code>	<code>systf_data_p</code>	Pointer to a structure with data about when callbacks should occur and the data to be passed
Related routines:	Use <code>vpi_register_cb()</code> to register callbacks for simulation-related events		

The VPI routine **vpi_register_systf()** shall register callbacks for user-defined system tasks or functions. Callbacks can be registered to occur when a user-defined system task or function is encountered during compilation or execution of Verilog HDL source code.

The *systf_data_p* argument shall point to a `s_vpi_systf_data` structure, which is defined in `vpi_user.h` and listed in Figure 23-16.

```
typedef struct t_vpi_systf_data {
    int type;           /* vpiSys[Task,Function] */
    int sysfunctype;    /* vpiSysFunc[Int,Real,Time,Sized] */
    char *tfname;       /* first character must be "$" */
    int (*calltf)();
    int (*compiletf)();
    int (*sizetf)();    /* for vpiSysFuncSized system functions only */
    char *user_data;
} s_vpi_systf_data, *p_vpi_systf_data;
```

Figure 23-16—The s_vpi_systf_data structure definition

23.25.1 System task and function callbacks

User-defined Verilog system tasks and functions that use VPI routines can be registered with **vpi_register_systf()**. The following system task/function-related callbacks are defined.

The *type* field of the `s_vpi_systf_data` structure shall register the user application to be a system task or a system function. The *type* field value shall be an integer constant of **vpiSysTask** or **vpiSysFunction**.

The *sysfunctype* field of the `s_vpi_systf_data` structure shall define the type of value that a system function shall return. The *sysfunctype* field shall be an integer constant of **vpiSysFuncInt**, **vpiSysFuncReal**, **vpiSysFuncTime**, or **vpiSysFuncSized**. This field shall only be used when the *type* field is set to **vpiSysFunction**.

The *compiletf*, *calltf*, and *sizetf* fields of the `s_vpi_systf_data` structure shall be pointers to the user-provided applications that are to be invoked by the system task/function callback mechanism. Callbacks to the applications pointed to by the *compiletf* and *sizetf* fields shall occur when the simulation data structure is compiled or built (or for

the first invocation if the system task or function is invoked from an interactive mode). Callbacks to the application pointed to by the *calltf* routine shall occur each time the system task or function is invoked during simulation execution.

The *user_data* field of the *s_vpi_systf_data* structure shall specify a user-defined value, which shall be passed back to the *compiletf*, *size*, and *calltf* applications when a callback occurs.

The following example application demonstrates dynamic linking of a VPI system task. The example uses an imaginary routine, *dlink()*, which accepts a file name and a function name and then links that function dynamically. This routine derives the target file and function names from the target *systf* name.

```
link_systf(target)
char *target;
{
    char task_name[strSize];
    char file_name[strSize];
    char compiletf_name[strSize];
    char calltf_name[strSize];
    static s_vpi_systf_data task_data_s = {vpiSysTask};
    static p_vpi_systf_data task_data_p = &task_data_s;

    sprintf(task_name, "$%s", target);
    sprintf(file_name, "%s.o", target);
    sprintf(compiletf_name, "%s_compiletf", target);
    sprintf(calltf_name, "%s_calltf", target);

    task_data_p->tfname = task_name;
    task_data_p->compiletf = (int (*)()) dlink(file_name,
        compiletf_name);
    task_data_p->calltf = (int (*)()) dlink(file_name, calltf_name);
    vpi_register_systf(task_data_p);
}
```

23.25.2 Initializing VPI system task/function callbacks

A means of initializing system task/function callbacks and performing any other desired task just after the simulator is invoked shall be provided by placing routines in a NULL-terminated static array, **vlog_startup_routines**. A C function using the array definition shall be provided as follows:

```
void (*vlog_startup_routines[]) ();
```

This C function shall be provided with a VPI-compliant product. Entries in the array shall be added by the user. The location of **vlog_startup_routines** and the procedure for linking **vlog_startup_routines** with a software product shall be defined by the product vendor. (Note that callbacks can also be registered or removed at any time during an application routine, not just at startup time).

A primary use of the **vlog_startup_routines** shall be for registering system tasks and functions. User tasks and functions that appear in a compiled description shall generally be registered by a routine in this array.

The following example uses **vlog_startup_routines** to register system tasks and functions and to run a user initialization routine.

```
/*In a vendor product file which contains vlog_startup_routines ...*/
extern void register_my_systfs();
extern void my_init();
void (*vlog_startup_routines[])() =
{
    setup_report_cpu,    /* user routine example in 23.24.3 */

```

```
        register_my_systfs, /* user routine listed below */
        0                  /* must be last entry in list */
    }

/* In a user provided file... */
void register_my_systfs()
{
    static s_vpi_systf_data systf_data_list[] = {
        {vpiSysTask, , "$my_task", my_task_calltf, my_task_compiletf},
        {vpiSysFunction, vpiFuncInt, "$my_func", my_func_calltf,
         my_func_compiletf},
        {vpiSysFunction, vpiSysFuncSized, "$my_func",
         my_sized_calltf, my_sized_compiletf, my_size},
    };
    p_vpi_systf_data systf_data_p = &(systf_data_list[0]);

    while (*systf_data_p)
        vpi_register_systf(systf_data_p++);
}
```


23.26 vpi_remove_cb()

vpi_remove_cb()			
Synopsis:	Remove a simulation callback registered with vpi_register_cb().		
Syntax:	vpi_remove_cb(cb_obj)		
Returns:	Type	Description	
	bool	1 (true) if successful; 0 (false) on a failure	
Arguments:	Type	Name	Description
	vpiHandle	cb_obj	Handle to the callback object
Related routines:	Use vpi_register_cb() to register callbacks for simulation-related events		

The VPI routine **vpi_remove_cb()** shall remove callbacks that were registered with *vpi_register_cb()*. The argument to this routine shall be a handle to the callback object. The routine shall return a **1** (true) if successful, and a **0** (false) on a failure.

23.27 vpi_scan()

vpi_scan()			
Synopsis:	Scan the Verilog HDL hierarchy for objects with a one-to-many relationship.		
Syntax:	<code>vpi_scan(itr)</code>		
Returns:	Type	Description	
	vpiHandle	Handle to an object	
Arguments:	Type	Name	Description
	vpiHandle	itr	Handle to an iterator object returned from vpi_iterate()
Related routines:	Use vpi_iterate() to obtain an iterator handle Use vpi_handle() to obtain handles to an object with a one-to-one relationship Use vpi_handle_multi() to obtain a handle to an object with a many-to-one relationship		

The VPI routine **vpi_scan()** shall traverse the instantiated Verilog HDL hierarchy and return handles to objects as directed by the iterator *itr*. The iterator handle shall be obtained by calling **vpi_iterate()** for a specific object type.

The following example application uses **vpi_iterate()** and **vpi_scan()** to display each net (including the size for vectors) declared in the module. The example assumes it shall be passed a valid module handle.

```

void display_nets(mod)
vpiHandle mod;
{
    vpiHandle net;
    vpiHandle itr;

    vpi_printf("Nets declared in module %s\n",
vpi_get_str(vpiFullName, mod));

    itr = vpi_iterate(vpiNet, mod);
    while (net = vpi_scan(itr))
    {
        vpi_printf("\t%s", vpi_get_str(vpiName, net));
        if (vpi_get(vpiVector, net))
        {
            vpi_printf(" of size %d\n", vpi_get(vpiSize, net));
        }
        else vpi_printf("\n");
    }
}

```

Annex A

(normative)

Formal syntax definition

The formal syntax of Verilog HDL is described using Backus-Naur Form (BNF).

A.1 Source text

```

source_text ::= {description}
description ::=
    module_declaration
    | udp_declaration
module_declaration ::=
    module_keyword module_identifier [ list_of_ports ] ; { module_item } endmodule
module_keyword ::= module | macromodule
list_of_ports ::= ( port { , port } )
port ::=
    [ port_expression ]
    | , port_identifier ( [ port_expression ] )
port_expression ::=
    port_reference
    | { port_reference { , port_reference } }
port_reference ::=
    port_identifier
    | port_identifier [ constant_expression ]
    | port_identifier [ msb_constant_expression : lsb_constant_expression ]
module_item ::=
    module_item_declaration
    | parameter_override
    | continuous_assign
    | gate_instantiation
    | udp_instantiation
    | module_instantiation
    | specify_block
    | initial_construct
    | always_construct
module_item_declaration ::=
    parameter_declaration
    | input_declaration
    | output_declaration
    | inout_declaration
    | net_declaration
    | reg_declaration
    | integer_declaration
    | real_declaration

```

```

| time_declaration
| realtime_declaration
| event_declaration
| task_declaration
| function_declaration

```

```
parameter_override ::= defparam list_of_param_assignments ;
```

A.2 Declarations

```

parameter_declaration ::= parameter list_of_param_assignments ;
list_of_param_assignments ::= param_assignment { , param_assignment }
param_assignment ::= parameter_identifier = constant_expression
input_declaration ::= input [range] list_of_port_identifiers ;
output_declaration ::= output [range] list_of_port_identifiers ;
inout_declaration ::= inout [range] list_of_port_identifiers ;
list_of_port_identifiers ::= port_identifier { , port_identifier }
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 ;
event_declaration ::= event event_identifier { , event_identifier } ;
list_of_real_identifiers ::= real_identifier { , real_identifier }
list_of_register_identifiers ::= register_name { , register_name }
register_name ::=
    register_identifier
    | memory_identifier [ upper_limit_constant_expression : lower_limit_constant_expression ]
range ::= [ msb_constant_expression : lsb_constant_expression ]
net_declaration ::=
    net_type [ vector | scalared ] [range] [delay3] list_of_net_identifiers ;
    | trireg [ vector | scalared ] [charge_strength] [range] [delay3] list_of_net_identifiers ;
    | net_type [ vector | scalared ] [drive_strength] [range] [delay3] list_of_net_decl_assignments ;
net_type ::= wire | tri | tri1 | supply0 | wand | triand | tri0 | supply1 | wor | trior
list_of_net_identifiers ::= net_identifier { , net_identifier }
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 ] ] )
delay2 ::= # 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
function_declaration ::=
    function [range_or_type] function_identifier ;

```

```

        function_item_declaration {function_item_declaration}
        statement
    endfunction
range_or_type ::= range | integer | real | realtime | time
function_item_declaration ::=
    block_item_declaration
  | input_declaration
task_declaration ::=
    task task_identifier ;
    {task_item_declaration}
    statement_or_null
    endtask
task_argument_declaration ::=
    block_item_declaration
  | output_declaration
  | inout_declaration
block_item_declaration ::=
    parameter_declaration
  | reg_declaration
  | integer_declaration
  | real_declaration
  | time_declaration
  | realtime_declaration
  | event_declaration

```

A.3 Primitive instances

```

gate_instantiation ::=
    n_input_gatetype [drive_strength] [delay2] n_input_gate_instance { , n_input_gate_instance } ;
  | n_output_gatetype [drive_strength] [delay2] n_output_gate_instance { , n_output_gate_instance } ;
  | enable_gatetype [drive_strength] [delay3] enable_gate_instance { , enable_gate_instance } ;
  | mos_swichtype [delay3] mos_switch_instance { , mos_switch_instance } ;
  | pass_swichtype pass_switch_instance { , pass_switch_instance } ;
  | pass_en_swichtype [delay3] pass_en_switch_instance { , pass_en_switch_instance } ;
  | cmos_swichtype [delay3] cmos_switch_instance { , cmos_switch_instance } ;
  | pullup [pullup_strength] pull_gate_instance { , pull_gate_instance } ;
  | pulldown [pulldown_strength] pull_gate_instance { , pull_gate_instance } ;
n_input_gate_instance ::= [name_of_gate_instance] ( output_terminal , input_terminal { , input_terminal } )
n_output_gate_instance ::= [name_of_gate_instance] ( output_terminal { , output_terminal } , input_terminal )
enable_gate_instance ::= [name_of_gate_instance] ( output_terminal , input_terminal , enable_terminal )
mos_switch_instance ::= [name_of_gate_instance] ( output_terminal , input_terminal , enable_terminal )
pass_switch_instance ::= [name_of_gate_instance] ( inout_terminal , inout_terminal )
pass_enable_switch_instance ::= [name_of_gate_instance] ( inout_terminal , inout_terminal , enable_terminal )
cmos_switch_instance ::= [name_of_gate_instance] ( output_terminal , input_terminal ,
    ncontrol_terminal , pcontrol_terminal )
pull_gate_instance ::= [name_of_gate_instance] ( output_terminal )
name_of_gate_instance ::= gate_instance_identifier [range]
pullup_strength ::=
    ( strength0 , strength1 )
  | ( strength1 , strength0 )
  | ( strength1 )
pulldown_strength ::=
    ( strength0 , strength1 )
  | ( strength1 , strength0 )

```

```

    | ( strength0 )
input_terminal ::= scalar_expression
enable_terminal ::= scalar_expression
ncontrol_terminal ::= scalar_expression
pcontrol_terminal ::= scalar_expression
output_terminal ::= terminal_identifier | terminal_identifier [ constant_expression ]
inout_terminal ::= terminal_identifier | terminal_identifier [ constant_expression ]
n_input_gatetype ::= and | nand | or | nor | xor | xnor
n_output_gatetype ::= buf | not
enable_gatetype ::= bufif0 | bufif1 | notif0 | notif1
mos_switchtype ::= nmos | pmos | rnmos | rpmos
pass_switchtype ::= tran | rtran
pass_en_switchtype ::= tranif0 | tranif1 | rtranif1 | rtranif0
cmos_switchtype ::= cmos | rcmos

```

A.4 Module instantiation

```

module_instantiation ::=
    module_identifier [ parameter_value_assignment ] module_instance { , module_instance } ;
parameter_value_assignment ::= # ( expression { , expression } )
module_instance ::= name_of_instance ( [ list_of_module_connections ] )
name_of_instance ::= module_instance_identifier [ range ]
list_of_module_connections ::=
    ordered_port_connection { , ordered_port_connection }
    | named_port_connection { , named_port_connection }
ordered_port_connection ::= [ expression ]
named_port_connection ::= . port_identifier ( [ expression ] )

```

A.5 UDP declaration and instantiation

```

udp_declaration ::=
    primitive udp_identifier ( udp_port_list ) ;
    udp_port_declaration { udp_port_declaration }
    udp_body
    endprimitive
udp_port_list ::= output_port_identifier , input_port_identifier { , input_port_identifier }
udp_port_declaration ::=
    output_declaration
    | input_declaration
    | reg_declaration
udp_body ::= combinational_body | sequential_body
combinational_body ::= table combinational_entry { combinational_entry } endtable
combinational_entry ::= level_input_list : output_symbol ;
sequential_body ::= [ udp_initial_statement ] table sequential_entry { sequential_entry } endtable
udp_initial_statement ::= initial udp_output_port_identifier = init_val ;
init_val ::= 1'b0 | 1'b1 | 1'bx | 1'bX | 1'B0 | 1'B1 | 1'Bx | 1'BX | 1 | 0
sequential_entry ::= seq_input_list : current_state : next_state ;
seq_input_list ::= level_input_list | edge_input_list
level_input_list ::= level_symbol { level_symbol }
edge_input_list ::= { level_symbol } edge_indicator { level_symbol }

```

```

edge_indicator ::= ( level_symbol level_symbol ) | edge_symbol
current_state ::= level_symbol
next_state ::= output_symbol | -
output_symbol ::= 0 | 1 | x | X
level_symbol ::= 0 | 1 | x | X | ? | b | B
edge_symbol ::= r | R | f | F | p | P | n | N | *

udp_instantiation ::= udp_identifier [ drive_strength ] [ delay2 ] udp_instance { , udp_instance } ;
udp_instance ::= [ name_of_udp_instance ] ( output_port_connection , input_port_connection
                                         { , input_port_connection } )
name_of_udp_instance ::= udp_instance_identifier [ range ]

```

A.6 Behavioral statements

```

continuous_assign ::= assign [drive_strength] [delay3] list_of_net_assignments ;
list_of_net_assignments ::= net_assignment { , net_assignment }
net_assignment ::= net_lvalue = expression

initial_construct ::= initial statement
always_construct ::= always statement

```

```

statement ::=
    blocking_assignment ;
    | non_blocking_assignment ;
    | procedural_continuous_assignments ;
    | procedural_timing_control_statement
    | conditional_statement
    | case_statement
    | loop_statement
    | wait_statement
    | disable_statement
    | event_trigger
    | seq_block
    | par_block
    | task_enable
    | system_task_enable

statement_or_null ::= statement | ;
blocking_assignment ::= reg_lvalue = [ delay_or_event_control ] expression
non-blocking_assignment ::= reg_lvalue <= [ delay_or_event_control ] expression
procedural_continuous_assignment ::=
    | assign reg_assignment ;
    | deassign reg_lvalue ;
    | force reg_assignment ;
    | force net_assignment ;
    | release reg_lvalue ;
    | release net_lvalue ;
procedural_timing_control_statement ::=
    delay_or_event_control statement_or_null
delay_or_event_control ::=
    delay_control
    | event_control
    | repeat ( expression ) event_control

```

```

delay_control ::=
    # delay_value
    | # ( mintymax_expression )
event_control ::=
    @ event_identifier
    | @ ( event_expression )
event_expression ::=
    expression
    | event_identifier
    | posedge expression
    | negedge expression
    | event_expression or event_expression
conditional_statement ::=
    if ( expression ) statement_or_null [ else statement_or_null ]
case_statement ::=
    case ( expression ) case_item { case_item } endcase
    | casez ( expression ) case_item { case_item } endcase
    | casex ( expression ) case_item { case_item } endcase
case_item ::=
    expression { , expression } : statement_or_null
    | default [ : ] statement_or_null
loop_statement ::=
    forever statement
    | repeat ( expression ) statement
    | while ( expression ) statement
    | for ( reg_assignment ; expression ; reg_assignment ) statement
reg_assignment ::= reg_lvalue = expression
wait_statement ::=
    wait ( expression ) statement_or_null
event_trigger ::=
    -> event_identifier ;
disable_statement ::=
    disable task_identifier ;
    | disable block_identifier ;
seq_block ::= begin [ : block_identifier { block_item_declaration } ] { statement } end
par_block ::= fork [ : block_identifier { block_item_declaration } ] { statement } join
task_enable ::= task_identifier [ ( expression { , expression } ) ] ;
system_task_enable ::= system_task_name [ ( expression { , expression } ) ] ;
system_task_name ::= $identifier    Note: The $ may not be followed by a space.

```

A.7 Specify section

```

specify_block ::= specify [ specify_item ] endspecify
specify_item ::=
    specparam_declaration
    | path_declaration
    | system_timing_check
specparam_declaration ::= specparam list_of_specparam_assignments ;
list_of_specparam_assignments ::= specparam_assignment { , specparam_assignment }
specparam_assignment ::=
    specparam_identifier = constant_expression
    | pulse_control_specparam
pulse_control_specparam ::=

```



```

    PATHPULSE$ = ( reject_limit_value [ , error_limit_value ] );
    | PATHPULSE$specify_input_terminal_descriptor$specify_output_terminal_descriptor
      = ( reject_limit_value [ , error_limit_value ] );

limit_value ::= constant_mintypmax_expression
path_declaration ::=
    simple_path_declaration ;
    | edge_sensitive_path_declaration ;
    | state-dependent_path_declaration ;

simple_path_declaration ::=
    parallel_path_description = path_delay_value
    | full_path_description = path_delay_value

parallel_path_description ::=
    ( specify_input_terminal_descriptor [ polarity_operator ] => specify_output_terminal_descriptor )

full_path_description ::=
    ( list_of_path_inputs [ polarity_operator ] *> list_of_path_outputs )

list_of_path_inputs ::=
    specify_input_terminal_descriptor { , specify_input_terminal_descriptor }

list_of_path_outputs ::=
    specify_output_terminal_descriptor { , specify_output_terminal_descriptor }

specify_input_terminal_descriptor ::=
    input_identifier
    | input_identifier [ constant_expression ]
    | input_identifier [ msb_constant_expression : lsb_constant_expression ]

specify_output_terminal_descriptor ::=
    output_identifier
    | output_identifier [ constant_expression ]
    | output_identifier [ msb_constant_expression : lsb_constant_expression ]

input_identifier ::= input_port_identifier | inout_port_identifier
output_identifier ::= output_port_identifier | inout_port_identifier
polarity_operator ::= + | -
path_delay_value ::=
    list_of_path_delay_expressions
    | ( list_of_path_delay_expressions )

list_of_path_delay_expressions ::=
    t_path_delay_expression
    | trise_path_delay_expression , tfall_path_delay_expression
    | trise_path_delay_expression , tfall_path_delay_expression , tz_path_delay_expression
    | t0l_path_delay_expression , t10_path_delay_expression , t0z_path_delay_expression ,
      tzl_path_delay_expression , t1z_path_delay_expression , tz0_path_delay_expression
    | t0l_path_delay_expression , t10_path_delay_expression , t0z_path_delay_expression ,
      tzl_path_delay_expression , t1z_path_delay_expression , tz0_path_delay_expression ,
      t0x_path_delay_expression , txl_path_delay_expression , t1x_path_delay_expression ,
      tx0_path_delay_expression , txz_path_delay_expression , tzx_path_delay_expression

path_delay_expression ::= constant_mintypmax_expression
edge_sensitive_path_declaration ::=
    parallel_edge_sensitive_path_description = path_delay_value
    | full_edge_sensitive_path_description = path_delay_value

parallel_edge_sensitive_path_description ::=
    ( [ edge_identifier ] specify_input_terminal_descriptor =>
      specify_output_terminal_descriptor [ polarity_operator ] : data_source_expression ) )

full_edge_sensitive_path_description ::=
    ( [ edge_identifier ] list_of_path_inputs *>
      list_of_path_outputs [ polarity_operator ] : data_source_expression ) )

data_source_expression ::= expression
edge_identifier ::= posedge | negedge

```

```

state_dependent_path_declaration ::=
    if ( conditional_expression ) simple_path_declaration
  | if ( conditional_expression ) edge_sensitive_path_declaration
  | ifnone simple_path_declaration

system_timing_check ::=
    $setup ( timing_check_event , timing_check_event , timing_check_limit [ , notify_register ] ) ;
  | $hold ( timing_check_event , timing_check_event , timing_check_limit [ , notify_register ] ) ;
  | $period ( controlled_timing_check_event , timing_check_limit [ , notify_register ] ) ;
  | $width ( controlled_timing_check_event , timing_check_limit ,
              constant_expression [ , notify_register ] ) ;
  | $skew ( timing_check_event , timing_check_event , timing_check_limit [ , notify_register ] ) ;
  | $recovery ( controlled_timing_check_event , timing_check_event ,
                timing_check_limit [ , notify_register ] ) ;
  | $setuphold ( timing_check_event , timing_check_event , timing_check_limit ,
                 timing_check_limit [ , notify_register ] ) ;

timing_check_event ::=
    [ timing_check_event_control ] specify_terminal_descriptor [ &&& timing_check_condition ]

specify_terminal_descriptor ::=
    specify_input_terminal_descriptor
  | specify_output_terminal_descriptor

controlled_timing_check_event ::=
    timing_check_event_control specify_terminal_descriptor [ &&& timing_check_condition ]

timing_check_event_control ::=
    posedge
  | negedge
  | edge_control_specifier

edge_control_specifier ::= edge [ edge_descriptor [ , edge_descriptor ] ]

edge_descriptor ::=
    01
  | 10
  | 0x
  | x1
  | 1x
  | x0

timing_check_condition ::=
    scalar_timing_check_condition
  | ( scalar_timing_check_condition )

scalar_timing_check_condition ::=
    expression
  | ~ expression
  | expression == scalar_constant
  | expression === scalar_constant
  | expression != scalar_constant
  | expression !== scalar_constant

timing_check_limit ::= expression

scalar_constant ::=
    1'b0 | 1'b1 | 1'B0 | 1'B1 | 'b0 | 'b1 | 'B0 | 'B1 | 1 | 0

notify_register ::= register_identifier

```

A.8 Expressions

```

net_lvalue ::=
    net_identifier
  | net_identifier [ expression ]

```

```

    | net_identifier [ msb_constant_expression : lsb_constant_expression ]
    | net_concatenation
reg_lvalue ::=
    reg_identifier
    | reg_identifier [ expression ]
    | reg_identifier [ msb_constant_expression : lsb_constant_expression ]
    | reg_concatenation
constant_expression ::=
    constant_primary
    | unary_operator constant_primary
    | constant_expression binary_operator constant_expression
    | constant_expression ? constant_expression : constant_expression
    | string
constant_primary ::=
    number
    | parameter_identifier
    | constant_concatenation
    | constant_multiple_concatenation
constant_mintypmax_expression ::=
    constant_expression
    | constant_expression : constant_expression : constant_expression
mintypmax_expression ::=
    expression
    | expression : expression : expression
expression ::=
    primary
    | unary_operator primary
    | expression binary_operator expression
    | expression ? expression : expression
    | string
unary_operator ::=
    + | - | ! | ~ | & | ~& | | | ~ | ^ | ~^ | ^^
binary_operator ::=
    + | - | * | / | % | == | != | === | !== | && | ||
    < | <= | > | >= | & | | | ^ | ^~ | ~^ | >> | <<
primary ::=
    number
    | identifier
    | identifier [ expression ]
    | identifier [ msb_constant_expression : lsb_constant_expression ]
    | concatenation
    | multiple_concatenation
    | function_call
    | ( mintypmax_expression )
number ::=
    decimal_number
    | octal_number
    | binary_number
    | hex_number
    | real_number
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
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 }
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 | 0
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

```

```

concatenation ::= { expression { , expression } }
multiple_concatenation ::= { expression { expression { , expression } } }
function_call ::=
    function_identifier ( expression { , expression } )
    | name_of_system_function [ ( expression { , expression } ) ]
name_of_system_function ::= $identifier
string ::= “ { Any_ASCII_Characters_except_new_line } ”

```

NOTES

- 1—Embedded spaces are illegal.
- 2—The \$ in name_of_system_function may not be followed by a space.

A.9 General

```

comment ::=
    short_comment
    | long_comment
short_comment ::= // comment_text \n
long_comment ::= /* comment_text */
comment_text ::= { Any_ASCII_character }
identifier ::= IDENTIFIER [ { . IDENTIFIER } ]
IDENTIFIER ::=
    simple_identifier
    | escaped_identifier
simple_identifier ::= [a-zA-Z][a-zA-Z-Z_$]
escaped_identifier ::= \ { Any_ASCII_character_except_white_space } white_space
white_space ::= space | tab | newline

```

NOTE—The period in identifier may not be preceded or followed by a space.

Annex B

(normative)

List of keywords

Keywords are predefined nonescaped identifiers that define Verilog language constructs. An escaped identifier shall not be treated as a keyword.

always	ifnone	rpmos
and	initial	rtran
assign	inout	rtranif0
begin	input	rtranif1
buf	integer	scalared
bufif0	join	small
bufif1	large	specify
case	macromodule	specparam
casex	medium	strong0
casez	module	strong1
cmos	nand	supply0
deassign	negedge	supply1
default	nmos	table
defparam	nor	task
disable	not	time
edge	notif0	tran
else	notif1	tranif0
end	or	tranif1
endcase	output	tri
endmodule	parameter	tri0
endfunction	pmos	tri1
endprimitive	posedge	triand
endspecify	primitive	trior
endtable	pull0	trireg
endtask	pull1	vectored
event	pullup	wait
for	pulldown	wand
force	rcmos	weak0
forever	real	weak1
fork	realtime	while
function	reg	wire
highz0	release	wor
highz1	repeat	xnor
if	rnmos	xor

Annex C

(normative)

The `acc_user.h` file

```

/*****
 * acc_user.h
 *
 * IEEE Std 1364-1995 Verilog Hardware Description Language (HDL)
 * Programming Language Interface (PLI).
 *
 * This file contains the constant definitions, structure definitions, and
 * routine declarations used by the Verilog PLI procedural interface
 * access (ACC) routines.
 *
 * The file should be included with all C routines that use the PLI ACC
 * routines.
 *****/
#ifndef ACC_USER_H
#define ACC_USER_H
/*-----*/
/*----- definitions -----*/
/*-----*/

/*----- general defines -----*/
typedef int    *HANDLE;
typedef int    *handle;

#define bool    int
#define true    1
#define TRUE    1
#define false   0
#define FALSE   0

#define global  extern
#define exfunc
#define local   static
#define null    0L

/*----- object types -----*/
#define accModule      20
#define accScope       21
#define accNet         25
#define accReg         30
#define accRegister    accReg
#define accPort        35
#define accTerminal    45

```

#define	accInputTerminal	46
#define	accOutputTerminal	47
#define	accInoutTerminal	48
#define	accCombPrim	140
#define	accSeqPrim	142
#define	accAndGate	144
#define	accNandGate	146
#define	accNorGate	148
#define	accOrGate	150
#define	accXorGate	152
#define	accXnorGate	154
#define	accBufGate	156
#define	accNotGate	158
#define	accBufif0Gate	160
#define	accBufif1Gate	162
#define	accNotif0Gate	164
#define	accNotif1Gate	166
#define	accNmosGate	168
#define	accPmosGate	170
#define	accCmosGate	172
#define	accRnmosGate	174
#define	accRpmosGate	176
#define	accRcmosGate	178
#define	accRtranGate	180
#define	accRtranif0Gate	182
#define	accRtranif1Gate	184
#define	accTranGate	186
#define	accTranif0Gate	188
#define	accTranif1Gate	190
#define	accPullupGate	192
#define	accPulldownGate	194
#define	accIntegerParam	200
#define	accIntParam	accIntegerParam
#define	accRealParam	202
#define	accStringParam	204
#define	accPath	206
#define	accTchk	208
#define	accPrimitive	210
#define	accPortBit	214
#define	accNetBit	216
#define	accRegBit	218
#define	accParameter	220
#define	accSpecparam	222
#define	accTopModule	224
#define	accModuleInstance	226
#define	accCellInstance	228
#define	accModPath	230
#define	accWirePath	234
#define	accInterModPath	236
#define	accScalarPort	250
#define	accPartSelectPort	254
#define	accVectorPort	256
#define	accConcatPort	258
#define	accWire	260
#define	accWand	261
#define	accWor	262

#define	accTri	263
#define	accTriand	264
#define	accTrior	265
#define	accTri0	266
#define	accTri1	267
#define	accTrireg	268
#define	accSupply0	269
#define	accSupply1	270
#define	accNamedEvent	280
#define	accEventVar	accNamedEvent
#define	accIntegerVar	281
#define	accIntVar	281
#define	accRealVar	282
#define	accTimeVar	283
#define	accScalar	300
#define	accVector	302
#define	accCollapsedNet	304
#define	accExpandedVector	306
#define	accUnExpandedVector	307
#define	accSetup	366
#define	accHold	367
#define	accWidth	368
#define	accPeriod	369
#define	accRecovery	370
#define	accSkew	371
#define	accNochange	376
#define	accNoChange	accNochange
#define	accSetuphold	377
#define	accInput	402
#define	accOutput	404
#define	accInout	406
#define	accMixedIo	407
#define	accPositive	408
#define	accNegative	410
#define	accUnknown	412
#define	accPathTerminal	420
#define	accPathInput	422
#define	accPathOutput	424
#define	accDataPath	426
#define	accTchkTerminal	428
#define	accBitSelect	500
#define	accPartSelect	502
#define	accTask	504
#define	accFunction	506
#define	accStatement	508
#define	accSystemTask	514
#define	accSystemFunction	516
#define	accSystemRealFunction	518
#define	accUserTask	520
#define	accUserFunction	522
#define	accUserRealFunction	524
#define	accNamedBeginStat	560
#define	accNamedForkStat	564
#define	accConstant	600
#define	accConcat	610
#define	accOperator	620


```

#define      accMinTypMax          696
#define      accModPathHasIfnone   715
/*----- parameter values for acc_configure() -----*/
#define      accPathDelayCount     1
#define      accPathDelimStr       2
#define      accDisplayErrors      3
#define      accDefaultAttr0       4
#define      accToHiZDelay         5
#define      accEnableArgs         6
#define      accDisplayWarnings    8
#define      accDevelopmentVersion 11
#define      accMapToMipd          17
#define      accMinTypMaxDelays    19

/*----- edge information used by acc_handle_tchk(), etc. -----*/
#define accNoedge                0
#define accNoEdge                0
#define accEdge01                1
#define accEdge10                2
#define accEdge0x                4
#define accEdgex1                8
#define accEdgex1x              16
#define accEdgex0               32
#define accPosedge               13
#define accPosEdge               accPosedge
#define accNegedge               50
#define accNegEdge               accNegedge

/*----- delay modes -----*/
#define accDelayModeNone         0
#define accDelayModePath         1
#define accDelayModeDistrib      2
#define accDelayModeUnit         3
#define accDelayModeZero         4
#define accDelayModeMTM          5

/*----- values for type field in t_setval_delay structure -----*/
#define accNoDelay               0
#define accInertialDelay         1
#define accTransportDelay        2
#define accPureTransportDelay    3
#define accForceFlag             4
#define accReleaseFlag           5
#define accAssignFlag            6
#define accDeassignFlag          7

/*----- values for type field in t_setval_value structure -----*/
#define accBinStrVal             1
#define accOctStrVal             2
#define accDecStrVal             3
#define accHexStrVal             4
#define accScalarVal             5
#define accIntVal                6
#define accRealVal               7
#define accStringVal             8
#define accVectorVal             10

```

```

/*----- scalar values -----*/
#define acc0          0
#define acc1          1
#define accX          2
#define accZ          3

/*----- VCL scalar values -----*/
#define vcl0          acc0
#define vcl1          acc1
#define vclX          accX
#define vclx          vclX
#define vclZ          accZ
#define vclz          vclZ

/*----- values for vc_reason field in t_vc_record structure -----*/
#define logic_value_change      1
#define strength_value_change   2
#define real_value_change       3
#define vector_value_change     4
#define event_value_change      5
#define integer_value_change     6
#define time_value_change       7
#define sregister_value_change   8
#define vregister_value_change   9

/*----- VCL strength values -----*/
#define vclSupply      7
#define vclStrong      6
#define vclPull        5
#define vclLarge       4
#define vclWeak        3
#define vclMedium      2
#define vclSmall       1
#define vclHighZ       0

/*----- flags used with acc_vcl_add -----*/
#define vcl_verilog_logic      2
#define VCL_VERILOG_LOGIC      vcl_verilog_logic
#define vcl_verilog_strength   3
#define VCL_VERILOG_STRENGTH    vcl_verilog_strength

/*----- flags used with acc_vcl_delete -----*/
#define vcl_verilog      vcl_verilog_logic
#define VCL_VERILOG      vcl_verilog

/*----- values for the type field in the t_acc_time structure ----- */
#define accTime          1
#define accSimTime       2
#define accRealTime      3

/*----- product types -----*/
#define accSimulator      1
#define accTimingAnalyzer 2
#define accFaultSimulator 3

```

```
#define accOther                                4

/*-----*/
/*----- global variable definitions -----*/
/*-----*/
extern bool acc_error_flag;
typedef int (*consumer_function)();

/*-----*/
/*----- structure definitions -----*/
/*-----*/

/*----- data structure used with acc_set_value() -----*/
typedef struct t_acc_time
{
    int type;
    int low,
        high;
    double real;
} s_acc_time, *p_acc_time;

/*----- data structure used with acc_set_value() -----*/
typedef struct t_setval_delay
{
    s_acc_time time;
    int model;
} s_setval_delay, *p_setval_delay;

/*----- data structure of vector values -----*/
typedef struct t_acc_vecval
{
    int aval;
    int bval;
} s_acc_vecval, *p_acc_vecval;

/*----- data structure used with acc_set_value() and acc_fetch_value() -----*/
typedef struct t_setval_value
{
    int format;
    union
    {
        {
            char *str;
            int scalar;
            int integer;
            double real;
            p_acc_vecval vector;
        } value;
    }
} s_setval_value, *p_setval_value, s_acc_value, *p_acc_value;

/*----- structure for VCL strengths -----*/
typedef struct t_strengths
{
    unsigned char logic_value;
    unsigned char strength1;
    unsigned char strength2;
} s_strengths, *p_strengths;
```

```

/*----- structure passed to callback routine for VCL -----*/
typedef struct t_vc_record
{
    int vc_reason;
    int vc_hightime;
    int vc_lowtime;
    char *user_data;
    union
    {
        unsigned char logic_value;
        double real_value;
        handle vector_handle;
        s_strengths strengths_s;
    } out_value;
} s_vc_record, *p_vc_record;

/*----- structure used with acc_fetch_location() routine -----*/
typedef struct t_location
{
    int line_no;
    char *filename;
} s_location, *p_location;

/*----- structure used with acc_fetch_timescale_info() routine -----*/
typedef struct t_timescale_info
{
    short unit;
    short precision;
} s_timescale_info, *p_timescale_info;

/*-----*/
/*----- routine declarations -----*/
/*-----*/
#if defined(__STDC__) || defined(__cplusplus)

#ifndef PROTO_PARAMS
#define PROTO_PARAMS(params) params
#define DEFINED_PROTO_PARAMS
#endif
#ifndef EXTERN
#define EXTERN
#define DEFINED_EXTERN
#endif

#else

#ifndef PROTO_PARAMS
#define PROTO_PARAMS(params) (/* nothing */)
#define DEFINED_PROTO_PARAMS
#endif
#ifndef EXTERN
#define EXTERN extern
#define DEFINED_EXTERN

```

```
#endif
#endif /* __STDC__ */
EXTERN bool    acc_append_delays PROTO_PARAMS((handle object, ...));
EXTERN bool    acc_append_pulsere PROTO_PARAMS((handle object, double vallr,
double vallx, ...));
EXTERN void    acc_close PROTO_PARAMS((void));
EXTERN handle *acc_collect PROTO_PARAMS((handle (*p_next_routine)(),
handle scope_object, int *aof_count));
EXTERN bool    acc_compare_handles PROTO_PARAMS((handle h1, handle h2));
EXTERN bool    acc_configure PROTO_PARAMS((int item, char *value));
EXTERN int     acc_count PROTO_PARAMS((handle (*next_func)(),
handle object_handle));
EXTERN int     acc_fetch_argc PROTO_PARAMS((void));
EXTERN char **acc_fetch_argv PROTO_PARAMS((void));
EXTERN double  acc_fetch_attribute PROTO_PARAMS((handle object, ...));
EXTERN int     acc_fetch_attribute_int PROTO_PARAMS((handle object, ...));
EXTERN char *acc_fetch_attribute_str PROTO_PARAMS((handle object, ...));
EXTERN char *acc_fetch_defname PROTO_PARAMS((handle object_handle));
EXTERN int     acc_fetch_delay_mode PROTO_PARAMS((handle object_p));
EXTERN bool    acc_fetch_delays PROTO_PARAMS((handle object, ...));
EXTERN int     acc_fetch_direction PROTO_PARAMS((handle object_handle));
EXTERN int     acc_fetch_edge PROTO_PARAMS((handle acc_obj));
EXTERN char *acc_fetch_fullname PROTO_PARAMS((handle object_handle));
EXTERN int     acc_fetch_fulltype PROTO_PARAMS((handle object_h));
EXTERN int     acc_fetch_index PROTO_PARAMS((handle object_handle));
EXTERN double  acc_fetch_itfarg PROTO_PARAMS((int n, handle tfinst));
EXTERN int     acc_fetch_itfarg_int PROTO_PARAMS((int n, handle tfinst));
EXTERN char *acc_fetch_itfarg_str PROTO_PARAMS((int n, handle tfinst));
EXTERN int     acc_fetch_location PROTO_PARAMS((p_location location_p,
handle object));
EXTERN char *acc_fetch_name PROTO_PARAMS((handle object_handle));
EXTERN int     acc_fetch_paramtype PROTO_PARAMS((handle param_p));
EXTERN double  acc_fetch_paramval PROTO_PARAMS((handle param));
EXTERN int     acc_fetch_polarity PROTO_PARAMS((handle path));
EXTERN int     acc_fetch_precision PROTO_PARAMS((void));
EXTERN bool    acc_fetch_pulsere PROTO_PARAMS((handle path_p, double *vallr,
double *valle, ...));
EXTERN int     acc_fetch_range PROTO_PARAMS((handle node, int *msb,
int *lsb));
EXTERN int     acc_fetch_size PROTO_PARAMS((handle obj_h));
EXTERN double  acc_fetch_tfarg PROTO_PARAMS((int n));
EXTERN int     acc_fetch_tfarg_int PROTO_PARAMS((int n));
EXTERN char *acc_fetch_tfarg_str PROTO_PARAMS((int n));
EXTERN void    acc_fetch_timescale_info PROTO_PARAMS((handle obj,
p_timescale_info aof_timescale_info));
EXTERN int     acc_fetch_type PROTO_PARAMS((handle object_handle));
EXTERN char *acc_fetch_type_str PROTO_PARAMS((int type));
EXTERN char *acc_fetch_value PROTO_PARAMS((handle object_handle,
char *format_str, p_acc_value acc_value_p));
EXTERN void    acc_free PROTO_PARAMS((handle *array_ptr));
EXTERN handle  acc_handle_by_name PROTO_PARAMS((char *inst_name,
handle scope_p));
EXTERN handle  acc_handle_condition PROTO_PARAMS((handle obj));
EXTERN handle  acc_handle_conn PROTO_PARAMS((handle term_p));
EXTERN handle  acc_handle_datapath PROTO_PARAMS((handle path));
EXTERN handle  acc_handle_hiconn PROTO_PARAMS((handle port_ref));
```

```

EXTERN handle acc_handle_interactive_scope PROTO_PARAMS((void));
EXTERN handle acc_handle_itfarg PROTO_PARAMS((int n, void *suena_inst));
EXTERN handle acc_handle_loconn PROTO_PARAMS((handle port_ref));
EXTERN handle acc_handle_modpath PROTO_PARAMS((handle mod_p,
        char *pathin_name, char *pathout_name, ...));
EXTERN handle acc_handle_notifier PROTO_PARAMS((handle tchk));
EXTERN handle acc_handle_object PROTO_PARAMS((char *inst_name, ...));
EXTERN handle acc_handle_parent PROTO_PARAMS((handle object_p));
EXTERN handle acc_handle_path PROTO_PARAMS((handle source,
        handle destination));
EXTERN handle acc_handle_pathin PROTO_PARAMS((handle path_p));
EXTERN handle acc_handle_pathout PROTO_PARAMS((handle path_p));
EXTERN handle acc_handle_port PROTO_PARAMS((handle mod_handle,
        int port_num, ...));
EXTERN handle acc_handle_scope PROTO_PARAMS((handle object));
EXTERN handle acc_handle_simulated_net PROTO_PARAMS((handle net_h));
EXTERN handle acc_handle_tchk PROTO_PARAMS((handle mod_p, int tchk_type,
        char *arg1_conn_name, int arg1_edgetype, ...));
EXTERN handle acc_handle_tchkarg1 PROTO_PARAMS((handle tchk));
EXTERN handle acc_handle_tchkarg2 PROTO_PARAMS((handle tchk));
EXTERN handle acc_handle_terminal PROTO_PARAMS((handle gate_handle,
        int terminal_index));
EXTERN handle acc_handle_tfarg PROTO_PARAMS((int n));
EXTERN handle acc_handle_tfinst PROTO_PARAMS((void));
EXTERN bool acc_initialize PROTO_PARAMS((void));
EXTERN handle acc_next PROTO_PARAMS((int *type_list, handle h_scope,
        handle h_object));
EXTERN handle acc_next_bit PROTO_PARAMS ((handle vector, handle bit));
EXTERN handle acc_next_cell PROTO_PARAMS((handle scope, handle cell));

EXTERN handle acc_next_cell_load PROTO_PARAMS((handle net_handle,
        handle load));
EXTERN handle acc_next_child PROTO_PARAMS((handle mod_handle, handle child));
EXTERN handle acc_next_driver PROTO_PARAMS((handle net, handle driver));
EXTERN handle acc_next_hiconn PROTO_PARAMS((handle port, handle hiconn));
EXTERN handle acc_next_input PROTO_PARAMS((handle path, handle pathin));
EXTERN handle acc_next_load PROTO_PARAMS((handle net, handle load));
EXTERN handle acc_next_loconn PROTO_PARAMS((handle port, handle loconn));
EXTERN handle acc_next_modpath PROTO_PARAMS((handle mod_p, handle path));
EXTERN handle acc_next_net PROTO_PARAMS((handle mod_handle, handle net));
EXTERN handle acc_next_output PROTO_PARAMS((handle path, handle pathout));
EXTERN handle acc_next_parameter PROTO_PARAMS((handle module_p,
        handle param));
EXTERN handle acc_next_port PROTO_PARAMS((handle ref_obj_p, handle port));
EXTERN handle acc_next_portout PROTO_PARAMS((handle mod_p, handle port));
EXTERN handle acc_next_primitive PROTO_PARAMS((handle mod_handle,
        handle prim));
EXTERN handle acc_next_scope PROTO_PARAMS((handle ref_scope_p,
        handle scope));
EXTERN handle acc_next_specparam PROTO_PARAMS((handle module_p,
        handle sparam));
EXTERN handle acc_next_tchk PROTO_PARAMS((handle mod_p, handle tchk));
EXTERN handle acc_next_terminal PROTO_PARAMS((handle gate_handle,
        handle term));
EXTERN handle acc_next_topmod PROTO_PARAMS((handle topmod));
EXTERN bool acc_object_of_type PROTO_PARAMS((handle object, int type));

```

```

EXTERN bool    acc_object_in_typelist PROTO_PARAMS((handle object,
                                                    int *type_list));
EXTERN int     acc_product_type PROTO_PARAMS((void));
EXTERN char    *acc_product_version PROTO_PARAMS((void));
EXTERN int     acc_release_object PROTO_PARAMS((handle obj));
EXTERN bool    acc_replace_delays PROTO_PARAMS((handle object, ...));
EXTERN bool    acc_replace_pulsere PROTO_PARAMS((handle object, double vallr,
                                                    double vallx, ...));
EXTERN void    acc_reset_buffer PROTO_PARAMS((void));
EXTERN bool    acc_set_interactive_scope PROTO_PARAMS((handle scope,
                                                    int callback_flag));
EXTERN bool    acc_set_pulsere PROTO_PARAMS((handle path_p, double vallr,
                                                    double valle));
EXTERN char    *acc_set_scope PROTO_PARAMS((handle object, ...));
EXTERN int     acc_set_value PROTO_PARAMS((handle obj,
                                                    p_setval_value setval_p, p_setval_delay delay_p));
EXTERN void    acc_vcl_add PROTO_PARAMS((handle object_p, int (*consumer)(),
                                                    char *user_data, int vcl_flags));
EXTERN void    acc_vcl_delete PROTO_PARAMS((handle object_p,
                                                    int (*consumer)(), char *user_data, int vcl_flags));
EXTERN char    *acc_version PROTO_PARAMS((void));

#ifdef DEFINED_PROTO_PARAMS
#undef DEFINED_PROTO_PARAMS
#undef PROTO_PARAMS
#endif

#ifdef DEFINED_EXTERN
#undef DEFINED_EXTERN
#undef EXTERN
#endif

/*-----*/
/*----- macro definitions -----*/
/*-----*/
#define acc_handle_calling_mod_m acc_handle_parent((handle)tf_getinstance())

#endif /* ACC_USER_H */

```

Annex D

(normative)

The veriuser.h file

```

/*****
 * veriuser.h
 *
 * IEEE Std 1364-1995 Verilog Hardware Description Language (HDL)
 * Programming Language Interface (PLI).
 *
 * This file contains the constant definitions, structure definitions, and
 * routine declarations used by the Verilog PLI procedural interface
 * task/function (TF) routines.
 *
 * The file should be included with all C routines that use the PLI TF
 * routines.
 *****/
#ifndef VERIUSER_H
#define VERIUSER_H

/*-----*/
/*----- definitions -----*/
/*-----*/

/*----- general defines -----*/
#define true          1
#define TRUE          1
#define false         0
#define FALSE         0
#define bool          int

/*----- defines for error interception -----*/
#define ERR_MESSAGE    1
#define ERR_WARNING    2
#define ERR_ERROR      3
#define ERR_INTERNAL   4
#define ERR_SYSTEM     5

/*----- values for reason parameter to misctf routines -----*/
#define reason_checktf  1
#define REASON_CHECKTF  reason_checktf
#define reason_sizetf   2
#define REASON_SIZETF   reason_sizetf
#define reason_calltf   3
#define REASON_CALLTF   reason_calltf
#define reason_save     4

```



```

#define REASON_SAVE          reason_save
#define reason_restart      5
#define REASON_RESTART      reason_restart
#define reason_disable      6
#define REASON_DISABLE      reason_disable
#define reason_paramvc      7
#define REASON_PARAMVC      reason_paramvc
#define reason_synch        8
#define REASON_SYNCH        reason_synch
#define reason_finish       9
#define REASON_FINISH       reason_finish
#define reason_reactivate  10
#define REASON_REACTIVATE   reason_reactivate
#define reason_rosynch     11
#define REASON_ROSYNCH      reason_rosynch
#define reason_paramdrc    15
#define REASON_PARAMDRC     reason_paramdrc
#define reason_endofcompile 16
#define REASON_ENDOF_COMPILE reason_endofcompile
#define reason_scope       17
#define REASON_SCOPE        reason_scope
#define reason_interactive  18
#define REASON_INTERACTIVE  reason_interactive
#define reason_reset       19
#define REASON_RESET        reason_reset
#define reason_endofreset  20
#define REASON_ENDOFRESET   reason_endofreset
#define reason_force       21
#define REASON_FORCE        reason_force
#define reason_release     22
#define REASON_RELEASE      reason_release
#define reason_startofsave  27
#define reason_startofrestart 28
#define REASON_MAX          28

/*-- types used by tf_typep() and expr_type field in tf_exprinfo structure --*/
#define tf_nullparam        0
#define TF_NULLPARAM        tf_nullparam
#define tf_string           1
#define TF_STRING           tf_string
#define tf_readonly         10
#define TF_READONLY         tf_readonly
#define tf_readwrite        11
#define TF_READWRITE        tf_readwrite
#define tf_rwbselect        12
#define TF_RWBSELECT        tf_rwbselect
#define tf_rwpselect        13
#define TF_RWPSELECT        tf_rwpselect
#define tf_rwmselect        14
#define TF_RWMSELECT        tf_rwmselect
#define tf_readonlyreal     15
#define TF_READONLYREAL     tf_readonlyreal
#define tf_readwritereal    16
#define TF_READWRITEREAL    tf_readwritereal

```

```

/*----- types used by node_type field in tf_nodeinfo structure -----*/
#define tf_null_node          100
#define TF_NULL_NODE          tf_null_node
#define tf_reg_node           101
#define TF_REG_NODE           tf_reg_node
#define tf_integer_node       102
#define TF_INTEGER_NODE       tf_integer_node
#define tf_time_node          103
#define TF_TIME_NODE          tf_time_node
#define tf_netvector_node     104
#define TF_NETVECTOR_NODE     tf_netvector_node
#define tf_netscalar_node     105
#define TF_NETSCALAR_NODE     tf_netscalar_node
#define tf_memory_node        106
#define TF_MEMORY_NODE        tf_memory_node
#define tf_real_node          107
#define TF_REAL_NODE          tf_real_node

/*-----*/
/*----- structure definitions -----*/
/*-----*/

/*---- structure used with tf_exprinfo() to get expression information ----*/
typedef struct t_tfexprinfo
{
    short expr_type;
    short padding;
    struct t_vecval *expr_value_p;
    double real_value;
    char *expr_string;
    int expr_ngroups;
    int expr_vec_size;
    int expr_sign;
    int expr_lhs_select;
    int expr_rhs_select;
} s_tfexprinfo, *p_tfexprinfo;

/*----- structure for use with tf_nodeinfo() to get node information -----*/
typedef struct t_tfnodeinfo
{
    short node_type;
    short padding;
    union
    {
        {
            struct t_vecval *vecval_p;
            struct t_strengthval *strengthval_p;
            char *memoryval_p;
            double *real_val_p;
        } node_value;
    }
    char *node_symbol;
    int node_ngroups;
    int node_vec_size;
    int node_sign;
    int node_ms_index;
}

```

```

    int node_ls_index;
    int node_mem_size;
    int node_lhs_element;
    int node_rhs_element;
    int *node_handle;
} s_tfnodeinfo, *p_tfnodeinfo;

/*----- data structure of vector values -----*/
typedef struct t_vecval
{
    int avalbits;
    int bvalbits;
} s_vecval, *p_vecval;

/*----- data structure of scalar net strength values -----*/
typedef struct t_strengthval
{
    int strength0;
    int strength1;
} s_strengthval, *p_strengthval;

/*-----*/
/*----- routine definitions -----*/
/*-----*/
#if defined(__STDC__) || defined(__cplusplus)

#ifndef PROTO_PARAMS
#define PROTO_PARAMS(params) params
#define DEFINED_PROTO_PARAMS
#endif
#ifndef EXTERN
#define EXTERN
#define DEFINED_EXTERN
#endif

#else

#ifndef PROTO_PARAMS
#define PROTO_PARAMS(params) (/* nothing */)
#define DEFINED_PROTO_PARAMS
#endif
#ifndef EXTERN
#define EXTERN extern
#define DEFINED_EXTERN
#endif

#endif /* __STDC__ */

EXTERN void          io_mcdprintf PROTO_PARAMS((int mcd, char *format, ...));
EXTERN void          io_printf PROTO_PARAMS((char *format, ...));
EXTERN char          *mc_scan_plusargs PROTO_PARAMS((char *plusarg));
EXTERN int           tf_add_long PROTO_PARAMS((int *aof_lowtime1,
                                                int *aof_hightime1, int lowtime2, int hightime2));
EXTERN int           tf_asynchoff PROTO_PARAMS((void));
EXTERN int           tf_asynchon PROTO_PARAMS((void));
EXTERN int           tf_clearalldelays PROTO_PARAMS((void));

```

```

EXTERN int      tf_compare_long PROTO_PARAMS((unsigned int low1,
        unsigned int high1, unsigned int low2,
        unsigned int high2));
EXTERN int      tf_copypvc_flag PROTO_PARAMS((int nparam));
EXTERN void     tf_divide_long PROTO_PARAMS((int *aof_low1,
        int *aof_high1, int low2, int high2));
EXTERN int      tf_dofinish PROTO_PARAMS((void));
EXTERN int      tf_dostop PROTO_PARAMS((void));
EXTERN int      tf_error PROTO_PARAMS((char *fmt, ...));
EXTERN int      tf_evaluatep PROTO_PARAMS((int pnum));
EXTERN p_tfexprinfo tf_exprinfo PROTO_PARAMS((int pnum, p_tfexprinfo pinfo));
EXTERN char     *tf_getcstringp PROTO_PARAMS((int nparam));
EXTERN char     *tf_getinstance PROTO_PARAMS((void));
EXTERN int      tf_getlongp PROTO_PARAMS((int *aof_highvalue, int pnum));
EXTERN int      tf_getlongtime PROTO_PARAMS((int *aof_hightime));
EXTERN int      tf_getnextlongtime PROTO_PARAMS((int *aof_lowtime,
        int *aof_hightime));
EXTERN int      tf_getp PROTO_PARAMS((int pnum));
EXTERN int      tf_getpchange PROTO_PARAMS((int nparam));
EXTERN double   tf_getrealp PROTO_PARAMS((int pnum));
EXTERN double   tf_getrealtime PROTO_PARAMS((void));
EXTERN int      tf_gettime PROTO_PARAMS((void));
EXTERN int      tf_gettimeprecision PROTO_PARAMS((void));
EXTERN int      tf_gettimeunit PROTO_PARAMS((void));
EXTERN char     *tf_getworkarea PROTO_PARAMS((void));
EXTERN int      tf_iasynchoff PROTO_PARAMS((char *inst));
EXTERN int      tf_iasynchon PROTO_PARAMS((char *inst));
EXTERN int      tf_iclearalldelays PROTO_PARAMS((char *inst));
EXTERN int      tf_icopypvc_flag PROTO_PARAMS((int nparam, char *inst));
EXTERN int      tf_ievaluatep PROTO_PARAMS((int pnum, char *inst));
EXTERN p_tfexprinfo tf_iexprinfo PROTO_PARAMS((int pnum, p_tfexprinfo pinfo,
        char *inst));
EXTERN char     *tf_igetcstringp PROTO_PARAMS((int nparam, char *inst));
EXTERN int      tf_igetlongp PROTO_PARAMS((int *aof_highvalue, int pnum,
        char *inst));
EXTERN int      tf_igetlongtime PROTO_PARAMS((int *aof_hightime,
        char *inst));
EXTERN int      tf_igetp PROTO_PARAMS((int pnum, char *inst));
EXTERN int      tf_igetpchange PROTO_PARAMS((int nparam, char *inst));
EXTERN double   tf_igetrealp PROTO_PARAMS((int pnum, char *inst));
EXTERN double   tf_igetrealtime PROTO_PARAMS((char *inst));
EXTERN int      tf_igettime PROTO_PARAMS((char *inst));
EXTERN int      tf_igettimeprecision PROTO_PARAMS((char *inst));
EXTERN int      tf_igettimeunit PROTO_PARAMS((char *inst));
EXTERN char     *tf_igetworkarea PROTO_PARAMS((char *inst));
EXTERN char     *tf_imipname PROTO_PARAMS((char *cell));
EXTERN int      tf_imovepvc_flag PROTO_PARAMS((int nparam, char *inst));
EXTERN p_tfnodeinfo tf_inodeinfo PROTO_PARAMS((int pnum, p_tfnodeinfo pinfo,
        char *inst));
EXTERN int      tf_inump PROTO_PARAMS((char *inst));
EXTERN int      tf_ipropagatep PROTO_PARAMS((int pnum, char *inst));
EXTERN int      tf_iputlongp PROTO_PARAMS((int pnum, int lowvalue,
        int highvalue, char *inst));
EXTERN int      tf_iputp PROTO_PARAMS((int pnum, int value, char *inst));

```

```

EXTERN int      tf_iputrealp PROTO_PARAMS((int pnum, double value,
char *inst));
EXTERN int      tf_irosynchronize PROTO_PARAMS((char *inst));
EXTERN int      tf_isetdelay PROTO_PARAMS((int delay, char *inst));
EXTERN int      tf_isetlongdelay PROTO_PARAMS((int lowdelay,
int highdelay, char *inst));
EXTERN int      tf_isetrealdelay PROTO_PARAMS((double realdelay,
char *inst));
EXTERN int      tf_isetworkarea PROTO_PARAMS((char *workarea,
char *inst));
EXTERN int      tf_isizep PROTO_PARAMS((int pnum, char *inst));
EXTERN char     *tf_ispname PROTO_PARAMS((char *cell));
EXTERN int      tf_istrdelputp PROTO_PARAMS((int nparam, int bitlength,
int format_char, char *value_p, int delay,
int delaytype, char *inst));
EXTERN char     *tf_istrgetp PROTO_PARAMS((int pnum, int format_char,
char *inst));
EXTERN int      tf_istrlongdelputp PROTO_PARAMS((int nparam,
int bitlength, int format_char, char *value_p,
int lowdelay, int highdelay, int delaytype,
char *inst));
EXTERN int      tf_istrrealdelputp PROTO_PARAMS((int nparam,
int bitlength, int format_char, char *value_p,
double realdelay, int delaytype, char *inst));
EXTERN int      tf_isynchronize PROTO_PARAMS((char *inst));
EXTERN int      tf_itestpvc_flag PROTO_PARAMS((int nparam, char *inst));
EXTERN int      tf_itypep PROTO_PARAMS((int pnum, char *inst));
EXTERN void     tf_long_to_real PROTO_PARAMS((int int_lo, int int_hi,
double *aof_real));
EXTERN char     *tf_longtime_tostr PROTO_PARAMS((int lowtime,
int hightime));
EXTERN int      tf_message PROTO_PARAMS((int level, char *facility,
char *messno, char *message, ...));
EXTERN char     *tf_mipname PROTO_PARAMS((void));
EXTERN int      tf_movepvc_flag PROTO_PARAMS((int nparam));
EXTERN void     tf_multiply_long PROTO_PARAMS((int *aof_low1,
int *aof_high1, int low2, int high2));
EXTERN p_tfnodeinfo tf_nodeinfo PROTO_PARAMS((int pnum, p_tfnodeinfo pinfo));
EXTERN int      tf_nump PROTO_PARAMS((void));
EXTERN int      tf_propagatep PROTO_PARAMS((int pnum));
EXTERN int      tf_putlongp PROTO_PARAMS((int pnum, int lowvalue,
int highvalue));
EXTERN int      tf_putp PROTO_PARAMS((int pnum, int value));
EXTERN int      tf_putrealp PROTO_PARAMS((int pnum, double value));
EXTERN int      tf_read_restart PROTO_PARAMS((char *blockptr,
int blocklen));
EXTERN void     tf_real_to_long PROTO_PARAMS((double real,
int *aof_int_lo, int *aof_int_hi));
EXTERN int      tf_rosynchronize PROTO_PARAMS((void));
EXTERN void     tf_scale_longdelay PROTO_PARAMS((char *cell,
int delay_lo, int delay_hi, int *aof_delay_lo,
int *aof_delay_hi));
EXTERN void     tf_scale_realdelay PROTO_PARAMS((char *cell,
double realdelay, double *aof_realdelay));
EXTERN int      tf_setdelay PROTO_PARAMS((int delay));
EXTERN int      tf_setlongdelay PROTO_PARAMS((int lowdelay,

```

```

        int highdelay));
EXTERN int      tf_setrealdelay PROTO_PARAMS((double realdelay));
EXTERN int      tf_setworkarea PROTO_PARAMS((char *workarea));
EXTERN int      tf_sizep PROTO_PARAMS((int pnum));
EXTERN char     *tf_spname PROTO_PARAMS((void));
EXTERN int      tf_strdelputp PROTO_PARAMS((int nparam, int bitlength,
        int format_char, char *value_p, int delay,
        int delaytype));
EXTERN char     *tf_strgetp PROTO_PARAMS((int pnum, int format_char));
EXTERN char     *tf_strgettime PROTO_PARAMS((void));
EXTERN int      tf_strlongdelputp PROTO_PARAMS((int nparam,
        int bitlength, int format_char, char *value_p,
        int lowdelay, int highdelay, int delaytype));
EXTERN int      tf_strrealdelputp PROTO_PARAMS((int nparam,
        int bitlength, int format_char, char *value_p,
        double realdelay, int delaytype));
EXTERN int      tf_subtract_long PROTO_PARAMS((int *aof_lowtime1,
        int *aof_hightime1, int lowtime2, int hightime2));
EXTERN int      tf_synchronize PROTO_PARAMS((void));
EXTERN int      tf_testpvc_flag PROTO_PARAMS((int nparam));
EXTERN int      tf_text PROTO_PARAMS((char *fmt, ...));
EXTERN int      tf_typep PROTO_PARAMS((int pnum));
EXTERN void     tf_unscale_longdelay PROTO_PARAMS((char *cell,
        int delay_lo, int delay_hi, int *aof_delay_lo,
        int *aof_delay_hi));
EXTERN void     tf_unscale_realdelay PROTO_PARAMS((char *cell,
        double realdelay, double *aof_realdelay));
EXTERN int      tf_warning PROTO_PARAMS((char *fmt, ...));
EXTERN int      tf_write_save PROTO_PARAMS((char *blockptr,
        int blocklen));

#ifdef DEFINED_PROTO_PARAMS
#undef DEFINED_PROTO_PARAMS
#undef PROTO_PARAMS
#endif

#ifdef DEFINED_EXTERN
#undef DEFINED_EXTERN
#undef EXTERN
#endif

#endif /* VERIUSER_H */

```

Annex E

(normative)

The vpi_user.h file

```

/*****
 * vpi_user.h
 *
 * IEEE Std 1364-1995 Verilog Hardware Description Language (HDL)
 * Programming Language Interface (PLI).
 *
 * This file contains the constant definitions, structure definitions,
 * and routine declarations used by the Verilog PLI procedural
 * interface (VPI).
 *
 * The file should be included with all C routines that use the VPI
 * routines.
 *****/
#ifndef VPI_USER_H
#define VPI_USER_H

/* basic typedefs */

typedef unsigned long *vpiHandle;

/* The following are the constant definitions. They are divided into three
   major areas:

   1) Object types
   2) Access methods
   3) Properties

   Note that most of the object types can also be used as access
   methods and that some methods can also be used a properties.
 */

/***** OBJECT TYPES *****/
#define vpiAlways      1      /* always block */
#define vpiAssignStmnt 2      /* quasi-continuous assignment */
#define vpiAssignment  3      /* procedural assignment */
#define vpiBegin       4      /* block statement */
#define vpiCase        5      /* case statement */
#define vpiCaseItem    6      /* case statement item */
#define vpiConstant    7      /* numerical constant or literal string */
#define vpiContAssign  8      /* continuous assignment */
#define vpiDeassign    9      /* deassignment statement */
#define vpiDefParam    10     /* defparam */

```

```

#define vpiDelayControl      11      /* delay statement (e.g. #10) */
#define vpiDisable           12      /* named block disable statement */
#define vpiEventControl      13      /* wait on event, e.g. @e */
#define vpiEventStmt         14      /* event trigger, e.g. ->e */
#define vpiFor               15      /* for statement */
#define vpiForce             16      /* force statement */
#define vpiForever           17      /* forever statement */
#define vpiFork              18      /* fork-join block */
#define vpiFuncCall          19      /* HDL function call */
#define vpiFunction           20      /* HDL function */
#define vpiGate              21      /* primitive gate */
#define vpiIf                22      /* if statement */
#define vpiIfElse            23      /* if-else statement */
#define vpiInitial           24      /* initial block */
#define vpiIntegerVar        25      /* integer variable */
#define vpiInterModPath      26      /* intermodule wire delay */
#define vpiIterator          27      /* iterator */
#define vpiIODecl            28      /* input/output declaration */
#define vpiMemory            29      /* behavioral memory */
#define vpiMemoryWord        30      /* single word of memory */
#define vpiModPath           31      /* module path for path delays */
#define vpiModule            32      /* module instance */
#define vpiNamedBegin        33      /* named block statement */
#define vpiNamedEvent        34      /* event variable */
#define vpiNamedFork         35      /* named fork-join block */
#define vpiNet               36      /* scalar or vector net */
#define vpiNetBit            37      /* bit of vector net */
#define vpiNullStmt          38      /* a semicolon (e.g., #10) */
#define vpiOperation         39      /* behavioral operation */
#define vpiParamAssign       40      /* module parameter assignment */
#define vpiParameter         41      /* module parameter */
#define vpiPartSelect        42      /* part select */
#define vpiPathTerm          43      /* terminal of module path */
#define vpiPort              44      /* module port */
#define vpiPortBit           45      /* bit of vector module port */
#define vpiPrimTerm          46      /* primitive terminal */
#define vpiRealVar           47      /* real variable */
#define vpiReg               48      /* scalar or vector register */
#define vpiRegBit            49      /* bit of vector register net */
#define vpiRelease           50      /* release statement */
#define vpiRepeat            51      /* repeat statement */
#define vpiRepeatControl     52      /* repeat control in an assign stmt */
#define vpiSchedEvent        53      /* vpi_put_value() event */
#define vpiSpecParam         54      /* specparam */
#define vpiSwitch            55      /* transistor switch */
#define vpiSysFuncCall       56      /* system function call */
#define vpiSysTaskCall       57      /* system task call */
#define vpiTableEntry        58      /* UDP state table entry */
#define vpiTask              59      /* HDL task */
#define vpiTaskCall          60      /* HDL task call */
#define vpiTchk              61      /* timing check */
#define vpiTchkTerm          62      /* terminal of timing check */
#define vpiTimeVar           63      /* time variable */
#define vpiTimeQueue         64      /* simulation event queue */
#define vpiUdp               65      /* user-defined primitive */
#define vpiUdpDefn           66      /* UDP definition */

```



```

#define vpiUserSystf      67      /* user-defined system task or function */
#define vpiVarSelect      68      /* variable array selection */
#define vpiWait           69      /* wait statement */
#define vpiWhile          70      /* while statement */

/***** METHODS *****/
/***** methods used to traverse one-to-one relationships *****/
#define vpiCondition      71      /* condition expression */
#define vpiDelay          72      /* net or gate delay */
#define vpiElseStmt       73      /* else statement */
#define vpiForIncStmt     74      /* increment statement in for loop */
#define vpiForInitStmt    75      /* initialization statement in for loop */
#define vpiHighConn       76      /* higher connection to port */
#define vpiLhs            77      /* left-hand side of assignment */
#define vpiIndex          78      /* index of var select, bit select, etc. */
#define vpiLeftRange      79      /* left range of vector or part select */
#define vpiLowConn        80      /* lower connection to port */
#define vpiParent         81      /* parent object */
#define vpiRhs            82      /* right-hand side of assignment */
#define vpiRightRange     83      /* right range of vector or part select */
#define vpiScope          84      /* containing scope object */
#define vpiSysTfCall      85      /* task function call */
#define vpiTchkDataTerm   86      /* timing check data term */
#define vpiTchkNotifier   87      /* timing check notifier */
#define vpiTchkRefTerm    88      /* timing check reference term */

/***** methods used to traverse one-to-many relationships *****/
#define vpiArgument       89      /* argument to (system) task or function */
#define vpiBit            90      /* bit of vector net or port */
#define vpiDriver         91      /* driver for a net */
#define vpiInternalScope  92      /* internal scope in module */
#define vpiLoad           93      /* load on net or register */
#define vpiModDataPathIn  94      /* data terminal of a module path */
#define vpiModPathIn      95      /* input terminal of a module path */
#define vpiModPathOut     96      /* output terminal of a module path */
#define vpiOperand        97      /* operand of expression */
#define vpiPortInst       98      /* connected port instance */
#define vpiProcess        99      /* process in module */
#define vpiVariables     100     /* variables in module */
#define vpiUse           101     /* usage */

/**** methods that can traverse one-to-one or one-to-many relationships ****/
#define vpiExpr          102     /* connected expression */
#define vpiPrimitive     103     /* primitive (gate, switch, UDP) */
#define vpiStmt          104     /* statement in process or task */

/***** PROPERTIES *****/
/***** generic object properties *****/
#define vpiUndefined     -1     /* undefined property */
#define vpiType          1      /* type of object */
#define vpiName          2      /* local name of object */
#define vpiFullName      3      /* full hierarchical name */
#define vpiSize          4      /* size of gate, net, port, etc. */
#define vpiFile          5      /* File name in which the object is used */
#define vpiLineNo        6      /* File line number where object is used */

```

```

/***** modules properties *****/
#define vpiTopModule      7    /* top-level module (Boolean) */
#define vpiCellInstance   8    /* cell (Boolean) */
#define vpiDefName        9    /* module definition name */
#define vpiProtected     10   /* source-protected module (Boolean) */
#define vpiTimeUnit      11   /* module time unit */
#define vpiTimePrecision  12   /* module time precision */
#define vpiDefNetType     13   /* default net type */
#define vpiUnconnDrive    14   /* unconnected port drive strength */
#define vpiHighZ          1    /* No default drive given */
#define vpiPull1          2    /* default pull1 drive */
#define vpiPull0          3    /* default pull0 drive */
#define vpiDefFile        15   /* File name where the module is defined */
#define vpiDefLineNo      16   /* File line number where module is defined*/
#define vpiDefDelayMode   17   /* Delay mode of the module */
#define vpiDelayModeNone  1    /* No delay mode specified */
#define vpiDelayModePath  2    /* Path delay mode */
#define vpiDelayModeDistrib 3    /* Distributed delay mode */
#define vpiDelayModeUnit  4    /* Unit delay mode */
#define vpiDelayModeZero  5    /* Zero delay mode */
#define vpiDelayModeMTM   6    /* min:typ:max delay mode */
#define vpiDefDecayTime   18   /* Decay time for trireg net */

/***** port and net properties *****/
#define vpiScalar         19   /* scalar (Boolean) */
#define vpiVector         20   /* vector (Boolean) */
#define vpiExplicitName   21   /* port is explicitly named */
#define vpiDirection      22   /* direction of port: */
#define vpiInput          1    /* input */
#define vpiOutput         2    /* output */
#define vpiInout          3    /* inout */
#define vpiMixedIO        4    /* mixed input-output */
#define vpiNoDirection    5    /* no direction */
#define vpiConnByName     23   /* connected by name (Boolean) */

#define vpiNetType        24   /* net subtypes: */
#define vpiWire           1    /* wire net */
#define vpiWand           2    /* wire-and net */
#define vpiWor            3    /* wire-or net */
#define vpiTri            4    /* tri-state net */
#define vpiTri0           5    /* pull-down net */
#define vpiTri1           6    /* pull-up net */
#define vpiTriReg         7    /* tri-state reg net */
#define vpiTriAnd         8    /* tri-state wire-and net */
#define vpiTriOr          9    /* tri-state wire-or net */
#define vpiSupply1        10   /* supply 1 net */
#define vpiSupply0        11   /* supply zero net */

#define vpiExplicitScalared 25 /* explicitly scalared (Boolean) */
#define vpiExplicitVectored 26 /* explicitly vectored (Boolean) */
#define vpiExpanded        27 /* expanded vector net (Boolean) */
#define vpiImplicitDecl    28 /* implicitly declared net (Boolean) */
#define vpiChargeStrength  29 /* charge decay strength of net */

```

```

#define vpiArray          30  /* variable array (Boolean) */
#define vpiPortIndex      31  /* port index */
/***** gate and terminal properties *****/
#define vpiTermIndex      32  /* Index of a primitive terminal */
#define vpiStrength0      33  /* 0-strength of net or gate */
#define vpiStrength1      34  /* 1-strength of net or gate */
#define vpiPrimType       35  /* primitive subtypes: */
#define vpiAndPrim        1   /* and gate */
#define vpiNandPrim       2   /* nand gate */
#define vpiNorPrim        3   /* nor gate */
#define vpiOrPrim         4   /* or gate */
#define vpiXorPrim        5   /* xor gate */
#define vpiXnorPrim       6   /* xnor gate */
#define vpiBufPrim        7   /* buffer */
#define vpiNotPrim        8   /* not gate */
#define vpiBufif0Prim     9   /* zero-enabled buffer */
#define vpiBufif1Prim    10   /* one-enabled buffer */
#define vpiNotif0Prim    11   /* zero-enabled not gate */
#define vpiNotif1Prim    12   /* one-enabled not gate */
#define vpiNmosPrim      13   /* nmos switch */
#define vpiPmosPrim      14   /* pmos switch */
#define vpiCmosPrim      15   /* cmos switch */
#define vpiRnmosPrim     16   /* resistive nmos switch */
#define vpiRpmsPrim     17   /* resistive pmos switch */
#define vpiRcmsPrim     18   /* resistive cmos switch */
#define vpiRtranPrim     19   /* resistive bidirectional */
#define vpiRtranif0Prim  20   /* zero-enable resistive bidirectional */
#define vpiRtranif1Prim  21   /* one-enable resistive bidirectional */
#define vpiTranPrim      22   /* bidirectional */
#define vpiTranif0Prim   23   /* zero-enabled bidirectional */
#define vpiTranif1Prim   24   /* one-enabled bidirectional */
#define vpiPullupPrim    25   /* pullup */
#define vpiPulldownPrim  26   /* pulldown */
#define vpiSeqPrim       27   /* sequential UDP */
#define vpiCombPrim      28   /* combinational UDP */

/***** path, path terminal, timing check properties *****/
#define vpiPolarity       36  /* polarity of module path... */
#define vpiDataPolarity   37  /* ...or data path: */
#define vpiPositive       1   /* positive */
#define vpiNegative       2   /* negative */
#define vpiUnknown        3   /* unknown (unspecified) */

#define vpiEdge           38  /* edge type of module path: */
#define vpiNoEdge         0x00000000 /* no edge */
#define vpiEdge01         0x00000001 /* 0 -> 1 */
#define vpiEdge10         0x00000002 /* 1 -> 0 */
#define vpiEdge0x         0x00000004 /* 0 -> x */
#define vpiEdgex1         0x00000008 /* x -> 1 */
#define vpiEdgelx         0x00000010 /* 1 -> x */
#define vpiEdgex0         0x00000020 /* x -> 0 */
#define vpiPosedge        (vpiEdgex1 | vpiEdge01 | vpiEdge0x)
#define vpiNegedge        (vpiEdgex0 | vpiEdge10 | vpiEdgelx)
#define vpiAnyEdge        (vpiPosedge | vpiNegedge)

#define vpiPathType       39  /* path delay connection subtypes: */

```

```

#define vpiPathFull          1      /* ( a *> b ) */
#define vpiPathParallel      2      /* ( a => b ) */
#define vpiModPathHasIfnone 40     /* state-dependent module path has ifnone
                                     condition specified */
#define vpiTchkType          41     /* timing check subtypes: */
#define vpiSetup              1      /* $setup */
#define vpiHold               2      /* $hold */
#define vpiPeriod             3      /* $period */
#define vpiWidth              4      /* $width */
#define vpiSkew               5      /* $skew */
#define vpiRecovery           6      /* $recovery */
#define vpiNoChange           7      /* $nochange */
#define vpiSetupHold          8      /* $setuphold */

/***** expression properties *****/
#define vpiOpType             42     /* operation subtypes: */
#define vpiMinusOp            1      /* unary minus */
#define vpiPlusOp             2      /* unary plus */
#define vpiNotOp              3      /* unary not */
#define vpiBitNegOp           4      /* bitwise negation */
#define vpiUnaryAndOp         5      /* bitwise reduction and */
#define vpiUnaryNandOp        6      /* bitwise reduction nand */
#define vpiUnaryOrOp          7      /* bitwise reduction or */
#define vpiUnaryNorOp         8      /* bitwise reduction nor */
#define vpiUnaryXorOp         9      /* bitwise reduction xor */
#define vpiUnaryXNorOp        10     /* bitwise reduction xnor */
#define vpiSubOp              11     /* binary subtraction */
#define vpiDivOp              12     /* binary division */
#define vpiModOp              13     /* binary modulus */
#define vpiEqOp               14     /* binary equality */
#define vpiNeqOp              15     /* binary inequality */
#define vpiCaseEqOp           16     /* case (x and z) equality */
#define vpiCaseNeqOp          17     /* case inequality */
#define vpiGtOp               18     /* binary greater than */
#define vpiGeOp               19     /* binary greater than or equal to */
#define vpiLtOp               20     /* binary less than */
#define vpiLeOp               21     /* binary less than or equal to */
#define vpiLShiftOp           22     /* binary left shift */
#define vpiRShiftOp           23     /* binary right shift */
#define vpiAddOp              24     /* binary addition */
#define vpiMultOp             25     /* binary multiplication */
#define vpiLogAndOp           26     /* binary logical and */
#define vpiLogOrOp            27     /* binary logical or */
#define vpiBitAndOp           28     /* binary bitwise and */
#define vpiBitOrOp            29     /* binary bitwise or */
#define vpiBitXorOp           30     /* binary bitwise xor */
#define vpiBitXNorOp          31     /* binary bitwise xnor */
#define vpiConditionOp        32     /* ternary conditional */
#define vpiConcatOp           33     /* n-ary concatenation */
#define vpiMultiConcatOp      34     /* repeated concatenation */
#define vpiEventOrOp          35     /* event or */
#define vpiNullOp             36     /* null operation */
#define vpiListOp             37     /* list of expressions */
#define vpiMinTypMaxOp        38     /* min:typ:max: delay expression */
#define vpiPosedgeOp          39     /* posedge */
#define vpiNegedgeOp          40     /* negedge */

```

```

#define vpiConstType      43  /* constant subtypes: */
#define vpiDecConst       1   /* decimal integer */
#define vpiRealConst      2   /* real */
#define vpiBinaryConst    3   /* binary integer */
#define vpiOctConst       4   /* octal integer */
#define vpiHexConst       5   /* hexadecimal integer */
#define vpiStringConst    6   /* string literal */

#define vpiBlocking       44  /* blocking assignment (Boolean) */
#define vpiCaseType       45  /* case statement subtypes: */
#define vpiCaseExact      1   /* exact match */
#define vpiCaseX          2   /* ignore X's */
#define vpiCaseZ          3   /* ignore Z's */
#define vpiNetDeclAssign  46  /* assign part of declaration (Boolean) */

/***** system taskfunc properties *****/
#define vpiSysFuncType    47  /* system function type */
#define vpiSysFuncInt     1   /* returns integer */
#define vpiSysFuncReal    2   /* returns real */
#define vpiSysFuncTime    3   /* returns time */
#define vpiSysFuncSized   4   /* returns sized */
#define vpiUserDefn       48  /* user defined system tf (Boolean) */

#define vpiScheduled      49  /* is object vpiSchedEvent still scheduled */

/***** I/O related definitions *****/
#define VPI_MCD_STDOUT    0x00000001
#define VPI_MCD_STDERR    0x00000002
#define VPI_MCD_LOG       0x00000004

/***** STRUCTURE DEFINITIONS *****/
/***** time structure *****/
typedef struct t_vpi_time
{
    int type; /* [vpiScaledRealTime,vpiSimTime,vpiSuppressTime]*/
    unsigned int high, low; /* for vpiSimTime */
    double real; /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;

/* time types */
#define vpiScaledRealTime 1
#define vpiSimTime 2
#define vpiSuppressTime 3

/***** delay structures *****/
typedef struct t_vpi_delay
{
    struct t_vpi_time *da; /* ptr to user allocated array of delay values */
    int no_of_delays; /* number of delays */
    int time_type; /* [vpiScaledRealTime,vpiSimTime,vpiSuppressTime]*/
    int mtm_flag; /* true for mtm values */
    int append_flag; /* true for append */
    int pulser_flag; /* true for pulser values */
} s_vpi_delay, *p_vpi_delay;

```

```

/***** value structures *****/
/* vector value */
typedef struct t_vpi_vecval
{
    /* following fields are repeated enough times to contain vector */
    int aval, bval;          /* bit encoding: ab: 00=0, 10=1, 11=X, 01=Z */
} s_vpi_vecval, *p_vpi_vecval;

/* strength (scalar) value */
typedef struct t_vpi_strengthval
{
    int logic;               /* vpi[0,1,X,Z] */
    int s0, s1;             /* refer to strength coding below */
} s_vpi_strengthval, *p_vpi_strengthval;

/* strength values */
#define vpiSupplyDrive      0x80
#define vpiStrongDrive      0x40
#define vpiPullDrive        0x20
#define vpiWeakDrive        0x08
#define vpiLargeCharge      0x10
#define vpiMediumCharge     0x04
#define vpiSmallCharge      0x02
#define vpiHiZ              0x01

/* generic value */
typedef struct t_vpi_value
{
    int format; /* vpi[[Bin,Oct,Dec,Hex]Str,Scalar,Int,Real,String,Vector,
                                                Strength,Suppress,Time,ObjType]Val */

    union
    {
        {
            char *str;                /* string value */
            int scalar;               /* vpi[0,1,X,Z] */
            int integer;              /* integer value */
            double real;              /* real value */
            struct t_vpi_time *time;  /* time value */
            struct t_vpi_vecval *vector; /* vector value */
            struct t_vpi_strengthval *strength; /* strength value */
            char *misc;               /* ...other */
        } value;
    }
} s_vpi_value, *p_vpi_value;

/* value formats */
#define vpiBinStrVal        1
#define vpiOctStrVal        2
#define vpiDecStrVal        3
#define vpiHexStrVal        4
#define vpiScalarVal        5
#define vpiIntVal           6
#define vpiRealVal          7
#define vpiStringVal        8
#define vpiVectorVal        9

```

```

#define vpiStrengthVal      10
#define vpiTimeVal         11
#define vpiObjTypeVal      12
#define vpiSuppressVal     13

/* delay modes */
#define vpiNoDelay          1
#define vpiInertialDelay    2
#define vpiTransportDelay   3
#define vpiPureTransportDelay 4

/* force and release flags */
#define vpiForceFlag        5
#define vpiReleaseFlag      6

/* scheduled event cancel flag */
#define vpiCancelEvent      7

/* bit mask for the flags argument to vpi_put_value() */
#define vpiReturnEvent      0x1000

/* scalar values */
#define vpi0                 0
#define vpi1                 1
#define vpiZ                 2
#define vpiX                 3
#define vpiH                 4
#define vpiL                 5
#define vpiDontCare          6
/*
#define vpiNoChange          7   Defined under vpiTchkType, but can be used
here.
*/

/***** system taskfunc structure *****/
typedef struct t_vpi_systf_data
{
    int type;                /* vpiSysTask, vpiSysFunc */
    int subtype;             /* vpiSys[Task, Func[Int,Real,Time,Sized]] */
    char *tfname;            /* first character has to be '$' */
    int (*calltf)();
    int (*compiletf)();
    int (*sizetf)();         /* for vpiSysFuncSized callbacks only */
    char *user_data;
} s_vpi_systf_data, *p_vpi_systf_data;

#define vpiSysTask          1
#define vpiSysFunc          2
/* the subtypes are defined under the vpiSysFuncType property */

/***** Verilog execution information structure *****/
typedef struct t_vpi_vlog_info
{
    int argc;
    char **argv;
    char *product;

```

```

    char *version;
} s_vpi_vlog_info, *p_vpi_vlog_info;
/***** PLI error information structure *****/
typedef struct t_vpi_error_info
{
    int state;                /* vpi[Compile,PLI,Run] */
    int level;                /* vpi[Notice,Warning,Error,System,Internal]*/
    char *message;
    char *product;
    char *code;
    char *file;
    int line;
} s_vpi_error_info, *p_vpi_error_info;

/* error types */
#define vpiCompile            1
#define vpiPLI                2
#define vpiRun                3

#define vpiNotice             1
#define vpiWarning            2
#define vpiError              3
#define vpiSystem             4
#define vpiInternal           5

/***** callback structures *****/
/* normal callback structure */
typedef struct t_cb_data
{
    int reason;                /* callback reason */
    int (*cb_rtn)();           /* call routine */
    vpiHandle obj;             /* trigger object */
    p_vpi_time *time;          /* callback time */
    p_vpi_value *value;        /* trigger object value */
    int index;                 /* index of the memory word or var select
                                that changed value */

    char *user_data;
} s_cb_data, *p_cb_data;

/* Callback Reasons */
/* Simulation-related */
#define cbValueChange         1
#define cbStmt                2
#define cbForce               3
#define cbRelease             4

/* Time-related */
#define cbAtStartOfSimTime    5
#define cbReadWriteSynch      6
#define cbReadOnlySynch       7
#define cbNextSimTime         8
#define cbAfterDelay          9

/* Action-related */
#define cbEndOfCompile        10

```



```

#define cbStartOfSimulation      11
#define cbEndOfSimulation        12
#define cbError                  13
#define cbTchkViolation          14
#define cbStartOfSave            15
#define cbEndOfSave              16
#define cbStartOfRestart         17
#define cbEndOfRestart           18
#define cbStartOfReset           19
#define cbEndOfReset             20
#define cbEnterInteractive        21
#define cbExitInteractive         22
#define cbInteractiveScopeChange 23
#define cbUnresolvedSystf        24

#if defined(__STDC__) || defined(__cplusplus)

#ifndef PROTO_PARAMS
#define PROTO_PARAMS(params) params
#define DEFINED_PROTO_PARAMS
#endif
#ifndef EXTERN
#define EXTERN
#define DEFINED_EXTERN
#endif

#else

#ifndef PROTO_PARAMS
#define PROTO_PARAMS(params) (/* nothing */)
#define DEFINED_PROTO_PARAMS
#endif
#ifndef EXTERN
#define EXTERN extern
#define DEFINED_EXTERN
#endif

#endif /* __STDC__ */

/***** FUNCTION DECLARATIONS *****/

/* callback related */
EXTERN vpiHandle    vpi_register_cb      PROTO_PARAMS((p_cb_data cb_data_p));
EXTERN int          vpi_remove_cb        PROTO_PARAMS((vpiHandle cb_obj));
EXTERN void         vpi_get_cb_info      PROTO_PARAMS((vpiHandle object,
    p_cb_data cb_data_p));
EXTERN void         vpi_register_systf   PROTO_PARAMS((p_vpi_systf_data
    systf_data_p));
EXTERN void         vpi_get_systf_info    PROTO_PARAMS((vpiHandle object,
    p_vpi_systf_data systf_data_p));

/* for obtaining handles */
EXTERN vpiHandle     vpi_handle_by_name   PROTO_PARAMS((char *name,
    vpiHandle scope));
EXTERN vpiHandle     vpi_handle_by_index  PROTO_PARAMS((vpiHandle object,
    int indx));

```

```

/* for traversing relationships */
EXTERN vpiHandle    vpi_handle
EXTERN vpiHandle    vpi_iterate
EXTERN vpiHandle    vpi_scan

/* for accesssing properties */
EXTERN int          vpi_get
EXTERN char *       vpi_get_str

/* delay processing */
EXTERN void         vpi_get_delays
EXTERN void         vpi_put_delays

/* value processing */
EXTERN void         vpi_get_value
EXTERN vpiHandle    vpi_put_value

/* time processing */
EXTERN void         vpi_get_time

/* I/O routines */
EXTERN unsigned int vpi_mcd_open
EXTERN unsigned int vpi_mcd_close
EXTERN char *       vpi_mcd_name
#ifdef VPI_IO_C
EXTERN int          vpi_mcd_printf
EXTERN int          vpi_printf
#else
EXTERN int          vpi_mcd_printf
EXTERN int          vpi_printf
#endif

/* utility routines */
EXTERN int          vpi_compare_objects
EXTERN int          vpi_chk_error
EXTERN int          vpi_free_object
EXTERN int          vpi_get_vlog_info

#ifdef DEFINED_PROTO_PARAMS
PROTO_PARAMS((int type,
              vpiHandle referenceHandle));
PROTO_PARAMS((int type,
              vpiHandle referenceHandle));
PROTO_PARAMS((vpiHandle iterator));

PROTO_PARAMS((int property,
              vpiHandle object));
PROTO_PARAMS((int property,
              vpiHandle object));

PROTO_PARAMS((vpiHandle object,
              p_vpi_delay delay_p));
PROTO_PARAMS((vpiHandle object,
              p_vpi_delay delay_p));

PROTO_PARAMS((vpiHandle expr,
              p_vpi_value value_p));
PROTO_PARAMS((vpiHandle object,
              p_vpi_value value_p,
              p_vpi_time time_p, int flags));

PROTO_PARAMS((vpiHandle object,
              p_vpi_time time_p));

PROTO_PARAMS((char *fileName));
PROTO_PARAMS((unsigned int mcd));
PROTO_PARAMS((unsigned int cd));

PROTO_PARAMS((unsigned int mcd,
              char *format,...));
PROTO_PARAMS((char *format,...));

PROTO_PARAMS(( ));
PROTO_PARAMS(( ));

PROTO_PARAMS((vpiHandle object1,
              vpiHandle object2));
PROTO_PARAMS((p_vpi_error_info
              error_info_p));
PROTO_PARAMS((vpiHandle object));
PROTO_PARAMS((p_vpi_vlog_info
              vlog_info_p));
#endif
#endif

```

```
#undef  PROTO_PARAMS
#endif
#ifdef  DEFINED_EXTERN
#undef  DEFINED_EXTERN
#undef  EXTERN
#endif

/***** GLOBAL VARIABLES *****/
extern void (*vlog_startup_routines[])(); /* array of function pointers, */
                                           /* last pointer should be null */
#endif /* VPI_USER_H */
```

Annex F

(informative)

System tasks and functions

The system tasks and functions described in this annex are for informative purposes only and are not part of the IEEE standard Verilog HDL.

This annex describes system tasks and functions as companions to the system tasks and functions described in Section 14. The system tasks and functions described in this annex may not be available in all implementations of the Verilog HDL. The following system tasks and functions are described in this annex:

\$countdrivers	[F.1]	\$reset_value	[F.7]
\$getpattern	[F.2]	\$restart	[F.8]
\$incsave	[F.8]	\$save	[F.8]
\$input	[F.3]	\$scale	[F.9]
\$key	[F.4]	\$scope	[F.10]
\$list	[F.5]	\$showscopes	[F.11]
\$log	[F.6]	\$showvars	[F.12]
\$nokey	[F.4]	\$sreadmemb	[F.13]
\$nolog	[F.6]	\$sreadmemh	[F.13]
\$reset	[F.7]		
\$reset_count	[F.7]		

The word *tool* in this annex refers to an implementation of Verilog HDL, typically a logic simulator.

F.1 \$countdrivers

Syntax:

```
$countdrivers (net, [ net_is_forced, number_of_01x_drivers, number_of_0_drivers,
                 number_of_1_drivers, number_of_x_drivers ] );
```

The **\$countdrivers** system function is provided to count the number of drivers on a specified net so that bus contention can be identified.

This system function returns a 0 if there is no more than one driver on the net and returns a 1 otherwise (indicating contention). The specified net shall be a scalar or a bit-select of a vector net. The number of parameters to the system function may vary according to how much information is desired.

If additional parameters are supplied to the **\$countdrivers** function, each parameter returns the information described in Table F-1.

Table F-1 — Parameter return value for \$countdriver function

Parameter	Return value
net_is_forced	1 if net is forced 0 otherwise
number_of_01x_drivers	An integer representing the number of drivers on the net that are in 0, 1, or x state. This represents the total number of drivers that are not forced
number_of_0_drivers	An integer representing the number of drivers on the net that are in 0 state
number_of_1_drivers	An integer representing the number of drivers on the net that are in 1 state
number_of_x_drivers	An integer representing the number of drivers on the net that are in x state

F.2 \$getpattern

Syntax:

```
$getpattern ( mem_element );
```

The system function **\$getpattern** provides for fast processing of stimulus patterns that have to be propagated to a large number of scalar inputs. The function reads stimulus patterns that have been loaded into a memory using the **\$readmemb** or **\$readmemh** system tasks.

Use of this function is limited, however: it may only be used in a continuous assignment statement where the lefthand side is a concatenation of scalar nets, and the parameter to the system function is a memory element reference.

Example:

The following example shows how stimuli stored in a file can be read into a memory using **\$readmemb** and applied to the circuit one pattern at a time using **\$getpattern**.

The memory `in_mem` is initialized with the stimulus patterns by the **\$readmemb** task. The integer variable `index` selects which pattern is being applied to the circuit. The `for` loop increments the integer variable `index` periodically to sequence the patterns.

```

module top;
parameter in_width=10,
           patterns=200,
           step=20;
reg [1:in_width] in_mem[1:patterns];
integer index;

// declare scalar inputs
wire i1,i2,i3,i4,i5,i6,i7,i8,i9,i10;

// assign patterns to circuit scalar inputs (a new pattern
// is applied to the circuit each time index changes value)
assign {i1,i2,i3,i4,i5,i6,i7,i8,i9,i10} = $getpattern(in_mem[index]);
initial begin
    // read stimulus patterns into memory
    $readmemb("patt.mem", in_mem);

    // step through patterns (note that each assignment
    // to index will drive a new pattern onto the circuit
    // inputs from the $getpattern system task specified above
    for (index = 1; index <= patterns; index = index + 1)
        #step;
end

    // instantiate the circuit module - e.g.
    mod1 cct (o1,o2,o3,o4,o5, i1,i2,i3,i4,i5,i6,i7,i8,i9,i10);

endmodule

```

F.3 \$input

Syntax:

```
$input ("filename");
```

The **\$input** system task allows command input text to come from a named file instead of from the terminal. At the end of the command file, the input is switched back to the terminal.

F.4 \$key and \$nokey

Syntax:

```
$key [ ("filename") ];
$nokey ;
```

A key file is created whenever interactive mode is entered for the first time during simulation. The key file contains all of the text that has been typed in from the standard input. The file also contains information about asynchronous interrupts.

The **\$nokey** and **\$key** system tasks are used to disable and re-enable output to the key file. An optional file name parameter for **\$key** causes the old key file to be closed, a new file to be created, and output to be directed to the new file.

F.5 \$list

Syntax:

```
$list [ ( hierarchical_name ) ] ;
```

When invoked without a parameter, **\$list** produces a listing of the module, task, function, or named block that is defined as the current scope setting. If an optional parameter is supplied, it shall refer to a specific module, task, function or named block, in which case the specified object is listed.

F.6 \$log and \$nolog

Syntax:

```
$log [ ("filename") ] ;  
$nolog ;
```

A log file contains a copy of all the text that is printed to the standard output. The log file may also contain, at the beginning of the file, the host command that was used to run the tool.

The **\$nolog** and **\$log** system tasks are used to disable and re-enable output to the log file. The **\$nolog** task disables output to the log file, while the **\$log** task re-enables the output. An optional file name parameter for **\$log** causes the old file to be closed, a new log file to be created, and output to be directed to the new log file.

F.7 \$reset, \$reset_count, and \$reset_value

Syntax:

```
$reset [ ( stop_value [ , reset_value , [ diagnostics_value ] ] ) ] ;  
$reset_count ;  
$reset_value ;
```

The **\$reset** system task enables a tool to be reset to its “Time 0” state so that processing (e.g., simulation) can begin again.

The **\$reset_count** system function keeps track of the number of times the tool is reset. The **\$reset_value** system function returns the value specified by the `reset_value` parameter argument to the **\$reset** system task. The **\$reset_value** system function is used to communicate information from before a reset of a tool to the time 0 state to after the reset.

The following are some of the simulation methods that can be employed with this system task and these system functions:

- Determine the **force** statements a design needs to operate correctly, reset the simulation time to 0, enter these **force** statements, and start to simulate again
- Reset the simulation time to 0 and apply new stimuli
- Determine that debug system tasks, such as **\$monitor** and **\$strobe**, are keeping track of the correct nets or registers, reset the simulation time to 0, and begin simulation again

The **\$reset** system task tells a tool to return the processing of the design to its logical state at time 0. When a tool executes the **\$reset** system task, it takes the following actions to stop the process:

- 1) Disables all concurrent activity, initiated in either **initial** and **always** procedural blocks in the source description or through interactive mode (disables, for example, all **force** and **assign** statements, the current **\$monitor** system task, and any other active tasks)
- 2) Cancels all scheduled simulation events

After a simulation tool executes the **\$reset** system task, the simulation is in the following state:

- The simulation time is 0.
- All registers and nets contain their initial values.
- The tool begins to execute the first procedural statements in all **initial** and **always** blocks.

The **stop_value** argument indicates whether interactive mode or processing is entered immediately after resetting of the tool. A value of 0 or no argument causes interactive mode to be entered after resetting the tool. A nonzero value passed to **\$reset** causes the tool to begin processing immediately.

The **reset_value** argument is an integer that specifies whose value is returned by the **\$reset_value** system function after the tool is reset. All declared integers return to their initial value after reset, but entering an integer as this argument allows access to what its value was before the reset with the **\$reset_value** system function. This argument provides a means of communicating information from before the reset of a tool to after the reset of the tool.

The **diagnostic_value** specifies the kind of diagnostic messages a tool displays before it resets the time to 0. Increasing integer values results in increased information. A value of zero results in no diagnostic message.

F.8 \$save, \$restart, and \$incsave

Three system tasks **\$save**, **\$restart**, and **\$incsave** work in conjunction with one another to save the complete state of simulation into a permanent file such that the simulation state can be reloaded at a later time and processing can continue where it left off.

Syntax:

```
$save("file_name");
$restart("file_name");
$incsave("incremental_file_name");
```

All three system tasks take a file name as a parameter. The file name has to be supplied as a string enclosed in quotation marks.

The **\$save** system task saves the complete state into the host operating system file specified as a parameter.

The **\$incsave** system task saves only what has changed since the last invocation of **\$save**. It is not possible to do an incremental save on any file other than the one produced by the last **\$save**.

The **\$restart** system task restores a previously saved state from a specified file.

Restarting from an incremental save is similar to restarting from a full save, except that the name of the incremental save file is specified in the restart command. The full save file that the incremental save file was based upon shall still be present, as it is required for a successful restart. If the full save file has been changed in any way since the incremental save was performed, errors will result.

The incremental restart is useful for going back in time. If a full save is performed near the beginning of processing, and an incremental save is done at regular intervals, then going back in time is as simple as restarting from the appropriate file.

Example:

```
module checkpoint;

initial
    #500 $save("save.dat"); // full save

always begin // incremental save every 10000 units,
              // files are recycled every 40000 units
    #100000 $incsave("inc1.dat");
    #100000 $incsave("inc2.dat");
    #100000 $incsave("inc3.dat");
    #100000 $incsave("inc4.dat");
end
endmodule
```

F.9 \$scale

Syntax:

\$scale (hierarchical_name);

The **\$scale** function takes a time value from a module with one time unit to be used in a module with a different time unit. The time value is converted from the time unit of one module to the time unit of the module that invokes **\$scale**.

F.10 \$scope

Syntax:

\$scope (hierarchical_name);

The **\$scope** system task allows a particular level of hierarchy to be specified as the scope for identifying objects. This task accepts a single parameter argument that shall be the complete hierarchical name of a module, task, function, or named block. The initial setting of the interactive scope is the first top-level module.

F.11 \$showscopes

Syntax:

\$showscopes [(n)];

The **\$showscopes** system task produces a complete list of modules, tasks, functions, and named blocks that are defined *at the current scope level*. An optional integer parameter can be given to **\$showscopes**. A nonzero parameter value causes all the modules, tasks, functions, and named blocks in or below the current hierarchical scope to be listed. No parameter or a zero value results in only objects at the current scope level to be listed.

F.12 \$showvars

Syntax:

```
$showvars [ ( list_of_variables ) ] ;
```

The **\$showvars** system task produces status information for register and net variables, both scalar and vector. When invoked without parameters, **\$showvars** displays the status of all variables in the current scope. When invoked with a list of variables, **\$showvars** shows only the status of the specified variables. If the list of variables includes a bit-select or part-select of a register or net, then the status information for all the bits of that register or net are displayed.

F.13 \$sreadmemb and \$sreadmemh

Syntax:

```
$sreadmemb ( mem_name , start_address , finish_address , string { , string } ) ;  
$sreadmemh ( mem_name , start_address , finish_address , string { , string } ) ;
```

The system tasks **\$sreadmemb** and **\$sreadmemh** load data into memory **mem_name** from a character string.

The **\$sreadmemh** and **\$sreadmemb** system tasks take memory data values and addresses as string arguments. The start and finish addresses indicate the bounds for where the data from strings will be stored in the memory. These strings take the same format as the strings that appear in the input files passed as arguments to **\$readmemb** and **\$readmemh**.

Annex G

(informative)

Compiler directives

The compiler directives described in this annex are for informative purposes only and are not part of the IEEE standard Verilog HDL.

This annex describes additional compiler directives as companions to the compiler directives described in Section 16. The compiler directives described in this annex may not be available in all implementations of the Verilog HDL. The following compiler directives are described in this annex:

<code>`default_decay_time</code>	[G.1]	<code>`delay_mode_path</code>	[G.4]
<code>`default_trireg_strength</code>	[G.2]	<code>`delay_mode_unit</code>	[G.5]
<code>`delay_mode_distributed</code>	[G.3]	<code>`delay_mode_zero</code>	[G.6]

The word *tool* in this annex refers to an implementation of Verilog HDL, typically a logic simulator.

G.1 ``default_decay_time`

The ``default_decay_time` compiler directive specifies the decay time for the trireg nets that do not have any decay time specified in the declaration. This compiler directive applies to all of the trireg nets in all the modules that follow it in the source description. An argument specifying the charge decay time shall be used with this compiler directive.

Syntax:

``default_decay_time`*integer_constant* | *real_constant* | **infinite**

Examples:

Example 1—The following example shows how the default decay time for all trireg nets can be set to 100 time units:

```
`default_decay_time100
```

Example 2—The following example shows how to avoid charge decay on trireg nets:

```
`default_decay_timeinfinite
```

The keyword **infinite** specifies no charge decay for all the trireg nets that do not have decay time specification.

G.2 ``default_trireg_strength`

The ``default_trireg_strength` compiler directive specifies the charge strength of **trireg** nets.

Syntax:

`default_trireg_strength*integer_constant*

The integer constant shall be between 0 and 250. It indicates the relative strength of the capacitance on the trireg net.

G.3 **`delay_mode_distributed**

The **`delay_mode_distributed** compiler directive specifies the distributed delay mode for all modules that follow this directive in the source description.

Syntax:

`delay_mode_distributed

This compiler directive shall be used before the declaration of the module whose delay mode is being controlled.

G.4 **`delay_mode_path**

The **`delay_mode_path** compiler directive specifies the path delay mode for all modules that follow this directive in the source description.

Syntax:

`delay_mode_path

This compiler directive shall be used before the declaration of the module whose delay mode is being controlled.

G.5 **`delay_mode_unit**

The **`delay_mode_unit** compiler directive specifies the unit delay mode for all modules that follow this directive in the source description.

Syntax:

`delay_mode_unit

This compiler directive shall be used before the declaration of the module whose delay mode is being controlled.

G.6 **`delay_mode_zero**

The **`delay_mode_zero** compiler directive specifies the zero-delay mode for all modules that follow this directive in the source description.

Syntax:

`delay_mode_zero

This compiler directive shall be used before the declaration of the module whose delay mode is being controlled.

Annex H

(informative)

Bibliography

[B1] IEEE Std 754-1985 (Reaff 1990), IEEE Standard for Binary Floating-Point Arithmetic (ANSI).¹

¹IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.