

INTERNATIONAL STANDARD

**IEC
61691-4**

First edition
2004-10

IEEE 1364™

Behavioural languages –

Part 4:

Verilog® hardware description language



IEEE

Reference number
IEC 61691-4(E):2004
IEEE Std. 1364(E):2001

Publication numbering

As from 1 January 1997 all IEC publications are issued with a designation in the 60000 series. For example, IEC 34-1 is now referred to as IEC 60034-1.

Consolidated editions

The IEC is now publishing consolidated versions of its publications. For example, edition numbers 1.0, 1.1 and 1.2 refer, respectively, to the base publication, the base publication incorporating amendment 1 and the base publication incorporating amendments 1 and 2.

Further information on IEC publications

The technical content of IEC publications is kept under constant review by the IEC, thus ensuring that the content reflects current technology. Information relating to this publication, including its validity, is available in the IEC Catalogue of publications (see below) in addition to new editions, amendments and corrigenda. Information on the subjects under consideration and work in progress undertaken by the technical committee which has prepared this publication, as well as the list of publications issued, is also available from the following:

- **IEC Web Site** (www.iec.ch)

- **Catalogue of IEC publications**

The on-line catalogue on the IEC web site (www.iec.ch/searchpub) enables you to search by a variety of criteria including text searches, technical committees and date of publication. On-line information is also available on recently issued publications, withdrawn and replaced publications, as well as corrigenda.

- **IEC Just Published**

This summary of recently issued publications (www.iec.ch/online_news/justpub) is also available by email. Please contact the Customer Service Centre (see below) for further information.

- **Customer Service Centre**

If you have any questions regarding this publication or need further assistance, please contact the Customer Service Centre:

Email: custserv@iec.ch
Tel: +41 22 919 02 11
Fax: +41 22 919 03 00

INTERNATIONAL STANDARD

IEC
61691-4

First edition
2004-10

IEEE 1364™

Behavioural languages –

Part 4: Verilog® hardware description language

Copyright © IEEE 2004 — All rights reserved

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

No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

International Electrotechnical Commission, 3, rue de Varembé, PO Box 131, CH-1211 Geneva 20, Switzerland
Telephone: +41 22 919 02 11 Telefax: +41 22 919 03 00 E-mail: inmail@iec.ch Web: www.iec.ch

The Institute of Electrical and Electronics Engineers, Inc, 3 Park Avenue, New York, NY 10016-5997, USA
Telephone: +1 732 562 3800 Telefax: +1 732 562 1571 E-mail: stds-info@ieee.org Web: www.standards.ieee.org



Commission Electrotechnique Internationale
International Electrotechnical Commission
Международная Электротехническая Комиссия



CONTENTS

FOREWORD	19
IEEE Introduction.....	23
1. Overview.....	25
1.1 Objectives of this standard.....	25
1.2 Conventions used in this standard.....	25
1.3 Syntactic description.....	26
1.4 Contents of this standard.....	26
1.5 Header file listings	28
1.6 Examples.....	29
1.7 Prerequisites.....	29
2. Lexical conventions	30
2.1 Lexical tokens	30
2.2 White space.....	30
2.3 Comments	30
2.4 Operators.....	30
2.5 Numbers.....	30
2.5.1 Integer constants	31
2.5.2 Real constants	34
2.5.3 Conversion	34
2.6 Strings	34
2.6.1 String variable declaration.....	35
2.6.2 String manipulation.....	35
2.6.3 Special characters in strings.....	35
2.7 Identifiers, keywords, and system names	36
2.7.1 Escaped identifiers	36
2.7.2 Generated identifiers.....	37
2.7.3 Keywords	37
2.7.4 System tasks and functions.....	37
2.7.5 Compiler directives.....	38
2.8 Attributes.....	38
2.8.1 Examples.....	39
2.8.2 Syntax.....	40
3. Data types.....	44
3.1 Value set.....	44
3.2 Nets and variables	44
3.2.1 Net declarations	44
3.2.2 Variable declarations	46
3.3 Vectors.....	47
3.3.1 Specifying vectors.....	47
3.3.2 Vector net accessibility	48
3.4 Strengths	48
3.4.1 Charge strength	48
3.4.2 Drive strength.....	48
3.5 Implicit declarations.....	49
3.6 Net initialization.....	49
3.7 Net types	49
3.7.1 Wire and tri nets.....	49
3.7.2 Wired nets	50

3.7.3	Trireg net.....	50
3.7.4	Tri0 and tri1 nets.....	54
3.7.5	Supply nets.....	55
3.8	regs.....	55
3.9	Integers, reals, times, and realsimes.....	55
3.9.1	Operators and real numbers.....	56
3.9.2	Conversion.....	56
3.10	Arrays.....	57
3.10.1	Net arrays.....	57
3.10.2	reg and variable arrays.....	57
3.10.3	Memories.....	57
3.11	Parameters.....	58
3.11.1	Module parameters.....	59
3.11.2	Local parameters—localparam.....	60
3.11.3	Specify parameters.....	61
3.12	Name spaces.....	62
4.	Expressions.....	64
4.1	Operators.....	64
4.1.1	Operators with real operands.....	65
4.1.2	Binary operator precedence.....	66
4.1.3	Using integer numbers in expressions.....	67
4.1.4	Expression evaluation order.....	67
4.1.5	Arithmetic operators.....	68
4.1.6	Arithmetic expressions with regs and integers.....	69
4.1.7	Relational operators.....	70
4.1.8	Equality operators.....	70
4.1.9	Logical operators.....	71
4.1.10	Bit-wise operators.....	71
4.1.11	Reduction operators.....	72
4.1.12	Shift operators.....	73
4.1.13	Conditional operator.....	74
4.1.14	Concatenations.....	75
4.1.15	Event or.....	76
4.2	Operands.....	76
4.2.1	Vector bit-select and part-select addressing.....	76
4.2.2	Array and memory addressing.....	78
4.2.3	Strings.....	79
4.3	Minimum, typical, and maximum delay expressions.....	81
4.4	Expression bit lengths.....	83
4.4.1	Rules for expression bit lengths.....	83
4.4.2	An example of an expression bit-length problem.....	84
4.4.3	Example of self-determined expressions.....	85
4.5	Signed expressions.....	86
4.5.1	Rules for expression types.....	86
4.5.2	Steps for evaluating an expression.....	86
4.5.3	Steps for evaluating an assignment.....	87
4.5.4	Handling X and Z in signed expressions.....	87
5.	Scheduling semantics.....	88
5.1	Execution of a model.....	88
5.2	Event simulation.....	88

5.3	The stratified event queue	88
5.4	The Verilog simulation reference model	89
5.4.1	Determinism	90
5.4.2	Nondeterminism	90
5.5	Race conditions	90
5.6	Scheduling implication of assignments	90
5.6.1	Continuous assignment	91
5.6.2	Procedural continuous assignment	91
5.6.3	Blocking assignment	91
5.6.4	Nonblocking assignment	91
5.6.5	Switch (transistor) processing	91
5.6.6	Port connections	92
5.6.7	Functions and tasks	92
6.	Assignments	93
6.1	Continuous assignments	93
6.1.1	The net declaration assignment	94
6.1.2	The continuous assignment statement	94
6.1.3	Delays	96
6.1.4	Strength	96
6.2	Procedural assignments	97
6.2.1	Variable declaration assignment	97
6.2.2	Variable declaration syntax	98
7.	Gate and switch level modeling	99
7.1	Gate and switch declaration syntax	99
7.1.1	The gate type specification	101
7.1.2	The drive strength specification	101
7.1.3	The delay specification	102
7.1.4	The primitive instance identifier	102
7.1.5	The range specification	102
7.1.6	Primitive instance connection list	103
7.2	and, nand, nor, or, xor, and xnor gates	105
7.3	buf and not gates	106
7.4	bufif1, bufif0, notif1, and notif0 gates	107
7.5	MOS switches	109
7.6	Bidirectional pass switches	110
7.7	CMOS switches	110
7.8	pullup and pulldown sources	111
7.9	Logic strength modeling	112
7.10	Strengths and values of combined signals	113
7.10.1	Combined signals of unambiguous strength	113
7.10.2	Ambiguous strengths: sources and combinations	114
7.10.3	Ambiguous strength signals and unambiguous signals	119
7.10.4	Wired logic net types	123
7.11	Strength reduction by nonresistive devices	126
7.12	Strength reduction by resistive devices	126
7.13	Strengths of net types	126
7.13.1	tri0 and tri1 net strengths	126
7.13.2	trireg strength	126
7.13.3	supply0 and supply1 net strengths	126

7.14	Gate and net delays	127
7.14.1	min:typ:max delays.....	128
7.14.2	trireg net charge decay.....	129
8.	User-defined primitives (UDPs)	131
8.1	UDP definition.....	131
8.1.1	UDP header.....	133
8.1.2	UDP port declarations.....	133
8.1.3	Sequential UDP initial statement	133
8.1.4	UDP state table	133
8.1.5	Z values in UDP.....	134
8.1.6	Summary of symbols	134
8.2	Combinational UDPs	135
8.3	Level-sensitive sequential UDPs	136
8.4	Edge-sensitive sequential UDPs	136
8.5	Sequential UDP initialization	137
8.6	UDP instances.....	139
8.7	Mixing level-sensitive and edge-sensitive descriptions.....	140
8.8	Level-sensitive dominance.....	141
9.	Behavioral modeling.....	142
9.1	Behavioral model overview	142
9.2	Procedural assignments.....	143
9.2.1	Blocking procedural assignments	143
9.2.2	The nonblocking procedural assignment.....	145
9.3	Procedural continuous assignments.....	148
9.3.1	The assign and deassign procedural statements.....	149
9.3.2	The force and release procedural statements	150
9.4	Conditional statement.....	151
9.4.1	If-else-if construct.....	152
9.5	Case statement	154
9.5.1	Case statement with don't-cares	157
9.5.2	Constant expression in case statement.....	157
9.6	Looping statements.....	158
9.7	Procedural timing controls.....	160
9.7.1	Delay control.....	161
9.7.2	Event control.....	162
9.7.3	Named events.....	162
9.7.4	Event or operator.....	163
9.7.5	Implicit event_expression list	164
9.7.6	Level-sensitive event control	165
9.7.7	Intra-assignment timing controls	166
9.8	Block statements	170
9.8.1	Sequential blocks	170
9.8.2	Parallel blocks.....	171
9.8.3	Block names.....	172
9.8.4	Start and finish times	172
9.9	Structured procedures	173
9.9.1	Initial construct	174
9.9.2	Always construct.....	174

10.	Tasks and functions.....	176
10.1	Distinctions between tasks and functions	176
10.2	Tasks and task enabling	176
10.2.1	Task declarations	177
10.2.2	Task enabling and argument passing	178
10.2.3	Task memory usage and concurrent activation.....	180
10.3	Functions and function calling.....	181
10.3.1	Function declarations	182
10.3.2	Returning a value from a function	183
10.3.3	Calling a function.....	184
10.3.4	Function rules.....	184
10.3.5	Use of constant functions.....	185
11.	Disabling of named blocks and tasks.....	187
12.	Hierarchical structures	190
12.1	Modules.....	190
12.1.1	Top-level modules	192
12.1.2	Module instantiation	192
12.1.3	Generated instantiation	194
12.2	Overriding module parameter values.....	204
12.2.1	defparam statement	206
12.2.2	Module instance parameter value assignment	207
12.2.3	Parameter dependence	209
12.3	Ports	209
12.3.1	Port definition	209
12.3.2	List of ports.....	209
12.3.3	Port declarations.....	210
12.3.4	List of ports declarations.....	212
12.3.5	Connecting module instance ports by ordered list.....	212
12.3.6	Connecting module instance ports by name	213
12.3.7	Real numbers in port connections.....	214
12.3.8	Connecting dissimilar ports	215
12.3.9	Port connection rules.....	215
12.3.10	Net types resulting from dissimilar port connections	216
12.3.11	Connecting signed values via ports.....	217
12.4	Hierarchical names	217
12.5	Upwards name referencing	220
12.6	Scope rules	222
13.	Configuring the contents of a design	224
13.1	Introduction.....	224
13.1.1	Library notation	224
13.1.2	Basic configuration elements.....	225
13.2	Libraries	225
13.2.1	Specifying libraries - the library map file.....	225
13.2.2	Using multiple library mapping files	227
13.2.3	Mapping source files to libraries.....	227
13.3	Configurations.....	227
13.3.1	Basic configuration syntax.....	227
13.3.2	Hierarchical configurations.....	230

13.4	Using libraries and configs	231
13.4.1	Precompiling in a single-pass use-model.....	231
13.4.2	Elaboration-time compiling in a single-pass use-model	231
13.4.3	Precompiling using a separate compilation tool	231
13.4.4	Command line considerations.....	231
13.5	Configuration examples	232
13.5.1	Default configuration from library map file	232
13.5.2	Using the default clause.....	232
13.5.3	Using the cell clause	233
13.5.4	Using the instance clause	233
13.5.5	Using a hierarchical config	233
13.6	Displaying library binding information	234
13.7	Library mapping examples	234
13.7.1	Using the command line to control library searching.....	234
13.7.2	File path specification examples	234
13.7.3	Resolving multiple path specifications	235
14.	Specify blocks	236
14.1	Specify block declaration.....	236
14.2	Module path declarations	237
14.2.1	Module path restrictions	238
14.2.2	Simple module paths.....	238
14.2.3	Edge-sensitive paths.....	239
14.2.4	State-dependent paths	240
14.2.5	Full connection and parallel connection paths.....	244
14.2.6	Declaring multiple module paths in a single statement	245
14.2.7	Module path polarity	246
14.3	Assigning delays to module paths.....	247
14.3.1	Specifying transition delays on module paths	248
14.3.2	Specifying x transition delays.....	249
14.3.3	Delay selection.....	250
14.4	Mixing module path delays and distributed delays.....	251
14.5	Driving wired logic	252
14.6	Detailed control of pulse filtering behavior	253
14.6.1	Specify block control of pulse limit values.....	254
14.6.2	Global control of pulse limit values.....	255
14.6.3	SDF annotation of pulse limit values	255
14.6.4	Detailed pulse control capabilities	256
15.	Timing checks	262
15.1	Overview	262
15.2	Timing checks using a stability window.....	265
15.2.1	\$setup	266
15.2.2	\$hold	266
15.2.3	\$setuphold	267
15.2.4	\$removal	269
15.2.5	\$recovery.....	270
15.2.6	\$crem	271
15.3	Timing checks for clock and control signals	273
15.3.1	\$skew	274
15.3.2	\$timeskew	275
15.3.3	\$fullskew	277

15.3.4	\$width	279
15.3.5	\$period	280
15.3.6	\$nochange	281
15.4	Edge-control specifiers	283
15.5	Notifiers: user-defined responses to timing violations	284
15.5.1	Requirements for accurate simulation	286
15.5.2	Conditions in negative timing checks	288
15.5.3	Notifiers in negative timing checks	290
15.5.4	Option behavior	290
15.6	Enabling timing checks with conditioned events.....	290
15.7	Vector signals in timing checks	291
15.8	Negative timing checks.....	292
16.	Backannotation using the Standard Delay Format (SDF).....	294
16.1	The SDF annotator.....	294
16.2	Mapping of SDF constructs to Verilog.....	294
16.2.1	Mapping of SDF delay constructs to Verilog declarations.....	294
16.2.2	Mapping of SDF timing check constructs to Verilog.....	296
16.2.3	SDF annotation of specparams	297
16.2.4	SDF annotation of interconnect delays	298
16.3	Multiple annotations	299
16.4	Multiple SDF files.....	300
16.5	Pulse limit annotation	300
16.6	SDF to Verilog delay value mapping.....	301
17.	System tasks and functions	302
17.1	Display system tasks	302
17.1.1	The display and write tasks.....	303
17.1.2	Strobed monitoring	310
17.1.3	Continuous monitoring	311
17.2	File input-output system tasks and functions.....	311
17.2.1	Opening and closing files.....	311
17.2.2	File output system tasks.....	313
17.2.3	Formatting data to a string	314
17.2.4	Reading data from a file.....	315
17.2.5	File positioning	319
17.2.6	Flushing output	319
17.2.7	I/O error status	319
17.2.8	Loading memory data from a file	320
17.2.9	Loading timing data from an SDF file.....	321
17.3	Timescale system tasks.....	322
17.3.1	\$printtimescale.....	322
17.3.2	\$timeformat.....	323
17.4	Simulation control system tasks.....	326
17.4.1	\$finish	326
17.4.2	\$stop.....	326
17.5	PLA modeling system tasks.....	327
17.5.1	Array types.....	327
17.5.2	Array logic types.....	328
17.5.3	Logic array personality declaration and loading.....	328
17.5.4	Logic array personality formats	328
17.6	Stochastic analysis tasks	331

17.6.1	\$q_initialize.....	331
17.6.2	\$q_add.....	332
17.6.3	\$q_remove.....	332
17.6.4	\$q_full.....	332
17.6.5	\$q_exam.....	332
17.6.6	Status codes.....	333
17.7	Simulation time system functions.....	333
17.7.1	\$time.....	333
17.7.2	\$stime.....	334
17.7.3	\$realtime.....	334
17.8	Conversion functions.....	335
17.9	Probabilistic distribution functions.....	336
17.9.1	\$random function.....	336
17.9.2	\$dist_functions.....	337
17.9.3	Algorithm for probabilistic distribution functions.....	338
17.10	Command line input.....	345
17.10.1	\$test\$plusargs (string).....	346
17.10.2	\$value\$plusargs (user_string, variable).....	346
18.	Value change dump (VCD) files.....	349
18.1	Creating the four state value change dump file.....	349
18.1.1	Specifying the name of the dump file (\$dumpfile).....	349
18.1.2	Specifying the variables to be dumped (\$dumpvars).....	350
18.1.3	Stopping and resuming the dump (\$dumpoff/\$dumpon).....	351
18.1.4	Generating a checkpoint (\$dumpall).....	352
18.1.5	Limiting the size of the dump file (\$dumplimit).....	352
18.1.6	Reading the dump file during simulation (\$dumpflush).....	353
18.2	Format of the four state VCD file.....	354
18.2.1	Syntax of the four state VCD file.....	354
18.2.2	Formats of variable values.....	356
18.2.3	Description of keyword commands.....	357
18.2.4	Four state VCD file format example.....	363
18.3	Creating the extended value change dump file.....	364
18.3.1	Specifying the dumpfile name and the ports to be dumped (\$dumpports).....	364
18.3.2	Stopping and resuming the dump (\$dumpportsoff/\$dumpportson).....	365
18.3.3	Generating a checkpoint (\$dumpportsall).....	366
18.3.4	Limiting the size of the dump file (\$dumpportslimit).....	366
18.3.5	Reading the dump file during simulation (\$dumpportsflush).....	367
18.3.6	Description of keyword commands.....	367
18.3.7	General rules for extended VCD system tasks.....	368
18.4	Format of the extended VCD file.....	368
18.4.1	Syntax of the extended VCD file.....	368
18.4.2	Extended VCD node information.....	370
18.4.3	Value changes.....	372
18.4.4	Extended VCD file format example.....	373
19.	Compiler directives.....	375
19.1	`celldefine and `endcelldefine.....	375
19.2	`default_nettype.....	375
19.3	`define and `undef.....	376
19.3.1	`define.....	376
19.3.2	`undef.....	378

19.4	`ifdef, `else, `elsif, `endif, `ifndef	378
19.5	`include	382
19.6	`resetall.....	382
19.7	`line	383
19.8	`timescale	383
19.9	`unconnected_drive and `nounconnected_drive	385
20.	PLI overview.....	386
20.1	PLI purpose and history (informative).....	386
20.2	User-defined system task or function names	386
20.3	User-defined system task or function types	387
20.4	Overriding built-in system task and function names	387
20.5	User-supplied PLI applications.....	387
20.6	PLI interface mechanism	387
20.7	User-defined system task and function arguments	388
20.8	PLI include files.....	388
20.9	PLI Memory Restrictions.....	388
21.	PLI TF and ACC interface mechanism.....	389
21.1	User-supplied PLI applications.....	389
21.1.1	The sizetf class of PLI applications.....	389
21.1.2	The checktf class of PLI applications.....	389
21.1.3	The calltf class of PLI applications.....	390
21.1.4	The misttf class of PLI applications.....	390
21.1.5	The consumer class of PLI applications.....	390
21.2	Associating PLI applications to a class and system task/function name	390
21.3	PLI application arguments.....	391
21.3.1	The data C argument.....	391
21.3.2	The reason C argument.....	391
21.3.3	The paramvc C argument.....	392
22.	Using ACC routines.....	393
22.1	ACC routine definition.....	393
22.2	The handle data type.....	393
22.3	Using ACC routines.....	394
22.3.1	Header files.....	394
22.3.2	Initializing ACC routines.....	394
22.3.3	Exiting ACC routines.....	394
22.4	List of ACC routines by major category.....	394
22.4.1	Fetch routines.....	395
22.4.2	Handle routines.....	396
22.4.3	Next routines.....	397
22.4.4	Modify routines.....	399
22.4.5	Miscellaneous routines.....	399
22.4.6	VCL routines.....	400
22.5	Accessible objects.....	400
22.5.1	ACC routines that operate on module instances	402
22.5.2	ACC routines that operate on module ports	402
22.5.3	ACC routines that operate on bits of a port	403
22.5.4	ACC routines that operate on module paths or data paths	403
22.5.5	ACC routines that operate on intermodule paths	404

22.5.6	ACC routines that operate on top-level modules	404
22.5.7	ACC routines that operate on primitive instances	404
22.5.8	ACC routines that operate on primitive terminals	405
22.5.9	ACC routines that operate on nets	405
22.5.10	ACC routines that operate on reg types	406
22.5.11	ACC routines that operate on integer, real, and time variables	406
22.5.12	ACC routines that operate on named events	406
22.5.13	ACC routines that operate on parameters and specparams	407
22.5.14	ACC routines that operate on timing checks	407
22.5.15	ACC routines that operate on timing check terminals	407
22.5.16	ACC routines that operate on user-defined system task/function arguments	408
22.6	ACC routine types and fulltypes	408
22.7	Error handling	411
22.7.1	Suppressing error messages	412
22.7.2	Enabling warnings	412
22.7.3	Testing for errors	412
22.7.4	Example	412
22.7.5	Exception values	413
22.8	Reading and writing delay values	413
22.8.1	Number of delays for Verilog HDL objects	414
22.8.2	ACC routine configuration	414
22.8.3	Determining the number of arguments for ACC delay routines	415
22.9	String handling	419
22.9.1	ACC routines share an internal string buffer	419
22.9.2	String buffer reset	420
22.9.3	Preserving string values	421
22.9.4	Example of preserving string values	421
22.10	Using VCL ACC routines	421
22.10.1	VCL objects	422
22.10.2	The VCL record definition	422
22.10.3	Effects of acc_initialize() and acc_close() on VCL consumer routines	425
22.10.4	An example of using VCL ACC routines	425
23.	ACC routine definitions	428
23.1	acc_append_delays()	429
23.2	acc_append_pulse()	433
23.3	acc_close()	435
23.4	acc_collect()	436
23.5	acc_compare_handles()	438
23.6	acc_configure()	439
23.7	acc_count()	448
23.8	acc_fetch_argc()	449
23.9	acc_fetch_argv()	450
23.10	acc_fetch_attribute()	452
23.11	acc_fetch_attribute_int()	456
23.12	acc_fetch_attribute_str()	457
23.13	acc_fetch_defname()	458
23.14	acc_fetch_delay_mode()	459
23.15	acc_fetch_delays()	461
23.16	acc_fetch_direction()	465
23.17	acc_fetch_edge()	466
23.18	acc_fetch_fullname()	468
23.19	acc_fetch_fulltype()	470

23.20 acc_fetch_index()	473
23.21 acc_fetch_location()	475
23.22 acc_fetch_name()	477
23.23 acc_fetch_paramtype()	479
23.24 acc_fetch_paramval()	480
23.25 acc_fetch_polarity()	482
23.26 acc_fetch_precision()	483
23.27 acc_fetch_pulse()	484
23.28 acc_fetch_range()	487
23.29 acc_fetch_size()	488
23.30 acc_fetch_tfarg(), acc_fetch_itfarg()	489
23.31 acc_fetch_tfarg_int(), acc_fetch_itfarg_int()	491
23.32 acc_fetch_tfarg_str(), acc_fetch_itfarg_str()	492
23.33 acc_fetch_timescale_info()	493
23.34 acc_fetch_type()	495
23.35 acc_fetch_type_str()	497
23.36 acc_fetch_value()	498
23.37 acc_free()	503
23.38 acc_handle_by_name()	504
23.39 acc_handle_calling_mode_m()	506
23.40 acc_handle_condition()	507
23.41 acc_handle_conn()	508
23.42 acc_handle_datapath()	509
23.43 acc_handle_hiconn()	510
23.44 acc_handle_interactive_scope()	512
23.45 acc_handle_loconn()	513
23.46 acc_handle_modpath()	514
23.47 acc_handle_notifier()	516
23.48 acc_handle_object()	517
23.49 acc_handle_parent()	519
23.50 acc_handle_path()	520
23.51 acc_handle_pathin()	521
23.52 acc_handle_pathout()	522
23.53 acc_handle_port()	523
23.54 acc_handle_scope()	525
23.55 acc_handle_simulated_net()	526
23.56 acc_handle_tchk()	528
23.57 acc_handle_tchkarg1()	532
23.58 acc_handle_tchkarg2()	534
23.59 acc_handle_terminal()	535
23.60 acc_handle_tfarg(), acc_handle_itfarg()	536
23.61 acc_handle_tnst()	538
23.62 acc_initialize()	539
23.63 acc_next()	540
23.64 acc_next_bit()	544
23.65 acc_next_cell()	546
23.66 acc_next_cell_load()	547
23.67 acc_next_child()	549
23.68 acc_next_driver()	550
23.69 acc_next_hiconn()	551
23.70 acc_next_input()	553
23.71 acc_next_load()	555
23.72 acc_next_loconn()	557
23.73 acc_next_modpath()	558

23.74	acc_next_net()	559
23.75	acc_next_output()	560
23.76	acc_next_parameter()	562
23.77	acc_next_port()	563
23.78	acc_next_portout()	565
23.79	acc_next_primitive()	566
23.80	acc_next_scope()	567
23.81	acc_next_specparam()	568
23.82	acc_next_tchk()	569
23.83	acc_next_terminal()	571
23.84	acc_next_topmod()	572
23.85	acc_object_in_typelist()	573
23.86	acc_object_of_type()	575
23.87	acc_product_type()	577
23.88	acc_product_version()	579
23.89	acc_release_object()	580
23.90	acc_replace_delays()	581
23.91	acc_replace_pulsere()	585
23.92	acc_reset_buffer()	588
23.93	acc_set_interactive_scope()	589
23.94	acc_set_pulsere()	590
23.95	acc_set_scope()	592
23.96	acc_set_value()	594
23.97	acc_vcl_add()	599
23.98	acc_vcl_delete()	601
23.99	acc_version()	602
24.	Using TF routines	603
24.1	TF routine definition	603
24.2	TF routine system task/function arguments	603
24.3	Reading and writing system task/function argument values	603
24.3.1	Reading and writing 2-state parameter argument values	603
24.3.2	Reading and writing 4-state values	603
24.3.3	Reading and writing strength values	604
24.3.4	Reading and writing to memories	604
24.3.5	Reading and writing string values	604
24.3.6	Writing return values of user-defined functions	604
24.3.7	Writing the correct C data types	604
24.4	Value change detection	605
24.5	Simulation time	605
24.6	Simulation synchronization	605
24.7	Instances of user-defined tasks or functions	606
24.8	Module and scope instance names	606
24.9	Saving information from one system TF call to the next	606
24.10	Displaying output messages	606
24.11	Stopping and finishing	606
25.	TF routine definitions	607
25.1	io_mcdprintf()	608
25.2	io_printf()	609
25.3	mc_scan_plusargs()	610
25.4	tf_add_long()	611

25.5	tf_asynchoff(), tf_iasynchoff()	612
25.6	tf_asynchon(), tf_iasynchon()	613
25.7	tf_clearalldelays(), tf_iclearalldelays()	614
25.8	tf_compare_long()	615
25.9	tf_copypvc_ag(), tf_ico pypvc_ag()	616
25.10	tf_divide_long()	617
25.11	tf_do.nish()	618
25.12	tf_dostop()	619
25.13	tf_error()	620
25.14	tf_evaluatep(), tf_ievaluatep()	621
25.15	tf_exprinfo(), tf_iexprinfo()	622
25.16	tf_getcstringp(), tf_igetcstringp()	625
25.17	tf_getinstance()	626
25.18	tf_getlongp(), tf_igetlongp()	627
25.19	tf_getlongtime(), tf_igetlongtime()	628
25.20	tf_getnextlongtime()	629
25.21	tf_getp(), tf_igetp()	630
25.22	tf_getpchange(), tf_igetpchange()	631
25.23	tf_getrealp(), tf_igetrealp()	632
25.24	tf_getrealtime(), tf_igetrealtime()	633
25.25	tf_gettime(), tf_igettime()	634
25.26	tf_gettimeprecision(), tf_igettimeprecision()	635
25.27	tf_gettimeunit(), tf_igettimeunit()	636
25.28	tf_getworkarea(), tf_igetworkarea()	637
25.29	tf_long_to_real()	638
25.30	tf_longtime_tostr()	639
25.31	tf_message()	640
25.32	tf_mipname(), tf_imipname()	642
25.33	tf_movepvc_ag(), tf_im ovepvc_ag()	643
25.34	tf_multiply_long()	644
25.35	tf_nodeinfo(), tf_inodeinfo()	645
25.36	tf_nump(), tf_inump()	649
25.37	tf_propagatep(), tf_ipropagatep()	650
25.38	tf_putlongp(), tf_iputlongp()	651
25.39	tf_putp(), tf_iputp()	652
25.40	tf_putrealp(), tf_iputrealp()	653
25.41	tf_read_restart()	654
25.42	tf_real_to_long()	655
25.43	tf_rosynchronize(), tf_irosynchronize()	656
25.44	tf_scale_longdelay()	657
25.45	tf_scale_realdelay()	658
25.46	tf_setdelay(), tf_isetdelay()	659
25.47	tf_setlongdelay(), tf_isetlongdelay()	660
25.48	tf_setrealdelay(), tf_isetrealdelay()	661
25.49	tf_setworkarea(), tf_isetworkarea()	662
25.50	tf_sizep(), tf_isizep()	663
25.51	tf_spname(), tf_ispname()	664
25.52	tf_strdelputp(), tf_istrdelputp()	665
25.53	tf_strgetp(), tf_istrgetp()	667
25.54	tf_strgettime()	668
25.55	tf_strlongdelputp(), tf_istrlongdelputp()	669
25.56	tf_strrealdelputp(), tf_istrrealdelputp()	671
25.57	tf_subtract_long()	673
25.58	tf_synchronize(), tf_isynchronize()	675

25.59	tf_testpvc_.ag(), tf_itestpvc_.ag().....	676
25.60	tf_text()	677
25.61	tf_typep(), tf_itypep()	678
25.62	tf_unscale_longdelay().....	679
25.63	tf_unscale_realdelay().....	680
25.64	tf_warning()	681
25.65	tf_write_save()	682
26.	Using VPI routines.....	683
26.1	VPI system tasks and functions	683
26.2	The VPI interface	683
26.2.1	VPI callbacks	683
26.2.2	VPI access to Verilog HDL objects and simulation objects	684
26.2.3	Error handling	684
26.2.4	Function availability	684
26.2.5	Traversing expressions.....	684
26.3	VPI object classifications.....	685
26.3.1	Accessing object relationships and properties	686
26.3.2	Object type properties	687
26.3.3	Object file and line properties.....	687
26.3.4	Delays and values	688
26.4	List of VPI routines by functional category.....	688
26.5	Key to data model diagrams	690
26.5.1	Diagram key for objects and classes	691
26.5.2	Diagram key for accessing properties.....	691
26.5.3	Diagram key for traversing relationships	692
26.6	Object data model diagrams.....	693
26.6.1	Module	694
26.6.2	Instance arrays	695
26.6.3	Scope	696
26.6.4	IO declaration	696
26.6.5	Ports	697
26.6.6	Nets and net arrays.....	698
26.6.7	Regs and reg arrays.....	700
26.6.8	IO declaration	702
26.6.9	Memory	703
26.6.10	IO declaration	704
26.6.11	Named event	704
26.6.12	Parameter, specparam	705
26.6.13	Primitive, prim term.....	706
26.6.14	UDP.....	707
26.6.15	Model path, path term	708
26.6.16	Intermodule path	708
26.6.17	Timing check	709
26.6.18	Task, function declaration	709
26.6.19	Task and function call.....	710
26.6.20	Frames.....	711
26.6.21	Delay terminals	712
26.6.22	Net drivers and loads	712
26.6.23	Reg drivers and loads	712
26.6.24	Continuous assignment	713
26.6.25	Simple expressions.....	714
26.6.26	Expressions	715

26.6.27	Process, block, statement, event statement	716
26.6.28	Assignment	717
26.6.29	Delay control	717
26.6.30	Event control	717
26.6.31	Repeat control	717
26.6.32	While, repeat, wait	718
26.6.33	For	718
26.6.34	Forever	718
26.6.35	If, if-else	719
26.6.36	Case	719
26.6.37	Assign statement, deassign, force, release	720
26.6.38	Disable	720
26.6.39	Callback	721
26.6.40	Time queue	721
26.6.41	Active time format	721
26.6.42	Attributes	722
26.6.43	Iterator	723
27.	VPI routine definitions	724
27.1	vpi_chk_error()	725
27.2	vpi_compare_objects()	726
27.3	vpi_control()	727
27.4	vpi_flush()	728
27.5	vpi_free_object()	729
27.6	vpi_get()	730
27.7	vpi_get_cb_info()	731
27.8	vpi_get_data()	732
27.9	vpi_get_delays()	734
27.10	vpi_get_str()	737
27.11	vpi_get_sysf_info()	738
27.12	vpi_get_time()	739
27.13	vpi_get_userdata()	740
27.14	vpi_get_value()	741
27.15	vpi_get_vlog_info()	747
27.16	vpi_handle()	748
27.17	vpi_handle_by_index()	749
27.18	vpi_handle_by_multi_index()	750
27.19	vpi_handle_by_name()	751
27.20	vpi_handle_multi()	752
27.21	vpi_iterate()	753
27.22	vpi_mcd_close()	754
27.23	vpi_mcd_flush()	755
27.24	vpi_mcd_name()	756
27.25	vpi_mcd_open()	757
27.26	vpi_mcd_printf()	758
27.27	vpi_mcd_vprintf()	759
27.28	vpi_printf()	760
27.29	vpi_put_data()	761
27.30	vpi_put_delays()	763
27.31	vpi_put_userdata()	766
27.32	vpi_put_value()	767
27.33	vpi_register_cb()	770
27.33.1	Simulation-event-related callbacks	771

27.33.2 Simulation-time-related callbacks	774
27.33.3 Simulator action and feature related callbacks	775
27.34 vpi_register_systf()	778
27.34.1 System task and function callbacks	779
27.34.2 Initializing VPI system task/function callbacks	780
27.34.3 Registering multiple system tasks and functions	781
27.35 vpi_remove_cb()	782
27.36 vpi_scan()	783
27.37 vpi_vprintf()	784
Annex A (normative) Formal syntax definition	785
A.1 Source text	785
A.1.1 Library source text	785
A.1.2 Configuration source text	785
A.1.3 Module and primitive source text	786
A.1.4 Module parameters and ports	786
A.1.5 Module items	786
A.2 Declarations	787
A.2.1 Declaration types	787
A.2.2 Declaration data types	789
A.2.3 Declaration lists	789
A.2.4 Declaration assignments	790
A.2.5 Declaration ranges	790
A.2.6 Function declarations	790
A.2.7 Task declarations	790
A.2.8 Block item declarations	791
A.3 Primitive instances	792
A.3.1 Primitive instantiation and instances	792
A.3.2 Primitive strengths	792
A.3.3 Primitive terminals	793
A.3.4 Primitive gate and switch types	793
A.4 Module and generated instantiation	793
A.4.1 Module instantiation	793
A.4.2 Generated instantiation	793
A.5 UDP declaration and instantiation	794
A.5.1 UDP declaration	794
A.5.2 UDP ports	794
A.5.3 UDP body	795
A.5.4 UDP instantiation	795
A.6 Behavioral statements	795
A.6.1 Continuous assignment statements	795
A.6.2 Procedural blocks and assignments	795
A.6.3 Parallel and sequential blocks	796
A.6.4 Statements	796
A.6.5 Timing control statements	796
A.6.6 Conditional statements	797
A.6.7 Case statements	798
A.6.8 Looping statements	798
A.6.9 Task enable statements	798
A.7 Specify section	798
A.7.1 Specify block declaration	798
A.7.2 Specify path declarations	799
A.7.3 Specify block terminals	799

A.7.4 Specify path delays	795
A.7.5 System timing checks.....	801
A.8 Expressions	803
A.8.1 Concatenations	803
A.8.2 Function calls	803
A.8.3 Expressions	804
A.8.4 Primaries	805
A.8.5 Expression left-side values.....	805
A.8.6 Operators.....	806
A.8.7 Numbers	806
A.8.8 Strings	807
A.9 General	807
A.9.1 Attributes.....	807
A.9.2 Comments	807
A.9.3 Identifiers	807
A.9.4 Identifier branches.....	808
A.9.5 White space	809
Annex B (normative) List of keywords	810
Annex C (informative) System tasks and functions	812
C.1 \$countdrivers.....	812
C.2 \$getpattern.....	813
C.3 \$input	814
C.4 \$key and \$nokey	814
C.5 \$list.....	815
C.6 \$log and \$nolog.....	815
C.7 \$reset, \$reset_count, and \$reset_value.....	815
C.8 \$save, \$restart, and \$incsave.....	816
C.9 \$scale.....	817
C.10 \$scope	817
C.11 \$showscopes	817
C.12 \$showvars	818
C.13 \$sreadmemb and \$sreadmemh.....	818
Annex D (informative) Compiler directives	819
D.1 `default_decay_time.....	819
D.2 `default_trireg_strength.....	819
D.3 `delay_mode_distributed.....	820
D.4 `delay_mode_path.....	820
D.5 `delay_mode_unit	820
D.6 `delay_mode_zero.....	820
Annex E (normative) acc_user.h.....	821
Annex F (normative) veriusers.h.....	830
Annex G (normative) vpi_user.h.....	838
Annex H (informative) Bibliography.....	852
Annex I (informative) List of Participants.....	853

INTERNATIONAL ELECTROTECHNICAL COMMISSION

BEHAVIOURAL LANGUAGES –

Part 4: Verilog® hardware description language

FOREWORD

- 1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
- 2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
- 3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
- 4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
- 5) IEC provides no marking procedure to indicate its approval and cannot be rendered responsible for any equipment declared to be in conformity with an IEC Publication.
- 6) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC/IEEE 61691-4 has been processed through IEC technical committee 93: Design automation.

The text of this standard is based on the following documents:

IEEE Std	FDIS	Report on voting
1364 (2001)	93/192/FDIS	93/197/RVD

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

This publication has been drafted in accordance with the ISO/IEC Directives.

The committee has decided that the contents of this publication will remain unchanged until 2006.

IEC 61691 consists of the following parts, under the general title *Behavioural languages*:

IEC/IEEE 61691-1-1, Part 1: *VHDL language reference manual*

IEC 61691-2, Part 2: *VHDL multilogic system for model interoperability*

IEC 61691-3-1, Part 3-1: *Analog description in VHDL* (under consideration)

IEC 61691-3-2, Part 3-2: *Mathematical operation in VHDL*

IEC 61691-3-3, Part 3-3: *Synthesis in VHDL*

IEC 61691-3-4, Part 3-4: *Timing expressions in VHDL* (under consideration)

IEC 61691-3-5, Part 3-5: *Library utilities in VHDL* (under consideration)

IEC/IEEE 61691-4, Part 4: *Verilog® hardware description language*

IEC/IEEE 61691-5, Part 5: *VITAL ASIC (application specific integrated circuit) modeling specification*

IEC/IEEE Dual Logo International Standards

This Dual Logo International Standard is the result of an agreement between the IEC and the Institute of Electrical and Electronics Engineers, Inc. (IEEE). The original IEEE Standard was submitted to the IEC for consideration under the agreement, and the resulting IEC/IEEE Dual Logo International Standard has been published in accordance with the ISO/IEC Directives.

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

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

The IEC and IEEE do not warrant or represent the accuracy or content of the material contained herein, and expressly disclaim any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEC/IEEE Dual Logo International Standards documents are supplied "AS IS".

The existence of an IEC/IEEE Dual Logo International 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 IEC/IEEE Dual Logo International Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEC and IEEE are not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Neither the IEC nor IEEE is undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEC/IEEE Dual Logo International Standards or IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

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

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

Secretary, IEEE-SA Standards Board, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA and/or General Secretary, IEC, 3, rue de Varembé, PO Box 131, 1211 Geneva 20, Switzerland.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

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

IEEE Standard Verilog[®] Hardware Description Language

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 17 March 2001

IEEE-SA Standards Board

Abstract: The Verilog[®] Hardware Description Language (HDL) is defined in this standard. Verilog HDL is a formal notation intended for use in all phases of the creation of electronic systems. Because it is both machine readable and human readable, it supports the development, verification, synthesis, and testing of hardware designs; the communication of hardware design data; and the maintenance, modification, and procurement of hardware. The primary audiences for this standard are the implementors of tools supporting the language and advanced users of the language.

Keywords: computer, computer languages, digital systems, electronic systems, hardware, hardware description languages, hardware design, HDL, PLI, programming language interface, Verilog HDL, Verilog PLI, Verilog[®]

IEEE Introduction

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

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

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

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

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

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

Objective of the IEEE Std 1364-2001 effort

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

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

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

Consolidate existing IEEE Std 1364-1995

Verilog Generate statement

Multi-dimensional arrays

Enhanced Verilog file I/O

Re-entrant tasks

Standardize Verilog configurations

Enhance timing representation

Enhance the VPI routines

Achievements

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

Three new sections have been added. Clause 13, Configuring the contents of a design, deals with configuration management and has been added to facilitate both the sharing of Verilog designs between designers and/or design groups and the repeatability of the exact contents of a given simulation session. Clause 15, Timing checks, has been broken out of Clause 17, System tasks and functions, and details more fully how timing checks are used in specify blocks. Clause 16, Backannotation using the Standard Delay Format (SDF), addresses using back annotation (IEEE Std 1497-1999) within IEEE Std 1364-2001.

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

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

The IEEE Std 1364-2001 Verilog Standards Group organization

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

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

BEHAVIOURAL LANGUAGES –

Part 4: Verilog® hardware description language

1. Overview

1.1 Objectives of this standard

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

- The formal syntax and semantics of all Verilog HDL constructs
- The formal syntax and semantics of Standard Delay Format (SDF) constructs
- Simulation system tasks and functions, such as text output display commands
- Compiler directives, such as text substitution macros and simulation time scaling
- The Programming Language Interface (PLI) binding mechanism
- The formal syntax and semantics of access routines, task/function routines, and Verilog procedural interface routines
- Informative usage examples
- Informative delay model for SDF
- Listings of header files for PLI

1.2 Conventions used in this standard

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

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

- a) To the developers of tools that process the Verilog HDL, the term *shall* denotes a requirement that the standard imposes. The resulting implementation is required to enforce the requirements and to issue an error if the requirement is not met by the input.
- b) To the Verilog HDL model developer, the term *shall* denotes that the characteristics of the Verilog HDL are natural consequences of the language definition. The model developer is required to adhere to the constraint implied by the characteristic. The term *can* denotes optional features that the model

developer can exercise at discretion. If used, however, the model developer is required to follow the requirements set forth by the language definition.

- c) To the Verilog HDL model user, the term *shall* denotes that the characteristics of the models are natural consequences of the language definition. The model user can depend on the characteristics of the model implied by its Verilog HDL source text.

1.3 Syntactic description

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

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

`module_declaration`

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

module **=>** ;

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

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

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

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

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

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

- f) If the name of any category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *msb*_constant_expression and *lsb*_constant_expression are equivalent to constant_expression.

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

1.4 Contents of this standard

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

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

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

Clause 3—Data types: This clause describes net and variable data types. This clause also discusses the parameter data type for constant values and describes drive and charge strength of the values on nets.

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

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

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

Clause 7—Gate and switch level modeling: This clause describes the gate and switch level primitives and logic strength modeling.

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

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

Clause 10—Tasks and functions: This clause describes tasks and functions—procedures that can be called from more than one place in a behavioral model. It describes how tasks can be used like subroutines and how functions can be used to define new operators.

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

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

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

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

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

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

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

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

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

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

Clause 21—PLI TF and ACC interface mechanism

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.5 Header file listings

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

1.6 Examples

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

1.7 Prerequisites

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

Withdrawing
IECNORM.COM: Click to view the full PDF of IEC 61691-4:2004

2. Lexical conventions

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

2.1 Lexical tokens

Verilog HDL source text files shall be a stream of lexical tokens. A *lexical token* shall consist of one or more characters. The layout of tokens in a source file shall be free format—that is, spaces and newlines shall not be syntactically significant other than being token separators, except for escaped identifiers (see 2.7.1).

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

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

2.2 White space

White space shall contain the characters for spaces, tabs, newlines, and formfeeds. These characters shall be ignored except when they serve to separate other lexical tokens. However, blanks and tabs shall be considered significant characters in strings (see 2.6).

2.3 Comments

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

2.4 Operators

Operators are single-, double-, or triple-character sequences and are used in expressions. Clause 4 discusses the use of operators in expressions.

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

2.5 Numbers

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


```

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

```

*Embedded spaces are illegal.

Syntax 2-1—Syntax for integer and real numbers

2.5.1 Integer constants

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

There are two forms to express integer constants. The first form is a simple decimal number, which shall be specified as a sequence of digits 0 through 9, optionally starting with a plus or minus unary operator. The second form specifies a *size constant*, which shall be composed of up to three tokens—an optional size constant, a single quote followed by a base format character, and the digits representing the value of the number.

The first token, a size constant, shall specify the size of the constant in terms of its exact number of bits. It shall be specified as a non-zero unsigned decimal number. For example, the size specification for two hexadecimal digits is 8, because one hexadecimal digit requires 4 bits. Unsized unsigned constants where the high order bit is unknown (X or x) or three-state (Z or z) are extended to the size of the expression containing the constant.

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

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

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

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

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

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

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

Negative numbers shall be represented in 2's complement form.

An X represents the *unknown value* in hexadecimal, octal, and binary constants. A Z represents the *high-impedance value*. See 3.1 for a discussion of the Verilog HDL value set. An X shall set 4 bits to unknown in the hexadecimal base, 3 bits in the octal base, and 1 bit in the binary base. Similarly, a Z shall set 4 bits, 3 bits, and 1 bit, respectively, to the high-impedance value.

If the size of the unsigned number is smaller than the size specified for the constant, the unsigned number shall be padded to the left with zeros. If the leftmost bit in the unsigned number is an X or a Z, then an X or a Z shall be used to pad to the left respectively.

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

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

Examples:

Example 1—Unsize constant numbers

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

Example 2—Sized constant numbers

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

Example 3—Using sign with constant numbers

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

Example 4—Automatic left padding

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

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

Example 5—Using underscore character in numbers

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

NOTES:

- 1) Sized negative constant numbers and sized signed constant numbers are sign-extended when assigned to a reg data type, regardless of whether the reg itself is signed or not.
- 2) Each of the three tokens for specifying a number may be macro substituted.
- 3) The number of bits that make up an unsized number (which is a simple decimal number or a number without the size specification) shall be at least 32.

2.5.2 Real constants

The *real constant numbers* shall be represented as described by IEEE Std 754-1985 [B1],¹ an IEEE standard for double-precision floating-point numbers.

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

Examples:

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

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

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

2.5.3 Conversion

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

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

2.6 Strings

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

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

2.6.1 String variable declaration

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

Example:

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

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

2.6.2 String manipulation

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

Example:

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

The output is:

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

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

2.6.3 Special characters in strings

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

Table 1—Specifying special characters in string

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

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 is either a *simple identifier* or an *escaped identifier* (see 2.7.1). A *simple identifier* shall be any sequence of letters, digits, dollar signs (\$), and underscore characters (_).

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

Example:

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

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

2.7.1 Escaped identifiers

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

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

Example:

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

2.7.2 Generated identifiers

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

2.7.3 Keywords

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

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

2.7.4 System tasks and functions

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

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

```

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

```

*The \$ character in a `system_function_identifier` or `system_task_identifier` shall not be followed by white space. A `system_function_identifier` or `system_task_identifier` shall not be escaped.

Syntax 2-2—Syntax for system tasks and functions

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

- A standard set of \$identifier system tasks and functions, as defined in Clauses 17 and 19.
- Additional \$identifier system tasks and functions defined using the PLI, as described in Clause 20.
- Additional \$identifier system tasks and functions defined by software implementations.

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

Example:

```

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

```

2.7.5 Compiler directives

The ‘ character (the ASCII value 60, called open quote or accent grave) introduces a language construct used to implement compiler directives. The compiler behavior dictated by a compiler directive shall take effect as soon as the compiler reads the directive. The directive shall remain in effect for the rest of the compilation unless a different compiler directive specifies otherwise. A compiler directive in one description file can therefore control compilation behavior in multiple description files.

The ‘identifier compiler directive construct can be defined in two places

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

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

Example:

```
‘define wordsize 8
```

2.8 Attributes

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

The syntax for specifying an attribute is shown in Syntax 2-3.

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

Syntax 2-3—Syntax for attributes

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

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

2.8.1 Examples

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

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

or

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

or

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

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

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

or

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

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

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

or

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

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

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

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

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

Example 6—To attach an attribute to an operator:

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

This sets the value for the attribute mode to be the string c l a.

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

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

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

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

2.8.2 Syntax

The syntax for legal statements with attributes is shown in Syntax 2-4— Syntax 2-11.

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

```
module_declaration ::= (From Annex A - A.1.3)
    { attribute_instance } module_keyword module_identifier
    [ module_parameter_port_list ] [ list_of_ports ];
    { module_item }
endmodule
| { attribute_instance } module_keyword module_identifier
    [ module_parameter_port_list ] [ list_of_port_declarations ];
    { non_port_module_item }
endmodule
```

Syntax 2-4—Syntax for module declaration attributes

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

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

Syntax 2-5—Syntax for port declaration attributes

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

```
module_item ::= (From Annex A - A.1.5)
    module_or_generate_item
    | port_declaration ;
    | { attribute_instance } generated_instantiation
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration
    | { attribute_instance } specify_block
    | { attribute_instance } specparam_declaration
module_or_generate_item ::=
    { attribute_instance } module_or_generate_item_declaration
    | { attribute_instance } parameter_override
    | { attribute_instance } continuous_assign
    | { attribute_instance } gate_instantiation
    | { attribute_instance } udp_instantiation
    | { attribute_instance } module_instantiation
    | { attribute_instance } initial_construct
    | { attribute_instance } always_construct
non_port_module_item ::=
    { attribute_instance } generated_instantiation
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } module_or_generate_item
    | { attribute_instance } parameter_declaration
    | { attribute_instance } specify_block
    | { attribute_instance } specparam_declaration
```

Syntax 2-6—Syntax for module item attributes

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

```

function_port_list ::= (From Annex A - A.2.6)
    { attribute_instance } input_declaration { , { attribute_instance } input_declaration }
task_item_declaration ::= (From Annex A - A.2.7)
    block_item_declaration
    | { attribute_instance } input_declaration ;
    | { attribute_instance } output_declaration ;
    | { attribute_instance } inout_declaration ;
task_port_item ::=
    { attribute_instance } input_declaration
    | { attribute_instance } output_declaration
    | { attribute_instance } inout_declaration
block_item_declaration ::= (From Annex A - A.2.8)
    { attribute_instance } block_reg_declaration
    | { attribute_instance } event_declaration
    | { attribute_instance } integer_declaration
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration
    | { attribute_instance } real_declaration
    | { attribute_instance } realtime_declaration
    | { attribute_instance } time_declaration

```

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

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

```

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

```

Syntax 2-8—Syntax for port connection attributes

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

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

Syntax 2-9—Syntax for udp attributes

3. Data types

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

3.1 Value set

The Verilog HDL value set consists of four basic values:

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

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

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

Almost all of the data types in the Verilog HDL store all four basic values. The exception is the *event* type (see 9.7.3), which has no storage. All bits of vectors can be independently set to one of the four basic values.

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

3.2 Nets and variables

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

3.2.1 Net declarations

The *net* data types shall represent physical connections between structural entities, such as gates. A net shall not store a value (except for the *triereg* 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 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 *triereg*, in which case it shall hold the previously driven value. It is illegal to redeclare a name already declared by a net, parameter, or variable declaration (see 3.1.2).

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

```

net_declaration ::= (From Annex A - A.2.1.3)
    net_type [ signed ]
        [ delay3 ] list_of_net_identifiers ;
    net_type [ drive_strength ] [ signed ]
        [ delay3 ] list_of_net_decl_assignments ;
    net_type [ vectored | scalared ] [ signed ]
        range [ delay3 ] list_of_net_identifiers ;
    net_type [ drive_strength ] [ vectored | scalared ] [ signed ]
        range [ delay3 ] list_of_net_decl_assignments ;
    trireg [ charge_strength ] [ signed ]
        [ delay3 ] list_of_net_identifiers ;
    trireg [ drive_strength ] [ signed ]
        [ delay3 ] list_of_net_decl_assignments ;
    trireg [ charge_strength ] [ vectored | scalared ] [ signed ]
        range [ delay3 ] list_of_net_identifiers ;
    trireg [ drive_strength ] [ vectored | scalared ] [ signed ]
        range [ delay3 ] list_of_net_decl_assignments ;

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

drive_strength ::= (From Annex A - A.2.2.2)
    ( strength0 , strength1 )
    | ( strength1 , strength0 )
    | ( strength0 , highz1 )
    | ( strength1 , highz0 )
    | ( highz0 , strength1 )
    | ( highz1 , strength0 )

strength0 ::= supply0 | strong0 | pull0 | weak0
strength1 ::= supply1 | strong1 | pull1 | weak1
charge_strength ::= ( small ) | ( medium ) | ( large )
delay3 ::= (From Annex A - A.2.2.3)
    # delay_value | # ( delay_value [ , delay_value [ , delay_value ] ] )
delay2 ::=
    # delay_value | # ( delay_value [ , delay_value ] )
delay_value ::=
    unsigned_number
    | parameter_identifier
    | specparam_identifier
    | mintypmax_expression

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

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

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

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

range ::=
    [ msb_constant_expression : lsb_constant_expression ]

```

Syntax 3-1 Syntax for net declaration

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

3.2.2 Variable declarations

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

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

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

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

Syntax 3-2 Syntax for variable declaration

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

CAUTION

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

3.3 Vectors

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

3.3.1 Specifying vectors

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

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

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

Examples:

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

NOTES:

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

3.3.2 Vector net accessibility

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

Examples:

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

3.4 Strengths

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

charge strength shall only be used when declaring a net of type **trireg**

drive strength shall only be used when placing a continuous assignment on a net in the same statement that declares the net

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

3.4.1 Charge strength

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

- **small**
- **medium**
- **large**

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

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

Examples:

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

3.4.2 Drive strength

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

3.5 Implicit declarations

The syntax shown in 3.2 shall be used to declare nets and variables explicitly. In the absence of an explicit declaration, an implicit net of default net type shall be assumed in the following circumstances:

- If an identifier is used in a port expression declaration, then an implicit net of type **wire** shall be assumed, with the vector width of the port expression declaration. See 12.3.3 for a discussion of port expression declarations.
- If an identifier is used in the terminal list of a primitive instance or a module instance, and that identifier has not been explicitly declared previously in one of the declaration statements of the instantiating module, then an implicit scalar net of default net type shall be assumed. See Section 19 for a discussion of control of the type for implicitly declared nets with the ``default_nettype` compiler directive.

If an identifier appears on the left-hand side of a continuous assignment statement, and that identifier has not been declared previously, an implicit scalar net declaration of the default net type is assumed.

3.6 Net initialization

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

3.7 Net types

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

Table 2—Net types

wire	tri	tri0	supply0
wand	triand	tri1	supply1
wor	trior	trireg	

3.7.1 Wire and tri nets

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

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

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

Table 3—Truth table for wire and tri nets

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

3.7.2 Wired nets

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

The net types *wor* and *trior* shall be identical in their syntax and functionality. The net types *wand* and *triand* shall be identical in their syntax and functionality. Table 4 and Table 5 give the truth tables for wired nets. Note that they assume equal strengths for both drivers. See 7.9 for a discussion of logic strength modeling.

Table 4—Truth table for wand and triand nets

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

Table 5—Truth table for wor and 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 *triereg* net stores a value and is used to model charge storage nodes. A triereg net can be in one of two states:

- driven state* When at least one driver of a triereg net has a value of 1, 0, or x, the resolved value propagates into the triereg net and is the driven value of the triereg net.
- capacitive state* When all the drivers of a triereg net are at the high-impedance value (Z), the triereg net retains its last driven value; the high-impedance value does not propagate from the driver to the triereg.

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

Examples:

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

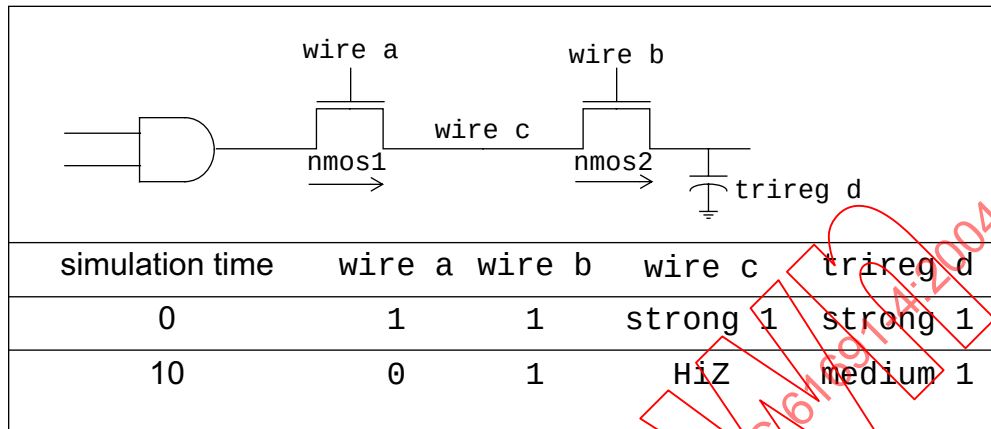


Figure 1—Simulation values of a trireg and its driver

- 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 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.

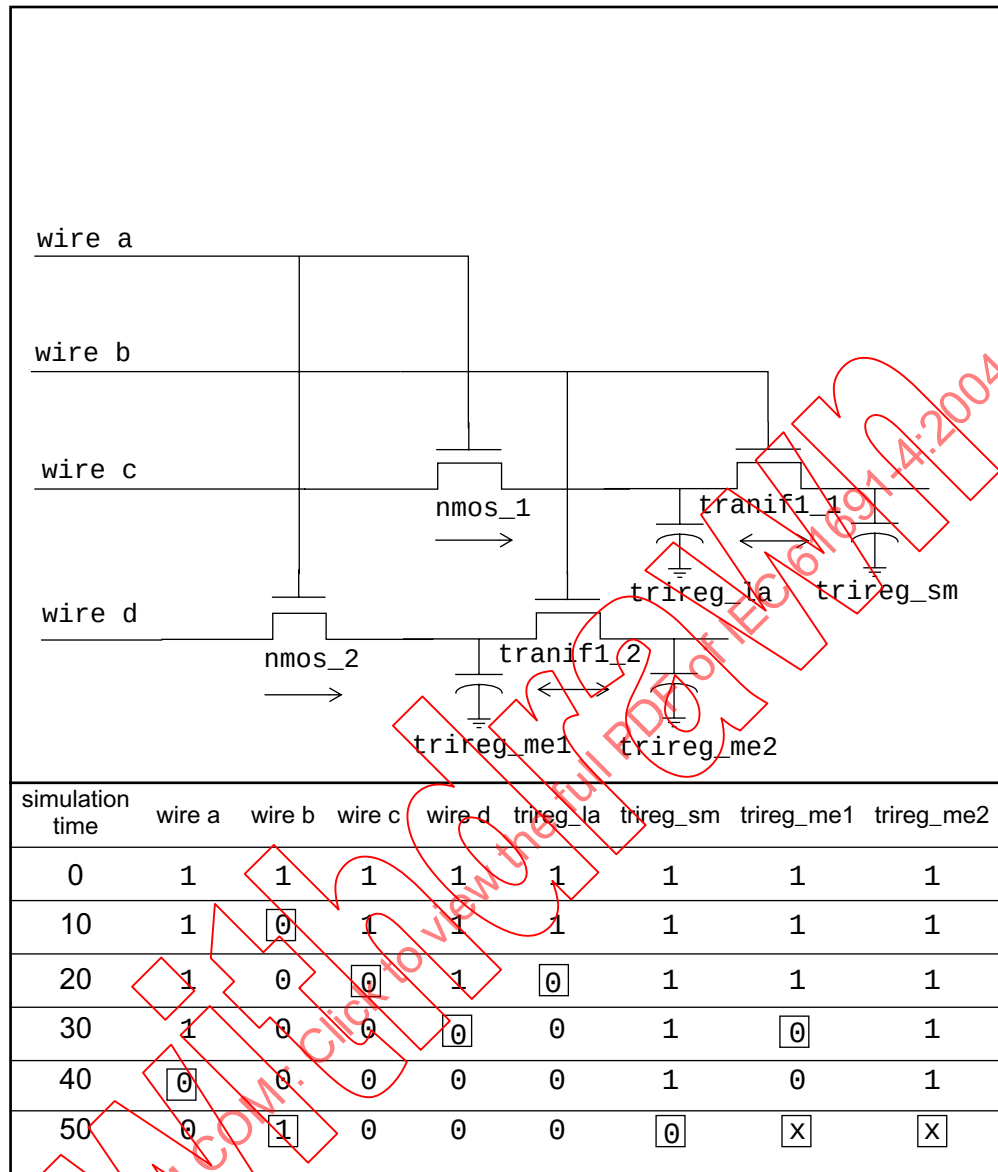


Figure 2—Simulation results of a capacitive network

In Figure 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.

- e) At simulation time 40, the value of wire a changes to 0, disconnecting `trireg_la` and `trireg_me1` from their drivers. These `trireg` nets enter the capacitive state and store the value 0.
- f) At simulation time 50, the value of wire b changes to 1.

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

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

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

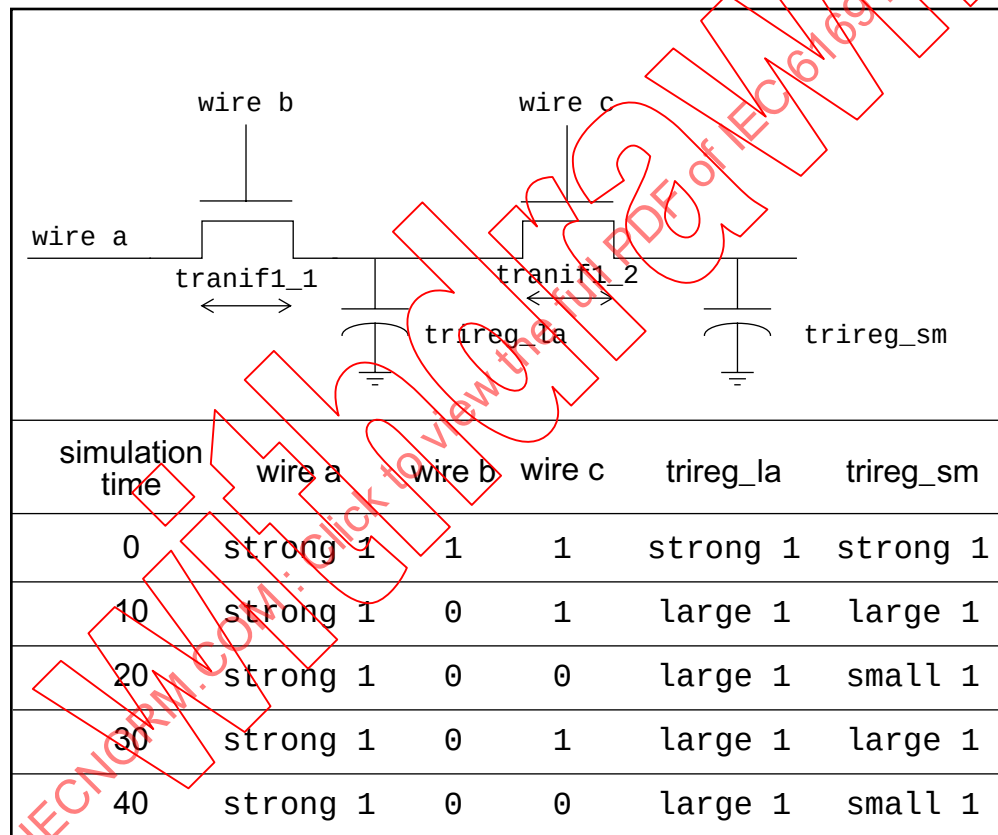


Figure 3—Simulation results of charge sharing

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

- a) At simulation time 0, the values of wire a, wire b, and wire c are 1, and wire a drives a strong 1 into `trireg_la` and `trireg_sm`.

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

3.7.3.2 Ideal capacitive state and charge decay

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

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 Clause 7. for a description of strength modeling.

A *tri0* net is equivalent to a wire net with a continuous 0 value of pull strength driving it. A *tri1* net is equivalent to a wire net with a continuous 1 value of pull strength driving it.

A truth table for *tri0* is shown in Table 6. A truth table for *tri1* is shown in Table 7.

Table 6— 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 7— 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 regs

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

3.9 Integers, reals, times, and realtimes

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

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

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

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

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

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

A **time** variable is used for storing and manipulating simulation time quantities in situations where timing checks are required and for diagnostics and debugging purposes. This data type is typically used in conjunction with the **\$time** system function (see 17.).

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

The **time** variables shall behave the same as a reg of at least 64 bits, with the least significant bit being bit 0. They shall be unsigned quantities, and unsigned arithmetic shall be performed on them. In contrast, integer variables shall be treated as signed regs with the least significant bit being zero. Arithmetic operations performed on integer variables shall produce 2's complement results.

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

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

- Not all Verilog HDL operators can be used with real number values. See Table 17 for lists of valid and invalid operators for real numbers and real variables.
- Real variables shall not use range in the declaration
- Real variables shall default to an initial value of zero.

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

Examples:

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

3.9.1 Operators and real numbers

The result of using logical or relational operators on real numbers and real variables is a single-bit scalar value. Not all Verilog HDL operators can be used with expressions involving real numbers and real variables. Table 4-9 lists the valid operators for use with real numbers and real variables. Real number constants and real variables are also prohibited in the following cases:

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

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. If the fractional part of the real number is exactly 0.5, it shall be rounded away from zero.

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

See Clause 17 for a discussion of system tasks that perform explicit conversion.

3.10 Arrays

An array declaration for a net or a variable declares an element type which is either scalar or vector (see 3.3). For example:

Declaration	Element Type
reg x[11:0];	scalar reg
wire [0:7] y[5:0];	eight-bit-wide vector wire indexed from 0 to 7
reg [31:0] x [127:0];	thirty-two-bit-wide reg

NOTE—Array size does not affect the element size..

Arrays can be used to group elements of the declared element type into multi-dimensional objects. Arrays shall be declared by specifying the element address range(s) after the declared identifier. Each dimension shall be represented by an address range. See 3.2.1 and 3.2.2 for net and variable declarations. The expression(s) 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 arrays and elements of the declared data type. This ability makes it convenient to declare both arrays and elements that match the element vector width in the same declaration statement.

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

Implementations may limit the maximum size of an array, but they shall at least be 16777216 (2^{24}).

3.10.1 Net arrays

Arrays of nets can be used to connect ports of generated instances. Each element of the array can be used in the same fashion as a scalar or vector net.

3.10.2 reg and variable arrays

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

3.10.3 Memories

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

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

3.10.3.1 Array examples

3.10.3.1.1 Array declarations

```

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

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

```

3.10.3.1.2 Assignment to array elements

The assignment statements in this section assume the presence of the declarations in 3.10.3.1.1.

```

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

```

3.10.3.1.3 Memory differences

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

```

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

```

3.11 Parameters

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

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

3.11.1 Module parameters

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

```

local_parameter_declaration ::= (From Annex A - A.2.1.1)
    localparam [ signed ] [ range ] list_of_param_assignments ;
    localparam integer list_of_param_assignments ;
    localparam real list_of_param_assignments ;
    localparam realtime list_of_param_assignments ;
    localparam time list_of_param_assignments ;
parameter_declaration ::=
    parameter [ signed ] [ range ] list_of_param_assignments ;
    parameter integer list_of_param_assignments ;
    parameter real list_of_param_assignments ;
    parameter realtime list_of_param_assignments ;
    parameter time list_of_param_assignments ;
list_of_param_assignments ::= (From Annex A - A.2.3)
    param_assignment { , param_assignment }
param_assignment ::= (From Annex A - A.2.4)
    parameter_identifier = constant_expression
range ::= (From Annex A - A.2.5)
    [ msb_constant_expression : lsb_constant_expression ]

```

Syntax 3-4 Syntax for module parameter declaration

The *list_of_param_assignments* shall be a comma-separated list of assignments, where the right hand side of the assignment shall be a constant expression; that is, an expression containing only constant numbers and previously defined parameters. (See 4.)

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

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

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

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

- A parameter with a signed type specification and with a range specification shall be signed, and shall be the range of its declaration. The sign and range shall not be affected by value overrides.
- A parameter with no range specification, and with either a signed type specification or no type specification, shall have an implied range with an *lsb* equal to 0 and an *msb* equal to one less than the size of the final value assigned to the parameter.
- A parameter with no range specification, and with either a signed type specification or no type specification, and for which the final value assigned to it is unsized, shall have an implied range with an *lsb* equal to 0 and an *msb* equal to an implementation-dependent value of at least 31.

The conversion rules between real and integer values described in 3.9.2 apply to parameters as well.

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;

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

3.11.2 Local parameters - localparam

Verilog HDL **localparam** - local parameter(s) are identical to parameters except that they can not directly be modified with the **defparam** statement or by the ordered or named parameter value assignment. Local parameters can be assigned to a constant expression containing a parameter which can be modified with the **defparam** statement or by the ordered or named parameter value assignment. See 12.1.3 for details.

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

3.11.3 Specify parameters

The syntax for declaring specify parameters is shown in Syntax 3-5.

```
specparam_declaration ::= (From Annex A - A.2.2.1)
    specparam [ range ] list_of_specparam_assignments ;
list_of_specparam_assignments ::= (From Annex A- A.2.3)
    specparam_assignment { , specparam_assignment }
specparam_assignment ::= (From Annex A - A.2.4)
    specparam_identifier = constant_mintypmax_expression
    | pulse_control_specparam
pulse_control_specparam ::=
    PATHPULSE$ = ( reject_limit_value [ , error_limit_value ] ) ;
    | PATHPULSE$specify_input_terminal_descriptor$specify_output_terminal_descriptor
    = ( reject_limit_value [ , error_limit_value ] ) ;
error_limit_value ::=
    limit_value
reject_limit_value ::=
    limit_value
limit_value ::=
    constant_mintypmax_expression
range ::= (From Annex A - A.2.5)
    [ msb_constant_expression : lsb_constant_expression ]
```

Syntax 3-5 Syntax of the specparam declaration

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

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

The specify parameters and module parameters shall not be interchangeable. In addition, module parameters shall not be assigned a constant expression that includes any specify parameters. Table 8 summarizes the differences between the two types of parameter declarations.

Table 8—Differences between specparams and parameters

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

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

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

Examples:

```
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.

Examples:

```
module RAM16GEN (DOUT, DIN, ADR, WE, CE)
specparam dhold = 1.0;
specparam ddly = 1.0;
parameter width = 1;
parameter regsize = dhold + 1.0; // Illegal - can't assign
                                // specparams to parameters
endmodule
```

3.12 Name spaces

In Verilog HDL, there are seven name spaces, two are global and five 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. Once a name is used to define a module, macromodule, or primitive, the name shall not be used again to declare another module, macromodule, or primitive.

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 19.3). The text macro names are defined in the linear order of appearance in the set of input files that make up the description of the design unit. Subsequent definitions of the same name override the previous definitions for the balance of the input files.

There are five local name spaces: *block*, *module*, *port*, *specify block*, and *attribute*. Once a name is defined within one of the five name spaces, it shall not be defined again in that space (with the same or a different type).

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, parameters, named events and the variable type of declaration (see 3.2.2). The variable type of declaration includes the **reg**, **integer**, **time**, **real**, and **realtime** declarations.

The *module name space* is introduced by the **module**, **macromodule**, and **primitive** constructs. It unifies the definition of functions, tasks, named blocks, instance names, parameters, named events, net type of declaration, and variable 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 variable or a wire with the same name as the port name.

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

The attribute name space is enclosed by the (* and *) constructs attached to a language element (see 2.8). An attribute name can be defined and used only in the attribute name space. Any other type of name cannot be defined in this name space.

4. Expressions

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

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

Some statement constructs require an expression to be a *constant expression*. The operands of a constant expression consist of constant numbers, parameter names, constant bit-selects of parameters, constant part-selects of parameters, and *constant function calls* (see 10.3.5) only, but they can use any of the operators defined in Table 9.

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 variable data types. Descriptions pertaining to variable usage apply to all of these data types.

An *operand* can be one of the following:

- Constant number (including real)
- Net
- 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
- Array element
- 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 9 lists these operators.

Table 9—Operators in the Verilog HDL

{ } { }	Concatenation, replication
+ - * / **	Arithmetic
%	Modulus
> >= < <=	Relational
!	Logical negation
&&	Logical and
	Logical or
==	Logical equality
!=	Logical inequality

Table 9—Operators in the Verilog HDL (continued)

===	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
<<	Logical left shift
>>	Logical right shift
<<<	Arithmetic left shift
>>>	Arithmetic right shift
?:	Conditional
or	Event or

4.1.1 Operators with real operands

The operators shown in Table 10 shall be legal when applied to real operands. All other operators shall be considered illegal when used with real operands.

Table 10—Legal operators for use in real expressions

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

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

Table 11 lists operators that shall not be used to operate on real numbers.

Table 11—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 12. The Verilog HDL has two equality operators. They are discussed in 4.1.8.

Table 12—Precedence rules for operators

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

Operators shown on the same row in Table 12 shall have the same precedence. Rows are arranged in order of decreasing precedence for the operators. For example, *, /, and % all have the same precedence, which is higher than that of the binary + and - operators.

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

$$A + B - C$$

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

$$A + B / C$$

Parentheses can be used to change the operator precedence.

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

4.1.3 Using integer numbers in expressions

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

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

A negative value for an integer with no base specifier shall be interpreted differently 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 an unsigned base specifier shall be interpreted as an unsigned value.

Example:

This example shows four ways to write the expression minus 12 divided by 3. Note that -12 and -'d12 both evaluate to the same 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.  
  
IntA = -'sd 12 / 3; // The result is -4.  
  
IntA = -4'sd 12 / 3; // -4'sd12 is the negative of the 4-bit  
// quantity 1100, which is -4. -(-4) = 4.  
// The result is 1.
```

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 13.

Table 13—Arithmetic operators defined

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

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

The result of the power operator shall be real if either operand is a real, integer, or signed. If both operands are unsigned then the result shall be unsigned. The result of the power operator is unspecified if the first operand is zero and the second operand is non-positive, or if the first operand is negative and the second operand is not an integral value.

The unary arithmetic operators shall take precedence over the binary operators. The unary operators are given in Table 14.

Table 14—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 15 gives examples of modulus operations.

Table 15—Examples of modulus operators

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
$-4 \text{ d}12 \% 3$	1	$-4 \text{ d}12$ is seen as a large, positive number that leaves a remainder of 1 when divided by 3

4.1.6 Arithmetic expressions with regs and integers

A reg data type shall be treated as an *unsigned* value unless explicitly declared to be signed. An integer variable shall be treated as signed. *Signed* values shall use a 2's complement representation. Conversions between signed and unsigned values shall keep the same bit representation; only the interpretation changes.

Table 16 lists how arithmetic operators interpret each data type.

Table 16—Data type interpretation by arithmetic operators

Data type	Interpretation
unsigned net	Unsigned
signed net	Signed, 2 s complement
unsigned reg	Unsigned
signed reg	Signed, 2 s complement
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;
reg signed [15:0] regS;

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

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

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

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

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

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

4.1.7 Relational operators

Table 17 lists and defines the relational operators.

Table 17—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 either operand of a relational operator contains an unknown (x) or high impedance (Z) value, then the result shall be a 1-bit unknown value (x).

When two operands of unequal bit lengths are used and one or both of the operands is unsigned, 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 .

When both operands of a relational expression are signed integral operands (an integer, a signed reg data type, or an unsigned, unbased integer) then the expression shall be interpreted as a comparison between signed values. When either operand of a relational expression is a real operand then the other operand shall be converted to an equivalent real value, and the expression shall be interpreted as a comparison between two real values.

Otherwise the expression shall be interpreted as a comparison between unsigned values.

4.1.8 Equality operators

The *equality operators* shall rank lower in precedence than the relational operators. Table 18 lists and defines the equality operators.

Table 18—Definitions of the equality operators

$a == b$	a equal to b, including x and z
$a != b$	a not equal to b, including x and z
$a === b$	a equal to b, result may be unknown
$a !== b$	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, due to unknown or high-impedance bits in the operands, the relation is ambiguous, then the result shall be a one bit unknown value (X).

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

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 non-zero or true operand into 0 and a zero or false operand into 1. An ambiguous truth value remains as X.

Examples:

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

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

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 19 through 23 show the results for each possible calculation.

Table 19—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 21—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 20—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 22—Bit-wise binary exclusive nor operator

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

Table 23—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 24 through 26. 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 24—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 25—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 26—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 27 shows the results of applying reduction operators on different operands.

Table 27—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

There are two types of *shift operators*, the logical shift operators, << and >>, and the arithmetic shift operators, <<< and >>>. The left shift operators, << and <<<, shall shift their left operand to the left by the number by the number of bit positions given by the right operand. In both cases, the vacated bit positions shall be filled with zeroes. The right shift operators, >> and >>>, shall shift their left operand to the right by the number of bit positions given by the right operand. The logical right shift shall fill the vacated bit positions with

zeroes. The arithmetic right shift shall fill the vacated bit positions with zeroes if the result type is unsigned. It shall fill the vacated bit positions with the value of the most-significant (i.e., *sign*) bit of the left operand if the result type is signed. If the right operand has an unknown or high impedance value, then the result shall be unknown. The right operand is always treated as an unsigned number and has no effect on the signedness of the result. The result signedness is determined by the left-hand operand and the remainder of the expression, as outlined in 4.5.1.

Examples:

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

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

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

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

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 ::= (From Annex A - A.8.3)
expression1 ? { attribute_instance } expression2 : expression3
expression1 ::=
    expression
expression2 ::=
    expression
expression3 ::=
    expression
```

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 28 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 28—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 three-state output bus illustrates a common use of the conditional operator.

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

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

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}
```

Another form of concatenation is the replication operation. The first expression shall be a non-zero, non-X and non-Z constant expression, the second expression follows the rules for concatenations. This example replicates "w" 4 times.

```
{4{w}} // This is equivalent to {w, w, w, w}
a[31:0] = {1'b1, {0{1'b0}} }; //illegal. RHS becomes {1'b1, ;
a[31:0] = {1'b1, {1'bz{1'b0}} }; //illegal. RHS becomes {1'b1, ;
a[31:0] = {1'b1, {1'bx{1'b0}} }; //illegal. RHS becomes {1'b1, ;
```

If the replication operator 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}
```

4.1.15 Event or

The event **or** operator shall perform an or of events. The **,** operator does the same thing. See 9.7 for events and triggering of events.

Example:

The following example shows both ways to make an assignment to *rega* when an event (change) occurs on *trig* or *enable*.

```
@(trig or enable) rega = regb ;  
@(trig , 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 variable in its complete form that is, just the name of the net or variable is given. In this case, all of the bits making up the net or variable value shall be used as the operand.

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

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 Vector bit-select and part-select addressing

Bit-selects extract a particular bit from a vector net, vector reg, integer variable, or time variable. 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. The bit-select or part-select of a variable declared as **real** or **realtime** shall be considered illegal.

Several contiguous bits in a vector net, vector reg, integer variable, or time variable can be addressed and are known as *part-selects*. There are two types of part-selects, a constant part-select and an indexed part-select.

A constant part-select of a vector reg or net is given with the following syntax:

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

Both 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.

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

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

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

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

The `width_expr` shall be a constant expression. It also shall not be affected by run-time parameter assignments. The `lsb_base_expr` and `msb_base_expr` can vary at run-time. The first two examples select bits starting at the base and ascending the bit range. The number of bits selected is equal to the width expression. The second two examples select bits starting at the base and descending the bit range. Part-selects that address a range of bits that are completely out of the address bounds of the net, reg, integer, or time, or when the part-select is X or Z, shall yield the value X when read, and shall have no effect on the data stored when written. Part-selects that are partially out of range shall when read return X for the bits that are out of range, and when written shall only affect the bits that are in range.

Examples:

```
reg [31:0] big_vect;
reg [0:31] little_vect;
reg [63:0] dword;
integer sel;
```

The first four if statements show the identity between the two part select constructs. The last one shows an indexable nature.

```
initial begin
  if (big_vect[0 +:8] == big_vect[7 : 0]) begin end
  if (little_vect[0 +:8] == little_vect[0 : 7]) begin end
  if (big_vect[15 -:8] == big_vect[15 : 8]) begin end
  if (little_vect[15 -:8] == little_vect[8 :15]) begin end
  if (sel >0 && sel < 8)
    dword[8*sel +:8] = big_vect[7:0]; // Replace the byte selected.
```

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 [2:17] acc
```

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

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

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

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 Array and memory addressing

Declaration of arrays and memories (one dimensional arrays of `reg`) are discussed in 3.10. This subclause discusses array 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`.

Examples:

The next example declares an array of 256 by 256 8-bit elements and an array 256 by 256 by 8 1-bit elements:

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

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

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

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

To express bit selects or part selects of array elements, the desired word shall first be selected by supplying an address for each dimension. Once selected, bit and part selects shall be addressed in the same manner as net and reg bit and part selects (see 4.2.1).

Examples:

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

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 regs, the regs should be at least $8 \cdot n$ bits (where n is the number of ASCII characters) in order to preserve the 8-bit ASCII code.

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 ($\backslash 0$, 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
```

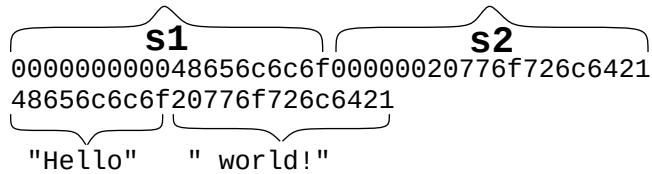
The comparison in this example fails because during the assignment the string variables are padded as illustrated in the next example:

```
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 and enclosed by parentheses. This is intended to represent minimum, typical, and maximum values in that order. The syntax is given in Syntax 4-2.

```

constant_expression ::= (From Annex A - A.8.3)
    constant_primary
    | unary_operator { attribute_instance } constant_primary
    | constant_expression binary_operator { attribute_instance } constant_expression
    | constant_expression ? { attribute_instance } constant_expression
    | constant_expression
    | string
constant_mintypmax_expression ::=
    constant_expression
    | constant_expression : constant_expression : constant_expression
expression ::=
    primary
    | unary_operator { attribute_instance } primary
    | expression binary_operator { attribute_instance } expression
    | conditional_expression
    | string
mintypmax_expression ::=
    expression
    | expression : expression : expression
constant_primary ::= (From Annex A - A.8.4)
    constant_concatenation
    | constant_function_call
    | ( constant_mintypmax_expression )
    | constant_multiple_concatenation
    | genvar_identifier
    | number
    | parameter_identifier
    | specparam_identifier
primary ::=
    number
    | hierarchical_identifier
    | hierarchical_identifier [ expression ] { [ expression ] }
    | hierarchical_identifier [ expression ] { [ expression ] } [ range_expression ]
    | hierarchical_identifier [ range_expression ]
    | concatenation
    | multiple_concatenation
    | function_call
    | system_function_call
    | constant_function_call
    | ( mintypmax_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 a typical expression that is 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 regs, then the result is a 16-bit value. However, in some situations it is not obvious how many bits are used to evaluate an expression, or what size the result should be.

For example, should an arithmetic add of two 16-bit values perform the evaluation using 16 bits, or should the evaluation use 17 bits in order to allow for a possible carry overflow? The answer depends on the type of device being modeled, and whether that device handles carry overflow. The Verilog HDL uses the bit length of the operands to determine how many bits to use while evaluating an expression. The bit length rules are given in 4.4.1. In the case of the addition operator, the bit length of the largest operand, including the left-hand side of an assignment, shall be used.

Examples:

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

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

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 29 shows how the form of an expression shall determine the bit lengths of the results of the expression. In Table 29, i , j , and k represent expressions of an operand, and $L(i)$ represents the bit length of the operand represented by i .

Table 29—Bit lengths resulting from self-determined expressions

Expression	Bit length	Comments
Unsize constant number*	Same as integer	
Sized constant number	As given	
i op j, where op is: + - * / % & ^ ^~ ^~^	max(L(i),L(j))	
op i, where op is: + - ~	L(i)	
i op j, where op is: == != == != && > >= < <=	1 bit	Operands are sized to max(L(i),L(j))
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

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

NOTE Multiplication without losing any overflow bits is still possible simply by assigning the result to something wide enough to hold it.

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 regs
```

The intent is to evaluate the expression

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

where a and b are to be added, which 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
```

In the following example:

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

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

the \$display statement will display:

```
answer = 01000
```

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

4.4.3 Example of self-determined expressions

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

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

Simulator output for this example:

```
a*b=16 // 96 was truncated since expression size
is 6
a**b=1 // expression size is 4 bits (size of a)
c=21 // example size is 16 bits (size of c)
```

4.5 Signed expressions

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

\$signed - returned value is signed

\$unsigned - returned value is unsigned

Example:

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

regA = $unsigned(-4); // regA = 4'b1100
regS = $signed(4'b1100); // regS = -4
```

4.5.1 Rules for expression types

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

- Expression type depends only on the operands. It does not depend on the LHS (if any).
- Decimal numbers are signed.
- Based_numbers are unsigned, except where the S notation is used in the base specifier (as in "4'sd12").
- Bit-select results are unsigned, regardless of the operands.
- Part-select results are unsigned, regardless of the operands.

NOTE—This is true even if the part-select specifies the entire vector.

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

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

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

4.5.2 Steps for evaluating an expression

- Determine the expression size based upon the standard rules of expression size determination.
- Determine the sign of the expression using the rules outlined in 4.5.1.
- Coerce the type of each operand of the expression (excepting those which are self-determined) to the type of the expression.

- Extend the size of each operand (excepting those which are self-determined) to the size of the expression. Perform sign extension if and only if the operand type (after type coercion) is signed.

4.5.3 Steps for evaluating an assignment

- Determine the size of the RHS by the standard assignment size determination rules (see 4.4)
- If needed, extend the size of the RHS, performing sign extension if and only if the type of the RHS is signed.

4.5.4 Handling X and Z in signed expressions

If a signed operand is to be resized to a larger signed width and the value of the sign bit is X, the resulting value shall be bit-filled with Xs. If the sign bit of the value is Z, then the resulting value shall be bit-filled with Zs. If any bit of a signed value is X or Z, then any non logical operation involving the value shall result in the entire resultant value being an X and the type consistent with the expression's type.

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 variable in the circuit being simulated, as well as the named event, is considered an *update event*.

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

In addition to events, another key aspect of a simulator is time. The term *simulation time* is used to refer to the time value maintained by the simulator to model the actual time it would take for the circuit being simulated. The term *time* is used interchangeably with simulation time in this 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 17.1) 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 25.58) or `vpi_register_cb(cb_readwrite)` (see 27.33) 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 (see 9.2.1) with a delay computes the right-hand side value using the current values, then causes the executing process to be suspended and scheduled as a future event. If the delay is 0, the process is scheduled as an inactive event for the current time.

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

5.6.4 Nonblocking assignment

A nonblocking assignment statement (see 9.2.2) always computes the updated value and schedules the update as a nonblocking assign update event, either in this time step if the delay is zero or as a future event if the delay is nonzero. The values in effect when the update is placed on the event queue are used to compute both the right-hand value and the left-hand target.

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

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.

6. Assignments

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

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

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

An assignment consists of two parts, a left-hand side and a right-hand side, separated by the equals (=) character; or, in the case of nonblocking procedural assignment, the less-than-equals (<=) character pair. The right-hand side can be any expression that evaluates to a value. The left-hand side indicates the variable to which the right-hand side value is to be assigned. The left-hand side can take one of the forms given in Table 30, depending on whether the assignment is a continuous assignment or a procedural assignment.

Table 30—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 Constant indexed part select of a vector net Concatenation of any of the above four LHS
Procedural assignment	Variables (vector or scalar) Bit-select of a vector reg, integer, or time variable Constant part select of a vector reg, integer, or time variable Memory word Indexed part select of a vector reg, integer, or time variable Concatenation of regs; bit or part selects of regs

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

```

Syntax 6-1 Syntax for continuous assignment

6.1.1 The net declaration assignment

The first two alternatives in the net declaration are discussed in 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 data type. The net may be explicitly declared, or may inherit an implicit declaration in accordance with the implicit declarations rules defined in 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 if the new value is different from the previous value, then the new value shall be 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:

- The value of **s**, a bus selector input variable, is checked in the assign statement. Based on the value of **s**, the net **data** receives the data from one of the four input buses.
- 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 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 following steps are taken:

- a) The value of the right-hand side expression is evaluated.
- b) If this RHS value differs from the value currently scheduled to propagate to the left-hand side, then the currently scheduled propagation event is descheduled.
- c) If the new RHS value equals the current left-hand side value, no event is scheduled.
- d) If the new RHS value differs from the current LHS value, a delay is calculated in the standard way using the current value of the left-hand side, the newly calculated value of the right-hand side, and the delays indicated on the statement; a new propagation event is then scheduled to occur delay time units in the future.

6.1.4 Strength

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

wire	tri	trireg
wand	triand	tri0
wor	trior	tri1

Continuous assignments driving strengths can be specified 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, procedural assignments put values in variables. The assignment does not have duration; instead, the variable holds the value of the assignment until the next procedural assignment to that variable.

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

6.2.1 Variable declaration assignment

The variable declaration assignment is a special case of procedural assignment as it assigns a value to a variable. It allows an initial value to be placed in a variable in the same statement that declares the variable. The assignment shall be to a constant expression. The assignment does not have duration; instead, the variable holds the value until the next assignment to that variable. Variable declaration assignments to an array are not allowed. Variable declaration assignments are only allowed at the module level.

Examples:

Example 1 Declare a 4 bit reg and assign it the value 4.

```
reg[3:0] a = 4'h4;
```

This is equivalent to writing:

```
reg[3:0] a;  
initial a = 4'h4;
```

Example 2 The following example is not legal.

```
reg [3:0] array [3:0] = 0;
```

Example 3 Declare two integers, the first is assigned the value of 0.

```
integer i = 0, j;
```

Example 4 Declare two real variables, assigned to the values 2.5 and 300,000.

```
real r1 = 2.5, n300k = 3E6;
```

Example 5 Declare a time variable and realtime variable with initial values.

```
time t1 = 25;  
realtime rt1 = 2.5;
```

NOTE—If the same variable is assigned different values both in an initial block and in a variable declaration assignment, the order of the evaluation is undefined.

6.2.2 Variable declaration syntax

The syntax for variable declaration assignments is given in Syntax 6-2.

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

Syntax 6-2 Syntax for reg declaration assignment

7. Gate and switch level modeling

This clause 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 ::= (From Annex A - A.3.1)
    cmos_switchtype [delay3] cmos_switch_instance { , cmos_switch_instance } ;
    enable_gatetype [drive_strength] [delay3] enable_gate_instance { , enable_gate_instance } ;
    mos_switchtype [delay3] mos_switch_instance { , mos_switch_instance } ;
    n_input_gatetype [drive_strength] [delay2] n_input_gate_instance { , n_input_gate_instance } ;
    n_output_gatetype [drive_strength] [delay2] n_output_gate_instance
        { , n_output_gate_instance } ;
    pass_en_switchtype [delay2] pass_enable_switch_instance { , pass_enable_switch_instance } ;
    pass_switchtype pass_switch_instance { , pass_switch_instance } ;
    pulldown [pulldown_strength] pull_gate_instance { , pull_gate_instance } ;
    pullup [pullup_strength] pull_gate_instance { , pull_gate_instance } ;

cmos_switch_instance ::= [ name_of_gate_instance ]
    ( output_terminal , input_terminal , ncontrol_terminal , pcontrol_terminal )
enable_gate_instance ::= [ name_of_gate_instance ]
    ( output_terminal , input_terminal , enable_terminal )
mos_switch_instance ::= [ name_of_gate_instance ]
    ( output_terminal , input_terminal , enable_terminal )
n_input_gate_instance ::= [ name_of_gate_instance ]
    ( output_terminal , input_terminal { , input_terminal } )
n_output_gate_instance ::= [ name_of_gate_instance ]
    ( output_terminal { , output_terminal } , input_terminal )
pass_switch_instance ::= [ name_of_gate_instance ] ( inout_terminal , inout_terminal )
pass_enable_switch_instance ::= [ name_of_gate_instance ]
    ( inout_terminal , inout_terminal , enable_terminal )
pull_gate_instance ::= [ name_of_gate_instance ] ( output_terminal )
name_of_gate_instance ::= gate_instance_identifier [ range ]
pulldown_strength ::= (From Annex A - A.3.2)
    ( strength0 , strength1 )
    | ( strength1 , strength0 )
    | ( strength0 )
pullup_strength ::= ( strength0 , strength1 )
    | ( strength1 , strength0 )
    | ( strength1 )
enable_terminal ::= (From Annex A - A.3.3)
    expression
inout_terminal ::= net_lvalue
input_terminal ::= expression
ncontrol_terminal ::= expression
output_terminal ::= net_lvalue
pcontrol_terminal ::= expression
cmos_switchtype ::= (From Annex A - A.3.4)
    cmos | rcmos
enable_gatetype ::= bufif0 | bufif1 | notif0 | notif1
mos_switchtype ::= nmos | pmos | rnmos | rpmos
n_input_gatetype ::= and | nand | or | nor | xor | xnor
n_output_gatetype ::= buf | not
pass_en_switchtype ::= tranif0 | tranif1 | rtranif1 | rtranif0
pass_switchtype ::= tran | rtran

```

Syntax 7-1 Syntax for gate instantiation

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 31 lists the keywords that shall begin a gate or a switch instance declaration.

Table 31 — Built-in gates and switches

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

Explanations of the built-in gates and switches shown in Table 31 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 32 can have the drive strength specification.

Table 32 — 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 7.9 through 7.13.

7.1.3 The delay specification

An optional delay specification shall specify the propagation delay through the gates and switches in a declaration. Gates and switches in declarations with no delay specification shall have no propagation delay. A delay specification can contain up to three delay values, depending on the gate type. The **pullup** and **pull-down** instance declarations shall not include delay specifications. Delays are discussed in more detail in 7.14.

7.1.4 The primitive instance identifier

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

7.1.5 The range specification

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

In order to specify an array of instances, the instance name shall be followed by the range specification. The range shall be specified by two constant expressions, left-hand index (**lhi**) and right-hand index (**rhi**), separated by a colon and enclosed within a pair of square brackets. A [**lhi**:**rhi**] range specification shall represent an array of $\text{abs}(\text{lhi} - \text{rhi}) + 1$ 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 regs.

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 three-state buffers

endmodule

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

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

endmodule
```

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

```

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

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

endmodule

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

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

endmodule

```

Example 4 This example demonstrates how a series of modules can be chained together. Figure 4 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

```

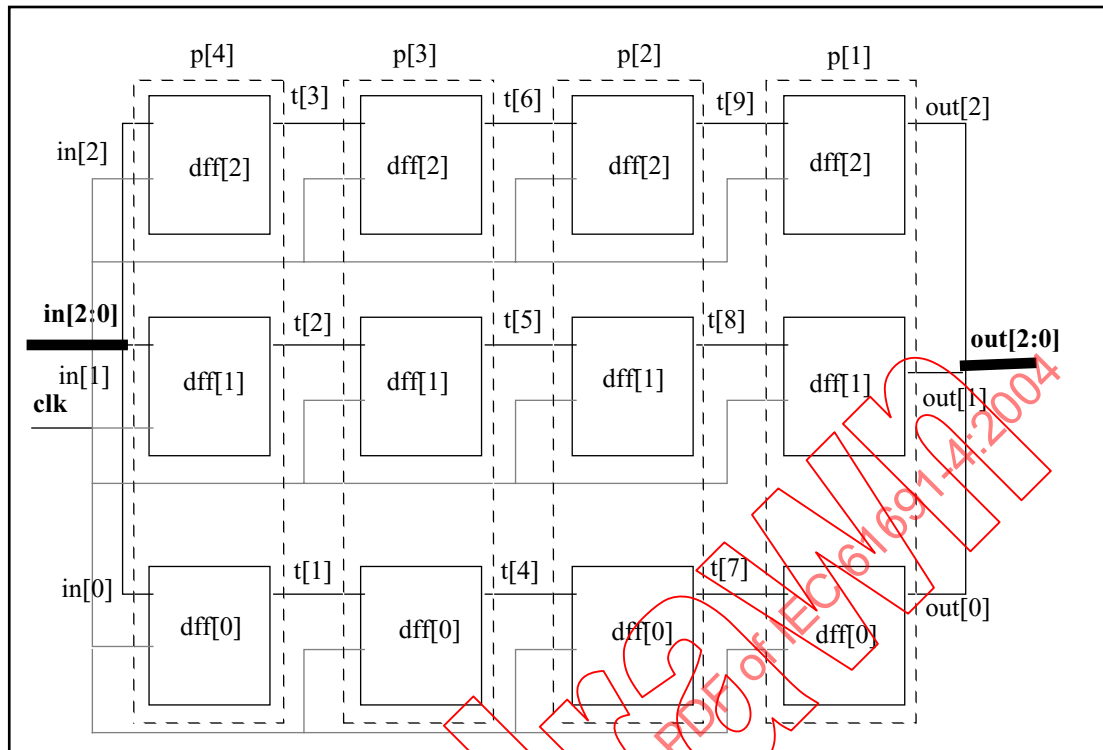


Figure 4—Schematic diagram of interconnections in array of instances

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 33.

Table 33— 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 34.

Table 34—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 these three-state logic gates shall begin with one of the following keywords:

bufif0 **bufif1** **notif1** **notif0**

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

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

Some combinations of data input values and control input values can cause these gates to output either of two values, without a preference for either value (see 7.10.2). These logic tables for these gates include two symbols representing such unknown results. The symbol L shall represent a result that has a value 0 or Z. The symbol H shall represent a result that has a value 1 or Z. Delays on transitions to H or L shall be treated the same as delays on transitions to X.

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

Table 35 presents the logic tables for these gates.

Table 35— Truth tables for three-state logic gates

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

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

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

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

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 7.11. The **rnmos** and **rpmos** switches shall reduce the strength of signals that propagate through them, as discussed in 7.12.

Table 36 presents the logic tables for these switches.

Table 36— Truth tables for MOS switches

	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

	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 7.11. The **rtran**, **rtranif0**, and **rtranif1** devices shall reduce the strength of the signals passing through them according to rules discussed in 7.12.

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 7.11. The **rcmos** gate shall reduce the strength of signals passing through it according to rules described in 7.12.

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

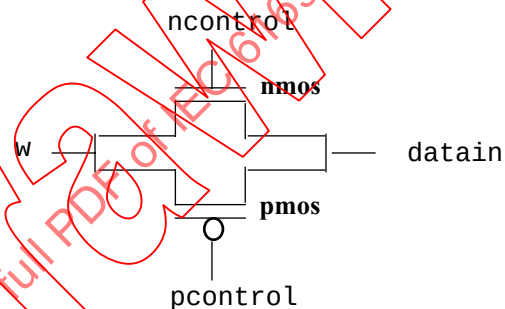
Example:

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

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

is equivalent to:

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



7.8 pullup and pulldown sources

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

pullup **pulldown**

A **pullup** source shall place a logic value 1 on the nets connected in its terminal list. A **pulldown** source shall place a logic value 0 on the nets connected in its terminal list. The signals that these sources place on nets shall have **pull** strength in the absence of a strength specification. If 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 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 37 demonstrates the continuum of strengths. The left column lists the keywords used in specifying strengths. The right column gives correlated strength levels.

Table 37—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 37, there are four *driving strengths*:

supply strong pull weak

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

In Table 37, 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 5.

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 5—Scale of strengths

Discussions of signal combinations later in this section employs graphics similar to those used in Figure 5.

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 5, or all strength levels shall be in its strength1 part. If the signal value of a net is unknown, it shall have strength levels in both the strength0 and the strength1 parts. A net with a signal value Z shall have a strength level only in one of the 0 subdivisions of the parts of the scale.

7.10 Strengths and values of combined signals

In addition to a signal value, a net shall have either a single unambiguous strength level or an ambiguous strength consisting of more than one level. When signals combine, their strengths and values shall determine the strength and value of the resulting signal in accordance with the principles in 7.10.1 through 7.10.4.

7.10.1 Combined signals of unambiguous strength

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

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

The combination of signals with unlike values and the same strength can have three possible results. Two of the results occur in the presence of wired logic and the third occurs in its absence. Wired logic is discussed in 7.10.4. The result in the absence of wired logic is the subject of Figure 7.

Example:

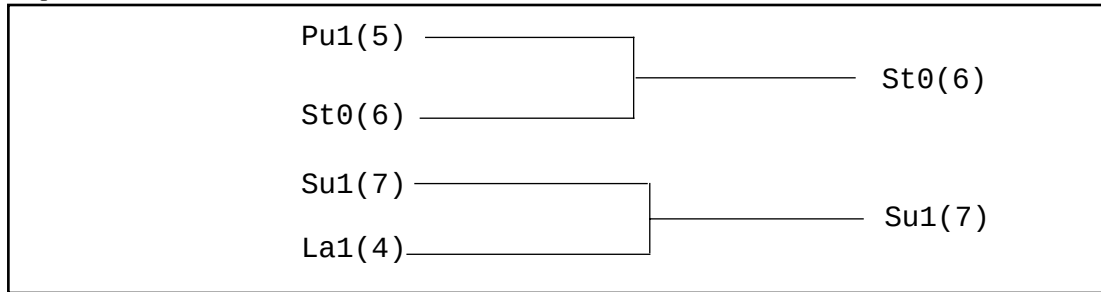


Figure 6—Combining unequal strengths

In Figure 6, 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.

7.10.2 Ambiguous strengths: sources and combinations

There are several classifications of signals possessing ambiguous strengths

- Signals with known values and multiple strength levels
- Signals with a value X, which have strength levels consisting of subdivisions of both the strength1 and the strength0 parts of the scale of strengths in Figure 5
- 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 5
- 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 5

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 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.

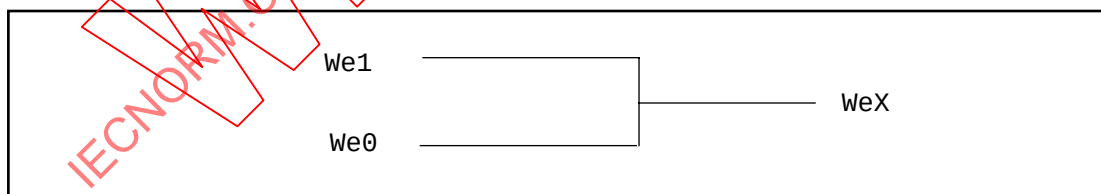


Figure 7—Combination of signals of equal strength and opposite values

This output signal is described in Figure 8.

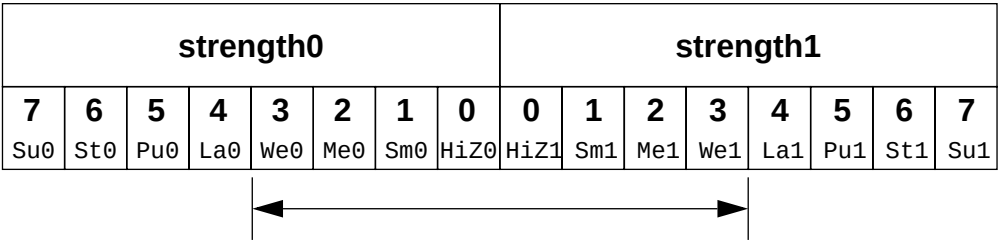


Figure 8—Weak x signal strength

An ambiguous signal strength can be a range of possible values. An example is the strength of the output from the three-state drivers with unknown control inputs as shown in Figure 9.

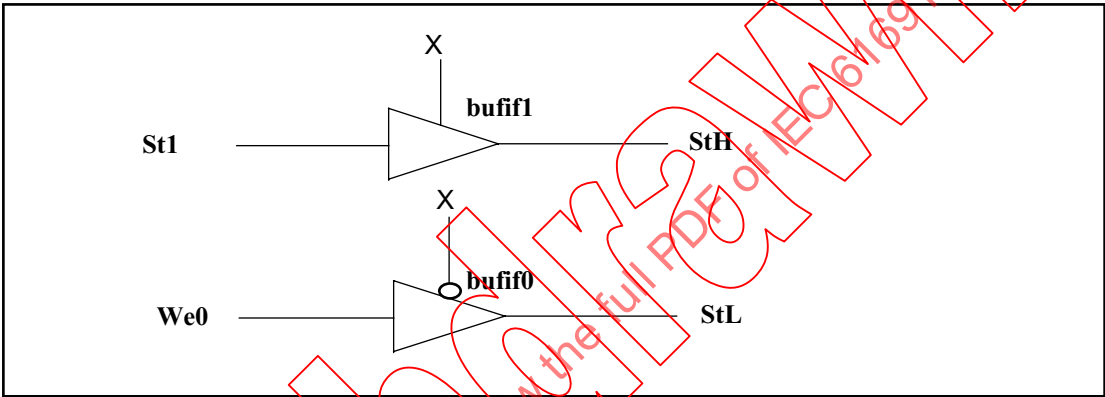


Figure 9—Bufifs with control inputs of x

The output of the **bufif1** in Figure 9 is a **strong H**, composed of the range of values described in Figure 10.

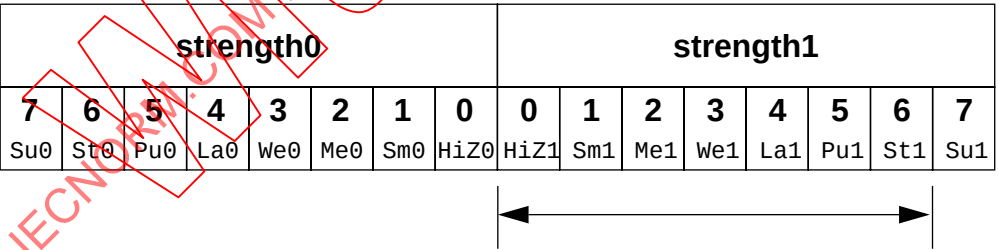


Figure 10—Strong H range of values

The output of the **bufif0** in Figure 9 is a **strong L**, composed of the range of values described in Figure 11.

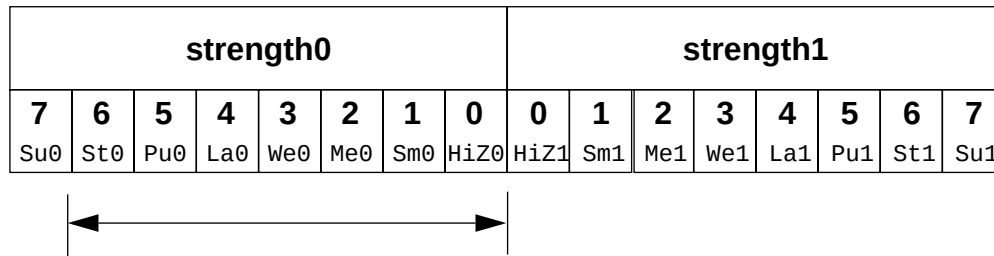


Figure 11—Strong L range of values

The combination of two signals of ambiguous strength shall result in a signal of ambiguous strength. The resulting signal shall have a range of strength levels that includes the strength levels in its component signals. The combination of outputs from two three-state drivers with unknown control inputs, shown in Figure 12, is an example.

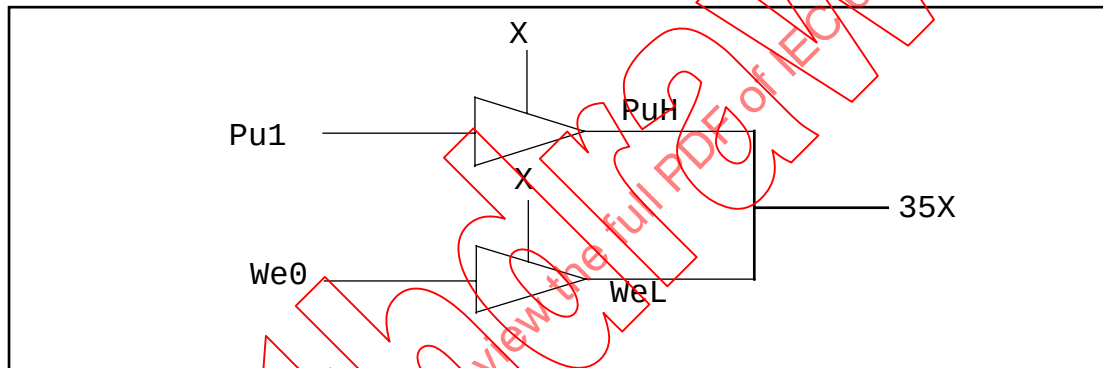


Figure 12—Combined signals of ambiguous strength

In Figure 12, 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 13.

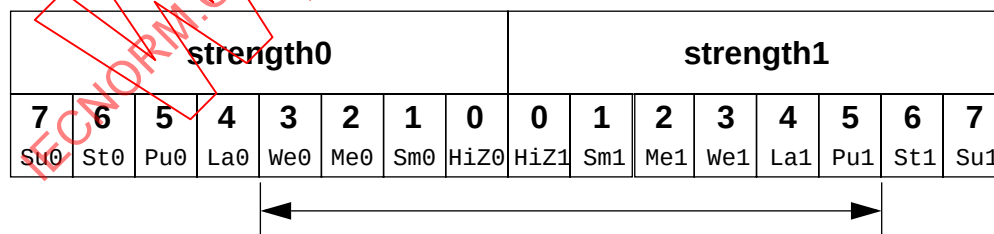


Figure 13—Range of strengths for an unknown signal

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 14.

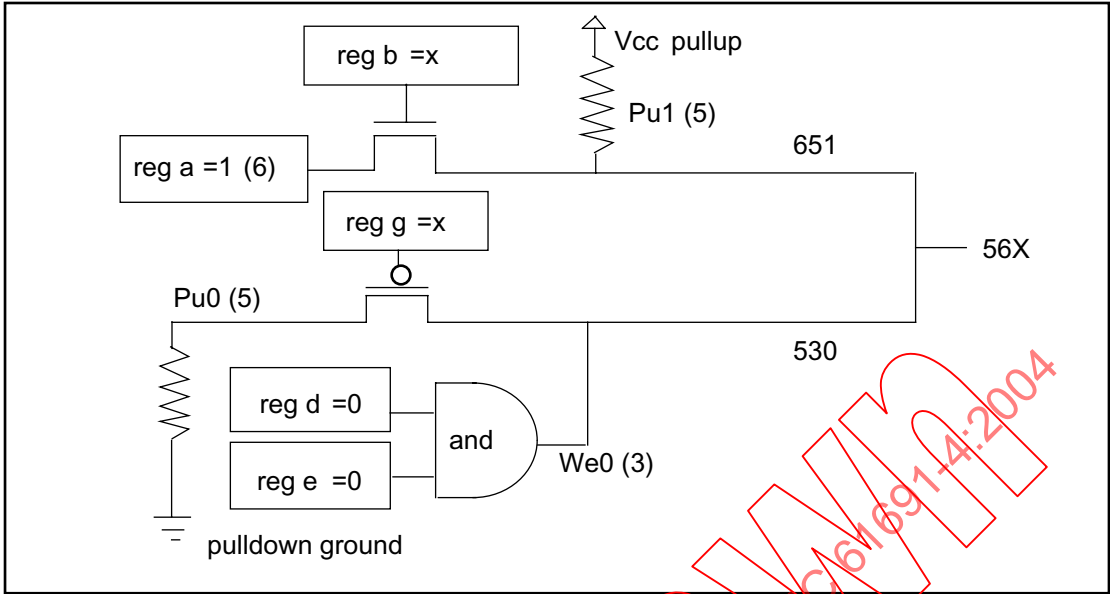


Figure 14—Ambiguous strengths from switch networks

In Figure 14, the upper combination of a reg, a gate controlled by a reg of unspecified value, and a pullup produces a signal with a value of 1 and a range of strengths (651) described in Figure 15.

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 15—Range of two strengths of a defined value

In Figure 14, the lower combination of a pulldown, a gate controlled by a reg of unspecified value, and an and gate produces a signal with a value 0 and a range of strengths (530) described in Figure 16.

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 16—Range of three strengths of a defined value

When the signals from the upper and lower configurations in Figure 14 combine, the result is an unknown with a range (56x) determined by the extremes of the two signals shown in Figure 17.

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 17—Unknown value with a range of strengths

In Figure 14, replacing the **pulldown** in the lower configuration with a **supply0** would change the range of the result to the range (StX) described in Figure 18.

The range in Figure 18 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 **supply** signal. This modeling feature is discussed in 7.11.

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 18—Strong X range

Logic gates produce results with ambiguous strengths as well as three-state drivers. Such a case appears in Figure 19. The **and** gate N1 is declared with **highz0** strength, and N2 is declared with **weak0** strength.

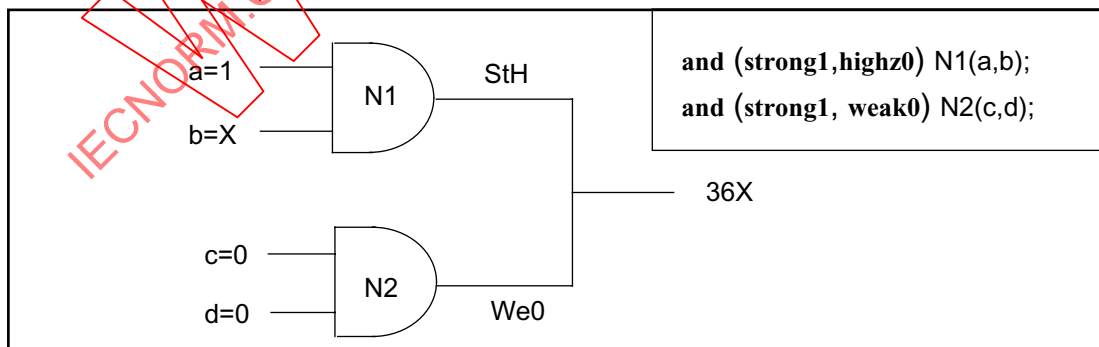


Figure 19—Ambiguous strength from gates

In Figure 19, reg **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 20.

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

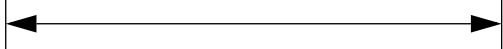


Figure 20—Ambiguous strength signal from a gate

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 21.

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 21—Weak 0

When the signals combine, the result is the range (36x) as described in Figure 22.

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

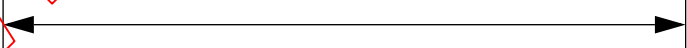


Figure 22—Ambiguous strength in combined gate signals

Figure 22 presents the combination of an ambiguous signal and an unambiguous signal. Such combinations are the topic of 7.10.3.

7.10.3 Ambiguous strength signals and unambiguous signals

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

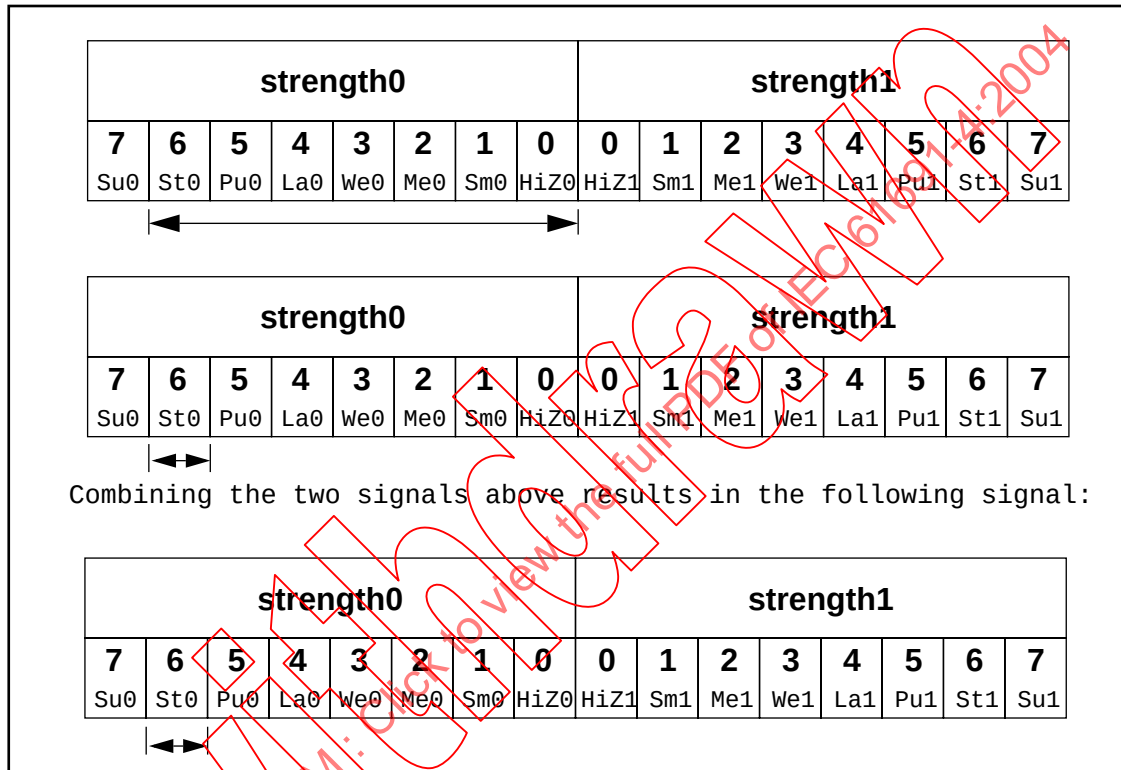


Figure 23— Elimination of strength levels

In Figure 23, 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.

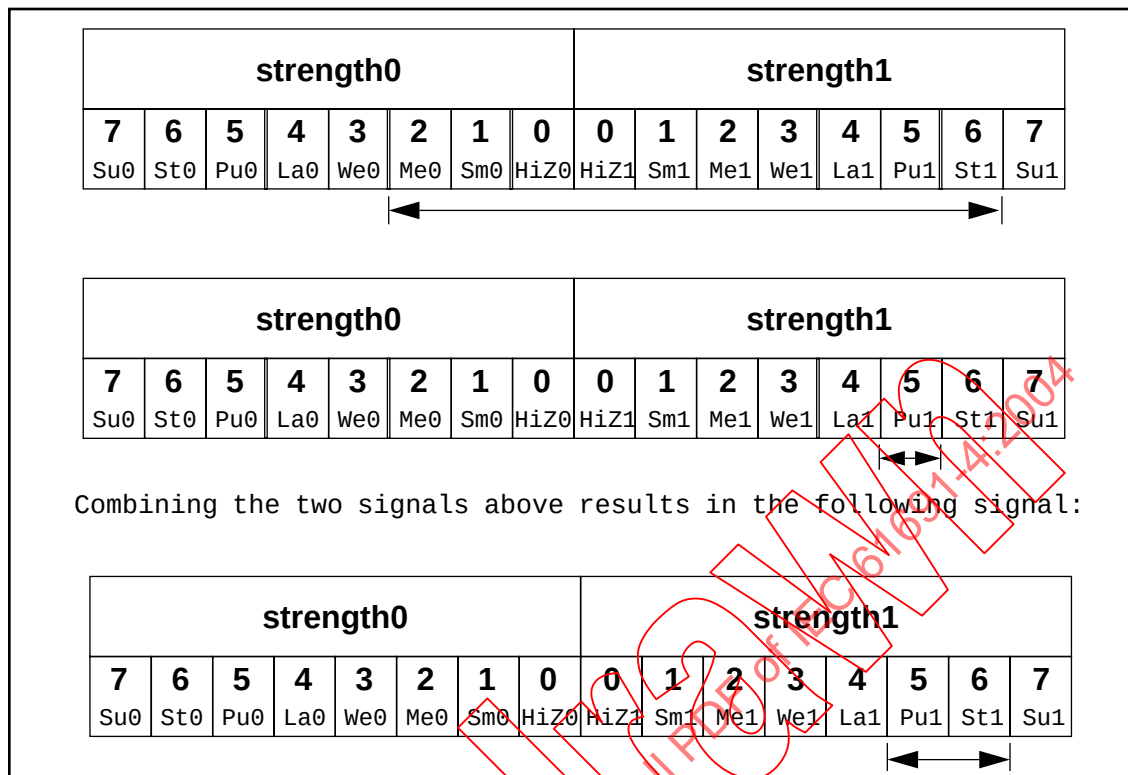


Figure 24—Result demonstrating a range and the elimination of strength levels of two values

In Figure 24, 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.

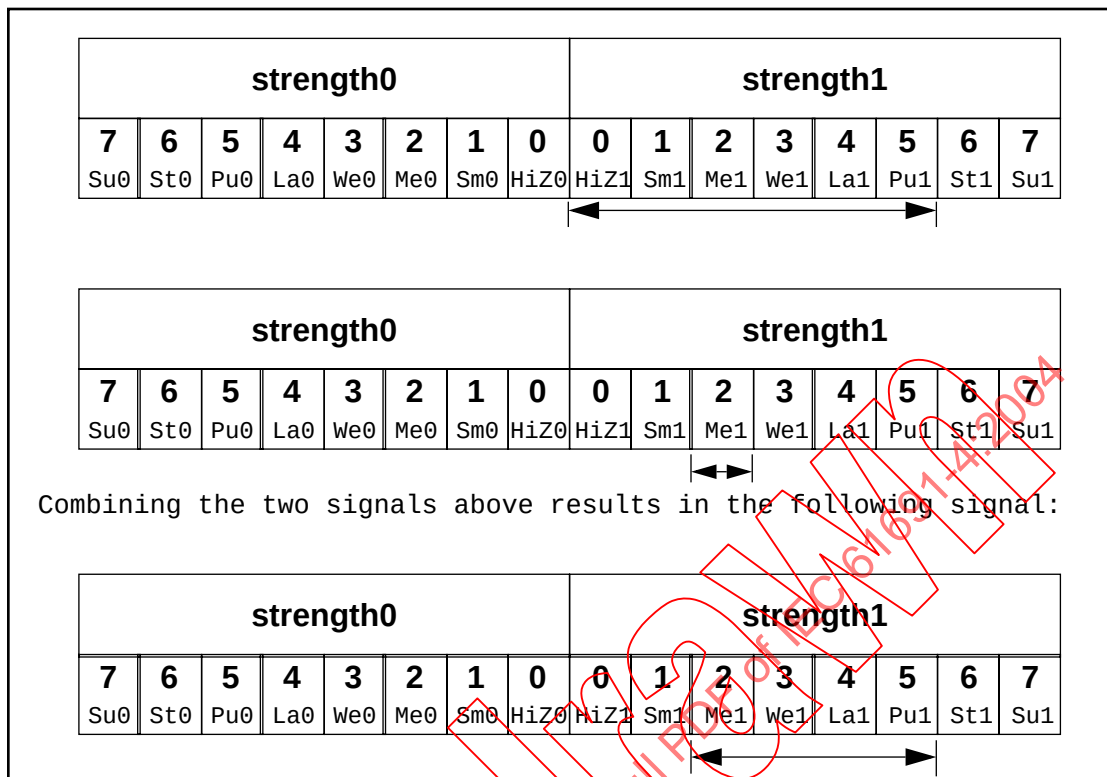


Figure 25—Result demonstrating a range and the elimination of strength levels of one value

In Figure 25, 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.

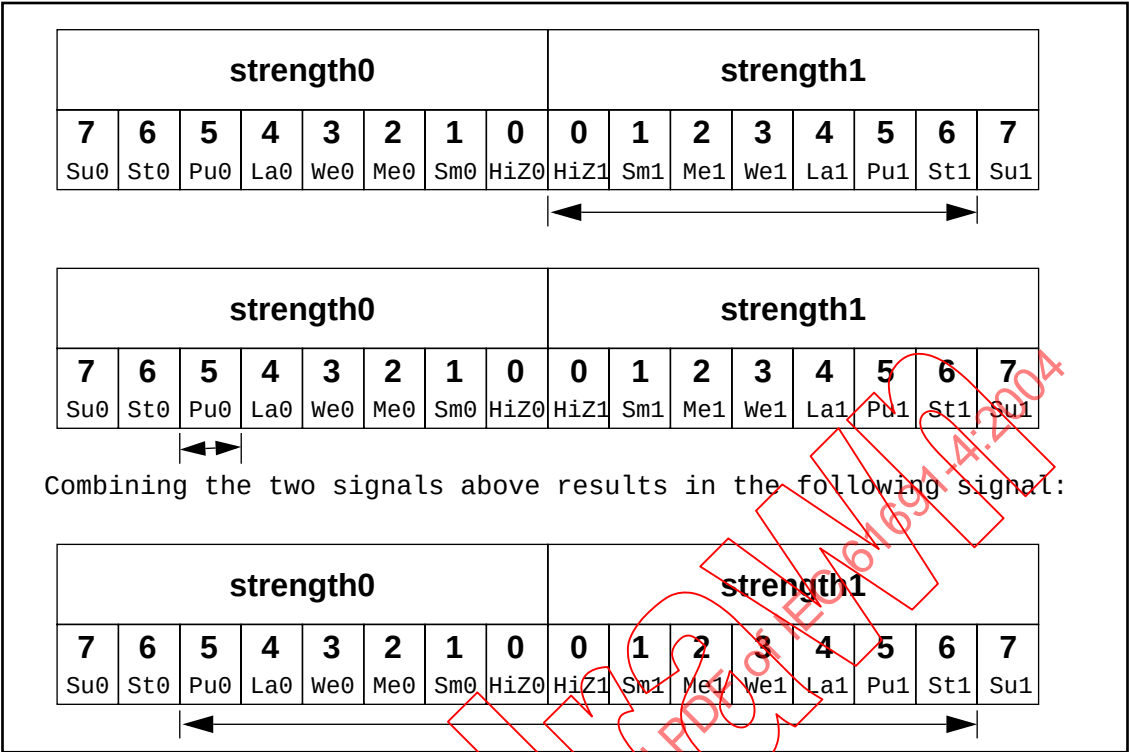


Figure 26—A range of both values

In Figure 26, 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.

7.10.4 Wired logic net types

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

Examples:

Consider the combination of two signals of unambiguous strength in Figure 27.

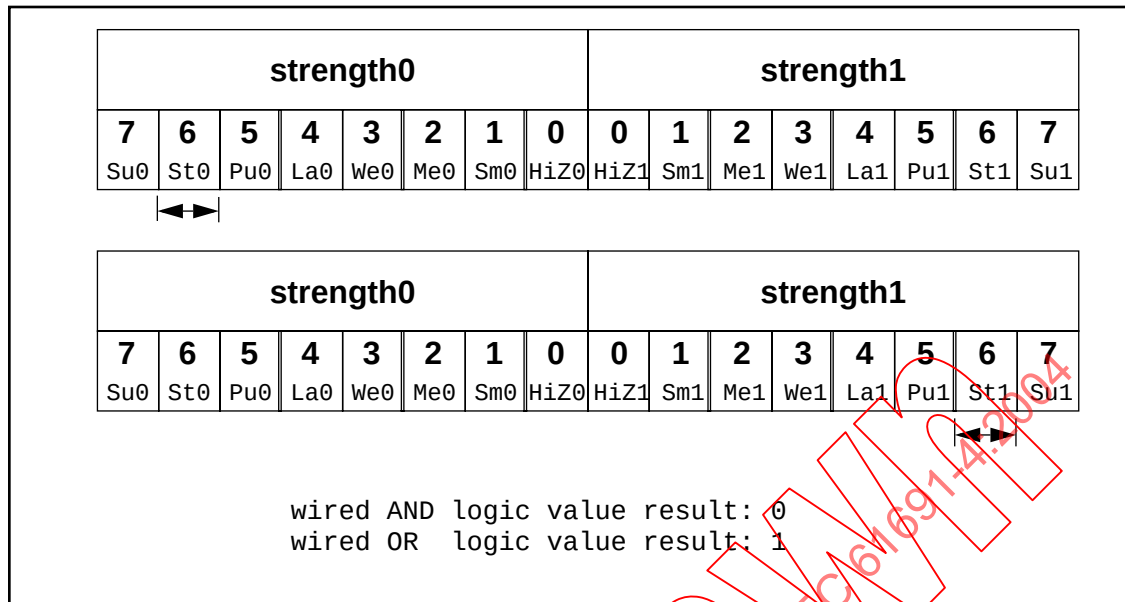


Figure 27—Wired logic with unambiguous strength signals

The combination of the signals in Figure 27, 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 27 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 28.

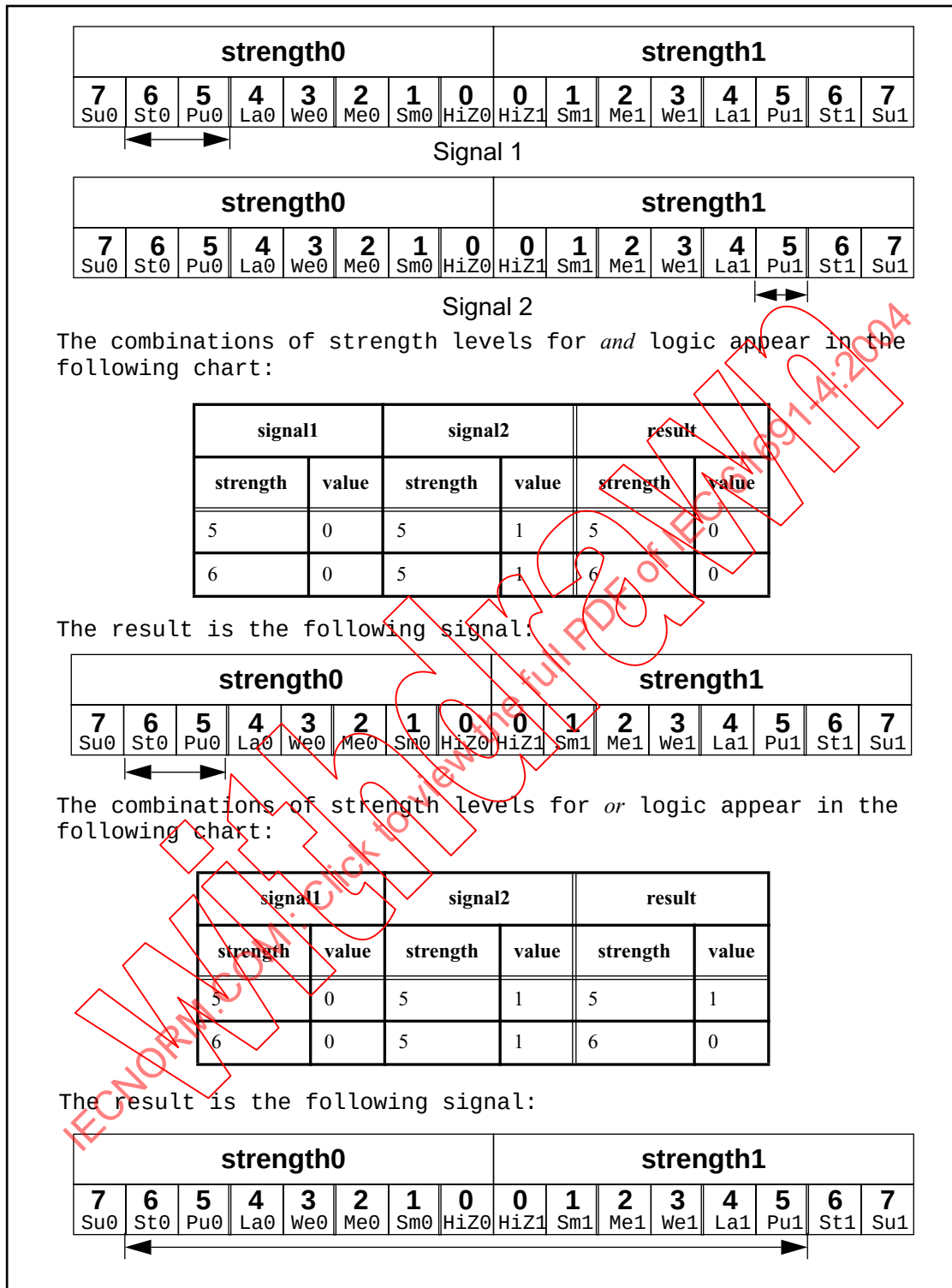


Figure 28—Wired logic and ambiguous strengths

7.11 Strength reduction by nonresistive devices

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

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

7.12 Strength reduction by resistive devices

The **rnmos**, **rpmos**, **rcmos**, **rtran**, **rtranif1**, and **rtranif0** devices shall reduce the strength of signals that pass through them according to Table 38.

Table 38—Strength reduction rules

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

7.13 Strengths of net types

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

7.13.1 tri0 and tri1 net strengths

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

7.13.2 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.

7.13.3 supply0 and supply1 net strengths

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

7.14 Gate and net delays

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

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

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

For a three-delay specification

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

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

Table 39 summarizes the from-to propagation delay choice for the two- and three-delay specifications.

Table 39—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 three-state outputs, where individual delays are given to the gates. The propagation delay from the primary inputs to the outputs of the module will be cumulative, and it depends on the signal path through the network.

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

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

endmodule
```

7.14.1 min:typ:max delays

The syntax for delays on gate primitives (including user-defined primitives; see Clause 8), nets, and continuous assignments shall allow three values each for the rising, falling, and turn-off delays. The minimum, typical, and maximum values for each delay shall be specified as 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
```

7.14.2 trireg net charge decay

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

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

A delay specification for charge decay models a charge storage node that is not ideal 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.

7.14.2.1 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:

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

7.14.2.2 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 **triereg** net named **cap1**. This **triereg** 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 29 shows an equivalent schematic for the source description.

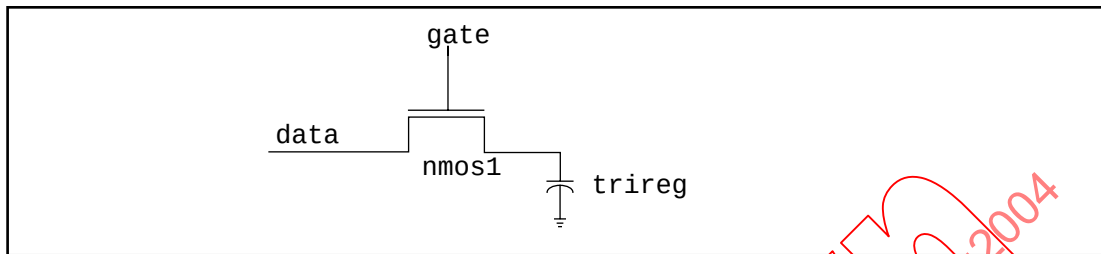


Figure 29—Triereg net with capacitance

```

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

```

8. User-defined primitives (UDPs)

This clause describes a modeling technique to augment the set of predefined gate primitives by designing and specifying new primitive elements called 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 three-state value z is not supported. In sequential UDPs, the output always has the same value as the internal state.

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

8.1 UDP definition

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

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 ::= (From Annex A - A.5.1)
    { attribute_instance } primitive udp_identifier ( udp_port_list ) ;
    udp_port_declaration { udp_port_declaration }
    udp_body
    endprimitive
    | { attribute_instance } primitive udp_identifier ( udp_declaration_port_list ) ;
    udp_body
    endprimitive

udp_port_list ::= (From Annex A - A.5.2)
    output_port_identifier , input_port_identifier { , input_port_identifier }

udp_declaration_port_list ::=
    udp_output_declaration , udp_input_declaration { , udp_input_declaration }

udp_port_declaration ::=
    udp_output_declaration ;
    | udp_input_declaration ;
    | udp_reg_declaration ;

udp_output_declaration ::=
    { attribute_instance } output port_identifier
    | { attribute_instance } output reg port_identifier [ = constant_expression ]

udp_input_declaration ::=
    { attribute_instance } input list_of_port_identifiers

udp_reg_declaration ::=
    { attribute_instance } reg variable_identifier

udp_body ::= (From Annex A - A.5.3)
    combinational_body | sequential_body

combinational_body ::=
    table combinational_entry { combinational_entry } endtable

combinational_entry ::=
    level_input_list : output_symbol ;

sequential_body ::=
    [ udp_initial_statement ] table sequential_entry { sequential_entry } endtable

udp_initial_statement ::=
    initial output_port_identifier = init_val ;

init_val ::= 1'b0 | 1'b1 | 1'bx | 1'bX | 1'B0 | 1'B1 | 1'Bx | 1'BX | 1 | 0

sequential_entry ::=
    seq_input_list : current_state : next_state ;

seq_input_list ::=
    level_input_list | edge_input_list

level_input_list ::=
    level_symbol { level_symbol }

edge_input_list ::=
    { level_symbol } edge_indicator { level_symbol }

edge_indicator ::=
    ( level_symbol level_symbol ) | edge_symbol

current_state ::= level_symbol

next_state ::= output_symbol | -

output_symbol ::= 0 | 1 | x | X

level_symbol ::= 0 | 1 | x | X | ? | b | B

edge_symbol ::= r | R | f | F | p | P | n | N | *

```

Syntax 8-1 Syntax for user-defined primitives

8.1.1 UDP header

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

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

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

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

8.1.2 UDP port declarations

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

Sequential UDPs shall contain a **reg** declaration for the output port, either in addition to the output declaration, when the UDP is declared using the first form of a UDP Header, or as part of the output declaration, in either case. 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 40), 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 40.

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 40 summarizes the meaning of all the value symbols that are valid in the table part of a UDP definition.

Table 40—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.
?	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      x      1 : 1 ;
  1      0      0 : 0 ;
  1      1      0 : 0 ;
  1      x      0 : 0 ;
  x      0      0 : 0 ;
  x      1      1 : 1 ;
endtable
endprimitive
```

The first entry in this example can be explained as follows: when control equals 0, 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 41.

Table 41—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 30 shows the schematic of the preceding module and the simulation propagation times of the initial value of the UDP output.

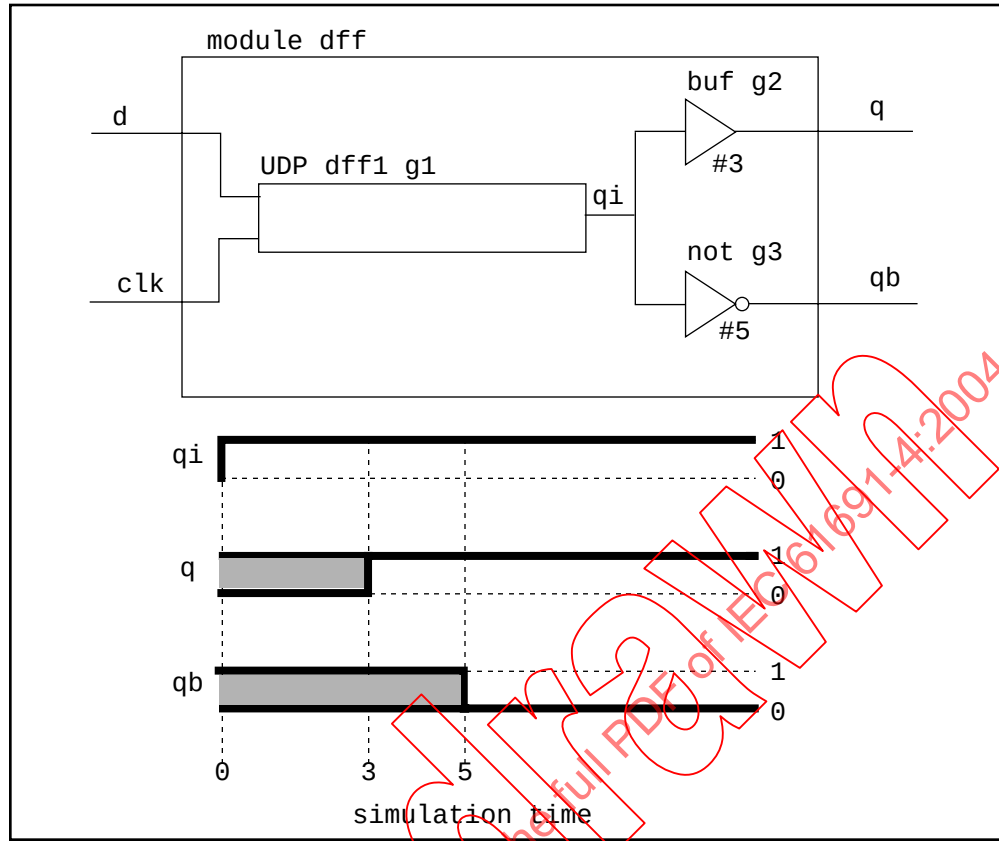


Figure 30—Module schematic and simulation times of initial value propagation

In Figure 30, 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 ::= (From Annex A- A.5.4)
    udp_identifier [ drive_strength ] [ delay2 ]
    [attribute_instance] udp_instance { , udp_instance } ;
udp_instance ::=
    [ name_of_udp_instance ] ( output_terminal , input_terminal
    { , input_terminal } )
name_of_udp_instance ::=
    udp_instance_identifier [ range ]

```

Syntax 8-2 Syntax for UDP instances

Instances of 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 40) 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 42 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 42—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.

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

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

9.2 Procedural assignments

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

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

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

reg, **integer**, **real**, **realtime**, or **time** data type: an assignment to the name reference of one of these data types.

Bit-select of a **reg**, **integer**, or **time** data type: an assignment to a single bit that leaves the other bits untouched.

Part-select of a **reg**, **integer**, or **time** data type: a part-select of one or more contiguous bits that leaves the rest of the bits untouched.

Memory word: a single word of a memory.

Concatenation 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 When the right-hand side evaluates to fewer bits than the left-hand side, then if the right-hand side is signed (see 4.5), it shall be sign-extended to the size of the left-hand side.

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 ::= (From Annex A - A.6.2)
    variable_lvalue = [ delay_or_event_control ] expression
delay_control ::= (From Annex A - A.6.5)
    # delay_value
    | # ( mintypmax_expression )
delay_or_event_control ::=
    delay_control
    | event_control
    | repeat ( expression ) event_control
event_control ::=
    @ event_identifier
    | @ ( event_expression )
    | @*
    | @ ( * )
event_expression ::=
    expression
    | hierarchical_identifier
    | posedge expression
    | negedge expression
    | event_expression or event_expression
    | event_expression , event_expression
variable_lvalue ::= (From Annex A - A.8.5)
    hierarchical_variable_identifier
    | hierarchical_variable_identifier [ expression ] { [ expression ] }
    | hierarchical_variable_identifier [ expression ] { [ expression ] }
      [ range_expression ]
    | hierarchical_variable_identifier [ range_expression ]
    | variable_concatenation

```

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 variable assignments within the same time step can be made without regard to order or dependence upon each other.

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

```

nonblocking_assignment ::= (From Annex A - A.6.2)
    variable_lvalue <= [ delay_or_event_control ] expression
delay_control ::= (From Annex A - A.6.5)
    # delay_value
    | # ( mintypmax_expression )
delay_or_event_control ::=
    delay_control
    | event_control
    | repeat ( expression ) event_control
event_control ::=
    @ event_identifier
    | @ ( event_expression )
    | @*
    | @ ( * )
event_expression ::=
    expression
    | hierarchical_identifier
    | posedge expression
    | negedge expression
    | event_expression or event_expression
    | event_expression , event_expression
variable_lvalue ::= (From Annex A - A.8.5)
    hierarchical_variable_identifier
    | hierarchical_variable_identifier [ expression ] { [ expression ] }
    | hierarchical_variable_identifier [ expression ] { [ expression ] }
    | [ range_expression ]
    | hierarchical_variable_identifier [ range_expression ]
    | variable_concatenation

```

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 Clause 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>	<i>Step 1:</i> At posedge c, the simulator evaluates the right-hand sides of the nonblocking assignments and schedules the assignments of the new values at the end of the <i>nonblocking assign update</i> events (see 5.4). <i>Step 2:</i> When the simulator activates the <i>nonblocking assign update</i> events, the simulator updates the left-hand side of each nonblocking assignment statement.	Nonblocking assignment schedules changes at time 5 a = 0 b = 1 Assignment values are: a = 1 b = 0

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

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

Example 2:

<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>	<i>scheduled changes at time 2</i> e = 0 <i>scheduled changes at time 4</i> f = 1 <i>scheduled changes at time 10</i> d = 1

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

Example 3:

```
//non_block1.v
module non_block1;
reg a, b;
initial begin
    a = 0;
    b = 1;
    a <= b; // evaluates, schedules, and
    b <= a; // executes in two steps
end
initial begin
    $monitor ($time, , "a = %b b = %b", a, b);
    #100 $finish;
end
endmodule
```

Step 1: The simulator evaluates the right-hand side of the nonblocking assignments and schedules the assignments for the end of the current time step.

Step 2:

At the end of the current time step, the simulator updates the left-hand side of each nonblocking assignment statement.

assignment values are:

a = 1
b = 0

The order of the execution of distinct nonblocking assignments to a given variable shall be preserved. This means that if there is clear ordering of the execution of a set of nonblocking assignments, then the order of the resulting updates of the destination of the nonblocking assignments shall be the same as the ordering of the execution.

Example 4:

```
module multiple2;
reg a;

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

If the simulator executes two procedural blocks concurrently, and if these procedural blocks contain non-blocking assignment operators to the same variable, the final value of that variable is indeterminate. For example, the value of reg a is indeterminate in the following example.

Example 5:

```
module 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 = ??
// The assigned value of the reg is indeterminate
endmodule
```

NOTE—The fact that two nonblocking assignments targeting the same variable are in different blocks is not by itself sufficient to make the order of assignments to a variable indeterminate. For example, the value of reg **a** at the end of time cycle 16 is determinate in the following example:

```

module multiple2 ;
reg a;

initial #8 a <= #8 1; // executed at time 8; schedules
                        // an update of 1 at time 16
initial #12 a <= #4 0; // executed at time 12; schedules
                        // an update of 0 at time 16
    // Because it is determinate that the update of a to
    // the value 1 is scheduled before the update of a to
    // the value 0, then it is determinate that a will have the
    // value 0 at the end of time slot 16. endmodule
endmodule

```

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

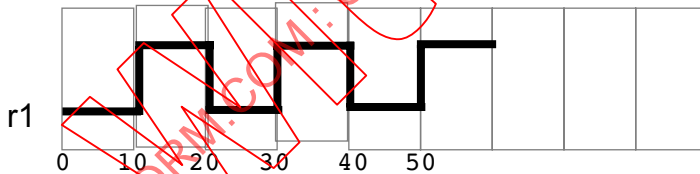
Example 6:

```

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 variables or nets. The syntax for these statements is given in Syntax 9-3.

```

net_assignment ::= (From Annex A - A.6.1)
    net_lvalue = expression
procedural_continuous_assignments ::= (From Annex A - A.6.2)
    assign variable_assignment
    | deassign variable_lvalue
    | force variable_assignment
    | force net_assignment
    | release variable_lvalue
    | release net_lvalue
variable_assignment ::= (From Annex A - A.6.3)
    variable_lvalue = expression
net_lvalue ::= (From Annex A - A.8.5)
    hierarchical_net_identifier
    | hierarchical_net_identifier [ constant_expression ] { [ constant_expression ] }
    | hierarchical_net_identifier [ constant_expression ] { [ constant_expression ] }
      [ constant_range_expression ]
    | hierarchical_net_identifier [ constant_range_expression ]
    | net_concatenation
variable_lvalue ::=
    hierarchical_variable_identifier
    | hierarchical_variable_identifier [ expression ] { [ expression ] }
    | hierarchical_variable_identifier [ expression ] { [ expression ] }
      [ range_expression ]
    | hierarchical_variable_identifier [ range_expression ]
    | variable_concatenation

```

Syntax 9-3 Syntax for procedural continuous assignments

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

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

9.3.1 The assign and deassign procedural statements

The *assign* procedural continuous assignment statement shall override all procedural assignments to a variable. The *deassign* procedural statement shall end a procedural continuous assignment to a variable. The value of the variable shall remain the same until the reg is assigned a new value through a procedural assignment or a procedural continuous assignment. The assign and deassign procedural statements allow, for example, modeling of asynchronous clear/preset on a D-type edge-triggered flip-flop, where the clock is inhibited when the clear or preset is active.

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

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 variables. The left-hand side of the assignment can be a variable, a net, a constant bit-select of a vector net, a part-select of a vector net, or a concatenation. It cannot be a memory word (array reference) or a bit-select or a part-select of a vector variable.

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

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 variable 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", $stime, d, e);
    assign d = a & b & c;
    a = 1;
    b = 0;
    c = 1;
    #10;
    force d = (a | b | c);
    force e = (a | b | c);
    #10 $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 ::= (From Annex A - A.6.6)
    if ( expression )
        statement_or_null [ else statement_or_null ]
    | if_else_if_statement
function_conditional_statement ::= (From Annex A - A.6.6)
    if ( expression ) function_statement_or_null
    [ else function_statement_or_null ]
    | function_if_else_if_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 ::= (From Annex A - A.6.6)
    if ( expression ) statement_or_null
    { else if ( expression ) statement_or_null }
    [ else statement_or_null ]

function_if_else_if_statement ::= (From Annex A - A.6.6)
    if ( expression ) function_statement_or_null
    { else if ( expression ) function_statement_or_null }
    [ else function_statement_or_null ]
```

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 multi-way 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` regs has to be added to the memory address, and which increment is to be added to the `index` reg. The first ten lines declare the regs and parameters.

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

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

9.5 Case statement

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

```

case_statement ::= (From Annex A - A.6.7)
    case ( expression )
        case_item { case_item } endcase
    | casez ( expression )
        case_item { case_item } endcase
    | casex ( expression )
        case_item { case_item } endcase
case_item ::=
    expression { , expression } : statement_or_null
    | default [ : ] statement_or_null
function_case_statement ::=
    case ( expression )
        function_case_item { function_case_item } endcase
    | casez ( expression )
        function_case_item { function_case_item } endcase
    | casex ( expression )
        function_case_item { function_case_item } endcase
function_case_item ::=
    expression { , expression } : function_statement_or_null
    | default [ : ] function_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 reg *rega* to produce a value for *result* as follows:

```

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

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

```

The *case 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'bx0x0x0x0;
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: - Executes an assignment normally used to initialize a variable that controls the number of loops executed. - 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. - Executes an assignment normally used to modify the value of the loop-control variable, then repeats step b.

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

```
function_loop_statement ::= (From Annex A - A.6.8)
    forever function_statement
    | repeat ( expression ) function_statement
    | while ( expression ) function_statement
    | for ( variable_assignment ; expression ; variable_assignment )
      function_statement
loop_statement ::=
    forever statement
    | repeat ( expression ) statement
    | while ( expression ) statement
    | for ( variable_assignment ; expression ; variable_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 9.7.7.

The second type of timing control is the *event expression*, which allows statement execution to be delayed until the occurrence of some simulation event occurring in a procedure executing concurrently with this procedure. A simulation event can be a change of value on a net or variable (an implicit event) or the occurrence of an explicitly named event that is triggered from other procedures (an explicit event). Most often, an event control is a positive or negative edge on a clock signal. Event control is discussed in 9.7.2 through 9.7.7.

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

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

Syntax 9-8 describes timing control in procedural statements.

```

delay_control ::= (From Annex A - A.6.5)
    # delay_value
    | # ( mintypmax_expression )
delay_or_event_control ::=
    delay_control
    | event_control
    | repeat ( expression ) event_control
event_control ::=
    @ event_identifier
    | @ ( event_expression )
    | @*
    | @ ( * )
event_expression ::=
    expression
    | hierarchical_identifier
    | posedge expression
    | negedge expression
    | event_expression or event_expression
    | event_expression , event_expression

```

Syntax 9-8 Syntax for procedural timing control

The gate and net delays also advance simulation time, as discussed in Clause 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. Specify parameters are permitted in the delay expression. They may be overridden by SDF annotation, in which case the expression is reevaluated.

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 variable or the occurrence of a declared event. The value changes on nets and variable can be used as events to trigger the execution of a statement. This is known as detecting an *implicit event*. The event can also be based on 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 43 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 43—Detecting posedge and negedge

	To	0	1	x	z
From					
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 reg r
@(posedge clock) rega = regb; // controlled by posedge on clock
forever @(negedge clock) rega = regb; // controlled by negative edge
```

9.7.3 Named events

A new data type, in addition to nets and variables, called *event* can be declared. An identifier declared as an event data type is called a *named event*. A named event can be triggered explicitly. It can be used in an event expression to control the execution of procedural statements in the same manner as event 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 ::= (From Annex A - A.2.1.3)
    event list_of_event_identifiers ;
list_of_event_identifiers ::= (From Annex A - A.2.3)
    event_identifier [ dimension { dimension } ]
    { , event_identifier [ dimension { dimension } ] }
dimension ::= (From Annex A - A.2.5)
    [ dimension_constant_expression : dimension_constant_expression ]
```

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 9.7.2.

A declared event is made to occur by the activation of an event triggering statement with the syntax given in Syntax 9-10. An event is not made to occur by changing the index of an event array in an event control expression.

```
event_trigger ::= (From Annex A - A.6.5)
    -> hierarchical_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** or a comma character (,) is used as an event logical or operator. A combination of these can be used in the same event expression. Comma-separated sensitivity lists shall be synonymous to **or**-separated sensitivity lists.

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;
```

The following examples show the use of the comma (,) as an event logical or operator.

```
always @(a, b, c, d, e)
always @(posedge clk, negedge rstn)
always @(a or b, c, d or e)
```

9.7.5 Implicit event_expression list

The `event_expression` list of an event control is a common source of bugs in RTL simulations. Users tend to forget to add some of the nets or variables read in the timing control statement. This is often found when comparing RTL and gate level versions of a design. The implicit event expression, `@*`, is a convenient shorthand that eliminates these problems by adding all nets and variables which are read by the statement (which can be a statement group) of a `procedural_timing_control_statement` to the `event_expression`.

All net and variable identifiers which appear in the statement will be automatically added to the event expression with these exceptions:

- Identifiers which only appear in wait or event expressions.
- Identifiers which only appear as a *hierarchical_reg_identifier* in the `reg_lvalue` of the left hand side of assignments.

Nets and variables which appear on the right hand side of assignments, in function and task calls, in case and conditional expressions, as an index variable on the left hand side of assignments or as variables in case item expressions shall all be included by these rules.

Examples:

Example 1

```
always @(*) // equivalent to @(a or b or c or d or f)
y = (a & b) | (c & d) | myfunction(f);
```

Example 2

```
always @* begin // equivalent to @(a or b or c or d or tmp1 or tmp2)
    tmp1 = a & b;
    tmp2 = c & d;
    y = tmp1 | tmp2;
end
```

Example 3

```
always @* begin // equivalent to @(b)
    @(i) kid = b; // i is not added to @*
end
```

Example 4

```

always @* begin // equivalent to @(a or b or c or d)
    x = a ^ b;
    @* // equivalent to @(c or d)
        x = c ^ d;
end

```

Example 5

```

always @* begin // same as @(a or en)
    y = 8'hff;
    y[a] = !en;
end

```

Example 6

```

always @* begin // same as @(state or go or ws)
    next = 4'b0;
    case (1'b1)
        state[IDLE] : if (go) next[READ] = 1'b1;
                        else next[IDLE] = 1'b1;
        state[READ] : next[DLY] = 1'b1;
        state[DLY] : if (!ws) next[DONE] = 1'b1;
                        else next[READ] = 1'b1;
        state[DONE] : next[IDLE] = 1'b1;
    endcase
end

```

9.7.6 Level-sensitive event control

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

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

wait_statement ::= (From Annex A - A.6.5)
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.

9.7.7 Intra-assignment timing controls

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

An intra-assignment delay or event control shall delay the assignment of the new value to the left-hand side, but the right-hand 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.

```
blocking_assignment ::= (From Annex A - A.6.2)
    variable_lvalue = [ delay_or_event_control ] expression
nonblocking_assignment ::=
    variable_lvalue <= [ delay_or_event_control ] expression
delay_control ::= (From Annex A - A.6.5)
    # delay_value
    | # ( mintypmax_expression )
delay_or_event_control ::=
    delay_control
    | event_control
    | repeat ( expression ) event_control
event_control ::=
    @ event_identifier
    | @ ( event_expression )
    | @*
    | @ ( * )
event_expression ::=
    expression
    | hierarchical_identifier
    | posedge expression
    | negedge expression
    | event_expression or event_expression
    | event_expression , event_expression
```

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

The intra-assignment delay and event control can be applied to both blocking assignments and nonblocking assignments. The 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. If the *repeat* count literal, or signed reg holding the *repeat* count, is less than or equal to 0 at the time of evaluation, the assignment occurs as if there is no *repeat* construct.

Examples:

```
repeat (-3) @ (event_expression)
    // will not execute event_expression.
```



```
repeat (a) @ (event_expression)
    // if a is assigned -3 it will execute the event_expression
    // if a is declared as an unsigned reg but not if it is
signed.
```

This construct is convenient when events have to be synchronized with counts of clock signals.

Examples:

Table 44 illustrates the philosophy of intra-assignment timing controls by showing the code that could accomplish the same timing effect without using intra-assignment.

Table 44—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 31 illustrates the activities that result from this `repeat` event control.

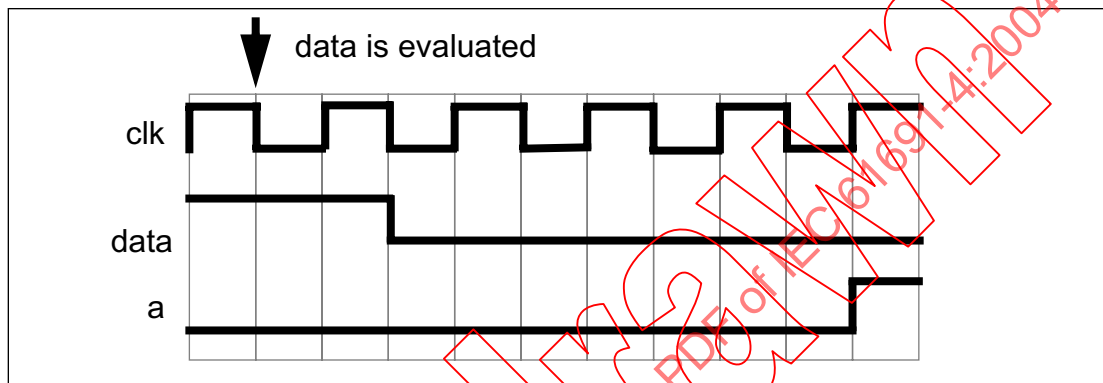


Figure 31 — Repeat event control utilizing a clock edge

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.

```
function_seq_block ::= (From Annex A - A.6.3)
    begin [ : block_identifier
        { block_item_declaration } ] { function_statement } end
seq_block ::=
    begin [ : block_identifier
        { block_item_declaration } ] { statement } end
block_item_declaration ::= (From Annex A - A.2.8)
    { attribute_instance } block_reg_declaration
    { attribute_instance } event_declaration
    { attribute_instance } integer_declaration
    { attribute_instance } local_parameter_declaration
    { attribute_instance } parameter_declaration
    { attribute_instance } real_declaration
    { attribute_instance } realtime_declaration
    { attribute_instance } time_declaration
```

Syntax 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 reg

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 ::= (From Annex A - A.6.3)
    fork [ : block_identifier
        { block_item_declaration } ] { statement } join
block_item_declaration ::= (From Annex A - A.2.8)
    { attribute_instance } block_reg_declaration
    { attribute_instance } event_declaration
    { attribute_instance } integer_declaration
    { attribute_instance } local_parameter_declaration
    { attribute_instance } parameter_declaration
    { attribute_instance } real_declaration
    { attribute_instance } realtime_declaration
    { attribute_instance } time_declaration
```

Syntax 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 reg is exactly the same for both implementations.

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

9.8.3 Block names

Both sequential and parallel blocks can be named by adding : `name_of_block` after the keywords **begin** or **fork**. The naming of blocks serves several purposes:

- It allows local variables, parameters, and named events to be declared for the block.
- It allows the block to be referenced in statements such as the disable statement (see Clause 11).

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

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

9.8.4 Start and finish times

Both sequential and 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.

initial_construct ::= (From Annex A - A.6.2)
initial statement

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 reg
    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.

always_construct ::= (From Annex A - A.6.2)
always statement

Syntax 9-16 Syntax for always construct

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

The following code, for example, creates a zero-delay infinite loop.

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

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

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

10. Tasks and functions

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

10.1 Distinctions between tasks and functions

The following rules distinguish tasks from functions:

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

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

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

10.2.1 Task declarations

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

```

task_declaration ::= (From Annex A - A.2.7)
    task [ automatic ] task_identifier ;
    { task_item_declaration }
    statement
endtask
| task [ automatic ] task_identifier ( task_port_list ) ;
    { block_item_declaration }
    statement
endtask

task_item_declaration ::=
    block_item_declaration
    | { attribute_instance } tf_input_declaration ;
    | { attribute_instance } tf_output_declaration ;
    | { attribute_instance } tf_inout_declaration ;

task_port_list ::=
    task_port_item { , task_port_item }

task_port_item ::=
    { attribute_instance } tf_input_declaration
    | { attribute_instance } tf_output_declaration
    | { attribute_instance } tf_inout_declaration

tf_input_declaration ::=
    input [ reg ] [ signed ] [ range ] list_of_port_identifiers
    | input [ task_port_type ] list_of_port_identifiers

tf_output_declaration ::=
    output [ reg ] [ signed ] [ range ] list_of_port_identifiers
    | output [ task_port_type ] list_of_port_identifiers

tf_inout_declaration ::=
    inout [ reg ] [ signed ] [ range ] list_of_port_identifiers
    | inout [ task_port_type ] list_of_port_identifiers

task_port_type ::=
    time | real | realtime | integer

block_item_declaration ::= (From Annex A - A.2.8)
    { attribute_instance } block_reg_declaration
    | { attribute_instance } event_declaration
    | { attribute_instance } integer_declaration
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration
    | { attribute_instance } real_declaration
    | { attribute_instance } realtime_declaration
    | { attribute_instance } time_declaration

block_reg_declaration ::=
    reg [ signed ] [ range ]
    list_of_block_variable_identifiers ;

list_of_block_variable_identifiers ::=
    block_variable_type { , block_variable_type }

block_variable_type ::=
    variable_identifier
    | variable_identifier dimension { dimension }

```

Syntax 10-1 Syntax for task declaration

There are two alternate task declaration syntaxes.

The first syntax shall begin with the keyword **task**, followed by the optional keyword **automatic**, followed by a name for the task and a semicolon, and ending with the keyword **endtask**. The keyword **automatic** declares an automatic task that is reentrant with all the task declarations allocated dynamically for each concurrent task entry. Task item declarations can specify the following:

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

The second syntax shall begin with the keyword **task**, followed by a name for the task and a parenthesis enclosed *task_port_list*. The *task_port_list* shall consist of zero or more comma separated *task_port_items*. There shall be a semicolon after the close parenthesis. The task body shall follow and then the keyword **end-task**.

In both syntaxes, the port declarations shall have the same syntax as defined by the *tf_input_declaration*, *tf_output_declaration* and *tf_inout_declaration*, as detailed in Syntax 10-1 above.

Tasks without the optional keyword **automatic** are static tasks, with all declared items being statically allocated. These items shall be shared across all uses of the task executing concurrently. Task with the optional keyword **automatic** are automatic tasks. All items declared inside automatic tasks are allocated dynamically for each invocation. Automatic task items can not be accessed by hierarchical references. Automatic tasks can be invoked through use of their hierarchical name.

10.2.2 Task enabling and argument passing

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

```
task_enable ::= (From Annex A - A.6.9)
hierarchical_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** variables
- Memory references
- Concatenations of **reg**, **integer**, **real**, **realtime** and **time** variables
- Concatenations of memory references
- Bit-selects and part-selects of **reg**, **integer**, and **time** variables

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

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 regs  
    d = foo2;  
    e = foo3;  
end  
endtask
```

Or using the second form of a task declaration, the task could be defined as:

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

The following statement enables the task:

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

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

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

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

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

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

```

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

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

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

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

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

```

10.2.3 Task memory usage and concurrent activation

A task may be enabled more than once concurrently. All variables of an automatic task shall be replicated on each concurrent task invocation to store state specific to that invocation. All variables of a static task shall be static in that there shall be a single variable corresponding to each declared local variable in a module instance, regardless of the number of concurrent activations of the task. However, static tasks in different instances of a module shall have separate storage from each other.

Variables declared in static tasks shall retain their values between invocations. They shall be initialized to the default initialization value as described in 3.2.2. Variables declared in automatic tasks shall be initialized to the default initialization value whenever execution enters their scope.

Because variables declared in automatic tasks are deallocated at the end of the task invocation, they shall not be used in certain constructs that might refer to them after that point.

- They shall not be assigned values using nonblocking assignments or procedural continuous assignments.
- They shall not be referenced by procedural continuous assignments or procedural force statements.
- They shall not be referenced in intra-assignment event controls of nonblocking assignments.
- They shall not be traced with system tasks such as **\$monitor** and **\$dumpvars**.

10.3 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 Function declarations

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

```

function_declaration ::= (From Annex A - A.2.6)
    function [ automatic ] [ signed ] [ range_or_type ]
        function_identifier ;
        function_item_declaration { function_item_declaration }
        function_statement
    endfunction
| function [ automatic ] [ signed ] [ range_or_type ]
    function_identifier ( function_port_list ) ;
    block_item_declaration { block_item_declaration }
    function_statement
    endfunction
function_item_declaration ::=
    block_item_declaration
    | tf_input_declaration ;
function_port_list ::=
    { attribute_instance } tf_input_declaration
    | { , { attribute_instance } tf_input_declaration }
tf_input_declaration ::=
    input [ reg ] [ signed ] [ range ] list_of_port_identifiers
    | input [ task_port_type ] list_of_port_identifiers
range_or_type ::=
    range | integer | real | realtime | time
block_item_declaration ::= (From Annex A - A.2.8)
    { attribute_instance } block_reg_declaration
    | { attribute_instance } event_declaration
    | { attribute_instance } integer_declaration
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration
    | { attribute_instance } real_declaration
    | { attribute_instance } realtime_declaration
    | { attribute_instance } time_declaration
block_reg_declaration ::=
    reg [ signed ] [ range ]
        list_of_block_variable_identifiers ;
list_of_block_variable_identifiers ::=
    block_variable_type { , block_variable_type }
block_variable_type ::=
    variable_identifier
    | variable_identifier dimension { dimension }

```

Syntax 10-3 Syntax for function declaration

A function definition shall begin with the keyword **function**, followed by the optional keyword **automatic**, followed by the optional **signed** designator, followed by the range or type of the return value from the function, followed by the name of the function, and then either a semicolon, or a function port list enclosed in parenthesis, and then a semicolon, and then shall end with the keyword **endfunction**. The use of a *range_or_type* shall be optional. A function specified without a range or type defaults to a one bit reg for the return value. If used, *range_or_type* shall specify 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.

The keyword **automatic** declares a recursive function with all the function declarations allocated dynamically for each recursive call. Automatic function items can not be accessed by hierarchical references. Automatic functions can be invoked through the use of their hierarchical name.

Function inputs shall be declared one of two ways. The first method shall have the name of the function followed by a semicolon. After the semicolon one or more input declarations optionally mixed with block item declarations shall follow. After the function item declarations there shall be a behavioral statement and then the **endfunction** keyword.

The second method shall have the name of the function, followed by an open parenthesis, and one or more input declarations, separated by commas. After all the input declarations, there shall be a close parenthesis, and a semicolon. After the semicolon, there shall be zero or more block item declarations, followed by a behavioral statement, and then the **endfunction** keyword.

Example:

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

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

Or using the second form of a function declaration, the function could be defined as:

```
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 variable, internal to the function, with the same name as the function. This variable either defaults to a 1-bit reg or is the same type as the type specified in the function declaration. The function definition initializes the return value from the function by assigning the function result to the internal variable with the same name as the function.

It is illegal to declare another object with the same name as the function in the scope where the function is declared. Inside a function, there is an implied variable with the name of the function, which may be used in expressions within the function. It is, therefore, also illegal to declare another object with the same name as the function inside the function scope.

The following line from the example in 10.3.1 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.

function_call ::= (From Annex A - A.8.2)
hierarchical_function_identifier { attribute_instance } (expression { , 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 six 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 variable that has the same name as the function name.
- f) A function shall not have any nonblocking assignments.

Example:

This example defines a function called `factorial` that returns an integer value. The `factorial` function is called iteratively and the results are printed.

```
module tryfact;

// define the function
function automatic integer factorial;
input [31:0] operand;
integer i;
if (operand >= 2)
    factorial = factorial (operand - 1) * operand;
else
    factorial = 1;
endfunction

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

The simulation results are as follows:

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

10.3.5 Use of constant functions

Constant function calls are used to support the building of complex calculations of values at elaboration time (see 12.1.3). A *constant function call* shall be a function invocation of a *constant function* local to the calling module where the arguments to the function are *constant expressions*. *Constant functions* are a subset of normal Verilog functions that shall meet the following constraints:

- They shall contain no hierarchical references.
- Any function invoked within a *constant function* shall be a *constant function* local to the current module. System functions shall not be invoked.
- All system tasks within a constant function shall be ignored.
- All system functions within a constant function shall be illegal.
- The only system task that may be invoked is `$display`, and it shall be ignored when invoked at elaboration time.

- All parameter values used within the function shall be defined before the use of the invoking *constant function call* (i.e. any parameter use in the evaluation of a *constant function call* constitutes a use of that parameter at the site of the original *constant function call*).
- All identifiers which are not parameters or functions shall be declared locally to the current function.
- If they use any parameter value that is affected directly or indirectly by a **defparam** statement (see 12.2.1), the result is undefined. This can produce an error or the constant function can return an indeterminate value.
- They shall not be declared inside a generate scope.
- They shall not themselves use constant functions in any context requiring a constant expression.

Constant function calls are evaluated at elaboration time. Their execution has no effect on the initial values of the variables used either at simulation time or among multiple invocations of a function at elaboration time. In each of these cases, the variables are initialized as they would be for normal simulation.

Example:

This example defines a function called `clogb2` that returns an integer which has the value of the ceiling of the log base 2.

```

module ram_model (address, write, chip_select, data);
  parameter data_width = 8;
  parameter ram_depth = 256;
  localparam adder_width = clogb2(ram_depth);
  input [adder_width - 1:0] address;
  input write, chip_select;
  inout [data_width - 1:0] data;

  //define the clogb2 function
  function integer clogb2;
    input [31:0] value;
    for (clogb2=0; value>0; clogb2=clogb2+1;)
      value = value>>1;
  endfunction

  reg [data_width - 1:0] data_store[0:ram_depth - 1];
  //the rest to the ram model

```

An instance of this `ram_model` with parameters assigned:

```

ram_model #(32,421) ram_a0(a_addr,a_wr,a_cs,a_data);

```

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.

```
disable_statement ::= (From Annex A - A.6.5)
    disable hierarchical_task_identifier ;
    | disable hierarchical_block_identifier ;
```

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 can be used to disable named blocks within a function, but cannot be used to disable functions. In cases where a disable statement within a function disables a block or a task that called the function, the behavior is undefined. Disabling an automatic task or a block inside an automatic task proceeds as for regular tasks for all concurrent executions of the task.

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 forward *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
```

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, muxes, and alus.

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

12.1 Modules

This 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 parameter definitions shall specify an ordered list of the parameters for the module. The optional list of ports or port declarations shall specify an ordered list of the ports for the module. The order used in defining the list of parameters in the `module_parameter_port_list` and in the list of ports can be significant when instantiating the module (see 12.2.2.1 and 12.3.5). The identifiers in this list shall be declared in input, output, and inout statements within the module definition. Ports declared in the list of port declarations shall not be redeclared within the body of the module. The module items define what constitutes a module and they include many different types of declarations and definitions; many of which have already been introduced.

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


```

module_declaration ::= (From Annex A - A.1.3)
    { attribute_instance } module_keyword module_identifier [ module_parameter_port_list ]
    [ list_of_ports ] ; { module_item }
    endmodule
    { attribute_instance } module_keyword module_identifier [ module_parameter_port_list ]
    [ list_of_port_declarations ] ; { non_port_module_item }
    endmodule

module_keyword ::= module | macromodule

module_parameter_port_list ::= (From Annex A - A.1.4
    # ( parameter_declaration { , parameter_declaration } )
list_of_ports ::= ( port { , port } )
list_of_port_declarations ::= ( port_declaration { , port_declaration } ) ( )
port ::= [ port_expression ] . port_identifier ( [ port_expression ] )
port_expression ::= port_reference | { port_reference { , port_reference } }
port_reference ::= port_identifier | port_identifier [ constant_expression ]
    | port_identifier [ range_expression ]
port_declaration ::= { attribute_instance } inout_declaration
    | { attribute_instance } input_declaration
    | { attribute_instance } output_declaration
module_item ::= module_or_generate_item (From Annex A - A.1.5)
    | port_declaration ;
    | { attribute_instance } generated_instantiation
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration
    | { attribute_instance } specify_block
    | { attribute_instance } specparam_declaration
module_or_generate_item ::= { attribute_instance } module_or_generate_item_declaration
    | { attribute_instance } parameter_override
    | { attribute_instance } continuous_assign
    | { attribute_instance } gate_instantiation
    | { attribute_instance } udp_instantiation
    | { attribute_instance } module_instantiation
    | { attribute_instance } initial_construct
    | { attribute_instance } always_construct
module_or_generate_item_declaration ::= net_declaration
    | reg_declaration
    | integer_declaration
    | real_declaration
    | time_declaration
    | realtime_declaration
    | event_declaration
    | genvar_declaration
    | task_declaration
    | function_declaration
non_port_module_item ::= { attribute_instance } generated_instantiation
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } module_or_generate_item
    | { attribute_instance } parameter_declaration
    | { attribute_instance } specify_block
    | { attribute_instance } specparam_declaration
parameter_override ::= defparam list_of_param_assignments ;

```

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

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

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

```

module_instantiation ::= (From Annex A - A.4.1)
    module_identifier [ parameter_value_assignment ]
    module_instance { , module_instance } ;
parameter_value_assignment ::=
    # ( list_of_parameter_assignments )
list_of_parameter_assignments ::=
    ordered_parameter_assignment { , ordered_parameter_assignment }
    | named_parameter_assignment { , named_parameter_assignment }
ordered_parameter_assignment ::=
    expression
named_parameter_assignment ::=
    . parameter_identifier ( [ expression ] )
module_instance ::=
    name_of_instance ( [ list_of_port_connections ] )
name_of_instance ::=
    module_instance_identifier [ range ]
list_of_port_connections ::=
    ordered_port_connection { , ordered_port_connection }
    | named_port_connection { , named_port_connection }
ordered_port_connection ::=
    { attribute_instance } [ expression ]
named_port_connection ::=
    { attribute_instance } .port_identifier ( [ expression ] )

```

Syntax 12-2 Syntax for module instantiation

The instantiations of modules can contain a range specification. This allows an array of instances to be created. The array of instances 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 port connections shall be provided only for modules defined with ports. The parentheses, however, are always required. When a list of port connections is given using the ordered port connection method, the first element in the list shall connect to the first port declared in the module, the second to the second port, and so on. See 12.3 for a more detailed discussion of ports and port connection rules.

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

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

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

```

12.1.3 Generated instantiation

After a Verilog design has been parsed, but before simulation begins, the design must have the modules being instantiated linked to the modules being defined, the parameters propagated among the various modules, and hierarchical references resolved. This phase in understanding a Verilog description is termed *elaboration*.

Generate instantiations are resolved during elaboration because that is when the parameters associated with a module become defined, hence, allowing the definition of the generated statements and declarations. Genvars are only defined during the evaluation of the generate instantiations and do not exist during simulation of a design.

Generate statements facilitate the creation of parameterized models. When used with constant functions (see 10.3.5), parameters can be used to constrain other parameter(s) or localparam(s) in a generated design.

All generate instantiations are coded within a module scope and require the keywords **generate - endgenerate**.

Generate statements allow control over the declaration of variables, functions and tasks, as well as control over instantiations. Generated instantiations are one or more: modules, user defined primitives, Verilog gate primitives, continuous assignments, initial blocks and always blocks. Generated declarations and instantiations can be conditionally instantiated into a design. Generated variable declarations and instantiations can be multiply instantiated into a design. Generated instances have unique identifier names and can be referenced hierarchically as described in 12.4.

To support the interconnection between structural elements and/or procedural blocks, generate statements permit the following Verilog data types to be declared within the generate scope: **net**, **reg**, **integer**, **real**, **time**, **realtime**, and **event**. Generated data types have unique identifier names and can be referenced hierarchically as described in 12.4 .

Parameter redefinition using ordered or named **parameter = value** assignment or **defparam** statements can also be declared within the generate scope. However, a **defparam** statement within the generate scope or within a hierarchy instantiated within the generate scope shall only modify the value of a parameter declared within the generate scope or within a hierarchy instantiated within the generate scope.

Tasks and functions declarations shall also be permitted within the generate scope, however not in a generate loop. Generated tasks and functions shall have unique identifier names and may be referenced hierarchically as described in 12.4.

Module declarations and module items that shall not be permitted in a generate statement include: parameters, local parameters, input declarations, output declarations, inout declarations and specify blocks.

Connections to generated module instances are handled the same way as they are handled with normal module instances as described in 12.1.2.

Generated statements are created using one of the following three methods: generate-loop, generate-conditional, or generate-case.

The syntax for generate instantiations is given in Syntax 12-3.

```

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

module_or_generate_item ::=
    { attribute_instance } module_or_generate_item_declaration
    { attribute_instance } parameter_override
    { attribute_instance } continuous_assign
    { attribute_instance } gate_instantiation
    { attribute_instance } udp_instantiation
    { attribute_instance } module_instantiation
    { attribute_instance } initial_construct
    { attribute_instance } always_construct

module_or_generate_item_declaration ::=
    net_declaration
    reg_declaration
    integer_declaration
    real_declaration
    time_declaration
    realtime_declaration
    event_declaration
    genvar_declaration
    task_declaration
    function_declaration

generated_instantiation ::= (From Annex A - A.4.2)
    generate { generate_item } endgenerate

generate_item_or_null ::=
    generate_item | ;

generate_item ::=
    generate_conditional_statement
    generate_case_statement
    generate_loop_statement
    generate_block
    module_or_generate_item

generate_conditional_statement ::=
    if ( constant_expression ) generate_item_or_null [ else generate_item_or_null ]

generate_case_statement ::= case ( constant_expression )
    genvar_case_item { genvar_case_item } endcase

genvar_case_item ::= constant_expression { , constant_expression } :
    generate_item_or_null | default [ : ] generate_item_or_null

generate_loop_statement ::=
    for ( genvar_assignment ; constant_expression ; genvar_assignment )
    begin : generate_block_identifier { generate_item } end

genvar_assignment ::=
    genvar_identifier = constant_expression

generate_block ::=
    begin [ : generate_block_identifier ] { generate_item } end

```

Syntax 12-3 Syntax for generate blocks

12.1.3.1 genvar - generate statement index variable

An index variable that shall only be declared for use in generate statements shall be declared as a *genvar* and is referred to as a *genvar* in the rest of this section.

The syntax for generate statement index variable declarations is given in Syntax 12-4.

```
genvar_declaration ::= (From Annex A - A.2.1.3)
    genvar list_of_genvar_identifiers ;
list_of_genvar_identifiers ::= (From Annex A - A.2.3)
    genvar_identifier { , genvar_identifier }
```

Syntax 12-4 Syntax for generate statement index variable declaration

A *genvar* shall be declared within the module where the *genvar* is used. A *genvar* can be declared either inside or outside of a generate scope. A *genvar* is an integer that is local to and shall only be used within a generate loop that uses it as an index variable. If any bit of the *genvar* ever is set to an X or Z or if the *genvar* is set to a negative value, this shall be an error.

Genvars are only defined during the evaluation of the generate blocks (see 12.1.3), and do not exist during simulation of a Verilog design.

The value of a *genvar* shall only be defined by a generate loop. Two generate loops using the same *genvar* as an index variable shall not be nested. The value of a *genvar* can be referenced in any context where the value of a parameter could be referenced.

12.1.3.2 generate-loop

A generate-loop permits one or more variable declarations, modules, user defined primitives, gate primitives, continuous assignments, initial blocks and always blocks to be instantiated multiple times using a for-loop. The index loop variable used in a generate for-loop shall be declared as a *genvar*. Both *genvar* assignments in the for-loop shall assign to the same *genvar*, which is the loop index variable. The first *genvar* assignment in the for-loop shall not reference the loop index variable on the right hand side.

Examples:

Example 1 A parameterized gray-code to binary-code converter module using a loop to generate continuous assignments

```
module gray2bin1 (bin, gray);
    parameter SIZE = 8; // this module is parameterizable
    output [SIZE-1:0] bin;
    input [SIZE-1:0] gray;

    genvar i;

    generate for (i=0; i<SIZE; i=i+1) begin:bit
        assign bin[i] = ^gray[SIZE-1-i];
    end generate
endmodule
```

Example 2 The same gray-code to binary-code converter module in example 1 is built using a loop to generate always blocks

```

module gray2bin2 (bin, gray);
  parameter SIZE = 8;          // this module is parameterizable
  output [SIZE-1:0] bin;
  input  [SIZE-1:0] gray;
  reg    [SIZE-1:0] bin;

  genvar i;

  generate for (i=0; i<SIZE; i=i+1) begin:bit
    always @(gray[SIZE-1:i]) // fixed part select
      bin[i] = ^gray[SIZE-1:i];
  end endgenerate
endmodule

```

The models in examples 3 and 4 are parameterized modules of ripple adders using a loop to generate Verilog gate primitives. Example 3 uses a two dimensional net declaration outside of the generate loop to make the connections between the gate primitives while example 4 makes the net declaration inside of the generate loop to generate the wires needed to connect the gate primitives for each iteration of the loop.

Example 3 Generated ripple adder with two-dimensional net declaration outside of the generate loop

```

module addergen1 (co, sum, a, b, ci);
  parameter SIZE = 4;
  output [SIZE-1:0] sum;
  output co;
  input [SIZE-1:0] a, b;
  input ci;
  wire [SIZE :0] c;
  wire [SIZE-1:0] t [1:3];
  genvar i;

  assign c[0] = ci;

  // Generated instance names are:
  // xor gates: bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
  //             bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
  // and gates: bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
  //             bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
  // or gates: bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
  // Generated instances are connected with
  // multi-dimensional nets t[1][3:0] t[2][3:0] t[3][3:0]
  // (12 multi-dimensional nets total)
  generate
    for(i=0; i<SIZE; i=i+1) begin:bit
      xor g1 ( t[1][i], a[i], b[i]);
      xor g2 ( sum[i], t[1][i], c[i]);
      and g3 ( t[2][i], a[i], b[i]);
      and g4 ( t[3][i], t[1][i], c[i]);
      or g5 ( c[i+1], t[2][i], t[3][i]);
    end
  endgenerate

  assign co = c[SIZE];
endmodule

```

Example 4 Generated ripple adder with net declaration inside of the generate loop

```

module addergen1 (co, sum, a, b, ci);
  parameter SIZE = 4;
  output [SIZE-1:0] sum;
  output co;
  input [SIZE-1:0] a, b;
  input ci;
  wire [SIZE :0] c;

  genvar i;

  assign c[0] = ci;

  // Generated instance names are:
  // xor gates: bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
  //             bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
  // and gates: bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
  //             bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
  // or gates: bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
  // Generated instances are connected with
  // generated nets: bit[0].t1 bit[1].t1 bit[2].t1 bit[3].t1
  //                 bit[0].t2 bit[1].t2 bit[2].t2 bit[3].t2
  //                 bit[0].t3 bit[1].t3 bit[2].t3 bit[3].t3
  generate
    for(i=0; i<SIZE; i=i+1) begin:bit
      wire t1, t2, t3; // generated net declaration

      xor g1 ( t1, a[i], b[i]);
      xor g2 ( sum[i], t1, c[i]);
      and g3 ( t2, a[i], b[i]);
      and g4 ( t3, t1, c[i]);
      or g5 ( c[i+1], t2, t3);
    end
  endgenerate

  assign co = c[SIZE];
endmodule

```

The generated instance names in a multi-level generate loop are shown in example 5. The generated name for the scope at each generate loop is created by adding the "[genvar s value]" string to the end of the generate block identifier for the loop. The generated names are now generated identifiers (see 2.7.2) which can be used in hierarchical path names (see 12.4).

Example 5 A multi-level generate loop

```

parameter SIZE = 2;
genvar      i, j, k, m;
generate
  for (i=0; i<SIZE+1; i=i+1) begin:B1 // scope B1[i]
    M1 N1(); // instantiates B1[i].N1[i]
    for (j=0; j<SIZE; j=j+1) begin:B2 // scope B1[i].B2[j]
      M2 N2(); // instantiates B1[i].B2[j].N2
      for (k=0; k<SIZE; k=k+1) begin:B3 // scope B1[i].B2[j].B3[k]
        M3 N3(); // instantiates B1[i].B2[j].B3[k].N3
      end
    end
  end
  if (i>0)
    for (m=0; m<SIZE; m=m+1) begin:B4 // scope B1[i].B4[m]
      M4 N4(); // instantiates B1[i].B4[m].N4
    end
  end
endgenerate

// some of the generated instance names are:
// B1[0].N1  B1[1].N1
// B1[0].B2[0].N2  B1[0].B2[1].N2
// B1[0].B2[0].B3[0].N3  B1[0].B2[0].B3[1].N3
//      B1[0].B2[1].B3[0].N3
// B1[1].B4[0].N4  B1[1].B4[1].N4

```

12.1.3.3 generate-conditional

A generate-conditional is an if-else-if generate construct that permits modules, user defined primitives, Verilog gate primitives, continuous assignments, initial blocks and always blocks to be conditionally instantiated into another module based on an expression that is deterministic at the time the design is elaborated.

Example 6 shows the implementation of a parameterized module. If either of the multiplier's `a_width` or `b_width` parameters are less than 8 (bits), a CLA multiplier is instantiated. If both of the multiplier's `a_width` or `b_width` parameters are greater than or equal to 8 (bits), a Wallace tree multiplier is instantiated.

Example 6 An implementation of a parameterized multiplier module

```

module multiplier(a,b,product);
parameter a_width = 8, b_width = 8;
localparam product_width = a_width+b_width; // can not be modified
// directly with the defparam statement
// or the module instance statement #
input    [a_width-1:0]    a;
input    [b_width-1:0]    b;
output   [product_width-1:0] product;

generate
  if((a_width < 8) || (b_width < 8))
    CLA_multiplier #(a_width,b_width) u1(a, b, product);
    // instance a CLA multiplier
  else
    WALLACE_multiplier #(a_width,b_width) u1(a, b, product);
    // instance a Wallace-tree multiplier
endgenerate
// The generated instance name is u1

endmodule

```

12.1.3.4 generate-case

A generate case construct permits modules, user defined primitives, Verilog gate primitives, continuous assignments, initial blocks and always blocks to be conditionally instantiated into another module based on a select one-of-many case construct. The selecting case expression must be deterministic at the time the design is elaborated.

Example 7 Generate with a case to handle widths less than 3

```

generate
  case (WIDTH)
    1: adder_1bit x1(co, sum, a, b, ci);
    // 1-bit adder implementation
    2: adder_2bit x1(co, sum, a, b, ci);
    // 2-bit adder implementation
    default: adder_cla #(WIDTH) x1(co, sum, a, b, ci);
    // others - carry look-ahead adder
  endcase
// The generated instance name is x1

endgenerate

```

Example 8 A module of memory dimm

```

module dimm (adr, ba, rasx, casx, csx, wex, dqm, cke, data, clk, dev_id);
  parameter [31:0] MEM_SIZE = 8, // in mbytes
              MEM_WIDTH = 16;
  input [10:0] adr;
  input ba;
  input rasx, casx, csx, wex;
  input [ 7:0] dqm;
  input cke;
  inout [63:0] data;
  input clk;
  input [ 4:0] dev_id;

  genvar i;

  generate
    case ({MEM_SIZE, MEM_WIDTH})
      {32'd8, 32'd16}: // 8Meg x 16 bits wide.
        begin
          // The generated instance names are word[3].p, word[2].p,
          // word[1].p, word[0].p, and the task read_mem
          for (i=0; i<4; i=i+1) begin:word
            sms_16b216t0 p (.clk(clk), .csb(csx), .cke(cke),
                          .ba(ba[0]), .addr(adr[10:0]), .rasb(rasx),
                          .casb(casx), .web(wex), .udqm(dqm[2*i+1]),
                          .ldqm(dqm[2*i]), .dqi(data[15+16*i:16*i]),
                          .dev_id(dev_id[4:0]));
          end
          task read_mem;
            input [31:0] address;
            output [63:0] data;
            begin
              word[3].p.read_mem(address, data[63:48]);
              word[2].p.read_mem(address, data[47:32]);
              word[1].p.read_mem(address, data[31:16]);
              word[0].p.read_mem(address, data[15:0]);
            end
          endtask
        end
      {32'd16, 32'd8}: // 16Meg x 8 bits wide.
        begin
          // The generated instance names are byte[7].p, byte[6].p,
          // byte[5].p, byte[4].p, byte[3].p, byte[2].p, byte[1].p,
          // byte[0].p and the task read_mem
          for (i=0; i<8; i=i+1) begin:byte
            sms_16b208t0 p (.clk(clk), .csb(csx), .cke(cke),
                          .ba(ba[0]), .addr(adr[10:0]), .rasb(rasx),
                          .casb(casx), .web(wex), .dqm(dqm[i]),
                          .dqi(data[7+8*i:8*i]), .dev_id(dev_id[4:0]));
          end
          task read_mem;
            input [31:0] address;
            output [63:0] data;
            begin
              byte[7].p.read_mem(address, data[63:56]);
              byte[6].p.read_mem(address, data[55:48]);
              byte[5].p.read_mem(address, data[47:40]);
              byte[4].p.read_mem(address, data[39:32]);
              byte[3].p.read_mem(address, data[31:24]);
              byte[2].p.read_mem(address, data[23:16]);
              byte[1].p.read_mem(address, data[15:8]);
              byte[0].p.read_mem(address, data[7:0]);
            end
          endtask
        end
      // Other memory cases ...
    endcase
  endgenerate
endmodule

```

```

// The generated instance names are word[3].p, word[2].p,
// word[1].p, word[0].p, and the task read_mem
{32'd16, 32'd8}: // 16Meg 8 bits wide.
begin
  for (i=0;i<4;i = i + 1)
    begin:byte
      sms_16b208t0 p
        (.clk(clk), .csb(csx), .cke(cke), .ba(ba[0]),
          .addr(adr[10:0]),
          ...rasb(rasx), .casb(casx), .web(wex), .dqm(dqm[i]),
          .dqi(data[8+8*i:8*i]),...dev_id(dev_id7[4:0])
        );
    end
  task read_mem;
    input [31:0] address;
    output [63:0] data;
    begin
      byte[7].p.read_mem(address, data[63:56]);
      byte[6].p.read_mem(address, data[55:48]);
      byte[5].p.read_mem(address, data[47:40]);
      byte[4].p.read_mem(address, data[39:32]);
      byte[3].p.read_mem(address, data[31:24]);
      byte[2].p.read_mem(address, data[23:16]);
      byte[1].p.read_mem(address, data[15:8]);
      byte[0].p.read_mem(address, data[7:0]);
    end
  endtask
  .....
endcase
endgenerate
// The generated instance names are byte[7].p, byte[6].p,
// byte[5].p, byte[4].p, byte[3].p, byte[2].p, byte[1].p,
// byte[0].p and the task read_mem

endmodule

```

12.2 Overriding module parameter values

There are two different ways that parameters can be defined. The first is the *module_parameter_port_list* (see 12.1), and the second is as a *module_item* (see 3.11). A module declaration can contain parameter definitions of either or both types, or no parameter definitions.

A module parameter can have a type specification and a range specification. The effect of parameter overrides on a parameter's type and range shall be in accordance with the following rules:

- A parameter declaration with no type or range specification shall default to the type and range of the final override value assigned to the parameter.
- A parameter with a range specification, but with no type specification, shall be the range of the parameter declaration and shall be unsigned. An override value shall be converted to the type and range of the parameter.
- A parameter with a type specification, but with no range specification, shall be of the type specified. An override value shall be converted to the type of the parameter. A signed parameter shall default to the range of the final override value assigned to the parameter.

- A parameter with a signed type specification and with a range specification shall be a signed, and shall be the range of its declaration. An override value shall be converted to the type and range of the parameter.

Examples:

```

module generic_fifo
    #(parameter MSB=3, LSB=0, DEPTH=4) // These parameters can be
    overridden
    (
        input [MSB:LSB] in,
        input clk, read, write, reset,
        output [MSB:LSB] out,
        output full, empty
    );

    localparam FIFO_MSB = DEPTH*MSB; // These parameters are local,
    and
    localparam FIFO_LSB = LSB;      // cannot be overridden. They can
    be                                // affected by altering the public
                                    // parameters above, and the module
                                    // will work correctly.

    reg [FIFO_MSB:FIFO_LSB] fifo;
    reg [LOG2(DEPTH):0] depth;

    always @(posedge clk or reset) begin
        case ({read,write,reset})
            // implementation of fifo
        endcase
    end
endmodule

```

There are two ways to alter non-local 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. If a defparam assignment conflicts with a module instance parameter, the parameter in the module will take the value specified by the defparam. The module instance parameter value assignment comes in two forms, by ordered list or by name. The next two subclauses describe these two methods.

There are two kinds of parameter declarations. The first kind of parameter declaration has a type and or range qualification, and second does not. When an untyped and unranged parameter's value is overridden, the parameter takes on the size and type of the override.

When a typed and/or ranged parameter is overridden, the new value is converted to the type and size of the destination, and assigned to that parameter.

Example:

```

module foo(a,b);
    real r1,r2;
    parameter [2:0] A = 3'h2;
    parameter B = 3'h2;
    initial begin
        r1 = A;
        r2 = B;
    end
endmodule

```

```

        $display("r1 is %f r2 is %f",r1,r2);
    end
endmodule // foo
module bar;
    wire a,b;
    defparam f1.A = 3.1415;
    defparam f1.B = 3.1415;
    foo f1(a,b);
endmodule // bar

```

Parameter A is a typed and/or ranged parameter, so when its value is redefined, the parameter retains its original type and sign. Therefore, the defparam of f1.A with the value 3.1415 is performed by converting the floating point number 3.1415 into a fixed point number 3 and then the low 3 bits of 3 are assigned to A.

Parameter B is not a typed and/or ranged parameter, so when its value is redefined, the parameter type and range take on the type and range of the new value. Therefore, the defparam of f1.B with the value 3.1415 replaces B's current value of 3 h2 with the floating point number 3.1415.

12.2.1 defparam statement

Using the *defparam statement*, parameter values can be changed in any module instance throughout the design using the hierarchical name of the parameter. However, a defparam statement in a hierarchy under a generate scope or array of instances shall not change a parameter value outside that hierarchy. 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.

In the case of multiple defparams for a single parameter, the parameter takes the value of the last defparam statement encountered in the source text. When defparams are encountered in multiple source files, e.g., found by library searching, the defparam from which the parameter takes its value is undefined.

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 to use one of the two forms of module instance parameter value assignment. They are assignment by ordered list and assignment by name. The two types of module instance parameter value assignment shall not be mixed; parameter assignments to a particular module instance shall be entirely by order or entirely by name.

Module instance parameter value assignment by ordered list is similar in appearance to the assignment of delay values to gate instances and assignment by name is similar to connecting module ports by name. It supplies values for particular instances of a module to any parameters that have been specified in the definition of that module.

12.2.2.1 Parameter value assignment by ordered list

The order of the assignments in the module instance parameter value assignment by ordered list shall follow the order of declaration of the parameters within the module. It is not necessary to assign values to all of the parameters within a module when using this method. However, it is not possible to skip over a parameter.

Therefore, to assign values to a subset of the parameters declared within a module, the declarations of the parameters that make up this subset shall precede the declarations of the remaining parameters. An alternative is to assign values to all of the parameters, but to use the default value (the same value assigned in the declaration of the parameter within the module definition) for those parameters that do not need new values.

Example:

Consider the following example, where the parameters within module instance `mod_a` are changed during instantiation.

```

module m;
reg clk;
wire [0:4] out_c, in_c;
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);
// create an instance and set one parameter
vdff #(.delay(12)) mod_c(out_c, in_c, clk);
endmodule

module vdff (out, in, clk);
parameter size = 5, 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`. The construct `#(.delay(12))` assigns the parameter `delay` the value 12 in the instance of module `vdff` called `mod_c`.

12.2.2.2 Parameter value assignment by name

Parameter assignment by name consists of explicitly linking the parameter name and its new value. The name of the parameter shall be the name specified in the instantiated module.

It is not necessary to assign values to all of the parameters within a module when using this method. Only those parameters that are assigned new values need to be specified.

The parameter expression is optional so that the instantiating module can document the existence of a parameter without assigning anything to it. The parentheses are required and in this case the parameter retains its default value. Once a parameter is assigned a value, there shall not be another assignment to this parameter name.

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 variables specified in the definition of module B.

12.3.1 Port definition

The syntax for ports and a list of ports is given in Syntax 12-5.

```
list_of_ports ::= (From Annex A - A.1.4)
                ( port { , port } )
list_of_port_declarations ::=
    ( port_declaration { , port_declaration } )
    | ( )
port ::=
    [ port_expression ]
    . port_identifier ( [ port_expression ] )
port_expression ::=
    port_reference
    | { port_reference { , port_reference } }
port_reference ::=
    port_identifier
    | port_identifier { constant_expression ]
    | port_identifier [ range_expression ]
port_declaration ::=
    { attribute_instance } inout_declaration
    | { attribute_instance } input_declaration
    | { attribute_instance } output_declaration
```

Syntax 12-5 Syntax for port

12.3.2 List of ports

The port reference for each port in the list of ports at the top of each module declaration can be one of the following:

- A simple identifier or escaped identifier
- A bit-select of a vector declared within the module
- A part-select of a vector declared within the module
- A concatenation of any of the above

The port expression is optional because ports can be defined that do not connect to anything internal to the module. Once a port has been defined, there shall not be another port definition with this same name.

The first type of module port with only a `port_expression` is an implicit port. The second type is the explicit port. This explicitly specifies the `port_identifier` used for connecting module instance ports by name (see 12.3.6) and the `port_expression` which contains identifiers declared inside the module as described in 12.3.3. Use of named port connections shall not be used for implicit ports unless the `port_expression` is a simple `port_identifier`.

12.3.3 Port declarations

Each `port_expression` in the list of ports for the module declaration shall also be declared in the body of the module as one of the following port declarations: **input**, **output**, or **inout** (bidirectional). This is in addition to any other data type declaration for a particular port for example, a **reg** or **wire**. The syntax for port declarations is given in Syntax 12-6.

```

inout_declaration ::= (From Annex A - A.2.1.2)
    inout [ net_type ] [ signed ] [ range ] list_of_port_identifiers
input_declaration ::=
    input [ net_type ] [ signed ] [ range ] list_of_port_identifiers
output_declaration ::=
    output [ net_type ] [ signed ] [ range ]
        list_of_port_identifiers
    | output [ reg ] [ signed ] [ range ]
        list_of_port_identifiers
    | output reg [ signed ] [ range ]
        list_of_variable_port_identifiers
    | output [ output_variable_type ]
        list_of_port_identifiers
    | output output_variable_type
        list_of_variable_port_identifiers
list_of_port_identifiers ::= (From Annex A - A.2.3)
    port_identifier { , port_identifier }

```

Syntax 12-6 Syntax for port declarations

If a port declaration includes a net or variable type, then the port is considered completely declared and it is an error for the port to be declared again as a variable or net data type declaration. Because of this, all other aspects of the port shall be declared in such a port declaration, including the signed and range definitions if needed.

If a port declaration does not include a net or variable type, then the port can be again declared in a net or variable declaration. If the net or variable is declared as a vector, the range specification between the two declarations of a port shall be identical. Once a name is used in a port declaration it shall not be declared again in another port declaration or in a data type declaration.

NOTE—Implementations may limit maximum number of ports in a module definition, but they will at least be 256.

Example:

```

input  aport;    // First declaration - okay.
input  aport;    // Error - multiple declaration, port declaration
output aport;    // Error - multiple declaration, port declaration

```

The signed attribute can be attached either to a port declaration or to the corresponding net or reg declaration, or to both. If either the port or the net/reg is declared as signed, then the other shall also be considered signed.

Implicit nets shall be considered unsigned. Nets connected to ports without an explicit net declaration shall be considered unsigned, unless the port is declared as signed.

Example:

```

module test(a,b,c,d,e,f,g,h);
input [7:0] a;           // no explicit declaration - net is unsigned
input [7:0] b;
input signed [7:0] c;
input signed [7:0] d;    // no explicit net declaration - net is signed
output [7:0] e;          // no explicit declaration - net is unsigned
output [7:0] f;
output signed [7:0] g;
output signed [7:0] h;  // no explicit net declaration - net is signed

wire signed [7:0] b;    // port b inherits signed attribute from net decl.
wire [7:0] c;           // net c inherits signed attribute from port
reg signed [7:0] f;     // port f inherits signed attribute from reg decl.
reg [7:0] g;            // reg g inherits signed attribute from port

endmodule

module complex_ports ({c,d}, .e(f)); // Nets {c,d} receive the first
// port bits. Name 'f' is declared inside the module.
// Name 'e' is defined outside the module.
// Can't use named port connections of first port.

module split_ports (a[7:4], a[3:0]); // First port is upper 4 bits of
// 'a'.
// Second port is lower 4 bits of 'a'.
// Can't use named port connections because
// of part-select port 'a'.

module same_port (.a(i), .b(i)); // Name 'i' is declared inside the
// module as a inout port. Names 'a' and 'b' are
// defined for port connections.

module renamed_concat (.a({b,c}), f, .g(h[1]));
// Names 'b', 'c', 'f', 'h' are defined inside the module.
// Names 'a', 'f', 'g' are defined for port connections.
// Can use named port connections.

module same_input (a,a);
input a;           // This is legal. The inputs are tied together.

```

12.3.4 List of ports declarations

An alternate syntax which minimizes the duplication of data can be used to specify the ports of a module. Each module shall either be declared entirely with the list of ports syntax as described in 12.3.2 or entirely using the *list_of_port_declarations* as described in this section.

Each declared port provides the complete information about the port. The port's direction, width, net, or variable type, and whether the port is signed or unsigned is completely described. The same syntax for input, inout, and output declarations is used in the module header as would be used for the list of port style declaration, except the *list_of_port_declarations* is included in the module header rather than separately (after the ; which terminates the module header).

As an example, the module named test given in the previous example could alternatively be declared as:

Example:

```

module test (
  input [7:0] a,
  input signed [7:0] b, c, d, // multiple ports that share all
                                // attributes can be declared together
  output [7:0] e,             // every attribute of the declaration
                                // must be in the one declaration
  output signed reg [7:0] f, g,
  output signed [7:0] h) ;
  // It is illegal to redeclare any ports of the module in the body
  // of the module.
endmodule

```

The *port_reference* type of module port declaration shall not be done using *list_of_port_declarations* style of module declarations. Also ports declared using the *list_of_port_declarations* shall only be simple identifiers. They shall not be bit-selects, part-selects, or concatenations (as in the example *complex_ports*); nor can a port be split (as in the example *split_ports*); nor can they be named ports (as in the example *same_port*).

Designs may freely mix modules declared using each syntax; hence implementations desiring the above special cases of port declaration can be done using the first *list_of_ports* syntax.

12.3.5 Connecting module instance ports by ordered list

One method of making the connection between the port expressions listed in a module instantiation and the ports declared within the instantiated module is the ordered list that is, the ports expressions listed for the module instance shall be in the same order as the ports listed in the module declaration.

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;

  tranifl      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 **tranifl** gate **g1**.

12.3.6 Connecting module instance ports by name

The second way to connect module ports consists of explicitly linking the two names for each side of the connection, the port declaration name from the module declaration to the expression the name used in the module declaration, followed by the name used in the instantiating module. This compound name is then placed in the list of module connections. The port name shall be the name specified in the module declaration. The port name cannot be a bit-select, a part-select, or a concatenation of ports. If the module port declaration was implicit, the **port_expression** shall be a simple **port_idenfier** which is used as the port name. If the module port declaration was explicit, the explicit name is used as the name of port.

The port expression can be any valid expression.

The port expression is optional so that the instantiating module can document the existence of the port without connecting it to anything. The parentheses are required.

The two types of module port connections shall not be mixed; connections to the ports of a particular module instance shall be all by order or all by name.

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;

  tranifl      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.

Multiple module instance port connections are not allowed, e.g., the following example is illegal:

Example 3 This example shows illegal port connections.

```

module test;
  a ia (.i (a), .i (b), // illegal connection of input port twice.
        .o (c), .o (d), // illegal connection of output port twice.
        .e (e), .e (f)); // illegal connection of inout port twice.
endmodule

```

12.3.7 Real numbers in port connections

The `real` data type shall not be directly connected to a port. It shall be connected indirectly, as shown in the following example. The system functions `$realtobits` and `$bitstoreal` shall be used for passing the bit patterns across module ports. (See 17.8 for a description of these system tasks.)

Example:

```
module driver (net_r);  
    output net_r;  
    real r;  
    wire [64:1] net_r = $realtobits(r);  
endmodule  
  
module receiver (net_r);  
    input net_r;  
    wire [64:1] net_r;  
    real r;  
  
    initial assign r = $bitstoreal(net_r);  
  
endmodule
```

12.3.8 Connecting dissimilar ports

A port of a module can be viewed as providing a link or connection between two items (nets, regs, expressions, etc.) one internal to the module instance and one external to the module instance.

Examination of the port connection rules described in 12.3.9 will show that the item receiving the value through the port (the internal item for inputs, the external item for outputs) shall be a structural net expression. The item that provides the value can be any expression.

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.

12.3.9 Port connection rules

The following rules shall govern the way module ports are declared and the way they are interconnected.

12.3.9.1 Rule 1

An input or inout port shall be of type net.

12.3.9.2 Rule 2

Each port connection shall be a continuous assignment of source to sink, where one connected item shall be a signal source and the other shall be a signal sink. The assignment shall be a continuous assignment from source to sink for input or output ports. The assignment is a nonstrength reducing transistor connection for inout ports. Only nets or structural net expressions shall be the sinks in an assignment.

A *structural net expression* is a port expression whose operands can be the following:

- A scalar net
- A vector net
- A constant bit-select of a vector net
- A part-select of a vector net
- A concatenation of structural net expressions

The following external items shall not be connected to the output or inout ports of modules:

- Variables
- Expressions other than
 - i) A scalar net
 - ii) A vector net
 - iii) A constant bit-select of a vector net
 - iv) A part-select of a vector net
 - v) A concatenation of the expressions listed above

12.3.10 Net types resulting from dissimilar port connections

When different net types are connected through a module port, the nets on both sides of the port can take on the same type. The resulting net type can be determined as shown in Table 45. 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 **tri_{reg}**, any strength value specified for the tri_{reg} net shall apply to the simulated net.

12.3.10.1 Net type resolution rule

When the two nets connected by a port are of different net type, the resulting single net can be assigned one of the following:

- The dominating net type if one of the two nets is dominating, *or*
- The net type external to the module

When a dominating net type does not exist, the external net type shall be used.

12.3.10.2 Net type table

Table 45 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 **tri_{reg}**, any strength value specified for the tri_{reg} net applies to the simulated net.

Table 45—Net types resulting from dissimilar port connections

Internal net	External net							
	wire, tri	wand, triand	wor, trior	tri _{reg}	tri0	tri1	supply0	supply1
wire, tri	ext	ext	ext	ext	ext	ext	ext	ext
wand, triand	int	ext	warn	warn	warn	warn	ext	ext

Table 45—Net types resulting from dissimilar port connections *(continued)*

Internal net	External net							
	wire, tri	wand, triand	wor, trior	triereg	tri0	tri1	supply0	supply1
wor, trior	int	warn	ext	warn	warn	warn	ext	ext
triereg	int	warn	warn	ext	ext	ext	ext	ext
tri0	int	warn	warn	int	ext	warn	ext	ext
tri1	int	warn	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								

12.3.11 Connecting signed values via ports

The sign attribute shall not cross hierarchy. In order to have the signed type cross hierarchy, the signed keyword must be used in the object's declaration at the different levels of hierarchy. Any expressions on a port shall be treated as any other expression in an assignment. It shall be typed, sized, evaluated and the resulting value assigned to the object on the other side of the port using the same rules as an assignment.

12.4 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, generated 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 one or more root modules of which no instances have been created. This root or these parallel root modules make up one or more hierarchies in a *design description* or *description*. Inside any module, each module instance (including an arrayed or generated 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. Only non-recursively referenced automatic tasks and/or functions create visible branches that can be referenced. Recursively called tasks and functions, declared using the automatic keyword and recursively called from within the same task or function, do not create visible branches that can be referenced. See 10.2.1 and 10.3.1 for a discussion of automatic tasks and functions.

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 12.6 for a discussion of scope rules and 3.12 for a discussion of name spaces.

Any named Verilog object or *hierarchical name reference* can be referenced uniquely in its full form by concatenating the names of the modules, module instance names, tasks, functions, or named blocks that contain it. The period character shall be used to separate each of the names in the hierarchy, except for escaped identifiers embedded in the hierarchical name reference, which are followed by separators composed of white space and a period-character. The complete path name to any object shall start at a top-level (root) module. This path name can be used from any level in the hierarchy or from a parallel hierarchy. The first node name in a path name can also be the top of a hierarchy that starts at the level where the path is being used (which allows and enables downward referencing of items) with the exceptions of items of automatic tasks and automatic task item declarations. These declarations can not be accessed by their hierarchical names.

The syntax for hierarchical path names is given in Syntax 12-7.

```

escaped_hierarchical_identifier* ::= (From Annex A - A.9.3)
    escaped_hierarchical_branch
    [ { .simple_hierarchical_branch | .escaped_hierarchical_branch } ]
escaped_identifier ::=
    \ {Any_ASCII_character_except_white_space} white_space
hierarchical_identifier ::=
    simple_hierarchical_identifier
    | escaped_hierarchical_identifier
simple_hierarchical_identifier ::=
    simple_hierarchical_branch [ .escaped_identifier ]
simple_identifier ::= [ a-zA-Z_ ] { [ a-zA-Z0-9_$ ] }
simple_hierarchical_branch ::= (From Annex A - A.9.4)
    simple_identifier [ [ unsigned_number ] ]
    [ { .simple_identifier [ [ unsigned_number ] ] } ]
escaped_hierarchical_branch* ::=
    escaped_identifier [ [ unsigned_number ] ]
    [ { .escaped_identifier [ [ unsigned_number ] ] } ]
white_space ::= (From Annex A - A.9.5)
    space | tab | newline | eof

```

*The period in `escaped_hierarchical_identifier` and `escaped_hierarchical_branch` shall be preceded by `white_space`, but shall not be followed by `white_space`.

The period (.) in `simple_hierarchical_identifier` and `simple_hierarchical_branch` shall not be preceded or followed by `white_space`.

A `simple_identifier` and `arrayed_reference` shall start with an alpha or underscore () character, shall have at least one character, and shall not have any spaces.

/ End of file.

Syntax 12-7 Syntax for hierarchical path names

Examples:

Example 1 The code in this example defines a hierarchy of module instances and named blocks.

```

module mod (in);
input in;

always @(posedge in) begin : keep
  reg hold;
  hold = in;
end
endmodule

module wave;
reg stim1, stim2;

  cct a(stim1, stim2); // instantiate cct

initial begin :wave1
  #100 fork :innerwave
    reg hold;
  join
  #150 begin
    stim1 = 0;
  end
end
endmodule

module cct (stim1, stim2);
input stim1, stim2;

  // instantiate mod
  mod amod(stim1), bmod(stim2);
endmodule

```

Figure 32 illustrates the hierarchy implicit in this Verilog code.

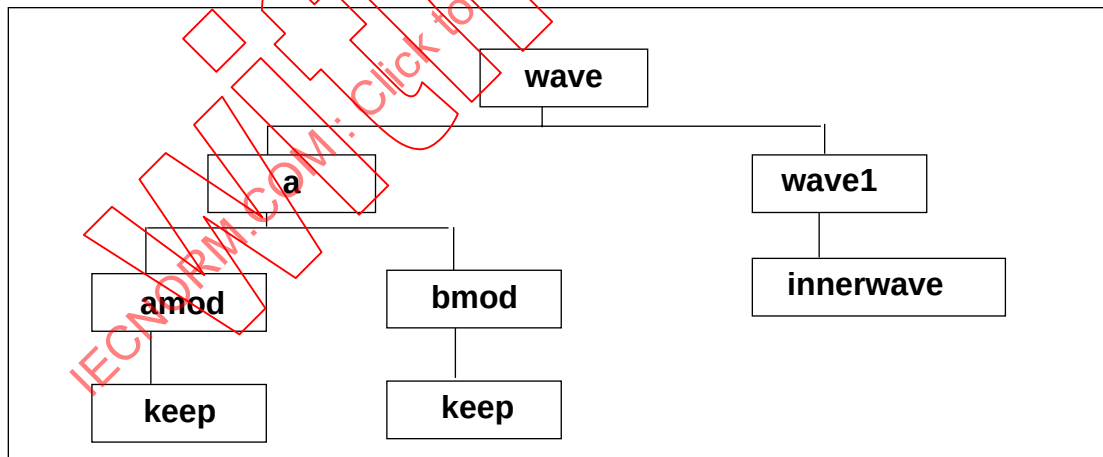


Figure 32—Hierarchy in a model

Figure 33 is a list of the hierarchical forms of the names of all the objects defined in the code.

wave	wave.a.bmod
wave.stim1	wave.a.bmod.in
wave.stim2	wave.a.bmod.keep
wave.a	wave.a.bmod.keep.hold
wave.a.stim1	wave.wave1
wave.a.stim2	wave.wave1.innerwave
wave.a.amod	wave.wave1.innerwave.hold
wave.a.amod.in	
wave.a.amod.keep	
wave.a.amod.keep.hold	

Figure 33—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

```

12.5 Upwards name referencing

The name of a module or module instance is sufficient to identify the module and its location in the hierarchy. A lower-level module can reference items in a module above it in the hierarchy. Variables can be referenced if the name of the higher-level module or its instance name is known. For tasks, functions, and named blocks, Verilog shall look in the enclosing module for the name until it is found or until the root of the hierarchy is reached. It shall only search in higher enclosing modules for the name, not instances.

The syntax for an upward reference is given in Syntax 12-8.

```

upward_name_reference ::=
  module_identifier.item_name
item_name ::=
  function_identifier
  block_identifier
  net_identifier
  parameter_identifier
  port_identifier
  task_identifier
  variable_identifier

```

Syntax 12-8 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, except for escaped identifiers embedded in the hierarchical name reference, which are followed by separators composed of white space and a period-character.

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

```

12.6 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. If the item is a variable, it shall stop at a module boundary; if the item is a task, function, or named block it continues to search higher-level modules until

found. 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.

If an identifier is referenced with a hierarchical name, the path can start with an module name, instance name, task, function, or named block. The names shall be searched first at the current level, then in higher-level modules until found. Since both module names and instance names can be used, precedence is given to instance names if there is a module named the same as an instance name.

Because of the upward searching, path names which are not strictly on a downward path can be used.

Example:

Example 1 In Figure 34, 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.

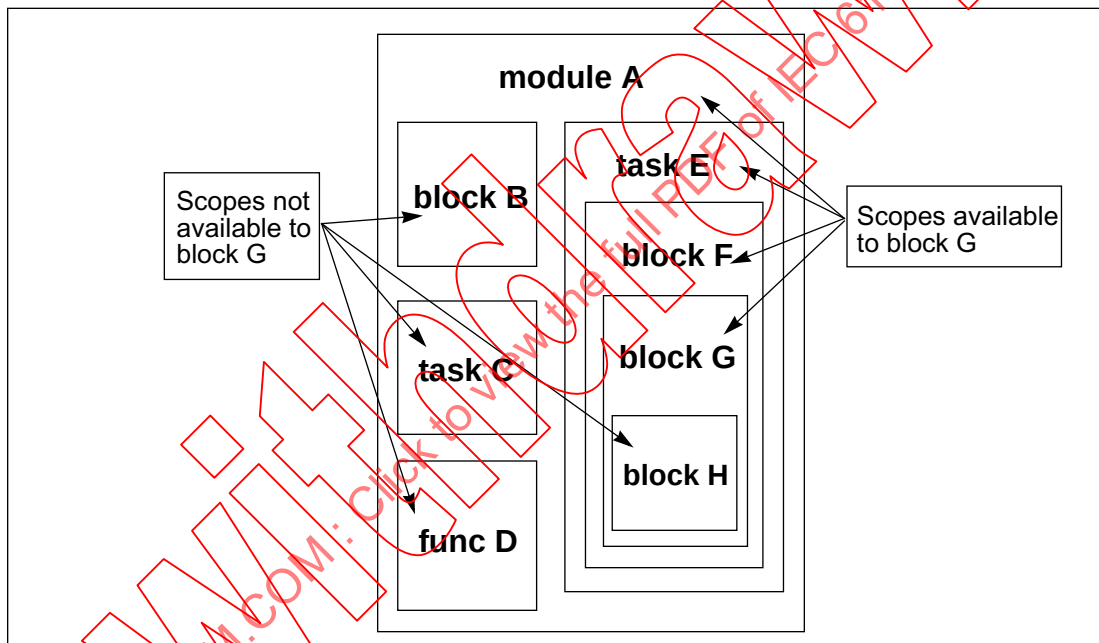


Figure 34—Scopes available to upward name referencing

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
```

13. Configuring the contents of a design

13.1 Introduction

To facilitate both the sharing of Verilog designs between designers and/or design groups, and the repeatability of the exact contents of a given simulation (or other tool) session, the concept of *configurations* is used in the Verilog language. A configuration is simply an explicit set of rules to specify the exact source description to be used to represent each instance in a design. The operation of selecting a source representation for an instance is referred to as *binding* the instance.

The example below shows a simple configuration problem.

Example:

file top.v	file adder.v	file adder.vg
module top();	module adder(...);	module adder(...);
adder a1(...); // rtl adder description		
	// gate-level adder description	
adder a2(...);	...	...
endmodule	endmodule	endmodule

Consider using the rtl adder description in adder.v for instance a1 in module top and the gate-level adder description in adder.vg for instance a2. In order to specify this particular set of instance bindings and to avoid having to change the source description to specify a new set, a configuration can be used.

```
config cfg1; // specify rtl adder for top.a1, gate-level adder for top.a2
  design rtlLib.top;
  default liblist rtlLib;
  instance top.a2 liblist gateLib;
endconfig
```

The elements of a *config* are explained in subsequent sections, but this simple example illustrates some important points about *configs*. As evidenced by the **config-endconfig** syntax, the config is a design element, similar to a module, which exists in the Verilog namespace. The config contains a set of rules which are applied when searching for a source description to *bind* to a particular instance of the design.

A Verilog design description starts with a top-level module (or modules), which is not instantiated elsewhere in the design. From this module's source description, the instantiated modules (or children) are found, and then the source descriptions for the module definitions of these subinstances shall be located, and so on until every instance in the design is mapped to a source description.

13.1.1 Library notation

In order to map a Verilog instance to a source description, the concept of a symbolic *library*, which is simply a logical collection of design elements (such as modules, macromodules, primitives, or configs) can be used. These design elements can be referred to as *cells*. The cell name shall be the same as the name of the module/macromodule/primitive/config being processed. Syntax 13-1 specifies a cell from a given library.

```
library_cell ::=  
    [library_identifier.]cell_identifier[:config]
```

Syntax 13-1 Syntax for cell

This notation gives a symbolic method of referring to source descriptions; the method of mapping source descriptions into libraries is shown in greater detail in 13.2.1. The optional `:config` extension shall be used explicitly to refer to a config in the case where a config has the same name as a module/macromodule/primitive.

For the purposes of this example, suppose the files `top.v` and `adder.v`, the `rtl` descriptions, have been mapped into the library `rtlLib`, and the file `adder.vg`, the gate-level description of the `adder`, has been mapped into the library `gateLib`. The actual mechanism for mapping source descriptions to libraries is detailed in 13.2.

13.1.2 Basic configuration elements

The **design** statement in `config cfg1` of the first example of 13.1 specifies the top-level module in the design and what source description is to be used. In this example, the `rtlLib.top` notation indicates the top-level module description shall be taken from `rtlLib`. Since `top.v` and `adder.v` were mapped to this library, the actual description for the module is known to come from `top.v`.

The **default** statement coupled with the **liblist** clause specifies, by default, all subinstances of `top` (i.e., `top.a1` and `top.a2`) shall be taken from `rtlLib`, which means the descriptions in `top.v` and `adder.v`, which were mapped to this library, shall be used. For a basic design, which can be completely `rtl`, this can be sufficient to specify completely the binding for the entire design. However, here the `top.a2` instance of `adder` to the gate-level description shall be bound.

The **instance** statement specifies, for the particular instance `top.a2`, the source description shall be taken from `gateLib`. The instance statement overrides the default rule for this particular instance. Since `adder.vg` was mapped to `gateLib`, this statement dictates the gate-level description in `adder.vg` be used for instance `top.a2`.

13.2 Libraries

As mentioned in the previous section, a library is a logical collection of cells which are mapped to particular source description files. The symbolic `lib.cell[:config]` notation supports the separate compilation of source files by providing a file system-independent name to refer to source descriptions when instances in a design are bound. It also allows multiple tools, which can have different invocation use-models, to share the same configuration.

13.2.1 Specifying libraries - the library map file

When parsing a source description file (or files), the parser shall first read the library mapping information from a pre-defined file prior to reading any source files. The name of this file and the mechanism for reading it shall be tool-specific, but all compliant tools shall provide a mechanism to specify one or more library mapping files to be used for a particular invocation of the tool. If multiple mapping files are specified, then they shall be read in the order in which they are specified.

For the purposes of this discussion, assume the existence of a file named `lib.map` in the current working directory, which is automatically read by the parser prior to parsing any source files specified on the command line. The syntax for declaring a library in the library map file is shown in Syntax 13-2.

```

escaped_hierarchical_identifier* ::= (From Annex A - A.1.1)
library_text ::=
    { library_descriptions }
library_descriptions ::=
    library_declaration
    | include_statement
    | config_declaration
library_declaration ::=
    library library_identifier file_path_spec [ { , file_path_spec } ]
    [ -incdir file_path_spec [ { , file_path_spec } ] ;
file_path_spec ::=
    file_path
include_statement ::=
    include <file_path_spec> ;

```

*The period in escaped_hierarchical_identifier and escaped_hierarchical_branch shall be preceded by white_space, but shall not be followed by white_space.

Syntax 13-2 Syntax for declaring library in the library map file

NOTES

1 The file_path uses file system-specific notation to specify an absolute or relative path to a particular file or set of files. The following shortcuts/wildcards can be used:

- ? single character wildcard (matches any single character)
- * multiple character wildcard (matches any number of characters in a directory/file name)
- ... hierarchical wildcard (matches any number of hierarchical directories)
- .. specifies the parent directory
- . specifies the directory containing the **lib.map**

Paths which end in / shall include all files in the specified directory. Identical to /*.

Paths which do not begin with / are relative to the directory in which the current lib.map file is located.

2 The paths ./*.v and *.v are identical and both specify all files with a .v suffix in the current directory.

Any file encountered by the compiler which does not match any library's file_path specification shall by default be compiled into a library named **work**.

To perform the library mapping discussed in the example in 13.1, use the following library definitions in the lib.map file:

```

library rtlLib *.v;           // matches all files in the current directory with a .v suffix
library gateLib ./*.vg;      // matches all files in the current directory with a .vg suffix

```

13.2.1.1 File path resolution

If a file name potentially matches multiple file path specifications, the path specifications shall be resolved in the following order:

- a) File path specifications which end with an explicit filename
- b) File path specifications which end with a wildcarded filename
- c) File path specifications which end with a directory

If a file name matches path specifications in multiple library definitions (after the above resolution rules have been applied), it shall be an error.

Using these rules with the library definitions in the `lib.map` file, all source files encountered by the parser/compiler can be mapped to a unique library. Once the source descriptions have been mapped to libraries, the cells defined therein are available for binding.

NOTE Tool implementers may find it convenient to provide a command-line argument to explicitly specify the library into which the file being parsed is to be mapped, which shall override any library definitions in the `lib.map` file. If these libraries do not exist in the `lib.map` file, they can only be accessed via an explicit config.

If multiple cells with the same name map to the same library, then the LAST cell encountered shall be written to the library. This is to support a separate-compile use-model (see 13.4.3), where it is assumed encountering a cell after it has previously been compiled is intended to be a recompiling of the cell. In the case where multiple modules with the same name are mapped to the same library in a single invocation of the compiler, then a warning message shall be issued.

13.2.2 Using multiple library mapping files

In addition to specifying library mapping information, a `lib.map` file can also include references to other `lib.map` files. The **include** command is used to insert the entire contents of a library mapping file in another file during parsing. The result is as though the contents of the included mapping file appear in place of the **include** command.

The syntax of a `lib.map` file is limited to library specifications, include statements, and standard Verilog comment syntax. Syntax 13-3 shows the syntax for the **include** command.

```
include_statement ::= (From Annex A - A.1.1)  
include <file_path_spec> ;
```

Syntax 13-3 Syntax for include command

If the file path specification, whether in an include or library statement, describes a relative path, it shall be relative to the location of the file which contains the file path. Library providers shall include a local library mapping file in addition to the source contents of the library. Individual users can then simply include the provider's library mapping file in their own map file to gain access to the contents of the provided library.

13.2.3 Mapping source files to libraries

For each cell definition encountered during parsing/compiling, the name of the source file being parsed is compared to the file path specifications of the library declarations in all of the library map files being used. The cell is mapped into the library whose file path specification matches the source file name.

13.3 Configurations

As mentioned in the introduction of this chapter, a configuration is simply a set of rules to apply when searching for library cells to which to bind instances. The syntax for configurations is shown in 13.3.1.

13.3.1 Basic configuration syntax

The configuration syntax is shown in Syntax 13-4.

```

config_declaration ::= (From Annex A - A.1.2)
    config config_identifier ;
    design_statement
    { config_rule_statement }
    endconfig
design_statement ::=
    design { [library_identifier.] cell_identifier } ;
config_rule_statement ::=
    default_clause liblist_clause
    | inst_clause liblist_clause
    | inst_clause use_clause
    | cell_clause liblist_clause
    | cell_clause use_clause

```

Syntax 13-4 Syntax for configuration

13.3.1.1 Design statement

The **design** statement names the library and cell of the top-level module or modules in the design hierarchy configured by the config. There shall be one and only one design statement, but multiple top-level modules can be listed in the design statement. The cell or cells identified can not be configurations themselves. It is possible the design identified can have the same name as configs, however.

The **design** statement shall appear before any config rule statements in the config.

If the library identifier is omitted, then the library which contains the config shall be used to search for the cell.

13.3.1.2 The default clause

The syntax for the **default** clause is specified in Syntax 13-5.

```

default_clause ::= (From Annex A - A.1.2)
    default

```

Syntax 13-5 Syntax for default clause

The **default** clause selects all instances which do not match a more specific selection clause. The **use** expansion clause (see 13.3.1.6) can not be used with a **default** selection clause. For other expansion clauses, there can not be more than one **default** clause which specifies the expansion clause.

For simple design configurations, it might be sufficient to specify a **default liblist** (see 13.3.1.5).

13.3.1.3 The instance clause

The **instance** clause is used to specify the specific instance to which the expansion clause shall apply. The syntax for the **instance** clause is specified in Syntax 13-6.

```
inst_clause ::= (From Annex A - A.1.2)
             instance inst_name
inst_name ::=
             topmodule_identifier{.instance_identifier}
```

Syntax 13-6 Syntax for instance clause

The instance name associated with the **instance** clause is a Verilog hierarchical name, starting at the top-level module of the config (i.e., the name of the cell in the **design** statement).

13.3.1.4 The cell clause

The **cell** selection clause names the cell to which it applies. The syntax for the **cell** clause is specified in Syntax 13-7.

```
cell_clause ::= (From Annex A - A.1.2)
             cell [ library_identifier.]cell_identifier
```

Syntax 13-7 Syntax for cell clause

If the optional library name is specified then the selection rule applies to any instance which is bound or is under consideration for being bound to the selected library and cell. It is an error if a library name is included in a **cell** selection clause and the corresponding expansion clause is a library list expansion clause.

13.3.1.5 The liblist clause

The **liblist** clause defines an ordered set of libraries to be searched to find the current instance. The syntax for the **liblist** clause is specified in Syntax 13-8.

```
liblist_clause ::= (From Annex A - A.1.2)
                liblist [{library_identifier}]
```

Syntax 13-8 Syntax for liblist clause

liblists are inherited hierarchically downward as instances are bound. When searching for a cell to bind to the current unbound instance, and in the absence of an applicable binding expansion clause, the specified library list is searched in the specified order.

The current library list is selected by the selection clauses. If no library list clause is selected, or the selected library list is empty, then the library list contains the single name which is the library in which the cell containing the unbound instance is found (i.e., the parent cell's library).

13.3.1.6 The use clause

The **use** clause specifies a specific binding for the selected cell. The syntax for the **use** clause is specified in Syntax 13-9.

```
use_clause ::= (From Annex A - A.1.2)
            use [library_identifier.]cell_identifier[:config]
```

Syntax 13-9 Syntax for use clause

A **use** clause can only be used in conjunction with an **instance** or **cell** selection clause. It specifies the exact library and cell to which a selected cell or instance is bound.

The **use** clause has no effect on the current value of the library list. It can be common in practice to specify multiple config rule statements, one of which specifies a binding and the other of which specifies a library list.

If the lib.cell being referred to by the **use** clause is a config which has the same name as a module/macro-module/primitive in the same library, then the optional **:config** suffix can be added to the lib.cell to specify the config explicitly.

If the library name is omitted, the library shall be inherited from the parent cell.

NOTE—The binding statement can create situations where the unbound instance's module name and the cell name to which it is bound are different.

13.3.2 Hierarchical configurations

For situations where it is desirable to specify a special set of configuration rules for a subsection of a design, it is possible to bind a particular instance directly to a configuration using the binding clause:

```
instance top.a1.foo use lib1.foo:config;
// bind to the config foo in library lib1
```

specifies the instance `top.a1.foo` is to be replaced with the design hierarchy specified by the configuration `lib1.foo:config`. The **design** statement in `lib1.foo:config` shall specify the actual binding for the instance `top.a1.foo`, and the rules specified in the config shall determine the configuration of all other subinstances under `top.a1.foo`.

It shall be an error for an instance clause to specify a hierarchical path to an instance which occurs within a hierarchy specified by another config.

```
config bot;
  design lib1.bot;
  default liblist lib1 lib2;
  instance bot.a1 liblist lib3;
endconfig

config top;
  design lib1.top;
  default liblist lib2 lib1;
  instance top.bot use lib1.bot:config;
  instance top.bot.a1 liblist lib4;
  // ERROR - can't set liblist for top.bot.a1 from this config
endconfig
```


13.4 Using libraries and configs

The following subclause describes potential use-models for referencing configs on the command line. It is included for clarification purposes.

The traditional Verilog simulation use-model takes a file-based approach, where the source descriptions for all cells in the design are specified on the command line for each invocation of the tool. With the advent of compiled-code simulators, the configuration mechanism shall also support a use-model which allows for the source files to be pre-compiled and then for the pre-compiled design objects to be referenced on the command line. This subclause shall explain how configurations can be used in both of these scenarios.

13.4.1 Precompiling in a single-pass use-model

The single-pass use-model is the traditional use-model with which most users are familiar. In this use-model, all of the source description files shall be provided to the simulator via the command line, and only these source descriptions can be used to bind cell instances in the current design. A precompiling strategy in this scenario actually parses every cell description provided on the command line and maps it into the library without regard to whether the cell actually is used in the design. The tool can optionally check to see if the cell already exists in the library, and if it is up-to-date (i.e. the source description has not changed since the last time the cell was compiled) the tool can skip recompiling the cell. After all cells on the command line have been compiled, then the tool can locate the top-level cell (discussed in Section 12), and proceed down the hierarchy, binding each instance as it is encountered in the hierarchy.

NOTE With this use-model it is not necessary for library objects to persist from one tool invocation to another (although for performance considerations it is recommended they do).

13.4.2 Elaboration-time compiling in a single-pass use-model

An alternate strategy which can be used with a single-pass tool is to parse the source files only to find the top-level module(s), without actually compiling anything into the library during this scanning process. Once the top-level module(s) has been found, then it can be compiled into the library, and the tool can proceed down the hierarchy, only compiling the source descriptions necessary to bind the design successfully. Based on the binding rules in place, only the source files which match the current library specification need to be parsed to find the current cell's source description to compile. As with the precompiled single-pass use-model, it is not necessary for library cells to persist from one invocation to another using this strategy.

13.4.3 Precompiling using a separate compilation tool

When using a separate compilation tool, it is essential library cells persist, and the compiled forms shall therefore exist somewhere in the file system. The exact format and location for holding these compiled forms shall be vendor/tool-specific. Using this separate compiler strategy, the source descriptions shall be parsed and compiled into the library using one or more invocations of the compiler tool. The only restriction is all cells in a design shall be precompiled prior to binding the design (typically via an invocation of a separate tool). Using this strategy, the tool which actually does the binding only needs to be told the top-level module(s) of the design to be bound, and then it shall use the precompiled form of the cell description(s) from the library to determine the subinstances and descend hierarchically down the design binding each cell as it is located.

13.4.4 Command line considerations

In each of the three preceding strategies, the binding rules can either be specified via a config, or the default rules (from the library map file) can be used. In the single-pass use-models, the config can be specified by including its source description file on the command line. In the case where the config includes a design statement, then the specified cell shall be the top-level module, regardless of the presence of any uninstantiated cells in the rest of the source files. When using a separate compilation tool, the tool which actually does the binding only needs to be given the *lib.cell* specification for the top-level cell(s) and/or the config to be used. In this strategy, the config itself shall also be precompiled.

13.5 Configuration examples

Consider the following set of source descriptions:

Example:

```

file top.v          file adder.v          file adder.vg          file lib.map
module top(...);  module adder(...);  module adder(...);  library rtlLib top.v;
...                ... // rtl                ... // gate-level  library aLib adder.*;
adder a1(...);     foo f1(...);     foo f1(...);        library gateLib
adder a2(...);     foo f2(...);     foo f2(...);        adder.vg;
endmodule        endmodule                endmodule
module foo(...);  module foo(...);  module foo(...);
... // rtl         ... // rtl         ... // gate-level
endmodule        endmodule                endmodule

```

All of the examples in this section shall assume the `top.v`, `adder.v` and `adder.vg` files get compiled with the given `lib.map` file. This yields the following library structure:

```

rtlLib.top // from top.v
rtlLib.foo // from top.v
aLib.adder // from adder.v
aLib.foo // rtl from adder.v
gateLib.adder // from adder.vg
gateLib.foo // from adder.vg

```

13.5.1 Default configuration from library map file

With no configuration, the libraries are searched according to the library declaration order in the library map file. This means all instances of module `adder` shall use `aLib.adder` (since `aLib` is the first library specified which contains a cell named `adder`), and all instances of module `foo` shall use `rtlLib.foo` (since `rtlLib` is the first library which contains `foo`).

13.5.2 Using the default clause

To always use the `foo` definition from file `adder.v`, use the following simple configuration:

```

config cfg1;
  design rtlLib.top
  default liblist aLib rtlLib;
endconfig

```

The **default liblist** statement overrides the library search order in the `lib.map` file, so `aLib` is always searched before `rtlLib`. Since the `gateLib` library is not included in the `liblist`, the gate-level descriptions of `adder` and `foo` shall not be used.

To use the gate-level representations of `adder` and `foo`, add to the config as follows:

```

config cfg2;
  design rtlLib.top
  default liblist gateLib aLib rtlLib;
endconfig

```

This shall cause the gate representation always to be taken before the `rtl` representation, using the module definitions for `adder` and `foo` from `adder.vg`. The `rtl` view of `top` shall be taken since there is no gate representation available.

13.5.3 Using the cell clause

To modify the config to use the `rtl` view of `adder` and the gate-level representation of `foo` from `gateLib`:

```
config cfg3;  
  design rtlLib.top  
  default liblist alib rtlLib;  
  cell foo use gateLib.foo;  
endconfig
```

The cell clause selects all cells named `foo` and explicitly binds them to the gate representation in `gateLib`.

13.5.4 Using the instance clause

To modify the config so the `top.a1 adder` (and its descendants) use the gate representation and the `top.a2 adder` (and its descendants) use the `rtl` representation from `alib`:

```
config cfg4  
  design rtlLib.top  
  default liblist gateLib rtlLib;  
  instance top.a2 liblist alib;  
endconfig
```

Since the `liblist` is inherited, all of the descendants of `top.a2` inherit its `liblist` from the instance selection clause.

13.5.5 Using a hierarchical config

Now suppose all this work has only been on the `adder` module by itself and a config which uses the `rtlLib.foo` cell for `f1`, and the `gateLib.foo` cell for `f2` has already been developed. Then use:

```
config cfg5;  
  design alib.adder;  
  default liblist gateLib alib;  
  instance adder.f1 liblist rtlLib;  
endconfig
```

To use this configuration `cfg5` for the `top.a2` instance of `adder` and take the full default `alib` `adder` for the `top.a1` instance, use the following config:

```
config cfg6;  
  design rtlLib.top;  
  default liblist alib rtlLib;  
  instance top.a2 use work.cfg5:config  
endconfig
```

The binding clause specifies the `work.cfg5:config` configuration is to be used to resolve the bindings of instance `top.a2` and its descendants. It is the design statement in config `cfg5` which defines the exact binding for the `top.a2` instance itself. The rest of `cfg5` defines the rules to bind the descendants of `top.a2`. Notice the instance clause in `cfg5` is relative to its own top-level module, `adder`.

13.6 Displaying library binding information

It shall be possible to display the actual library binding information for module instances during simulation. The format specifier %l or %L shall print out the `library.cell` binding information for the module instance containing the display (or other textual output) command. This is similar to the %m format specifier which prints out the hierarchical path name of the module containing it.

It shall also be able to use the VPI interface to display the binding information. The following new `vpiProperties` shall exist for objects of type `vpiModule`:

`vpiUseBinding` - the `library.cell` binding information for a module instance
`vpiLibrary` - the library name into which the module was compiled
`vpiCell` - the name of the cell bound to the module instance
`vpiConfig` - the `library.cell` name of the config controlling the binding of the module instance

These properties shall be of `String` type, similar to the `vpiName` and `vpiFullName` properties.

13.7 Library mapping examples

In the absence of a configuration, it is possible to perform basic control of the library searching order when binding a design.

When a config is used, the config overrides the rules specified here.

13.7.1 Using the command line to control library searching

In the absence of a configuration, it shall be necessary for all compliant tools to provide a mechanism of specifying a library search order on the command line which overrides the default order from the library mapping file. This mechanism shall include specification of library names only, with the definitions of these libraries to be taken from the library mapping file.

NOTE It is recommended all compliant tools use "-L <library_name>" to specify this search order.

13.7.2 File path specification examples

Example:

Given the following set of files:

```
/proj/lib1/rtl/a.v
/proj/lib2/gates/a.v
/proj/lib1/rtl/b.v
/proj/lib2/gates/b.v
```

From the `/proj` library, the following absolute `file_path_specs` are resolved as shown:

```
/proj/lib*/*/a.v =/proj/lib1/rtl/a.v, /proj/lib2/gates/a.v
.../a.v =/proj/lib1/rtl/a.v, /proj/lib2/gates/a.v
/proj/.../b.v =/proj/lib1/rtl/b.v, /proj/lib2/gates/b.v
.../rtl/*.v =/proj/lib1/rtl/a.v, /proj/lib1/rtl/b.v
```

From the /proj/lib1 directory, the following relative file_path_specs are resolved as shown:

```
../lib2/gates/*.v = /proj/lib2/gates/a.v, /proj/lib2/gates/b.v  
./rtl/?..v = /proj/lib1/rtl/a.v, /proj/lib1/rtl/b.v  
./rtl/ = /proj/lib1/rtl/a.v, /proj/lib1/rtl/b.v
```

13.7.3 Resolving multiple path specifications

Example:

```
library lib1 "/proj/lib1/foo*.v";  
library lib2 "/proj/lib1/foo.v";  
library lib3 "../lib1/";  
library lib4 "/proj/lib1/*ver.v";
```

When evaluated from the directory /proj/tb directory, the following source files shall map into the specified library:

```
../lib1/foobar.v - lib1 // potentially matches lib1 and lib3. Since lib1 includes  
a filename and lib3 only specifies a directory; lib1 takes  
precedence  
/proj/lib1/foo.v - lib2 // takes precedence over lib1 and lib3 path specifications  
/proj/lib1/bar.v - lib3  
/proj/lib1/barver.v - lib4 // takes precedence over lib3 path specification  
/proj/lib1/foover.v - ERROR // matches lib1 and lib4  
/test/tb/tb.v - work // does not match any library specifications.
```

14. Specify blocks

Two types of HDL constructs are often used to describe delays for structural models such as ASIC cells. They are as follows:

- *Distributed delays*, which specify the time it takes events to propagate through gates and nets inside the module (see 7.14)
- *Module path delays*, which describe the time it takes an event at a source (input port or inout port) to propagate to a destination (output port or inout port)

This clause describes how paths are specified in a module and how delays are assigned to these paths.

14.1 Specify block declaration

A block statement called the *specify block* is the vehicle for describing paths between a source and a destination and for assigning delays to these paths. The syntax for specify block is shown in Syntax 14-1.

```
specify_item ::= (From Annex A - A.7.1)
specparam_declaration
| pulsestyle_declaration
| showcanceled_declaration
| path_declaration
| system_timing_check
```

Syntax 14-1 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 Clause 15).

The paths described in the specify block, called *module paths*, pair a signal source with a signal destination. The source may be unidirectional (an input port) or bidirectional (an inout port) and is referred to as the *module path source*. Similarly, the destination may be unidirectional (an output port) or bidirectional (an inout port) and is referred to as the *module path destination*.

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 3.11.3. The line following the declarations of specify parameters describes a module path and assigns delays to that module path. The specify parameters determine the delay assigned to the module path. Specifying module paths is presented in 14.2. Assigning delays to module paths is discussed in 14.3. The line preceding the keyword **endspecify** instantiates one of the system timing checks, which are discussed further in Section 15.

14.2 Module path declarations

There are two steps required to set up module path delays in a specify block:

- Describe the module paths
- Assign delays to those paths (see 14.3)

The syntax of the module path declaration is described in Syntax 14-2.

```
path_declaration ::= (From Annex A - A.7.2)
    simple_path_declaration ;
    | edge_sensitive_path_declaration ;
    | state_dependent_path_declaration ;
```

Syntax 14-2 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 35 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.

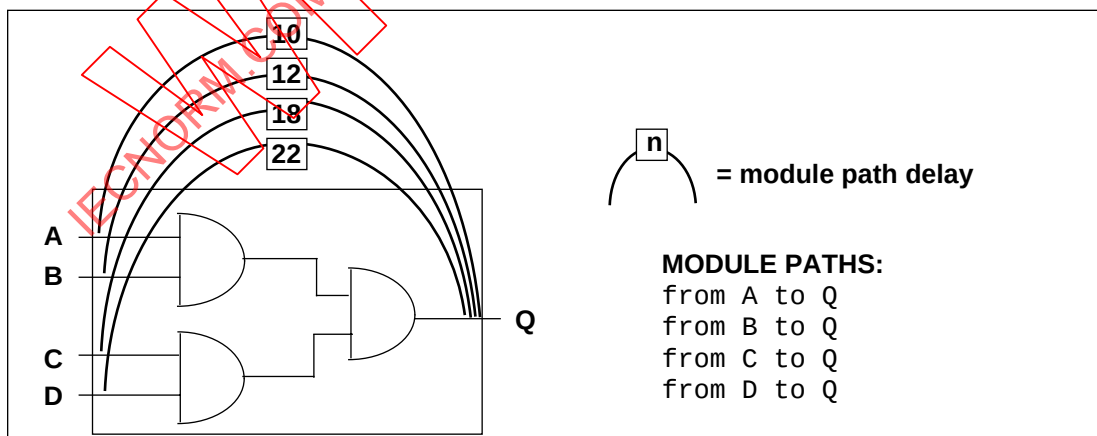


Figure 35—Module path delays

14.2.1 Module path restrictions

Module paths have the following restrictions:

- The module path source shall be a net that is connected to a module input port or inout port.
- The module path destination shall be a net or variable that is connected to a module output port or inout port.
- The module path destination shall have only one driver inside the module.

14.2.2 Simple module paths

The syntax for specifying a simple module path is given in Syntax 14-3.

```

simple_path_declaration ::= (From Annex A - A.7.2)
    parallel_path_description = path_delay_value
    | full_path_description = path_delay_value
parallel_path_description ::=
    ( specify_input_terminal_descriptor [ polarity_operator ] =>
      specify_output_terminal_descriptor )
full_path_description ::=
    ( list_of_path_inputs [ polarity_operator ] *> list_of_path_outputs )
list_of_path_inputs ::=
    specify_input_terminal_descriptor { , specify_input_terminal_descriptor }
list_of_path_outputs ::=
    specify_output_terminal_descriptor { , specify_output_terminal_descriptor }
specify_input_terminal_descriptor ::= (From Annex A - A.7.3)
    input_identifier
    | input_identifier [ constant_expression ]
    | input_identifier [ range_expression ]
specify_output_terminal_descriptor ::=
    output_identifier
    | output_identifier [ constant_expression ]
    | output_identifier [ range_expression ]
input_identifier ::=
    input_port_identifier | inout_port_identifier
output_identifier ::=
    output_port_identifier | inout_port_identifier
polarity_operator ::= (From Annex A - A.7.4)
    + | -

```

Syntax 14-3 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 14.2.5 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;
```

14.2.3 Edge-sensitive paths

When a module path is described using an edge transition at the source, it is called an *edge-sensitive path*. The edge-sensitive path construct is used to model the timing of input to output delays, which only occur when a specified edge occurs at the source signal.

The syntax of the edge-sensitive path declaration is shown in Syntax 14-4.

```
edge_sensitive_path_declaration ::= (From Annex A - A.7.4)  
    parallel_edge_sensitive_path_description = path_delay_value  
    | full_edge_sensitive_path_description = path_delay_value  
parallel_edge_sensitive_path_description ::=  
    ( [ edge_identifier ] specify_input_terminal_descriptor =>  
      specify_output_terminal_descriptor [ polarity_operator ] : data_source_expression )  
full_edge_sensitive_path_description ::=  
    ( [ edge_identifier ] list_of_path_inputs *>  
      list_of_path_outputs [ polarity_operator ] : data_source_expression )  
data_source_expression ::=  
    expression  
edge_identifier ::=  
    posedge | negedge
```

Syntax 14-4 Syntax 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 the bit-select of a vector output or inout port. For full connections (***>**), the destination shall be a list of one or more of the vector or scalar output and inout ports, and bit-selects or part-selects of vector output and inout ports. Refer to 14.2.5 for a description of parallel paths and full connection paths.

The data source expression is an arbitrary expression, which serves as a description of the flow of data to the path destination. This arbitrary data path description does not affect the actual propagation of data or events through the model; how an event at the data path source propagates to the destination depends on the internal logic of the module. The polarity operator describes whether the data path is inverting or noninverting.

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`.

14.2.4 State-dependent paths

A *state-dependent path* makes it possible to assign a delay to a module path that affects signal propagation delay through the path only if specified conditions are true.

A state-dependent path description includes the following items:

- A conditional expression that, when evaluated true, enables the module path
- A module path description
- A delay expression that applies to the module path

The syntax for the state-dependent path declaration is shown in Syntax 14-5.

```
state_dependent_path_declaration ::= (From Annex A - A.7.4)
    if ( module_path_expression ) simple_path_declaration
    | if ( module_path_expression ) edge_sensitive_path_declaration
    | ifnone simple_path_declaration
```

Syntax 14-5 Syntax of state-dependent paths

14.2.4.1 Conditional expression

The operands in the conditional expression shall be constructed from the following:

- Scalar or vector module input ports or inout ports or their bit-selects or part-selects
- Locally defined variables or nets or their bit-selects or part-selects
- Compile time constants (constant numbers and specify parameters)

Table 46 contains a list of valid operators that may be used in conditional expressions:

Table 46—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.

14.2.4.2 Simple state-dependent paths

If the path description of a state-dependent path is a simple path, then it is called a *simple state-dependent path*. The simple path description is discussed in 14.2.2.

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.

14.2.4.3 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 14.2.3.

Different delays can be assigned to the same edge-sensitive path as long as the following criteria are met:

- The edge, condition, or both make each declaration unique.
- The port is referenced in the same way in all path declarations (entire port, bit-select, or part-select).

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
```

14.2.4.4 The ifnone condition

The **ifnone** keyword is used to specify a default state-dependent path delay when all other conditions for the path are false. The **ifnone** condition shall specify the same module path source and destination as the state-dependent module paths. The following rules apply to module paths specified with the **ifnone** condition:

- Only simple module paths may be described with an **ifnone** condition.
- The state-dependent paths that correspond to the **ifnone** path may be either simple module paths or edge-sensitive paths.
- If there are no corresponding state-dependent module paths to the **ifnone** module path, then the **ifnone** module path shall be treated the same as an unconditional simple module path.
- It is illegal to specify both an **ifnone** condition for a module path and an unconditional simple module path for the same module path.

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);

```

14.2.5 Full connection and parallel connection paths

The operator ***>** shall be used to establish a *full connection* between source and destination. In a full connection, every bit in the source shall connect to every bit in the destination. The module path source need not have the same number of bits as the module path destination.

The full connection can handle most types of module paths, 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 14.2.6)

The operator **=>** shall be used to establish a *parallel connection* between source and destination. In a parallel connection, each bit in the source shall connect to one corresponding bit in the destination. Parallel module paths can be created only between sources and destinations that contain the same number of bits.

Parallel connections are more restrictive than full connections. They only connect one source to one destination, where each signal contains the same number of bits. Therefore, a parallel connection may only be used to describe a module path between two vectors of the same size. 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 36 illustrates how a parallel connection differs from a full connection between two 4-bit vectors.

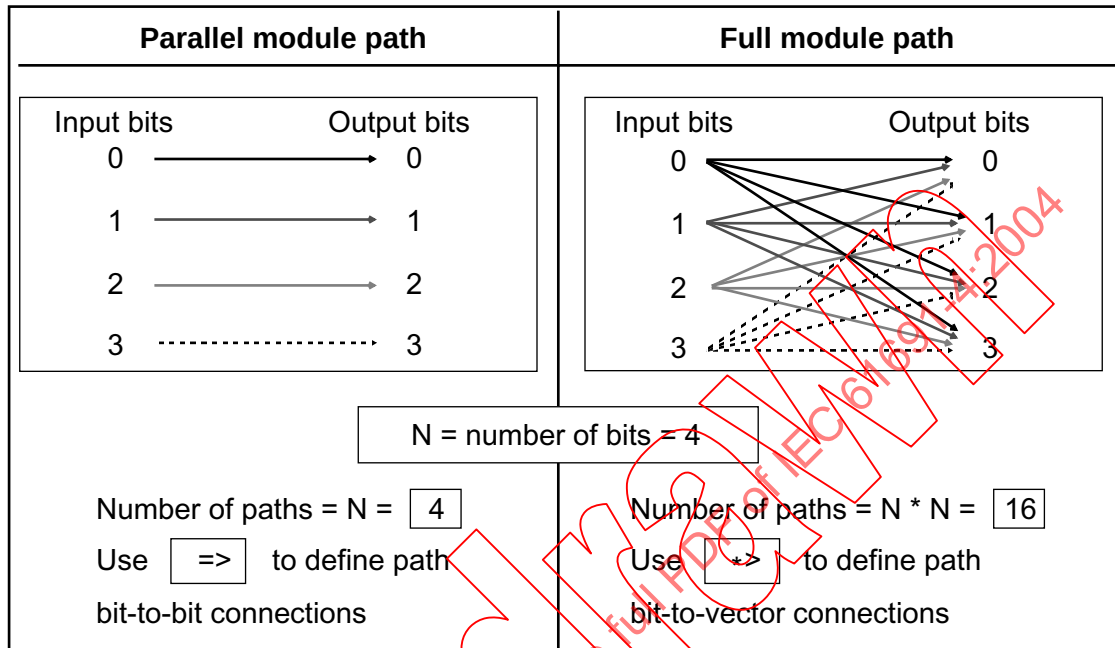


Figure 36—The difference between parallel and full connection paths

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.

14.2.6 Declaring multiple module paths in a single statement

Multiple module paths may be described in a single statement by using the symbol *> to connect a comma-separated list of sources to a comma-separated list of destinations. When describing multiple module paths in one statement, the lists of sources and destinations may contain a mix of scalars and vectors of any size.

The connection in a multiple module path declaration is always a full connection.

Example:

$$(a, b, c \text{ } ^*> \text{ } q1, q2) = 10;$$

is equivalent to the following six individual module path assignments:

$$\begin{aligned} (a \text{ } ^*> \text{ } q1) &= 10 ; \\ (b \text{ } ^*> \text{ } q1) &= 10 ; \\ (c \text{ } ^*> \text{ } q1) &= 10 ; \\ (a \text{ } ^*> \text{ } q2) &= 10 ; \\ (b \text{ } ^*> \text{ } q2) &= 10 ; \\ (c \text{ } ^*> \text{ } q2) &= 10 ; \end{aligned}$$

14.2.7 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

14.2.7.1 Unknown polarity

By default, module paths shall have *unknown polarity* that is, a transition at the path source may propagate to the destination in an unpredictable way, as follows:

- A rise at the source may cause 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.

14.2.7.2 Positive polarity

For module paths with *positive polarity*, any transition at the source may cause the same transition at the destination, as follows:

- A rise at the source may cause either a rise transition or no transition at the destination.
- A fall at the source may cause either a fall transition or no transition at the destination.

A module path with positive polarity shall be specified by prefixing the + polarity operator to => or *>.

14.2.7.3 Negative polarity

For module paths with *negative polarity*, any transition at the source may cause the opposite transition at the destination, as follows:

- A rise at the source may cause either a fall transition or no transition at the destination.
- A fall at the source may cause either a rise transition or no transition at the destination.

A module path with negative polarity shall be specified by prefixing the - polarity operator to ==> or *>.

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 ;
```

14.3 Assigning delays to module paths

The delays that occur at the module outputs where paths terminate shall be specified by assigning delay values to the module path descriptions. The syntax for specifying delay values is shown in Syntax 14-6.

```
path_delay_value ::= (From Annex A - A.7.4)
list_of_path_delay_expressions
| (list_of_path_delay_expressions )
list_of_path_delay_expressions ::=
t_path_delay_expression
| trise_path_delay_expression , tfall_path_delay_expression
| trise_path_delay_expression , tfall_path_delay_expression , tz_path_delay_expression
| t01_path_delay_expression , t10_path_delay_expression , t0z_path_delay_expression ,
  tz1_path_delay_expression , t1z_path_delay_expression , tz0_path_delay_expression
| t01_path_delay_expression , t10_path_delay_expression , t0z_path_delay_expression ,
  tz1_path_delay_expression , t1z_path_delay_expression , tz0_path_delay_expression ,
  t0x_path_delay_expression , tx1_path_delay_expression , t1x_path_delay_expression ,
  tx0_path_delay_expression , txz_path_delay_expression , tzx_path_delay_expression
t_path_delay_expression ::=
path_delay_expression
```

Syntax 14-6 Syntax for path delay value

In module path delay assignments, a module path description (see 14.2) is specified on the left-hand side, and one or more delay values are specified on the right-hand side. The delay values may be optionally enclosed in a pair of parentheses. There may be one, two, three, six, or twelve delay values assigned to a module path, as described in 14.3.1. The delay values shall be constant expressions containing literals or specparams, and there may be a delay expression of the form *min:typ:max*.

Example:

```
specify
// Specify Parameters
specparam tRise_clk_q = 45:150:270, tFall_clk_q=60:200:350;
specparam tRise_Control = 35:40:45, tFall_control=40:50:65;

// Module Path Assignments
(clk => q) = (tRise_clk_q, tFall_clk_q);
(clr, pre *> q) = (tRise_control, tFall_control);
endspecify
```

In the example above, the specify parameters declared following the **specparam** keyword specify values for the module path delays. The module path assignments assign those module path delays to the module paths.

14.3.1 Specifying transition delays on module paths

Each path delay expression may be a single value representing the typical delay or a colon-separated list of three values representing a *minimum*, *typical*, and *maximum* delay, in that order. If the path delay expression results in a negative value, it shall be treated as zero. Table 47 describes how different path delay values shall be associated with various transitions. The path delay expression names refer to the names used in Syntax 14-6.

Table 47 – Associating path delay expressions with transitions

Transitions	Number of path delay expressions specified				
	1	2	3	6	12
0 -> 1	t	trise	trise	t01	t01
1 -> 0	t	tfall	tfall	t10	t10
0 -> z	t	trise	tz	t0z	t0z
z -> 1	t	trise	trise	tz1	tz1
1 -> z	t	tfall	tz	t1z	t1z
z -> 0	t	tfall	tfall	tz0	tz0
0 -> x	*	*	*	*	t0x
x -> 1	*	*	*	*	tx1
1 -> x	*	*	*	*	t1x
x -> 0	*	*	*	*	tx0
x -> z	*	*	*	*	txz
z -> x	*	*	*	*	tzx

* See 14.3.2.

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) ;
```

14.3.2 Specifying x transition delays

If the x transition delays are not explicitly specified, the calculation of delay values for x transitions is based on the following two pessimistic rules:

- Transitions from a known state to x shall occur as quickly as possible—that is, the shortest possible delay shall be used for any transition to x.
- Transitions from x to a known state shall take as long as possible—that is, the longest possible delay shall be used for any transition from x.

Table 48 presents the general algorithm for calculating delay values for x transitions, along with specific examples. The following two groups of x transitions are represented in the table:

- a) Transition from a known state S to x: S → x
- b) Transition from x to a known state S: x → S

Table 48—Calculating delays for x transitions

X transition	Delay value
General algorithm	
s -> x	minimum (s -> other known signals)
x -> s	maximum (other known signals -> s)
Specific transitions	
0 -> x	minimum (0 -> z delay, 0 -> 1 delay)
1 -> x	minimum (1 -> z delay, 1 -> 0 delay)
z -> x	minimum (z -> 1 delay, z -> 0 delay)
x -> 0	maximum (z -> 0 delay, 1 -> 0 delay)
x -> 1	maximum (z -> 1 delay, 0 -> 1 delay)
x -> z	maximum (1 -> z delay, 0 -> z delay)
Usage: (C => Q) = (5, 12, 17, 10, 6, 22) ;	
0 -> x	minimum (17, 5) = 5
1 -> x	minimum (6, 12) = 6
z -> x	minimum (10, 22) = 10
x -> 0	maximum (22, 12) = 22
x -> 1	maximum (10, 5) = 10
x -> z	maximum (6, 17) = 17

14.3.3 Delay selection

The simulator shall determine the proper delay to use when a specify path output must be scheduled to transition. There may be specify paths to the output from more than one input, and the simulator must decide which specify path to use.

The simulator shall do this by first determining which specify paths to the output are active. Active specify paths are those whose input has transitioned most recently in time, and which have either no condition or whose conditions are true. In the presence of simultaneous input transitions, it is possible for many specify paths to an output to be simultaneously active.

Once the active specify paths are identified, a delay must be selected from among them. This is done by comparing the correct delay for the specific transition being scheduled from each specify path, and choosing the smallest.

Examples:

Example 1:

(A => Y) = (6, 9);
(B => Y) = (5, 11);

For a Y transition from 0 to 1, if A transitioned more recently than B a delay of 6 will be chosen. But if B transitioned more recently than A, a delay of 5 will be chosen. And if the last time they transitioned A and B did so simultaneously, then the smallest of the two rise delays would be chosen, which is the rise delay from B of 5. The fall delay from A of 9 would be chosen if Y was instead to transition from 1 to 0.

Example 2:

```
if (MODE < 5) (A => Y) = (5, 9);
if (MODE < 4) (A => Y) = (4, 8);
if (MODE < 3) (A => Y) = (6, 5);
if (MODE < 2) (A => Y) = (3, 2);
if (MODE < 1) (A => Y) = (7, 7);
```

Anywhere from zero to five of these specify paths might be active depending upon the value of MODE. For instance, when MODE is 2 the first three specify paths are active. A rise transition would select a delay of 4, because that is the smallest rise delay among the first three. A fall transition would select a delay of 5, because that is the smallest fall delay among the first three.

14.4 Mixing module path delays and distributed delays

If a module contains module path delays and distributed delays (delays on primitive instances within the module), the larger of the two delays for each path shall be used.

Examples:

Example 1 Figure 37 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.

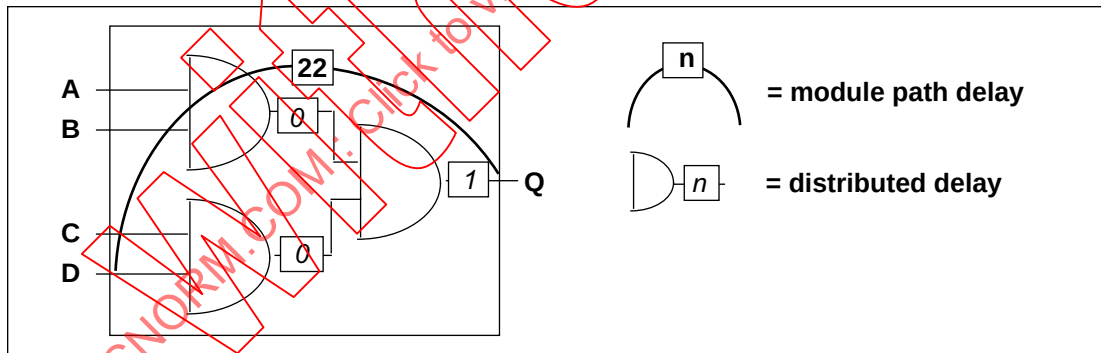


Figure 37—Module path delays longer than distributed delays

Example 2 In Figure 38, 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.

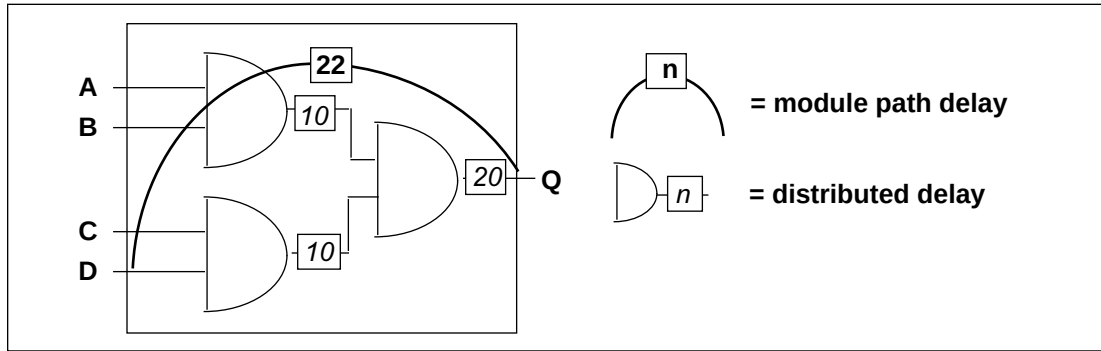


Figure 38—Module path delays shorter than distributed delays

14.5 Driving wired logic

Module path output nets shall not have more than one driver within the module. Therefore, wired logic is not allowed at module path outputs.

Figure 40 illustrates a violation of this wired-output rule and a method of avoiding the rule violation.

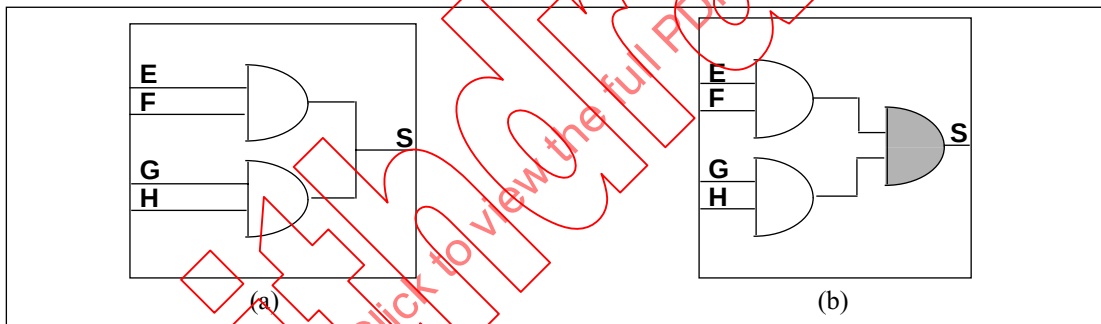


Figure 39—Legal and illegal module paths

In Figure 39 (a), any module path to S is illegal because the path destination has two drivers.

Assuming signal S in Figure 39 (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 39 (b) shows how adding a third **and** gate the shaded gate solves the problem for the module in Figure 39 (a).

The example in Figure 40 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.

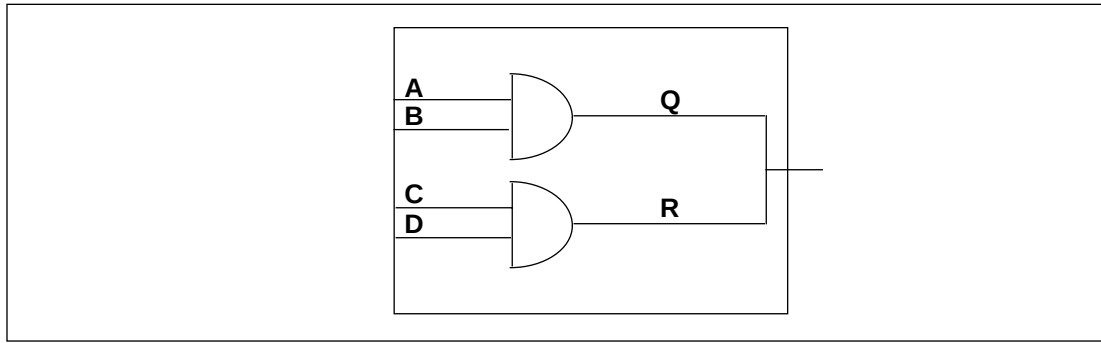


Figure 40—Illegal module paths

Although multiple output drivers to a path destination are prohibited *inside* the same module, they are allowed *outside* the module. The example in Figure 41 is legal since Q and R each have only one driver within the module in which the module paths are specified.

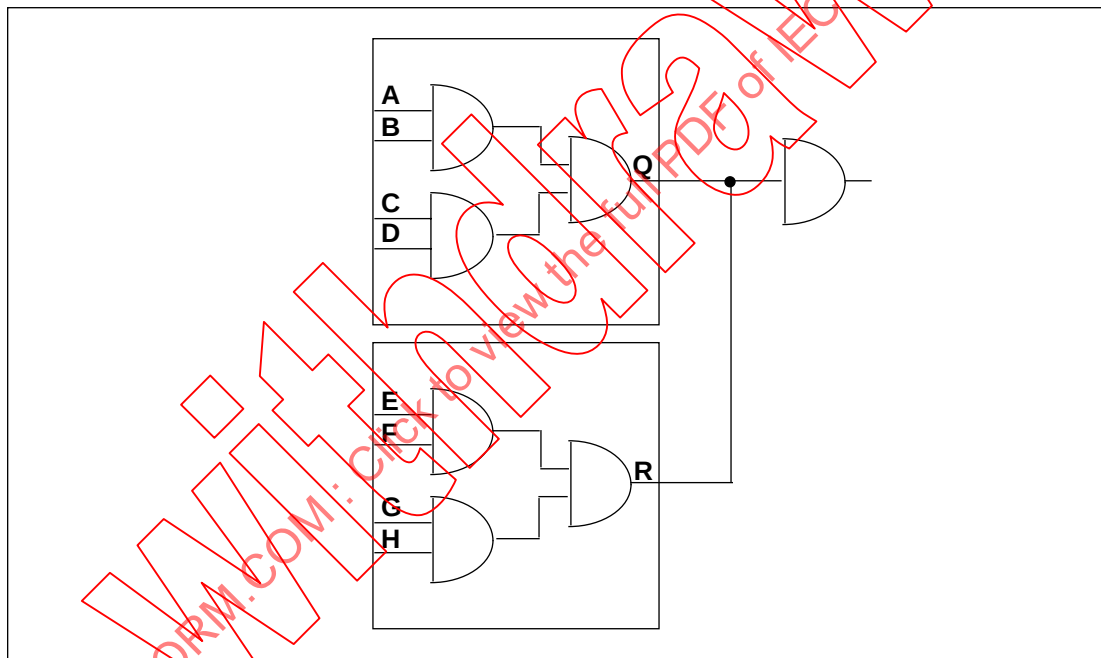


Figure 41—Legal module paths

14.6 Detailed control of pulse filtering behavior

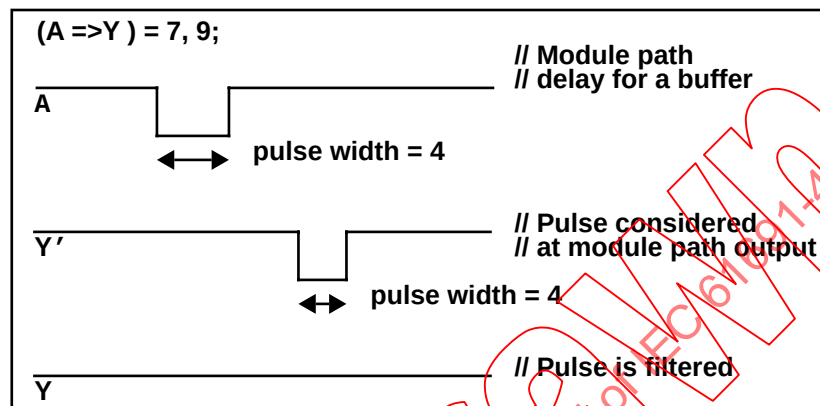
Two consecutive scheduled transitions closer together in time than the module path delay is deemed a pulse. By default, pulses on a module path output are rejected. Consecutive transitions cannot be closer together than the module path delay, and this is known as the inertial delay model of pulse propagation.

Pulse width ranges control how to handle a pulse presented at a module path output. They are:

- A pulse width range for which a pulse shall be rejected
- A pulse width range for which a pulse shall be allowed to propagate to the path destination
- A pulse width range for which a pulse shall generate a logic X on the path destination

Two pulse limit values define the pulse width ranges associated with each module path transition delay. The pulse limit values are called the error limit (e-limit) and the rejection limit (r-limit). The e-limit must always be at least as large as the r-limit. Pulses greater than or equal to the e-limit pass unfiltered. Pulses less than the e-limit but greater than or equal to the r-limit are filtered to X. Pulses less than the r-limit are rejected and no pulse emerges. By default, both the e-limit and the r-limit are set equal to the delay. These default values yield full inertial pulse behavior, rejecting all pulses smaller than the delay.

Example:



The rise delay from input A to output Y is 7, and the fall delay is 9. By default, the e-limit and the r-limit for the rise delay are both 7. The e-limit and the r-limit for the fall delay are both 9. The pulse limits associated with the delay forming the trailing edge of the pulse determine if and how the pulse should be filtered. Waveform Y' shows the waveform resulting from no pulse filtering. The width of the pulse is 2, which is less than the reject limit for the rise delay of 7, and so the pulse is filtered as shown in waveform Y.

There are three ways to modify the pulse limits from their default values. First, the Verilog language provides the **PATHPULSES** specparam to modify the pulse limits from their default values. Second, invocation options can specify percentages applying to all module path delays to form the corresponding e-limits and r-limits. Third, SDF annotation can individually annotate the e-limit and r-limit of each module path transition delay.

14.6.1 Specify block control of pulse limit values

Pulse limit values may be set from within the specify block with the **PATHPULSES** specparam. The syntax for using **PATHPULSES** to specify the reject limit and error limit values is given in Syntax 14-7.

```
pulse_control_specparam ::= (From Annex A - A.2.4)
    PATHPULSES = ( reject_limit_value [ , error_limit_value ] ) ;
    | PATHPULSES$specify_input_terminal_descriptor$specify_output_terminal_descriptor
      = ( reject_limit_value [ , error_limit_value ] ) ;

error_limit_value ::=
    limit_value
reject_limit_value ::=
    limit_value
limit_value ::=
    constant_mintypmax_expression
```

Syntax 14-7 Syntax for **PATHPULSES** pulse control

If only the reject limit value is specified, it shall apply to both the reject limit and the error limit.

The reject limit and error limit may be specified for a specific module path. When no module path is specified, the reject limit and error limit shall apply to all module paths defined in a module. If both path-specific **PATHPULSE\$** specparams and a 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.

When a module path declaration declares multiple paths, the **PATHPULSE\$** specparam shall only be specified for the first path input terminal and the first path output terminal. The reject limit and error limit specified shall apply to all other paths in the multiple path declaration. A **PATHPULSE\$** specparam which specifies anything other than the first path input and path output terminals shall be ignored.

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
```

14.6.2 Global control of pulse limit values

Two invocation options can specify percentages applying globally to all module path transition delays. The error limit invocation option specifies the percentage of each module path transition delay used for its error limit value. The reject limit invocation option specifies the percentage of each module path transition delay used for its reject limit value. The percentage values shall be an integer between 0 and 100.

The default values for both the reject and error limit invocation options are 100%. When neither option is present then 100% of each module transition delay is used as the reject and error limits.

It is an error if the error limit percentage is smaller than the reject limit percentage. In such cases the error limit percentage is set equal to the reject limit percentage.

When both **PATHPULSE\$** and global pulse limit invocation options are present, the **PATHPULSE\$** values shall take precedence.

14.6.3 SDF annotation of pulse limit values

SDF annotation can be used to specify the pulse limit values of module path transition delays. Section 16 describes this in greater detail.

When both **PATHPULSES**, global pulse limit invocation options, and SDF annotation of pulse limit values are present, SDF annotation values shall take precedence.

14.6.4 Detailed pulse control capabilities

The default style of pulse filtering behavior has two drawbacks. First, pulse filtering to the X state may be insufficiently pessimistic with an X state duration too short to be useful. Second, unequal delays can result in pulse rejection whenever the trailing edge precedes the leading edge, leaving no indication that a pulse was rejected. This section introduces more detailed pulse control capabilities.

14.6.4.1 On-event versus on-detect pulse filtering

When an output pulse must be filtered to X, greater pessimism can be expressed if the module path output transitions immediately to X (on-detect) instead of at the already scheduled transition time of the leading edge of the pulse (on-event).

The on-event method of pulse filtering to X is the default. When an output pulse must be filtered to X, the leading edge of the pulse becomes a transition to X and the trailing edge a transition from X. The times of transition of the edges do not change.

Just like on-event, the on-detect method of pulse filtering changes the leading edge of the pulse into a transition to X and the trailing edge to a transition from X, but the time of the leading edge is changed to occur immediately upon detection of the pulse.

Figure 42 illustrates this behavior using a simple buffer with asymmetric rise/fall times and both the r-limits and e-limits equal to 0. An output waveform is shown for both on-detect and on-event approaches.

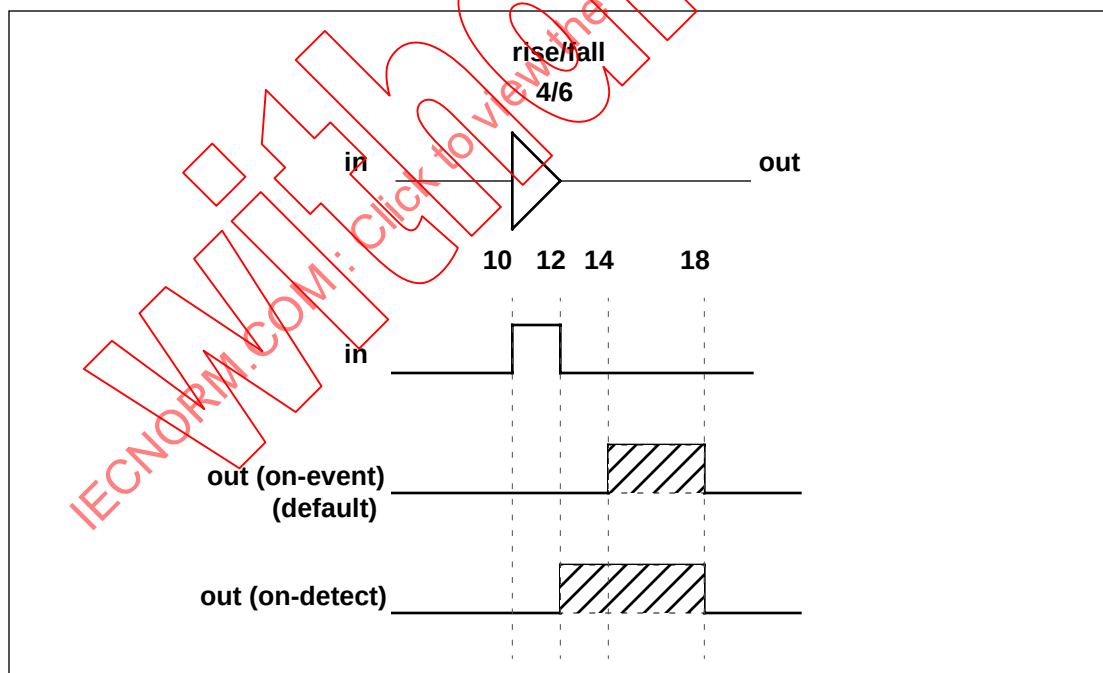


Figure 42—On-detect -vs.- on-event

On-detect versus on-event behavior can be selected in two different ways. First, one may be selected globally for all module path outputs through use of the on-detect or on-event invocation option. Second, one may be selected locally through use of specify block pulse style declarations.

The syntax for pulse style declarations is shown in Syntax 14-8.

```
pulstyle_declaration ::= (From Annex A- A.7.1)
    pulstyle_onevent list_of_path_outputs ;
    | pulstyle_ondetect list_of_path_outputs ;
```

Syntax 14-8 Syntax for pulse style declarations

It is an error if a module path output appears in a pulse style declaration after it has already appeared in a module path declaration.

The pulse style invocation options take precedence over pulse style specify block declarations.

14.6.4.2 Negative pulse detection

When the delays to a module path output are unequal, it is possible for the trailing edge of a pulse to be scheduled for a time earlier than the schedule time of the leading edge, yielding a pulse with a negative width. Under normal operation, if the schedule for a trailing pulse edge is earlier than the schedule for a leading pulse edge, then the leading edge is cancelled. No transition takes place when the initial and final states of the pulse are the same, leaving no indication a schedule was ever present.

Negative pulses can be indicated with the X state by use of the *showcancelled* style of behavior. When the trailing edge of a pulse would be scheduled before the leading edge, this style causes the leading edge to be scheduled to X, and the trailing edge to be scheduled from X. With on-event pulse style, the schedule to X replaces the leading edge schedule. With on-detect pulse style, the schedule to X is made immediately upon detection of the negative pulse.

showcancelled behavior can be enabled in two different ways. First, it may be enabled globally for all module path outputs through use of the **showcancelled** and **noshowcancelled** invocation options. Second, it may be enabled locally through use of specify block **showcancelled** declarations.

The syntax for *showcancelled* declarations is shown in Syntax 14-9.

```
showcancelled_declaration ::= (From Annex A- A.7.1)
    showcancelled list_of_path_outputs ;
    | noshowcancelled list_of_path_outputs ;
```

Syntax 14-9 Syntax for showcancelled declarations

It is an error if a module path output appears in a *showcancelled* declaration after it has already appeared in a module path declaration. The *showcancelled* invocation options take precedence over the *showcancelled* specify block declarations.

The *showcancelled* behavior is illustrated in Figure 43, which shows a narrow pulse presented at the input to a buffer with unequal rise/fall delays. This causes the trailing edge of the pulse to be scheduled earlier than leading edge. The leading edge of the input pulse schedules an output event 6 units later at the point marked by A. The pulse trailing edge occurs one time unit later, which schedules an output event 4 units later marked by point B. This second schedule on the output is for a time prior to the already existing schedule for the leading output pulse edge.

The output waveform is shown for three different operating modes. The first waveform shows the default behavior with showcancelled behavior not enabled and with the default on-event style. The second waveform shows showcancelled behavior in conjunction with on-event. The last waveform shows showcancelled behavior in conjunction with on-detect.

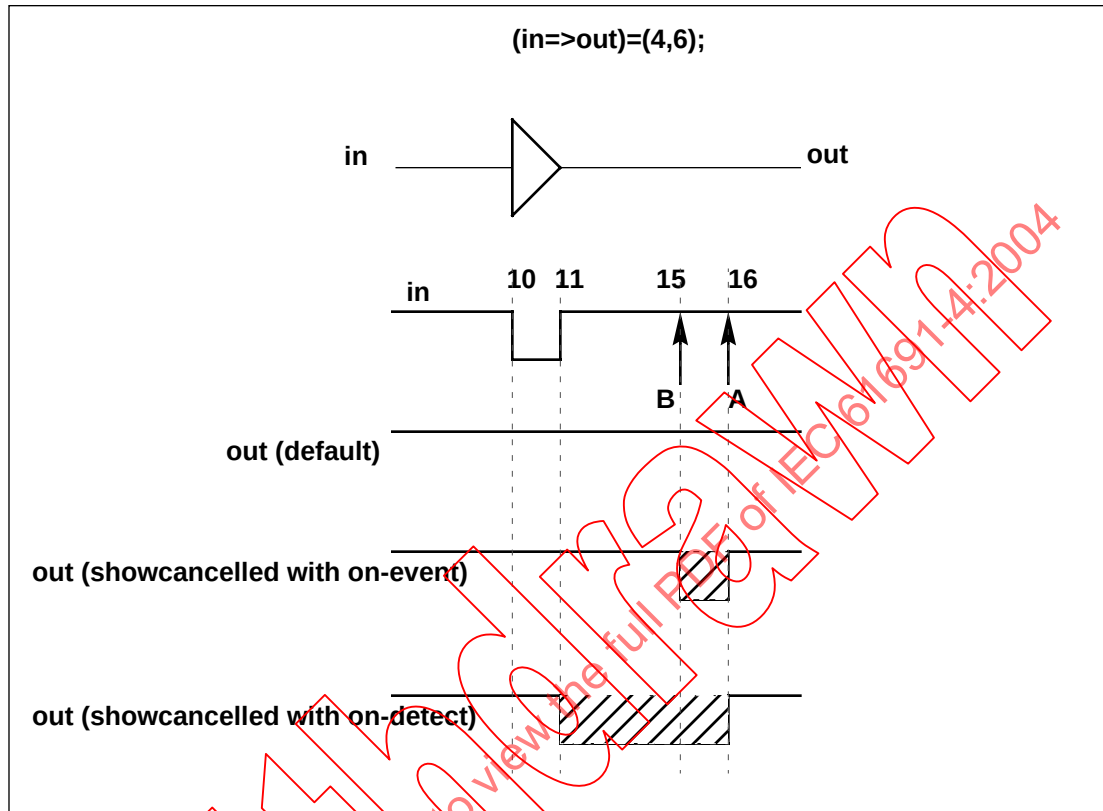


Figure 43 – Current event cancellation problem and correction

This same situation can also arise with nearly simultaneous input transitions, which is defined as two inputs transitioning closer together in time than the difference in their respective delays to the output. Figure 44 shows waveforms for a 2-input NAND gate where initially A is high and B is low. B transitions 0->1 at time 10, causing a 1->0 output schedule at time 24. A transitions 1->0 at time 12, causing a 0->1 schedule at time 22. Arrows mark the output transitions caused by the transitions on inputs A and B.

The output waveform is shown for three different operating modes. The first waveform shows the default behavior with showcancelled behavior not enabled and with the default on-event style. The second shows showcancelled behavior in conjunction with on-event. The third shows showcancelled behavior in conjunction with on-detect.

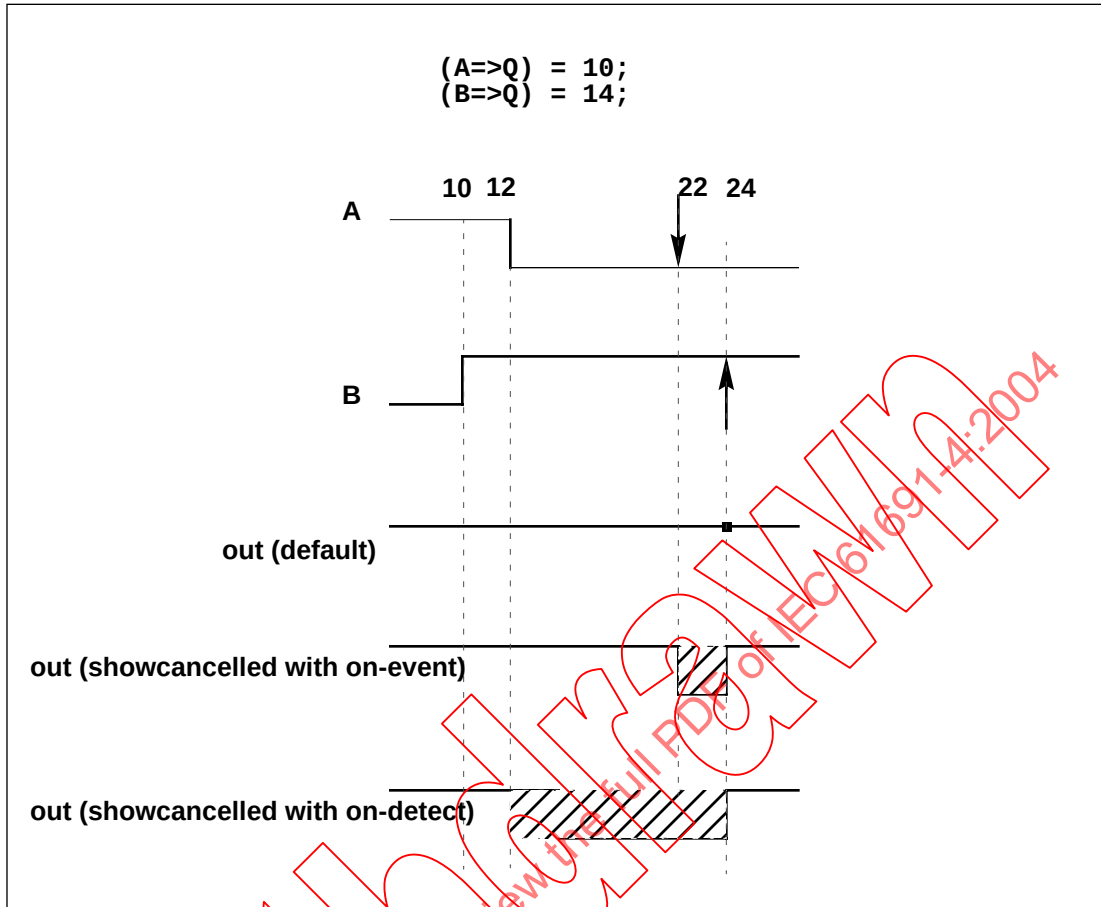


**Figure 44—NAND gate with nearly simultaneous input switching
where one event is scheduled prior to another that has not matured**

One drawback of the on-event style with showcancelled behavior is that as the output pulse edges draw closer together, the duration of the resulting X state becomes smaller. Figure 45 illustrates how the on-detect style solves this problem.

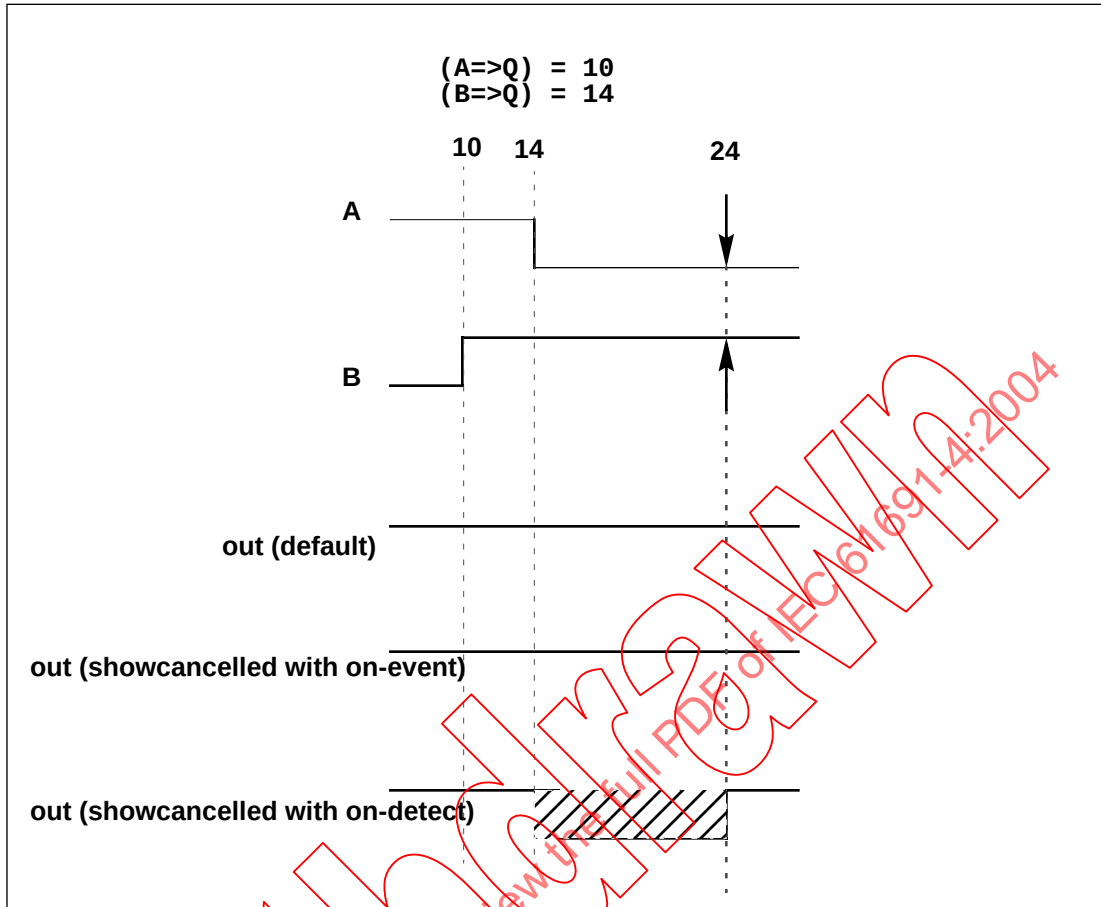


Figure 45—Input NAND gate with nearly simultaneous input switching with output event scheduled at same time.

Examples:

Example 1:

```

specify
  (a=>out)=(2,3);
  (b=>out)=(3,4);
endspecify

```

Since no pulse style or showcancelled declarations appear within the specify block, the compiler applies the default modes of on-event and noshowcancelled.

Example 2:

```

specify
  (a=>out)=(2,3);
  showcancelled out;
  (b=>out)=(3,4);
endspecify

```

This showcanceled declaration is in error because it follows use of out in a module path declaration. It would be contradictory for out to have n showcanceled behavior from input a, but showcanceled behavior from input b.

Example 2 Both these specify blocks produce the same result. Outputs out and out_b are both declared showcanceled and on_detect.

```
specify
  showcanceled out;
  pulsestyle_ondetect out;
  (a =>out)=(2,3);
  (a=>out)=(4,5);
  showcanceled out_b;
  pulsestyle_ondetect out_b;
  (b=>out_b)=(5,6);
  (b=>out_b)=(3,4);
endspecify

specify
  showcanceled out,out_b;
  pulsestyle_ondetect out,out_b;
  (a =>out)=(2,3);
  (b=>out)=(3,4);
  (a =>out_b)=(3,4);
  (b=>out_b)=(5,6);
endspecify
```

15. Timing checks

This section describes how timing checks are used in specify blocks to determine if signals obey the timing constraints.

15.1 Overview

Timing checks can be placed in specify blocks to verify the timing performance of a design by making sure critical events occur within given time limits. The syntax for system timing checks is given in Syntax 15-1.

Withdrawing
IECNORM.COM: Click to view the full PDF of IEC 61691-4:2004


```

system_timing_check ::= (From Annex A - A.7.5.1)
    $setup_timing_check
    $hold_timing_check
    $setuphold_timing_check
    $recovery_timing_check
    $removal_timing_check
    $recrem_timing_check
    $skew_timing_check
    $timeskew_timing_check
    $fullskew_timing_check
    $period_timing_check
    $width_timing_check
    $nochange_timing_check

$setup_timing_check ::=
    $setup ( data_event , reference_event , timing_check_limit [ , [ notify_reg ] ] );
$hold_timing_check ::=
    $hold ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] );
$setuphold_timing_check ::=
    $setuphold ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notify_reg ] ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
        [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] );
$recovery_timing_check ::=
    $recovery ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] );
$removal_timing_check ::=
    $removal ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] );
$recrem_timing_check ::=
    $recrem ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notify_reg ] ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
        [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] );
$skew_timing_check ::=
    $skew ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] );
$timeskew_timing_check ::=
    $timeskew ( reference_event , data_event , timing_check_limit
        [ , [ notify_reg ] ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] );
$fullskew_timing_check ::=
    $fullskew ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notify_reg ] ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] );
$period_timing_check ::=
    $period ( controlled_reference_event , timing_check_limit [ , [ notify_reg ] ] );
$width_timing_check ::=
    $width ( controlled_reference_event , timing_check_limit ,
        threshold [ , [ notify_reg ] ] );
$nochange_timing_check ::=
    $nochange ( reference_event , data_event , start_edge_offset ,
        end_edge_offset [ , [ notify_reg ] ] );

```

Syntax 15-1 Syntax for system timing checks

The syntax for check time conditions and timing check events are given in Syntax 15-2.

```

checktime_condition ::= (From Annex A - A.7.5.2)
    mintypmax_expression
controlled_reference_event ::=
    controlled_timing_check_event
data_event ::=
    timing_check_event
delayed_data ::=
    terminal_identifier
    | terminal_identifier [ constant_mintypmax_expression ]
delayed_reference ::=
    terminal_identifier
    | terminal_identifier [ constant_mintypmax_expression ]
end_edge_offset ::= mintypmax_expression
event_based_flag ::= constant_expression
notify_reg ::= variable_identifier
reference_event ::= timing_check_event
remain_active_flag ::= constant_mintypmax_expression
stamp_time_condition ::= mintypmax_expression
start_edge_offset ::= mintypmax_expression
threshold ::= constant_expression
timing_check_limit ::= expression
timing_check_event ::= (From Annex A - A.7.5.3)
    [timing_check_event_control] specify_terminal_descriptor [ &&& timing_check_condition ]
controlled_timing_check_event ::=
    timing_check_event_control specify_terminal_descriptor [ &&& timing_check_condition ]
timing_check_event_control ::= posedge | negedge | edge_control_specifier
specify_terminal_descriptor ::=
    specify_input_terminal_descriptor
    | specify_output_terminal_descriptor
edge_control_specifier ::= edge [ edge_descriptor [ , edge_descriptor ] ]
edge_descriptor * ::= 01 | 10 | z_or_x zero_or_one | zero_or_one z_or_x
zero_or_one ::= 0 | 1
z_or_x ::= x | X | z | Z
timing_check_condition ::=
    scalar_timing_check_condition
    | ( scalar_timing_check_condition )
scalar_timing_check_condition ::=
    expression
    | ~ expression
    | expression == scalar_constant
    | expression === scalar_constant
    | expression != scalar_constant
    | expression !== scalar_constant
scalar_constant ::= 1'b0 | 1'b1 | 1'B0 | 1'B1 | 'b0 | 'b1 | 'B0 | 'B1 | 1 | 0

```

* Embedded spaces are illegal.

Syntax 15-2 Syntax for check time conditions and timing check events

For ease of presentation, the timing checks are divided into two groups. The first group of timing checks are described in terms of stability time windows:

\$setup	\$hold	\$setuphold
\$recovery	\$removal	\$crem

The timing checks in the second group check clock and control signals, and are described in terms of the difference in time between two events (the **\$nochange** check involves three events):

\$skew	\$timeskew	\$fullskew
\$width	\$period	\$nochange

Although they begin with a \$, timing checks are not system tasks. The leading \$ is present because of historical reasons, and timing checks shall not be confused with system tasks. In particular, no system task can appear in a specify block, and no timing check can appear in procedural code.

Some timing checks can accept negative limit values. This topic is discussed in detail in 15.8.

All timing checks have both a reference event and a data event, and boolean conditions can be associated with each. Some of the checks have two signal arguments, one of which is the reference event and the other the data event. Other checks have only one signal argument, and the reference and data events are derived from it. Reference events and data events shall only be detected by timing checks when their associated conditions are true. See 15.6 for more information about conditions in timing checks.

Timing check evaluation is based upon the times of two events, called the timestamp event and the timecheck event. A transition on the timestamp event signal causes the simulator to record (stamp) the time of transition for future use in evaluating the timing check. A transition on the timecheck event signal causes the simulator to actually evaluate the timing check to determine whether a violation has taken place.

For some checks the reference event is always the timestamp event, while the data event is always the timecheck event, while for other checks the reverse is true. And for yet other checks the decision as to which is the timestamp and which the timecheck event is based upon factors to be discussed later in greater detail.

Every timing check can include an optional notifier which toggles whenever the timing check detects a violation. The model can use the notifier to make behavior a function of timing check violations. Notifiers are discussed in greater detail in 15.5.

Like expressions for module path delays, timing check limit values are constant expressions which can include specparams.

15.2 Timing checks using a stability window

These timing checks are discussed in this section:

\$setup	\$hold	\$setuphold
\$recovery	\$removal	\$crem

These checks accept two signals, the reference event and the data event, and define a time window with respect to one signal while checking the time of transition of the other signal with respect to the window. In general they all perform the following steps:

- Define a time window with respect to the reference signal using the specified limit or limits;
- Check the time of transition of the data signal with respect to the time window;
- Report a timing violation if the data signal transitions within the time window.

15.2.1 \$setup

The **\$setup** timing check syntax is shown in Syntax 15-3.

```

$setup_timing_check ::= (From Annex A - A.7.5.1)
$setup ( data_event , reference_event , timing_check_limit [ , [ notify_reg ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
    timing_check_event
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
timing_check_limit ::=
    expression

```

Syntax 15-3 Syntax for **\$setup**

Table 49 defines the **\$setup** timing check.

Table 49—**\$setup** arguments

Argument	Description
data_event	Timestamp event
reference_event	Timecheck event
limit	Non-negative constant expression
notifier (optional)	Reg

The data event is usually a data signal, while the reference event is usually a clock signal.

The endpoints of the time window are determined as follows:

$$\begin{aligned}
 (\text{beginning of time window}) &= (\text{timecheck time}) - \text{limit} \\
 (\text{end of time window}) &= (\text{timecheck time})
 \end{aligned}$$

The **\$setup** timing check reports a timing violation in the following case:

$$(\text{beginning of time window}) < (\text{timestamp time}) < (\text{end of time window})$$

The endpoints of the time window are not part of the violation region. When the limit is zero, the **\$setup** check shall never issue a violation.

15.2.2 \$hold

The **\$hold** timing check syntax is shown in Syntax 15-4.

```

$hold_timing_check ::= (From Annex A - A.7.5.1)
$hold ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
    timing_check_event
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
timing_check_limit ::=
    expression
    
```

Syntax 15-4 Syntax for \$hold

Table 50 defines the **\$hold** timing check.

Table 50—\$hold arguments

Argument	Description
reference_event	Timestamp event
data_event	Timecheck event
limit	Non-negative constant expression
notifier (optional)	Reg

The data event is usually a data signal, while the reference event is usually a clock signal.

The endpoints of the time window are determined as follows:

```

(beginning of time window) = (timestamp time)
(end of time window) = (timestamp time) + limit
    
```

The **\$hold** timing check reports a timing violation in the following case:

```

(beginning of time window) <= (timecheck time) < (end of time
window)
    
```

Only the end of the time window is not part of the violation region. When the limit is zero, the **\$hold** check shall never issue a violation.

15.2.3 \$setuphold

The **\$setuphold** timing check syntax is shown in Syntax 15-5.

```

$setuphold_timing_check ::= (From Annex A - A.7.5.1)
    $setuphold ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notify_reg ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
            [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ) ;
checktime_condition ::= (From Annex A - A.7.5.2)
    mintymax_expression
data_event ::=
    timing_check_event
delayed_data ::=
    terminal_identifier
    | terminal_identifier [ constant_mintymax_expression ]
delayed_reference ::=
    terminal_identifier
    | terminal_identifier [ constant_mintymax_expression ]
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
stamptime_condition ::=
    mintymax_expression
timing_check_limit ::=
    expression

```

Syntax 15-5 Syntax for \$setuphold

Table 51 defines the **\$setuphold** timing check.

Table 51—\$setuphold arguments

Argument	Description
reference_event	Timecheck or timestamp event when setup limit is positive Timestamp event when setup limit is negative
data_event	Timecheck or timestamp event when hold limit is positive Timestamp event when hold limit is negative
setup_limit	Constant expression
hold_limit	Constant expression
notifier (optional)	Reg
timestamp_cond (optional)	Timestamp condition for negative timing checks
timecheck_cond (optional)	Timecheck condition for negative timing checks
delayed_reference (optional)	Delayed reference signal for negative timing checks
delayed_data (optional)	Delayed data signal for negative timing checks

The **\$setuphold** timing check can accept negative limit values. This is discussed in greater detail in 15.8.

The data event is usually a data signal, while the reference event is usually a clock signal.

When both the setup limit and the hold limit are positive, either the reference event or the data event can be the timecheck event. It shall depend upon which occurs first in the simulation.

When either the setup limit or the hold limit is negative the restriction becomes:

$$\text{setup\_limit} + \text{hold\_limit} > (\text{simulation unit of precision})$$

The **\$setuphold** timing check combines the functionality of the **\$setup** and **\$hold** timing checks into a single timing check. Therefore, the 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 );
```

When both setup and hold limits are positive and the data event occurs first, the endpoints of the time window are determined as follows:

$$\begin{aligned} (\text{beginning of time window}) &= (\text{timecheck time}) - \text{limit} \\ (\text{end of time window}) &= (\text{timecheck time}) \end{aligned}$$

And the **\$setuphold** timing check reports a timing violation in the following case:

$$(\text{beginning of time window}) < (\text{timecheck time}) \leq (\text{end of time window})$$

Only the beginning of the time window is not part of the violation region. The **\$setuphold** check shall report a timing violation when the reference and data events occur simultaneously.

When both setup and hold limits are positive and the data event occurs second, the endpoints of the time window are determined as follows:

$$\begin{aligned} (\text{beginning of time window}) &= (\text{timestamp time}) \\ (\text{end of time window}) &= (\text{timestamp time}) + \text{limit} \end{aligned}$$

And the **\$setuphold** timing check reports a timing violation in the following case:

$$(\text{beginning of time window}) \leq (\text{timecheck time}) < (\text{end of time window})$$

Only the end of the time window is not part of the violation region. The **\$setuphold** check shall report a timing violation when the reference and data events occur simultaneously.

When both limits are zero, the **\$setuphold** check shall never issue a violation.

15.2.4 \$removal

The **\$removal** timing check syntax is shown in Syntax 15-6.

```

$removal_timing_check ::= (From Annex A - A.7.5.1)
    $removal ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
    timing_check_event
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
timing_check_limit ::=
    expression

```

Syntax 15-6 Syntax for \$removal

Table 52 defines the **\$removal** timing check.

Table 52—\$removal arguments

Argument	Description
reference_event	Timestamp event
data_event	Timecheck event
limit	Non-negative constant expression
notifier (optional)	Reg

The reference event is usually a control signal like clear, reset or set, while the data event is usually a clock signal.

The endpoints of the time window are determined as follows:

```

(beginning of time window) = (timecheck time) - limit
(end of time window) = (timecheck time)

```

The **\$removal** timing check reports a timing violation in the following case:

```

(beginning of time window) < (timestamp time) < (end of time
window)

```

The endpoints of the time window are not part of the violation region. When the limit is zero, the **\$removal** check shall never issue a violation.

15.2.5 \$recovery

The **\$recovery** timing check syntax is shown in Syntax 15-7.


```

$recovery_timing_check ::= (From Annex A - A.7.5.1)
    $recovery ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
    timing_check_event
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
timing_check_limit ::=
    expression
    
```

Syntax 15-7 Syntax for \$recovery

Table 53 defines the \$recovery timing check.

Table 53—\$recovery arguments

Argument	Description
reference_event	Timestamp event
data_event	Timecheck event
limit	Non-negative constant expression
notifier (optional)	Reg

The reference event is usually a control signal like clear, reset or set, while the data event is usually a clock signal.

The endpoints of the time window are determined as follows:

```

(beginning of time window) = (timestamp time)
(end of time window) = (timestamp time) + limit
    
```

The \$recovery timing check reports a timing violation in the following case:

```

(beginning of time window) <= (timecheck time) < (end of time
window)
    
```

Only the end of the time window is not part of the violation region. When the limit is zero, the \$recovery check shall never issue a violation.

15.2.6 \$crecm

The \$crecm timing check syntax is shown in Syntax 15-8.

```

$crem_timing_check ::= (From Annex A - A.7.5.1)
    $crem ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notify_reg ] [ , [ stamp_time_condition ] [ , [ check_time_condition ]
            [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ) ;
check_time_condition ::= (From Annex A - A.7.5.2)
    mintypmax_expression
data_event ::=
    timing_check_event
delayed_data ::=
    terminal_identifier
    | terminal_identifier [ constant_mintypmax_expression ]
delayed_reference ::=
    terminal_identifier
    | terminal_identifier [ constant_mintypmax_expression ]
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
stamp_time_condition ::=
    mintypmax_expression
timing_check_limit ::=
    expression

```

Syntax 15-8 Syntax for \$crem

Table 54 defines the \$crem timing check.

Table 54—\$crem arguments

Argument	Description
reference_event	Timecheck or timestamp event when removal limit is positive Timestamp event when removal limit is negative
data_event	Timecheck or timestamp event when recovery limit is positive Timestamp event when recovery limit is negative
recovery_limit	Constant expression
removal_limit	Constant expression
notifier (optional)	Reg
timestamp_cond (optional)	Timestamp condition for negative timing checks
timecheck_cond (optional)	Timecheck condition for negative timing checks
delayed_reference (optional)	Delayed reference signal for negative timing checks
delayed_data (optional)	Delayed data signal for negative timing checks

The \$crem timing check can accept negative limit values. This is discussed in greater detail in 15.8.

When both the removal limit and the recovery limit are positive, either the reference event or the data event can be the timecheck event. It shall depend upon which occurs first in the simulation.

When either the removal limit or the recovery limit is negative the restriction becomes:

$$\text{removal\_limit} + \text{recovery\_limit} > (\text{simulation unit of precision})$$

The **\$screm** timing check combines the functionality of the **\$removal** and **\$recovery** timing checks into a single timing check. Therefore, the following invocation:

```
$screm( posedge clear, posedge clk, tREC, tREM );
```

is equivalent in functionality to the following, if **tREC** and **tREM** are not negative:

```
$removal( posedge clear, posedge clk, tREM );  
$recovery( posedge clear, posedge clk, tREC );
```

When both removal and recovery limits are positive and the data event occurs first, the endpoints of the time window are determined as follows:

$$\begin{aligned} (\text{beginning of time window}) &= (\text{timecheck time}) - \text{limit} \\ (\text{end of time window}) &= (\text{timecheck time}) \end{aligned}$$

And the **\$screm** timing check reports a timing violation in the following case:

$$(\text{beginning of time window}) < (\text{timecheck time}) \leq (\text{end of time window})$$

Only the beginning of the time window is not part of the violation region. The **\$screm** check shall report a timing violation when the reference and data events occur simultaneously.

When both removal and recovery limits are positive and the data event occurs second, the endpoints of the time window are determined as follows:

$$\begin{aligned} (\text{beginning of time window}) &= (\text{timestamp time}) \\ (\text{end of time window}) &= (\text{timestamp time}) + \text{limit} \end{aligned}$$

And the **\$screm** timing check reports a timing violation in the following case:

$$(\text{beginning of time window}) \leq (\text{timecheck time}) < (\text{end of time window})$$

Only the end of the time window is not part of the violation region. The **\$screm** check shall report a timing violation when the reference and data events occur simultaneously.

When both limits are zero, the **\$screm** check shall never issue a violation.

15.3 Timing checks for clock and control signals

The following timing checks are discussed in this section:

\$skew \$timeskew \$fullskew \$period \$width \$nochange

These checks accept one or two signals and verify transitions on them are never separated by more than the limit. For those checks specifying only one signal, the reference event and data event are derived from that one signal. In general these checks all perform the following steps:

- a) Determine the elapsed time between two events;
- b) Compare the elapsed time to the specified limit;
- c) Report a timing violation if the elapsed time violates the limit.

The skew checks have two different violation detection mechanisms, event-based and timer-based. Event-based skew checking is performed only when a signal transitions, while timer-based skew checking takes place as soon as the simulation time equal to the skew limit has elapsed.

The **\$nochange** check involves three events rather than two.

15.3.1 \$skew

The **\$skew** timing check syntax is shown in Syntax 15-9.

```
$skew_timing_check ::= (From Annex A - A.7.5.1)
    $skew ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
    timing_check_event
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
timing_check_limit ::=
    expression
```

Syntax 15-9 Syntax for \$skew

Table 55 defines the **\$skew** timing check.

Table 55—\$skew arguments

Argument	Description
reference_event	Timestamp event
data_event	Timecheck event
limit	Non-negative constant expression
notifier (optional)	Reg

The **\$skew** timing check reports a violation in the following case:

$$(\text{timecheck time}) - (\text{timestamp time}) > \text{limit}$$

Simultaneous transitions on the reference and data signals can never cause **\$skew** to report a timing violation, even when the skew limit value is zero.

The **\$skew** timing check is event-based; it is evaluated only after a data event. If there is never a data event (i.e., the data event is infinitely late), the **\$skew** timing check shall never be evaluated, and no timing violation shall ever be reported. In contrast, the **\$timeskew** and **\$fullskew** checks are timer-based by default, and they shall be used if violation reports are absolutely required and the data event can be very late or even absent altogether. These checks are discussed in 15.3.2 and 15.3.3.

\$skew shall wait indefinitely for the data event once it has detected a reference event and it shall not report a timing violation until the data event takes place. A second consecutive reference event shall cancel the old wait for the data event and begin a new one.

After a reference event, the **\$skew** timing check shall never stop checking data events for a timing violation. **\$skew** shall report timing violations for all data events occurring beyond the limit after a reference event.

15.3.2 \$timeskew

The syntax for **\$timeskew** is shown in Syntax 15-10.

```
$timeskew_timing_check ::= (From Annex A - A.7.5.1)
    $timeskew ( reference_event , data_event , timing_check_limit
        [ , [ notify_reg ] ] [ , [ event_based_flag ] ] [ , [ remain_active_flag ] ] ] );
data_event ::= (From Annex A - A.7.5.2)
    timing_check_event
event_based_flag ::=
    constant_expression
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
remain_active_flag ::=
    constant_mintypmax_expression
timing_check_limit ::=
    expression
```

Syntax 15-10 Syntax for \$timeskew

Table 56 defines the **\$timeskew** timing check arguments.

Table 56—\$timeskew arguments

Argument	Description
reference_event	Timestamp event
data_event	Timecheck event
limit	Non-negative constant expression
notifier (optional)	Reg
event_based_flag (optional)	Constant expression
remain_active_flag (optional)	Constant expression

The **\$timeskew** timing check reports a violation only in the following cases:

$$(\text{timecheck time}) - (\text{timestamp time}) > \text{limit}$$

Simultaneous transitions on the reference and data signals can never cause **\$timeskew** to report a timing violation, even when the skew limit value is zero.

The default behavior for **\$timeskew** is timer-based. Violations are reported immediately upon an elapse of time after the reference event equal to the limit, and the check shall become dormant and report no more violations (even in response to data events) until after the next reference event. This check shall also become dormant if it detects a reference event when its condition is false.

The **\$timeskew** check's default timer-based behavior can be altered to event-based using the event based flag. It behaves like the **\$skew** check when both the event based flag and the remain active flag are set. The **\$timeskew** check behaves like the **\$skew** check when only the event based flag is set, except it becomes dormant after reporting the first violation.

Example:

```
$timeskew (posedge CP &&& MODE, negedge CPN, 50);
```

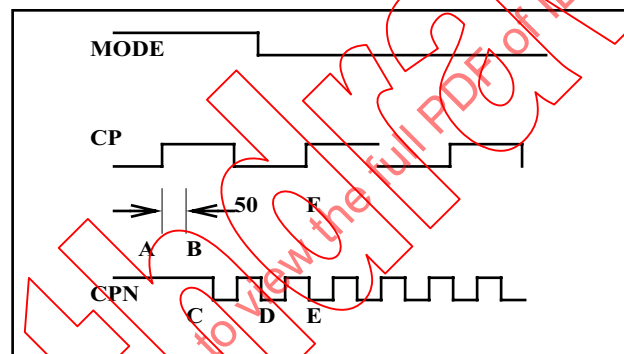


Figure 46—Sample \$timeskew

Case 1: Event based flag and remain active flag not set.

After the first reference event on **CP** at A, a violation is reported at B as soon as 50 time units have passed. No further violations are reported.

Case 2: Event based flag set, remain active flag not set.

The negative transition on **CPN** at point C shall cause a timing violation. Subsequent negative transitions at points D and E do not cause violations. The second reference event at F occurs while **MODE** is false, turning the **\$timeskew** check dormant, and no further violations are reported.

Case 3: Event based flag set, remain active flag set.

The first three negative transitions on **CPN** at points C, D and E shall cause timing violations. The second reference event at F occurs while **MODE** is false, turning the **\$timeskew** check dormant, and no further violations are reported.

Case 4: Event based flag and remain active flag both set.

Every negative edge on **CPN** is reported as a violation, which is identical to **\$skew** behavior.

15.3.3 \$fullskew

The syntax for **\$fullskew** is shown in Syntax 15-11.

```

$fullskew_timing_check ::= (From Annex A - A.7.5.1)
    $fullskew ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notify_reg ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
    timing_check_event
event_based_flag ::=
    constant_expression
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
remain_active_flag ::=
    constant_mintypmax_expression
timing_check_limit ::=
    expression
    
```

Syntax 15-11 Syntax for \$fullskew

Table 57 defines the **\$fullskew** timing check arguments.

Table 57 – \$fullskew arguments

Argument	Description
reference_event	Timestamp or timecheck event
data_event	Timestamp or timecheck event
limit 1	Non-negative constant expression
limit 2	Non-negative constant expression
notifier (optional)	Reg
event_based_flag (optional)	Constant expression
remain_active_flag (optional)	Constant expression

\$fullskew is identical to **\$timeskew** except the reference and data events can transition in either order. The first limit is the maximum time by which the data event can follow the reference event. The second limit is the maximum time by which the reference event can follow the data event.

The reference event is the timestamp event and the data event is the timecheck event when the reference event precedes the data event. The data event is the timestamp event and the reference event is the timecheck event when the data event precedes the reference event.

The **\$fullskew** timing check reports a violation only in the following case, where limit is set to limit1 when the reference event transitions first, and to limit2 when the data event transitions first:

$$(\text{timecheck time}) - (\text{timestamp time}) > \text{limit}$$

Simultaneous transitions on the reference and data signals shall never cause **\$fullskew** to report a timing violation, even when the skew limit value is zero.

The default behavior for **\$fullskew** is timer-based. Violations shall be reported immediately upon an elapse of time after the timestamp event equal to the limit. It then becomes dormant and reports no more violations, even in response to timecheck events, until after the next timestamp event. This check shall also become dormant if it detects a timestamp event when the associated condition is false.

The **\$fullskew** check's default timer-based behavior can be altered to event-based using the event based flag. It behaves like the **\$skew** check when both the event based flag and the remain active flag are set. The **\$timeskew** check behaves like the **\$skew** check when only the event based flag is set, except it becomes dormant after it reports the first violation.

Example:

\$fullskew (posedge CP && MODE, negedge CPN, 50, 70),

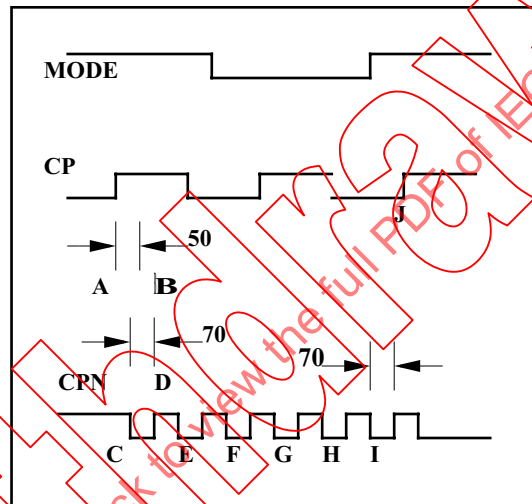


Figure 47—Sample \$fullskew

Case 1: Event based flag and remain active flag not set.

The transition at A of **CP** while **MODE** is true begins a wait for a negative transition on **CPN**, and a violation is reported at B as soon as a period of time equal to 50 time units has passed. This resets the check and readies it for the next active transition.

A negative transition on **CPN** occurs next at C, beginning a wait for a positive transition on **CP** while **MODE** is true. At D a time equal to 70 time units has passed without a positive edge on **CP** while **MODE** is true, so a violation is reported and the check is again reset to await the next active transition.

A transition on **CPN** at E also results in a timing violation, as does the transition at F, because even though **CP** transitions, **MODE** is no longer true. Transitions at G and H also result in timing violations, but not the transition at I, because it is followed by a positive transition on **CP** while **MODE** is true.

Case 2: Event based flag set, remain active flag not set.

The transition at A of **CP** while **MODE** is true begins a wait for a negative transition on **CPN**, and a violation is reported at C on **CPN** because it occurs beyond the 50 time unit limit. This transition at C also begins a wait of 70 time units for a positive transition on **CP** while **MODE** is true. But for transitions on **CPN** at B

through H there is no positive transition on CP while MODE is true, and so no timing violations are reported. The transition at I on CPN begins a wait of 70 time units, and this is satisfied by the positive transition on CP at J while MODE is true.

Case 3: Event based flag and remain active flag both set.

The transition at A of CP while MODE is true begins a wait for a negative transition on CPN, and a violation is reported at C on CPN, and it shall also begin a wait for a positive transition on CP while MODE is true. No such transition on CP ever takes place after CPN transitions C through H, but no violations are reported because CP never experiences a positive transition while MODE is true. Transition I also reports no violation because a positive transition at I on CP while MODE is true occurs within the 70 time unit skew limit.

15.3.4 \$width

The \$width timing check syntax is shown in Syntax 15-12.

```
$width_timing_check ::= (From Annex A - A.7.5.1)
    $width ( controlled_reference_event , timing_check_limit ,
        threshold [ , [ notify_reg ] ] ) ;
controlled_reference_event ::= (From Annex A - A.7.5.2)
    controlled_timing_check_event
notify_reg ::=
    variable_identifier
threshold ::=
    constant_expression
timing_check_limit ::=
    expression
```

Syntax 15-12 Syntax for \$width

If the comma before the threshold is present, the comma before the notifier shall also be present, even though both arguments are optional.

Table 58 defines the \$width timing check.

Table 58—\$width arguments

Argument	Description
reference_event	Timestamp edge triggered event
(data_event - implicit)	Timecheck edge triggered event
limit	Non-negative constant expression
threshold (optional)	Non-negative constant expression
notifier (optional)	Reg

The \$width timing check monitors the width of signal pulses by measuring the time from the timestamp event to the timecheck event. 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 shall occur if the reference event is not an edge specification.

While the **\$width** timing check can be defined in terms of a time window, it is simpler to express it as the difference between the timecheck and timestamp times. The **\$width** timing check reports a violation in the following case:

$$\text{threshold} < (\text{timecheck time}) - (\text{timestamp time}) < \text{limit}$$

The pulse width has to be greater than or equal to limit in order to avoid a timing violation, but no violation is reported for glitches smaller than the threshold.

The threshold argument shall be included if the notifier argument is required. It is permissible to not specify both the threshold and notifier arguments, making the default value for the threshold zero (0). If the notifier is present, a non-null value for the threshold shall also be present. Here is a legal **\$width** check when the notifier is required and the threshold is not:

```
$width (posedge clk, 6, 0, ntfr_reg);
```

The data event and the reference event shall never occur at the same simulation time because these events are triggered by opposite transitions.

Example:

The following example demonstrates some examples of legal and illegal calls:

```
// Legal Calls
$width ( negedge clr, lim );
$width ( negedge clr, lim, thresh, notif );
$width ( negedge clr, lim, 0, notif );

// Illegal Calls
$width ( negedge clr, lim, , notif );
$width ( negedge clr, lim, notif );
```

15.3.5 \$period

The **\$period** timing check syntax is shown in Syntax 15-13.

```
$period_timing_check ::= (From Annex A - A.7.5.1)
    $period ( controlled_reference_event , timing_check_limit [ , [ notify_reg ] ] );
controlled_reference_event ::= (From Annex A - A.7.5.2)
    controlled_timing_check_event
notify_reg ::=
    variable_identifier
timing_check_limit ::=
    expression
```

Syntax 15-13 Syntax for \$period

Table 59 defines the **\$period** timing check.

Table 59—\$period arguments

Argument	Description
reference_event	Timestamp edge triggered event
(data_event - implicit)	Timestamp edge triggered event
limit	Non-negative constant expression
notifier (optional)	Reg

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 shall be passed as the reference event. A compilation error shall occur if the reference event is not an edge specification.

While the **\$period** timing check can be defined in terms of a time window, it is simpler to express it as the difference between the timecheck and timestamp times. The **\$period** timing check reports a violation in the following case:

(timecheck time) - (timestamp time) < limit

15.3.6 \$nochange

The **\$nochange** syntax is shown in Syntax 15-14.

```
$nochange_timing_check := (From Annex A - A.7.5.1)
$nochange ( reference_event , data_event , start_edge_offset ,
            end_edge_offset [ , [ notify_reg ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
            timing_check_event
end_edge_offset ::=
            mintypmax_expression
notify_reg ::=
            variable_identifier
reference_event ::=
            timing_check_event
start_edge_offset ::=
            mintypmax_expression
```

Syntax 15-14 Syntax for \$nochange

Table 60 defines the **\$snochange** timing check arguments.

Table 60—\$snochange arguments

Argument	Description
reference_event	Edge triggered timestamp and/or timecheck event
data_event	Timestamp or timecheck event
start_edge_offset	Constant expression
end_edge_offset	Constant expression
notifier (optional)	Reg

The **\$snochange** timing check reports a timing violation if the data event occurs during the specified level of the control signal (the reference event). The reference event can be specified with the **posedge** or the **negedge** keyword, but the edge control specifiers (see 15.4) 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 after the edge. For example, if the reference event is a **posedge**, then the duration is the period during which the reference signal is high. A positive offset for start edge extends the region by starting the timing violation region earlier; a negative offset for start edge shrinks the region by starting the region later. Similarly, a positive offset for the end edge extends the timing violation region by ending it later, while a negative offset for the end edge shrinks the region by ending it earlier. If both the offsets are zero, the size of the region shall not change.

Unlike other timing checks, **\$snochange** involves three, rather than two, transitions. The leading edge of the reference event defines the beginning of the time window, while the trailing edge of the reference event defines the end of the time window. A violation results if the data event occurs anytime within the time window.

The endpoints of the time window are determined as follows:

$$\begin{aligned} \text{(beginning of time window)} &= \\ &= \text{(leading reference edge time)} - \text{start\_edge\_offset} \\ \text{(end of time window)} &= \text{(trailing reference edge time)} + \\ &+ \text{end\_edge\_offset} \end{aligned}$$

The **\$snochange** timing check reports a timing violation in the following case:

$$\text{(beginning of time window)} < \text{(data event time)} < \text{(end of time window)}$$

The endpoints of the time window are not included. The values of **start_edge_offset** and **end_edge_offset** play a significant role in determining which signal, the reference event or the data event, is the timestamp or timecheck event.

Example:

```
$snochange( posedge clk, data, 0, 0) ;
```

In this example, **\$snochange** system task shall report a violation if the **data** signal changes while **clk** is high. It shall not be a violation if **posedge clk** and a transition on **data** occur simultaneously.

15.4 Edge-control specifiers

The edge-control specifiers can be used to control events in timing checks based on specific edge transitions between 0, 1, and x. Syntax 15-15 shows the syntax for edge-control specifiers.

```

edge_control_specifier ::= (From Annex A - A.7.5.3)
    edge [ edge_descriptor [ , edge_descriptor ] ]
edge_descriptor* ::=
    01
    | 10
    | z_or_x zero_or_one
    | zero_or_one z_or_x
zero_or_one ::= 0 | 1
z_or_x ::= x | X | z | Z

```

*Embedded spaces are illegal.

Syntax 15-15 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 can be used as a shorthand for certain edge-control specifiers. For example, the construct:

posedge clr

is equivalent to the following:

edge[01, 0x, x1] clr

Similarly, the construct

negedge clr

is the same as the following:

edge[10, x0, 1x] clr

However, edge-control specifiers offer the flexibility to declare edge transitions other than **posedge** and **negedge**.

15.5 Notifiers: user-defined responses to timing violations

Timing check notifiers detect timing check violations behaviorally, and, therefore, take an action as soon as a violation occurs. Such notifiers can be used to print an informative error message describing the violation or to propagate an X value at the output of the device which reported the violation.

The notifier is a reg declared in the module where timing check tasks are invoked which 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 61 shows how the notifier values are toggled when timing violations occur.

Table 61 — User-defined responses to timing violations

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 following example uses a notifier to set the D flip-flop output to X when a timing violation occurs in an edge-sensitive UDP.

```

primitive posdff_udp(q, clock, data, preset, clear, notifier);
output q; reg q;
input clock, data, preset, clear, notifier;
table
//clock data  p c notifier state  q
//-----
r    0    1 1    ?    :  ?  : 0 ;
r    1    1 1    ?    :  ?  : 1 ;

p    1    ? 1    ?    :  1  : 1 ;
p    0    1 ?    ?    :  0  : 0 ;

n    ?    ? ?    ?    :  ?  : - ;
?    *    ? ?    ?    :  ?  : - ;

?    ?    0 1    ?    :  ?  : 1 ;
?    ?    * 1    ?    :  1  : 1 ;

?    ?    1 0    ?    :  ?  : 0 ;
?    ?    1 *    ?    :  0  : 0 ;
?    ?    ? ?    *    :  ?  : x ; // At any notifier event
                                     // output x

endtable
endprimitive

```

```

module dff(q, qbar, clock, data, preset, clear);
output q, qbar;
input clock, data, preset, clear;
reg notifier;

and (enable, preset, clear);
not (qbar, ffout);
buf (q, ffout);
posedge_udp (ffout, clock, data, preset, clear, notifier);

specify
  // Define timing check specparam values
  specparam tSU = 10, tHD = 1, tPW = 25, tWPC = 10, tREC = 5;
  // Define module path delay rise and fall min:typ:max values
  specparam tPLHc = 4:6:9 , tPHLc = 5:8:11;
  specparam tPLHpc = 3:5:6 , tPHLpc = 4:7:9;

  // Specify module path delays
  (clock *> q,qbar) = (tPLHc, tPHLc);
  (preset,clear *> q,qbar) = (tPLHpc, tPHLpc);

  // Setup time : data to clock, only when preset and clear are 1
  $setup(data, posedge clock &&& enable, tSU, notifier);

  // Hold time: clock to data, only when preset and clear are 1
  $hold(posedge clock, data &&& enable, tHD, notifier);

  // Clock period check
  $period(posedge clock, tPW, notifier);
  // Pulse width : preset, clear
  $width(negedge preset, tWPC, 0, notifier);
  $width(negedge clear, tWPC, 0, notifier);

  // Recovery time: clear or preset to clock
  $recovery(posedge preset, posedge clock, tREC, notifier);
  $recovery(posedge clear, posedge clock, tREC, notifier);
endspecify
endmodule

```

NOTE—This model applies to edge-sensitive UDPs only; for level-sensitive models, an additional UDP for X propagation has to be generated.

15.5.1 Requirements for accurate simulation

In order to accurately model negative value timing checks:

- a) A timing violation shall be triggered if the signal changes in the violation window, exclusive of the endpoints. Violation windows smaller than two units of simulation precision can not yield timing violations.
- b) The value of the latched data shall be the one which is stable during the violation window, again, exclusive of the endpoints.

To facilitate these modeling requirements, delayed copies of the data and reference signals are generated in the timing checks, and these are used internally for timing check evaluation at run-time. The setup and hold times used internally are adjusted so as to shift the violation window and make it overlap the reference signal.

Delayed data and reference signals can be declared within the timing check so they can be used in the model's functional implementation to insure accurate simulation. If no delayed signals are declared in the timing check, and if a negative setup or hold value is present, then implicit delayed signals are created. Since implicit delayed signals can not be used in defining model behavior, such a model can possibly behave incorrectly.

Examples:

Example 1:

```
$setuphold (posedge CLK, DATA, -10, 20);
```

Implicit delayed signals shall be created for CLK and DATA, but it shall not be possible to access them. The **\$setuphold** check shall be properly evaluated, but functional behavior shall not always be accurate. The old DATA value shall be incorrectly clocked in if DATA transitions between **posedge CLK** and 10 time units later.

Example 2:

```
$setuphold (posedge CLK, DATA1, -10, 20);  
$setuphold (posedge CLK, DATA2, -15, 18);
```

Implicit delayed signals shall be created for CLK, DATA1 and DATA2, one for each. Even though CLK is referenced in two different timing checks, only one implicit delayed signal is created, and it is used for both timing checks.

Example 3:

If a given signal has a delayed signal in some timing checks but not in others, the delayed signal shall be used in both cases:

```
$setuphold (posedge CLK, DATA1, -10, 20, , , del_CLK, del_DATA1);  
$setuphold (posedge CLK, DATA2, -15, 18);
```

Explicit delayed signals of **del_CLK** and **del_DATA1** are created for CLK and DATA1, while an implicit delayed signal is created for DATA2. In other words, CLK has only one delayed signal created for it, **del_CLK**, rather than one explicit delayed signal for the first check, and another implicit delayed signal for the second check.

The delayed versions of the signals, whether implicit or explicit, shall be used in the **\$setup**, **\$hold**, **\$setuphold**, **\$recovery**, **\$removal**, **\$crem**, **\$width**, **\$period** and **\$nochange** timing checks, and these checks shall have their limits adjusted accordingly. This ensures the notifier shall be toggled at the proper moment. If the adjusted limit becomes less than or equal to 0, the limit shall be set to 0 and the simulator shall issue a warning.

The delayed versions of the signals shall not be used for the **\$skew**, **\$fullskew** and **\$timeskew** timing checks because it can possibly result in the reversal of the order of signal transitions. This causes the notifiers for these timing checks to toggle at the wrong time relative to the rest of the model, perhaps resulting in transitions to X due to a timing check violation being canceled. This issue shall be addressed in the model, possibly by using separate notifiers for these checks.

It is possible for a set of negative timing check values to be mutually inconsistent and produce no solution for the delay values of delayed signals. In these situations the simulator shall issue a warning message. The inconsistency shall be resolved by changing the smallest negative limit value to 0 and recalculating the delays for the delayed signals, and this shall be repeated until a solution is reached. This procedure shall always produce a solution because in the worst case all negative limit values become 0, and no delayed signals are needed.

The delayed timing check signals are only actually delayed when negative limit values are present. If a timing check signal becomes delayed by more than the propagation delay from that signal to an output, that output shall take longer than its propagation delay to change. It shall instead transition at the same time which the delayed timing check signal changes. Thus, the output shall behave as if its specify path delay were equal to the delay applied to the timing check signal. This situation can only arise when unique setup/hold or removal/recovery times are given for each edge of the data signal.

Example:

```
(CLK = Q) = 6;
$setuphold (posedge CLK, posedge D, -3, 8, , , , dCLK, dD);
$setuphold (posedge CLK, negedge D, -7, 13, , , , dCLK, dD);
```

The setup time of -7 (the larger in absolute value of -3 and -7) creates a delay of 7 for dCLK, and so output Q shall not change until 7 time units after a positive edge on CLK, rather than the 6 time units given in the specify path.

15.5.2 Conditions in negative timing checks

Conditions can be associated with both the reference and data signals by using the &&& operator, but when either the setup or hold time is negative the conditions need to be paired with reference and data signals in a more flexible way. This example illustrates why.

This pair of **\$setup** and **\$hold** checks work together to provide the same check as a single **\$setuphold**:

```
$setup (data, clk&&&cond1, tsetup, ntfr);
$hold (clk, data&&&cond1, thold, ntfr);
```

clk is the timecheck event for the **\$setup** check, while data is the timecheck event for the **\$hold** check. This can not be represented in a single **\$setuphold** check, and so additional arguments are provided to make this possible. These arguments are timestamp_cond and timecheck_cond, and they immediately follow the notifier (see 15.2.3). This **\$setuphold** check is equivalent to the separate **\$setup** and **\$hold** checks shown above:

```
$setuphold( clk, data, tsetup, thold, ntfr, , cond1);
```

The timestamp_cond argument is null, while the timecheck_cond argument is cond1.

The timestamp_cond and timecheck_cond arguments are associated with either the reference or data signals based on which delayed version of these signals occurs first. timestamp_cond is associated with the delayed signal which transitions first, while timecheck_cond is associated with the delayed signal which transitions second.

Delayed signals are only created for the reference and data signals, and not for any condition signals associated with them. Therefore, timestamp_cond and timecheck_cond are not implicitly delayed by the simulator. Delayed condition signals for the timestamp_cond and timecheck_cond fields can be created by making them a function of the delayed signals.

Example:

```

assign TE_cond_D = (dTE !== 1'b1);
assign TE_cond_TI = (dTE !== 1'b0);
assign DXTI_cond = (dTI !== dD);

specify
  $setuphold(posedge CP, D, -10, 20, notifier, ,TE_cond_D, dCP, dD);
  $setuphold(posedge CP, TI, 20, -10, notifier, ,TE_cond_TI, dCP, dTI);
  $setuphold(posedge CP, TE, -4, 8, notifier, ,DXTI_cond, dCP, dTE);
endspecify

```

The assign statements create condition signals which are functions of the delayed signals. Creating delayed signal conditions synchronizes the conditions with the delayed versions of the reference and data signals used to perform the checks.

The first **\$setuphold** has a negative setup time, and so the timecheck condition **TE_cond_D** is associated with data signal **D**. The second **\$setuphold** has a negative hold time, and so the timecheck condition **TE_cond_TI** is associated with reference signals **CP**. The third **\$setuphold** has a negative setup time, and so the timecheck condition **DXTI_cond** is associated with data signal **TE**.

The violation windows for the example are shown in Figure 48.

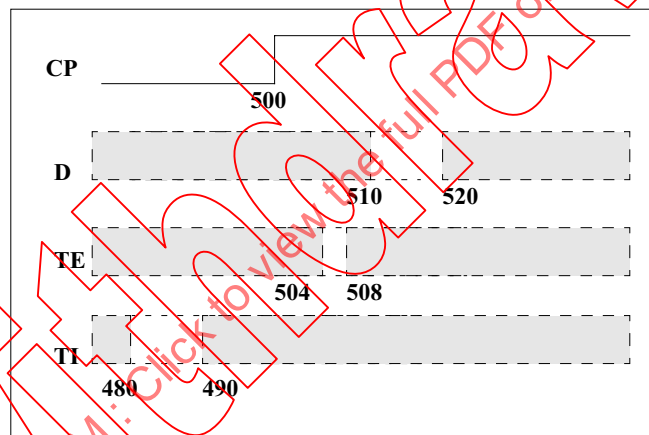


Figure 48—Timing check violation windows

These are the delay values calculated for the delayed signals:

dCP	10.01
dD	0.00
dTI	20.02
dTE	2.02

Use of delayed signals in creating the signals for the **timestamp_cond** and **timecheck_cond** arguments is not required, but it is usually closer to actual device behavior.

15.5.3 Notifiers in negative timing checks

Because the reference and data signals are delayed internally, the detection of the timing violation is also delayed. Notifier regs in negative timing checks shall be toggled when the timing check detects a timing violation, which occurs when the delayed signals as measured by the adjusted timing check values are in violation, not when the undelayed signals at the model inputs as measured by the original timing check values are in violation.

15.5.4 Option behavior

As already mentioned, the ability of Verilog simulators to handle negative values in **\$setuphold** and **\$recrem** timing checks shall be enabled with an invocation option. It is possible models written to accept negative timing check values with delayed reference and/or delayed data signals can be run without this invocation option enabled. In this circumstance the delayed reference and data signals become copies of the original reference and data signals. The same occurs if an invocation option turning off all timing checks is used.

15.6 Enabling timing checks with conditioned events

A construct called a conditioned event ties the occurrence of timing checks to the value of a conditioning signal. Syntax 15-16 shows the syntax for controlled timing check event.

```

timing_check_event ::= (From Annex A - A.7.5.3)
    [timing_check_event_control] specify_terminal_descriptor [ &&& timing_check_condition ]
controlled_timing_check_event ::=
    timing_check_event_control specify_terminal_descriptor [ &&& timing_check_condition ]
timing_check_event_control ::=
    posedge
  | negedge
  | edge_control_specifier
specify_terminal_descriptor ::=
    specify_input_terminal_descriptor
  | specify_output_terminal_descriptor
timing_check_condition ::=
    scalar_timing_check_condition
  | ( scalar_timing_check_condition )
scalar_timing_check_condition ::=
    expression
  | ~ expression
    expression == scalar_constant
    expression === scalar_constant
    expression != scalar_constant
    expression !== scalar_constant
scalar_constant ::=
    1'b0 | 1'b1 | 1'B0 | 1'B1 | 'b0 | 'b1 | 'B0 | 'B1 | 1 | 0

```

Syntax 15-16 Syntax for controlled timing check event

The comparisons used in the condition can be deterministic, as in ==, !=, ~, or no operation, or nondeterministic, as in == or !=. When comparisons are deterministic, an x value on the conditioning signal shall not enable the timing check. For nondeterministic comparisons, an x on the conditioning signal shall enable the timing check.

The conditioning signal shall be a scalar net; if a vector net or an expression resulting in a 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 can 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 shall occur every time there is a positive edge on the signal `clk`.

To trigger the setup check on the positive edge on the signal `clk` only when the signal `clr` is high, rewrite the command as

```
$setup( data, posedge clk &&& clr, 10 );
```

Example 2 This example shows two ways to trigger the same timing check as in example 1 (on the positive `clk` edge) only when `clr` is low. The second method uses the `===` operator, which makes the comparison deterministic.

```
$setup( data, posedge clk &&& (~clr), 10 );  
$setup( data, posedge clk &&& (clr===0), 10 );
```

Example 3 To perform the previous sample setup check on the positive `clk` edge only when `clr` and `set` are high, add the following statement outside the specify block:

```
and new_gate( clr_and_set, clr, set );
```

Then add the condition to the timing check using the signal `clr_and_set` as follows:

```
$setup( data, posedge clk &&& clr_and_set, 10 );
```

15.7 Vector signals in timing checks

Either or both signals in a timing check can be a vector. This shall be interpreted as a single timing check where the transition of one or more bits of a vector is considered a single transition of that vector.

Example:

```
module DFF (Q, CLK, DAT);  
input CLK;  
input [7:0] DAT;  
output [7:0] Q;  
always @(posedge clk)  
Q = DAT;  
specify  
$setup (DAT, posedge CLK, 10);  
endspecify  
endmodule
```

If `DAT` transitions from `'b00101110` to `'b01010011` at time 100, and `CLK` transitions from 0 to 1 at time 105, then the **\$setup** timing check shall still only report a single timing violation.

Simulators can provide an option causing vectors in timing checks to result in the creation of multiple single-bit timing checks. For timing checks with only a single signal, such as **\$period** or **\$width**, a vector of width N results in N unique timing checks. For timing checks with two signals, such as **\$setup**, **\$hold**, **\$setuphold**, **\$skew**, **\$timeskew**, **\$fullskew**, **\$recovery**, **\$removal**, **\$recrem** and **\$nochange**, where M and N are the widths of the signals, the result is M*N unique timing checks. If there is a notifier, all the timing checks trigger that notifier.

With such an option enabled, the above example yields six timing violation because six bits of DAT transitioned.

15.8 Negative timing checks

Both the **\$setuphold** and **\$recrem** timing checks can accept negative values when the negative timing check option is enabled. The behavior of these two timing checks is identical with respect to negative values. The descriptions in this section are for the **\$setuphold** timing check, but apply equally to the **\$recrem** timing check.

The setup and hold timing check values define a timing violation window with respect to the reference signal edge during which the data shall remain constant. Any change of the data during the specified window causes a timing violation. The timing violation is reported and, through the notifier reg, other actions can take place in the model, such as forcing the output of a flip-flop to X when it detects a timing violation.

A positive value for both setup and hold times implies this violation window straddles the reference signal shown in Figure 49.

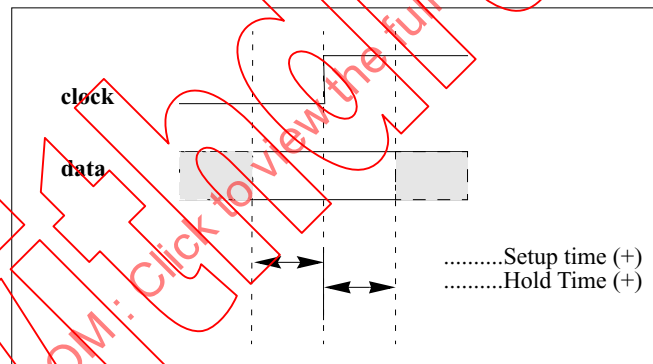


Figure 49—Data constraint interval, positive setup/hold

A negative hold or setup time means the violation window is shifted to either before or after the reference edge. This can happen in a real device because of disparate internal device delays between the internal clock and data signal paths. These internal device delays are illustrated in Figure 50 showing how significant differences in these delays can cause negative setup or hold values.

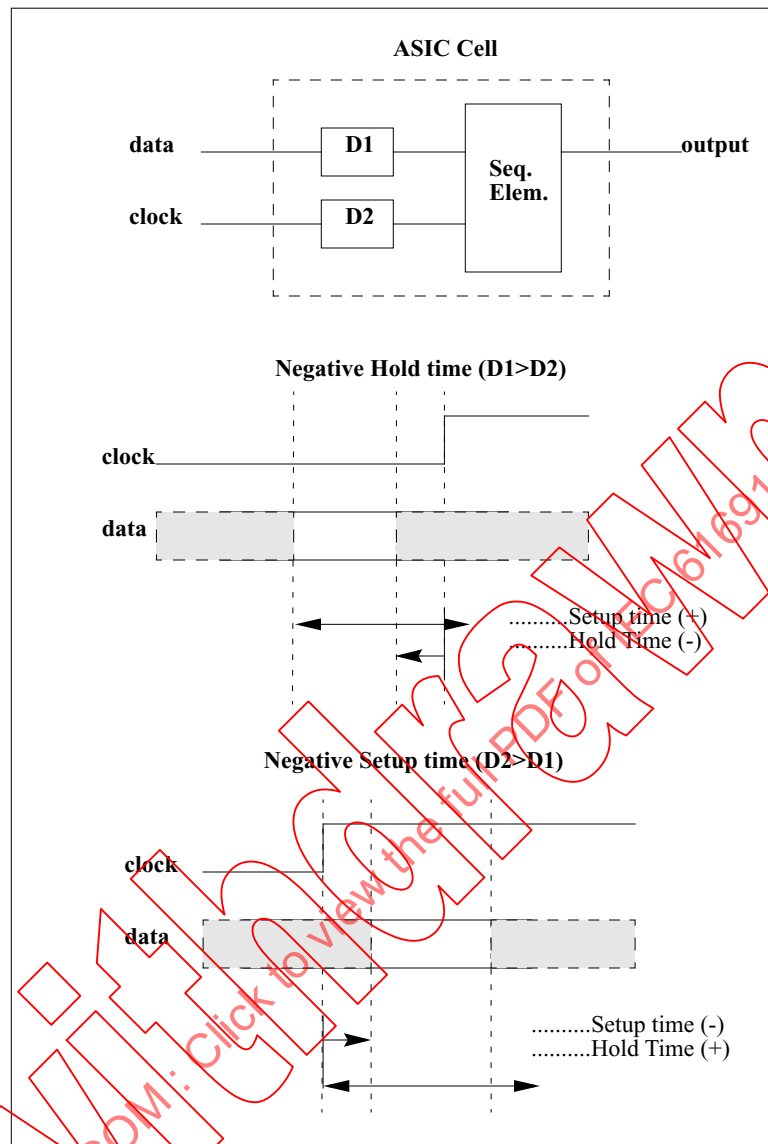


Figure 50—Data constraint interval, negative setup/hold

16. Backannotation using the Standard Delay Format (SDF)

SDF files contain timing values for specify path delays, specparam values, timing check constraints, and interconnect delays. SDF files can also contain other information in addition to simulation timing, but these need not concern Verilog simulation. The timing values in SDF files usually come from ASIC delay calculation tools that take advantage of connectivity, technology, and layout geometry information.

Verilog backannotation is the process by which timing values from the SDF file update specify path delays, specparam values, timing constraint values, and interconnect delays.

All this information is covered further in IEEE Std 1497-1999 [B2].

16.1 The SDF annotator

The term SDF Annotator refers to any tool capable of backannotating SDF data to a Verilog simulator. It shall report a warning for any data it is unable to annotate.

An SDF file can contain many constructs which are not related to specify path delays, specparam values, timing check constraint values, or interconnect delays. An example is any construct in the TIMINGENV section of the SDF file. All constructs unrelated to Verilog timing shall be ignored without any warnings issued.

Any Verilog timing value for which the SDF file does not provide a value shall not be modified during the backannotation process, and its pre-backannotation value shall be unchanged.

16.2 Mapping of SDF constructs to Verilog

SDF timing values appear within a CELL declaration which can contain one or more of DELAY, TIMINGCHECK and LABEL sections. The DELAY section contains propagation delay values for specify paths and interconnect delays. The TIMINGCHECK section contains timing check constraint values. The LABEL section contains new values for specparams. Backannotation into Verilog is done by matching SDF constructs to the corresponding Verilog declarations, then replacing the existing Verilog timing values with those from the SDF file.

16.2.1 Mapping of SDF delay constructs to Verilog declarations

When annotating DELAY constructs that are not interconnect delays (covered in 16.2.3), the SDF annotator looks for specify paths where the names and conditions match. When annotating TIMINGCHECK constructs, the SDF annotator looks for timing checks of the same type where the names and conditions match. Table 62 shows which Verilog structures can be annotated by each SDF construct in the DELAY section.

Table 62—Mapping of SDF delay constructs to Verilog declarations

SDF Construct	Verilog annotated structure
(PATHPULSE...	Conditional and non-conditional specify path pulse limits
(PATHPULSEPERCENT...	Conditional and non-conditional specify path pulse limits
(IOPATH...	Conditional and non-conditional specify path delays/pulse limits
(IOPATH (RETAIN...	Conditional and non-conditional specify path delays/pulse limits, RETAIN ignored without warning
(COND (IOPATH...	Conditional specify path delays/pulse limits
(COND (IOPATH (RETAIN...	Conditional specify path delays/pulse limits, RETAIN ignored without warning
(CONDELSE (IOPATH...	ifnone
(CONDELSE (IOPATH (RETAIN...	ifnone, RETAIN ignored without warning
(DEVICE...	All specify paths to module outputs. If no specify paths, all primitives driving module outputs.
(DEVICE port_instance...	If port_instance is a module instance, all specify paths to module outputs. If no specify paths, all primitives driving module outputs. If port_instance is a module instance output, all specify paths to that module output. If no specify path, all primitives driving that module output.

In this example the source SDF signal `sel` matches the source Verilog signal, and the destination SDF signal `zout` also matches the destination Verilog signal, and so the rise/fall times of 1.3 and 1.7 are annotated to the specify path.

SDF file:

```
(IOPATH sel zout (1.3) (1.7))
```

Verilog specify path:

```
(sel => zout) = 0;
```

A conditional **IOPATH** delay between two ports shall annotate only to Verilog specify paths between those same two ports with the same condition. In this example the rise/fall times of 1.3 and 1.7 are annotated only to the second specify path.

SDF file:

```
(COND mode (IOPATH sel zout (1.3) (1.7)))
```

Verilog specify paths:

```
if (!mode) (sel => zout) = 0;
if (mode) (sel => zout) = 0;
```

A non-conditional **IOPATH** delay between two ports shall annotate to all Verilog specify paths between those same two ports. In this example the rise/fall times of 1.3 and 1.7 are annotated to both specify paths.

SDF file:

```
(IOPATH sel zout (1.3) (1.7))
```

Verilog specify paths:

```
if (!mode) (sel => zout) = 0;
if (mode) (sel => zout) = 0;
```

16.2.2 Mapping of SDF timing check constructs to Verilog

Table 63 shows which Verilog timing checks are annotated to by each type of SDF timing check. **v1** is the first value of a timing check, **v2** is the second value, while **x** indicates no value is annotated.

Table 63—Mapping of SDF timing check constructs to Verilog

SDF Timing Check	Annotated Verilog Timing checks
(SETUP v1...	\$setup(v1), \$setuphold(v1,x)
(HOLD v1...	\$hold(v1), \$setuphold(x,v1)
(SETUPHOLD v1 v2...	\$setup(v1), \$hold(v2), \$setuphold(v1,v2)
(RECOVERY v1...	\$recovery(v1), \$recrem(v1,x)
(REMOVAL v1...	\$removal(v1), \$recrem(x,v1)
(RECREM v1 v2...	\$recovery(v1), \$removal(v2), \$recrem(v1,v2)
(SKEW v1...	\$skew(v1)
(TIMESKEW v1... *	\$timeskew(v1)
(FULLSKEW v1 v2... *	\$fullskew(v1,v2)
(WIDTH v1...	\$width(v1,x)
(PERIOD v1...	\$period(v1)
(NOCHANGE v1 v2...	\$nochange(v1,v2)

* Not part of current SDF standard

Not usually implemented in Verilog simulators

The reference and data signals of timing checks can have logical condition expressions and edges associated with them. An SDF timing check with no conditions or edges on any of its signals shall match all corresponding Verilog timing checks regardless of whether conditions are present or not. In this example the SDF timing check shall annotate to all the Verilog timing checks:

SDF file:

```
(SETUPHOLD data clk (3) (4))
```

Verilog timing checks:

```
$setuphold (posedge clk&&& mode, data, 1, 1, ntfr);
$setuphold (negedge clk&&&!mode, data, 1, 1, ntfr);
```

When conditions and/or edges are associated with the signals in an SDF timing check, then they shall match those in any corresponding Verilog timing check before annotation shall happen. In this example the SDF timing check shall annotate to the first Verilog timing check, but not the second:

SDF file:

```
(SETUPHOLD data (posedge clk) (3) (4))
```

Verilog timing checks:

```
$setuphold (posedge clk&&& mode, data, 1, 1, ntfr); // Annotated
$setuphold (negedge clk&&&!mode, data, 1, 1, ntfr); // Not annotated
```

Here, the SDF timing check shall not annotate to any of the Verilog timing checks:

SDF file:

```
(SETUPHOLD data (COND !mode (posedge clk)) (3) (4))
```

Verilog timing checks:

```
$setuphold (posedge clk&&& mode, data, 1, 1, ntfr); // Not annotated
$setuphold (negedge clk&&&!mode, data, 1, 1, ntfr); // Not annotated
```

16.2.3 SDF annotation of specparams

The SDF LABEL construct annotates to specparams. Any expression containing one or more specparams is reevaluated when annotated to from an SDF file.

This example shows SDF LABEL constructs annotating to specparams in a Verilog module. The specparams are used in procedural delays to control when the clock transitions. The SDF LABEL construct annotates the values of dhigh and dlow, thereby setting the period and duty cycle of the clock.

SDF file:

```
(LABEL
  (ABSOLUTE
    (dhigh 60)
    (dlow 40)))
```

Verilog file:

```
module clock(clk);
  output clk;
  reg clk;
  specparam dhigh=0, dlow=0;
  initial clk = 0;
  always
  begin
    #dhigh clk = 1; // Clock remains low for time dlow
                  // before transitioning to 1
    #dlow  clk = 0; // Clock remains high for time dhigh
                  // before transitioning to 0
  end;
endmodule
```

This example shows a specparam in an expression of a specify path. The SDF LABEL construct can be used to change the value of the specparam and cause reevaluation of the expression:

```
specparam cap = 0;
...
specify
  (A => Z) = 1.4 * cap + 0.7;
endspecify
```

16.2.4 SDF annotation of interconnect delays

SDF interconnect delay annotation differs from annotation of other constructs described above in that there exists no corresponding Verilog declaration to which to annotate. In Verilog simulation, interconnect delays are an abstraction that represents the signal propagation delay from an output or inout module port to an input or inout module port. The INTERCONNECT construct includes a source, a load, and delay values, while the PORT and NETDELAY constructs include only a load and delay values. Interconnect delays can only be annotated between module ports, never between primitive pins. Table 64 shows how the SDF interconnect constructs in the DELAY section are annotated:

Table 64—SDF annotation of interconnect delays

SDF Construct	Verilog annotated structure
(PORT...	Interconnect delay
(NETDELAY *	Interconnect delay
(INTERCONNECT	Interconnect delay

*Only QVI SDF version 1.0, 2.0, and 2.1, and IEEE SDF version 4.0

Interconnect delays can be annotated to both single source and multi-source nets.

When annotating a PORT construct, the SDF annotator shall search for the port and if it exists shall annotate an interconnect delay to that port which shall represent the delay from all sources on the net to that port.

When annotating a NETDELAY construct, the SDF annotator shall check to see if it is annotating to a port or a net. If it is a port then the SDF annotator shall annotate an interconnect delay to that port. If it is a net then it shall annotate an interconnect delay to all load ports connected to that net. If the port or net has more than one source then the delay shall represent the delay from all sources. NETDELAY delays can only be annotated to input or inout module ports, or to nets.

In the case of multi-source nets, unique delays can be annotated between each source/load pair using the INTERCONNECT construct. When annotating this construct, the SDF annotator shall find the source port and the load port, and if both exist it shall annotate an interconnect delay between the two. If the source port is not found, or if the source port and the load port are not actually on the same net, then a warning message is issued, but the delay to the load port is annotated anyway. If this happens for a load port that is part of a multi-source net, then the delay is treated as if it were the delay from all source ports, which is the same as the annotation behavior for a PORT delay. Source ports shall be output or input ports, while load ports shall be input or inout ports.

Interconnect delays share many of the characteristics of specify path delays. The same rules for specify path delays for filling in missing delays and pulse limits also apply for interconnect delays. Interconnect delays have twelve transition delays, and unique reject and error pulse limits are associated with each of the twelve. An unlimited number of future schedules are permitted.

In a Verilog module, a reference to an annotated port, wherever it occurs, whether in \$monitor and \$display statements or in expressions, shall provide the delayed signal value. A reference to the source shall yield the undelayed signal value, while a reference to the load shall yield the delayed signal value. In general, references to the signal value hierarchically before the load shall yield the undelayed signal value, while references to the signal at or hierarchically after the load shall yield the delayed signal value. An annotation to a hierarchical port shall affect all connected ports at higher or lower hierarchical levels, depending on the direction of annotation. An annotation from a source port shall be interpreted as being from all sources hierarchically higher or lower than that source port.

Up-hierarchy annotations shall be properly handled. This situation arises when the load is hierarchically above the source. The delay to all ports hierarchically above the load or which connect to the net at points hierarchically above the load is the same as the delay to that load.

Down-hierarchy annotation shall also be properly handled. This situation arises when the source is hierarchically above the load. The delay to the load is interpreted as being from all ports at or above the source or which connect to the net at points hierarchically above the source.

Hierarchically overlapping annotations are permitted. This occurs when annotations to or from the same port take place at different hierarchical levels, and therefore do not correspond to the same hierarchical subset of ports. In this example, the first INTERCONNECT statement annotates to all ports of the net that are at or hierarchically within i53/se1mode, while the second annotates to a smaller subset of ports, only those at or hierarchically within i53/u21/in:

```
(INTERCONNECT i14/u5/out i53/se1mode (1.43) (2.17))
(INTERCONNECT i14/u5/out i53/u21/in (1.58) (1.92))
```

Overlapping annotations can occur in many different ways, particularly on multi-source/multi-load nets, and SDF annotation shall properly resolve all the interactions.

16.3 Multiple annotations

SDF annotation is an ordered process. The constructs from the SDF file are annotated in their order of occurrence. This means that annotation of an SDF construct can be changed by annotation of a subsequent construct that either modifies (INCREMENT) or overwrites (ABSOLUTE) it. These do not have to be the same construct. This example first annotates pulse limits to an IOPATH, then annotates the entire IOPATH, thereby overwriting the pulse limits that were just annotated:

```
(DELAY
  (ABSOLUTE
    (PATHPULSE A Z (2.1) (3.4))
    (IOPATH A Z (3.5) (6.1))
```

Overwriting the pulse limits can be avoided by using empty parentheses to hold the current values of the pulse limits.

```
(DELAY
  (ABSOLUTE
    (PATHPULSE A Z (2.1) (3.4))
    (IOPATH A Z ((3.5) ( ) ( )) ((6.1) ( ) ( )) )
```

The above annotation can be simplified into a single statement like this:

```
(DELAY
  (ABSOLUTE
    (IOPATH A Z ((3.5) (2.1) (3.4)) ((6.1) (2.1) (3.4)) )
```

A PORT annotation followed by an INTERCONNECT annotation to the same load shall cause only the delay from the INTERCONNECT source to be affected. For this net with three sources and a single load, the delay from all sources except i13/out remains 6:

```
(DELAY
  (ABSOLUTE
    (PORT i15/in (6))
    (INTERCONNECT i13/out i15/in (5))
```

An INTERCONNECT annotation followed by a PORT annotation shall cause the INTERCONNECT annotation to be overwritten. Here, the delays from all sources to the load shall become 6.

```
(DELAY
  (ABSOLUTE
    (INTERCONNECT i13/out i15/in (5))
    (PORT i15/in (6))
```

16.4 Multiple SDF files

More than one SDF file can be annotated. Each call to the **\$sdf_annotate** task annotates the design with timing information from an SDF file. Annotated values either modify (INCREMENT) or overwrite (ABSOLUTE) values from earlier SDF files. Different regions of a design can be annotated from different SDF files by specifying the region's hierarchy scope as the second argument to **\$sdf_annotate**.

16.5 Pulse limit annotation

For SDF annotation of delays (not timing constraints), the default values annotated for pulse limits shall be calculated using the percentage settings for the reject and error limits. By default these limits are 100%, but they can be modified through invocation options. For example, assuming invocation options have set the reject limit to 40% and the error limit to 80%, this SDF construct shall annotate a delay of 5, a reject limit of 2, and an error limit of 4:

```
(DELAY
  (ABSOLUTE
    (IOPATH A Z (5))
```

Given that the specify path delay was originally 0, this annotation results in a delay of 5 and pulse limits of 0:

```
(DELAY
  (ABSOLUTE
    (IOPATH A Z ((5) () ())) )
```

Annotations in INCREMENT mode can result in pulse limits less than 0, in which case they shall be adjusted to 0. For example, if the specify path pulse limits were both 3, this annotation results in a 0 value for both pulse limits:

```
(DELAY
  (INCREMENT
    (IOPATH A Z (( ) (-4) (-5)) )
```

There are two SDF constructs that annotate only to pulse limits, **PATHPULSE** and **PATHPULSEPERCENT**. They do not affect the delay. When **PATHPULSE** sets the pulse limits to values greater than the delay Verilog shall exhibit the same behavior as if the pulse limits had been set equal to the delay.

16.6 SDF to Verilog delay value mapping

Verilog specify paths and interconnects can have unique delays for up to twelve state transitions (see 14.3.1). All other constructs, such as gate primitives and continuous assignments, can have only three state transition delays (see 7.14).

For Verilog specify path and interconnect delays, the number of transition delay values provided by SDF might be less than twelve.

Table 65 shows how fewer than twelve SDF delays are extended to be twelve delays. The Verilog transition types are shown down the left hand side, while the number of SDF delays provided is shown across the top. The SDF values are given the names v1 through v12.

Table 65—SDF to Verilog delay value mapping

Verilog transition	Number of SDF delay values provided				
	1 value	2 values	3 values	6 values	12 values
0 -> 1	v1	v1	v1	v1	v1
1 -> 0	v1	v2	v2	v2	v2
0 -> z	v1	v1	v3	v3	v3
z -> 1	v1	v1	v1	v4	v4
1 -> z	v1	v2	v3	v5	v5
z -> 0	v1	v2	v2	v6	v6
0 -> x	v1	v1	min(v1,v3)	min(v1,v3)	v7
x -> 1	v1	v1	v1	max(v1,v4)	v8
1 -> x	v1	v2	min(v2,v3)	min(v2,v5)	v9
x -> 0	v1	v2	v2	max(v2,v6)	v10
x -> z	v1	max(v1,v2)	v3	max(v3,v5)	v11
z -> x	v1	min(v1,v2)	min(v1,v2)	min(v4,v6)	v12

For other delays that can have at most three values, the expansion of less than three SDF delays into three Verilog delays is covered in Table 39. More than three SDF delays are reduced to three Verilog delays by simply ignoring the extra delays. The delay to the X-state is created from the minimum of the other three delays.

17. System tasks and functions

This clause describes system tasks and functions that are considered part of the Verilog HDL. These system tasks and functions are divided into ten categories as follows:

Display tasks		[17.1]	PLA modeling tasks		[17.5]
\$display	\$strobe		\$async\$and\$array	\$async\$and\$plane	
\$displayb	\$strobeb		\$async\$nand\$array	\$async\$nand\$plane	
\$displayh	\$strobeh		\$async\$or\$array	\$async\$or\$plane	
\$displayo	\$strobo		\$async\$nor\$array	\$async\$nor\$plane	
\$monitor	\$write		\$sync\$and\$array	\$sync\$and\$plane	
\$monitorb	\$writeb		\$sync\$nand\$array	\$sync\$nand\$plane	
\$monitorh	\$writeh		\$sync\$or\$array	\$sync\$or\$plane	
\$monitro	\$wrote		\$sync\$nor\$array	\$sync\$nor\$plane	
\$monitoroff	\$monitoron				
File I/O tasks		[17.2]	Stochastic analysis tasks		[17.6]
\$fclose	\$fopen		\$q_initialize	\$q_add	
\$fdisplay	\$fstrobe		\$q_remove	\$q_full	
\$fdisplayb	\$fstrobeb		\$q_exam		
\$fdisplayh	\$fstrobeh		Simulation time functions		[17.7]
\$fdisplayo	\$fstrobo		\$realtime	\$stime	
\$fgetc	\$fputc		\$time		
\$fflush	\$ferror		Conversion functions		[17.8]
\$fgets	\$frewind		\$bittoreal	\$realtobits	
\$fmonitor	\$fwrite		\$itor	\$rtol	
\$fmonitorb	\$fwriteb		\$signed	\$unsigned	
\$fmonitorh	\$fwriteh		Probabilistic distribution functions		[17.9]
\$fmonitro	\$fwrote		\$dist_chi_square	\$dist_erlang	
\$readmemb	\$readmemh		\$dist_exponential	\$dist_normal	
\$swrite	\$swriteb		\$dist_poisson	\$dist_t	
\$swriteo	\$swriteh		\$dist_uniform	\$random	
\$sformat	\$sdf_annotate				
\$fscanf	\$sscanf				
\$fread	\$fseek				
Timescale tasks		[17.3]	Command line input		[17.10]
\$prnttimescale	\$timeformat		\$test\$plusargs	\$value\$plusargs	
Simulation control tasks		[17.4]			
\$finish	\$stop				

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 Clause 18.

17.1 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.

17.1.1 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 17-1 Syntax for \$display and \$write system tasks

These are the main system task routines for displaying information. The two sets of tasks are identical except that **\$display** automatically adds a newline character to the end of its output, whereas the **\$write** task does not.

The **\$display** and **\$write** tasks display their arguments in the same order as they appear in the argument list. Each argument can be a quoted string, an expression that returns a value, or a null argument.

The contents of string arguments are output literally except when certain escape sequences are inserted to display special characters or to specify the display format for a subsequent expression.

Escape sequences are inserted into a string in three ways:

- The special character `\` indicates that the character to follow is a literal or nonprintable character (see Table 66).
- 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 67). For each `%` character, with the exception of `%m` 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 66).

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.

17.1.1.1 Escape sequences for special characters

The escape sequences given in Table 66, when included in a string argument, cause special characters to be displayed.

Table 66—Escape sequences for printing special characters

Argument	Description
\n	The newline character
\t	The tab character
\\	The \ character
\"	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

```

Simulating this example shall display the following:

```

\  \
"S

```

17.1.1.2 Format specifications

Table 67 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 67—Escape sequences for format specifications

Argument	Description
%h or %H	Display in hexadecimal format
%d or %D	Display in decimal format
%o or %O	Display in octal format
%b or %B	Display in binary format
%c or %C	Display in ASCII character format
%l or %L	Display library binding information
%v or %V	Display net signal strength

Table 67—Escape sequences for format specifications (continued)

%m or %M	Display hierarchical name
%s or %S	Display as a string
%t or %T	Display in current time format
%u or %U	Unformatted 2 value data
%z or %Z	Unformatted 4 value data

The formatting specification %l (or %L) is defined for displaying the library information of the specific module. This information shall be displayed as "*library.cell*" corresponding to the library name the current module instance was extracted from and the cell name of the current module instance. See Clause 13 for information on libraries and configuring designs.

The formatting specification %u (or %U) is defined for writing data without formatting (binary values). The application shall transfer the 2 value binary representation of the specified data to the output stream. This escape sequence can be used with any of the existing display system tasks, although \$fwrite should be the preferred one to use. Any unknown or high-impedance bits in the source shall be treated as zero. This formatting specifier is intended to be used to support transferring data to and from external programs that have no concept of x and z. Applications that require preservation of x and z are encouraged to use the %z I/O format specification.

The data shall be written to the file in the native endian format of the underlying system (i.e., in the same endian order as if the PLI was used, and the C language write(2) system call was used). The data shall be written in units of 32 bits with the word containing the LSB written first.

NOTE For POSIX applications: It might be necessary to open files for unformatted I/O with the wb, wb+, or w+b specifiers, to avoid the systems implementation of I/O altering patterns in the unformatted stream that match special characters.

The formatting specification %z (or %Z) is defined for writing data without formatting (binary values). The application shall transfer the 4 value binary representation of the specified data to the output stream. This escape sequence can be used with any of the existing display system tasks, although \$fwrite should be the preferred one to use.

This formatting specifier is intended to be used to support transferring data to and from external programs that recognize and support the concept of x and z. Applications that do not require the preservation of x and z are encouraged to use the %u I/O format specification.

The data shall be written to the file in the native endian format of the underlying system (i.e., in the same endian order as if the PLI was used, and the data were in a s_vpi_vecval structure (See 27.14, Figure 179), and the C language write(2) system call was used to write the structure to disk). The data shall be written in units of 32 bits with the structure containing the LSB written first.

NOTE For POSIX applications: It might be necessary to open files for unformatted I/O with the wb, wb+ or w+b specifiers, to avoid the systems implementation of I/O altering patterns in the unformatted stream that match special characters.

The format specifications in Table 68 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 68—Format specifications for real numbers

Argument	Description
%e or %E	Display <code>real</code> in an exponential format
%f or %F	Display <code>real</code> in a decimal format
%g or %G	Display <code>real</code> in exponential or decimal format, whichever format results in the shorter printed output

The net signal strength, hierarchical name, and string format specifications are described in 17.1.1.5 through 17.1.1.7.

The `%t` format specification works with the **Stimeformat** 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 **Stimeformat** task is described in 17.3.2.

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

```

Simulating this example shall display the following:

[illegible]

17.1.1.3 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 can be overridden by inserting a zero between the % character and the letter that indicates the radix, as shown in the following example.

```
$display("d=%0h a=%0h", data, addr);
```

Example:

```
module printval;
reg [11:0] r1;
initial begin
    r1 = 10;
    $display( "Printing with maximum size - :%d: :%h:", r1,r1);
    $display( "Printing with minimum size - :%0d: :%0h:", r1,r1 );
end
endmodule

Printing with maximum size - : 10: :00a:
Printing with minimum size - :10: :a:
```

In this example, the result of a 12-bit expression is displayed. The first call to **\$display** uses the standard format specifier syntax and produces results requiring four and three columns for the decimal and hexadecimal radices, respectively. The second **\$display** call uses the %0 form of the format specifier syntax and produces results requiring two columns and one column, respectively.

17.1.1.4 Unknown and high impedance values

When the result of an expression contains an unknown or high impedance value, 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>	x
<code>\$display ("%h", 14'bx01010);</code>	xxXa
<code>\$display ("%h %o", 12'b001xxx101x01, 12'b001xxx101x01);</code>	XXX 1x5X

17.1.1.5 Strength format

The %v format specification is used to display the strength of scalar nets. For each %v specification that appears in a string, a corresponding scalar reference shall follow the string in the argument list.

The strength of a scalar net is reported in a three-character format. The first two characters indicate the strength. The third character indicates the current logic value of the scalar and can be any one of the values given in Table 69.

Table 69—Logic value component of strength format

Argument	Description
0	For a logic 0 value
1	For a logic 1 value
X	For an unknown value
Z	For a high impedance value
L	For a logic 0 or high impedance value
H	For a logic 1 or high impedance value

The first two characters the strength characters are either a two-letter mnemonic or a pair of decimal digits. Usually, a mnemonic is used to indicate strength information; however, in less typical cases, a pair of decimal digits can be used to indicate a range of strength levels. Table 70 shows the mnemonics used to represent the various strength levels.

Table 70—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 Clause 7 for strength modeling.)

For the logic values 0 and 1, a mnemonic is used when there is no range of strengths in the signal. Otherwise, the logic value is preceded by two decimal digits, which indicate the maximum and minimum strength levels.

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 71 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 71 — Explanation of strength formats

Argument	Description
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

17.1.1.6 Hierarchical name format

The `%m` format specifier does not accept an argument. Instead, it causes the display task to print the hierarchical name of the module, task, function, or named block that invokes the system task containing the format specifier. This is useful when there are many instances of the module that calls the system task. One obvious application is timing check messages in a flip-flop or latch module; the `%m` format specifier shall pinpoint the module instance responsible for generating the timing check message.

17.1.1.7 String format

The `%s` format specifier is used to print ASCII codes as characters. For each `%s` specification that appears in a string, a corresponding 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.

17.1.2 Strobed monitoring

```
strobe_tasks ::=
    strobe_task_name ( list_of_arguments );
strobe_task_name ::=
    $strobe | $strobeb | $strobeo | $strobeh
```

Syntax 17-2 Syntax for \$strobe system tasks

The system task **\$strobe** provides the ability to display simulation data at a selected time. That time is the end of the current simulation time, when all the simulation events 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 17.1.1).

Example:

```
forever @(negedge clock)
    $strobe ("At time %d, data is %h", $time, data);
```

In this example, **\$strobe** writes the time and data information to the standard output and the log file at each negative edge of the clock. The action shall occur just before simulation time is advanced and after all other events at that time have occurred, so that the data written is sure to be the correct data for that simulation time.

17.1.3 Continuous monitoring

```
monitor_tasks ::=
    monitor_task_name [ ( list_of_arguments ) ] ;
    | $monitoron ;
    | $monitoroff ;
monitor_task_name ::=
    $monitor | $monitorb | $monitoro | $monitorh
```

Syntax 17-3 Syntax for \$monitor system tasks

The **\$monitor** task provides the ability to monitor and display the values of any variables or expressions specified as arguments to the task. The arguments for this task are specified in exactly the same manner as for the **\$display** system task including the use of escape sequences for special characters and format specifications (see 17.1.1).

When a **\$monitor** task is invoked with one or more arguments, the simulator sets up a mechanism whereby each time a variable or an expression in the argument list changes value with the exception of the **\$time**, **\$stime** or **\$realtime** system functions the entire argument list is displayed at the end of the time step as if reported by the **\$display** task. If two or more arguments change value at the same time, only one display is produced that shows the new values.

Only one **\$monitor** display list can be active at any one time; however, a new **\$monitor** task with a new display list can be issued any number of times during simulation.

The **\$monitoron** and **\$monitoroff** tasks control a monitor flag that enables and disables the monitoring. Use **\$monitoroff** to turn off the flag and disable monitoring. The **\$monitoron** system task can be used to turn on the flag so that monitoring is enabled and the most recent call to **\$monitor** can resume its display. A call to **\$monitoron** shall produce a display immediately after it is invoked, regardless of whether a value change has taken place; this is used to establish the initial values at the beginning of a monitoring session. By default, the monitor flag is turned on at the beginning of simulation.

17.2 File input-output system tasks and functions

The system tasks and functions for file-based operations are divided into three categories:

- Functions and tasks that open and close files
- Tasks that output values into files
- Tasks that output values into variables
- Tasks and functions that read values from files and load into variables or memories

17.2.1 Opening and closing files

```
file_open_function ::=
    integer multi_channel_descriptor = $fopen ( " file_name " );
    | integer fd = $fopen ( " file_name ", type );
file_close_task ::=
    $fclose ( multi_channel_descriptor );
    | $fclose ( fd );
```

Syntax 17-4 Syntax for \$fopen and \$fclose system tasks

The function **\$fopen** opens the file specified as the **filename** argument and returns either a 32 bit multi channel descriptor, or a 32 bit file descriptor, determined by the absence or presence of the **type** argument.

filename is a character string, or a reg containing a character string that names the file to be opened.

type is a character string, or a reg containing a character string of one of the following forms in the table below, which indicates how the file should be opened. If **type** is omitted, the file is opened for writing, and a multi channel descriptor **mcd** is returned. If **type** is supplied, the file is opened as specified by the value of **type**, and a file descriptor **fd** is returned.

The multi channel descriptor **mcd** is a 32 bit reg in which a single bit is set indicating which file is opened. The least significant bit (bit 0) of a **mcd** always refers to the standard output. Output is directed to two or more files opened with multi channel descriptors by bitwise oring together their **mcds** and writing to the resultant value.

The most significant bit (bit 32) of a multi channel descriptor is reserved, and shall always be cleared, limiting an implementation to at most 31 files opened for output via multi channel descriptors.

The file descriptor **fd** is a 32 bit value. The most significant bit (bit 32) of a **fd** is reserved, and shall always be set; this allows implementations of the file input and output functions to determine how the file was opened. The remaining bits hold a small number indicating what file is opened. Three file descriptors are pre opened; they are **STDIN**, **STDOUT** and **STDERR**, which have the values 32'h8000_0000, 32'h8000_0001 and 32'h8000_0002, respectively. **STDIN** is pre opened for reading, and **STDOUT** and **STDERR** are pre opened for append.

Unlike multi channel descriptors, file descriptors can not be combined via bitwise or in order to direct output to multiple files. Instead, files are opened via file descriptor for input, output, input and output, as well as for append operations, based on the value of **type**, according to the following table:

Table 72—Types for file descriptors

Argument	Description
"r" or "rb"	open for reading
"w" or "wb"	truncate to zero length or create for writing
"a" or "ab"	append; open for writing at end of file, or create for writing
"r+", "r+b", or "rb+"	open for update (reading and writing)
"w+", "w+b", or "wb+"	truncate or create for update
"a+", "a+b", or "ab+"	append; open or create for update at end-of-file

If a file can not be opened (either the file doesn't exist, and the **type** specified is "r", "rb", "r+", "r+b", or "rb+", or the permissions do not allow the file to be opened at that path, a zero is returned for either the **mcd** or the **fd**. Applications can call **\$ferror** to determine the cause of the most recent error (see 17.2.7).

The "b" in the above types exists to distinguish binary files from text files. Many systems (such as Unix) make no distinction between binary and text files, and on these systems the "b" is ignored. However, some systems (such as machines running NT or Windows) perform data mappings on certain binary values written to and read from files that are opened for text access.

The **\$fclose** system task closes the file specified by **fd** or closes the file(s) specified by the multi channel descriptor **mc_d**. No further output to or input from any file descriptor(s) closed by **\$fclose** is allowed. Active **\$fmonitor** and/or **\$fstrobe** operations on a file descriptor or multi channel descriptor are implicitly cancelled by an **\$fclose** operation. The **\$fopen** function shall reuse channels that have been closed.

NOTE The number of simultaneous input and output channels that can be open at any one time is dependent on the operating system. Some operating systems do not support opening files for update.

17.2.2 File output system tasks

```
file_output_tasks ::=
    file_output_task_name ( multi_channel_descriptor , list_of_arguments ) ;
    | file_output_task_name ( fd , list_of_arguments ) ;
file_output_task_name ::=
    $fdisplay | $fdisplayb | $fdisplayh | $fdisplayo
    | $fwrite | $fwriteb | $fwriteh | $fwriteo
    | $fstrobe | $fstrobeb | $fstrobeh | $fstrobeo
    | $fmonitor | $fmonitorb | $fmonitorh | $fmonitro
```

Syntax 17-5 Syntax for file output system tasks

Each of the four formatted display tasks **\$display**, **\$write**, **\$monitor**, and **\$strobe** has a counterpart that writes to specific files as opposed to the standard output. These counterpart tasks **\$fdisplay**, **\$fwrite**, **\$fmonitor**, and **\$fstrobe** accept the same type of arguments as the tasks upon which they are based, with one exception: The first parameter shall be either a multi channel descriptor or a file descriptor, which indicates where to direct the file output. Multi channel descriptors are described in detail in 17.2.1. A multichannel descriptor is either a variable or the result of an expression that takes the form of a 32-bit unsigned integer value.

The **\$fstrobe** and **\$fmonitor** system tasks work just like their counterparts, **\$strobe** and **\$monitor**, except that they write to files using the multi channel descriptor for control. Unlike **\$monitor**, any number of **\$fmonitor** tasks can be set up to be simultaneously active. However, there is no counterpart to **\$monitoron** and **\$monitoroff** tasks. The task **\$fclose** is used to cancel an active **\$fstrobe** or **\$fmonitor** task.

Example:

This example shows how to set up multi channel descriptors. In this example, three different channels are opened using the **\$fopen** function. The three multi channel 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 );

```

17.2.3 Formatting data to a string

```

string_output_tasks ::=
    string_output_tasks_name ( output_reg, list_of_arguments );
string_output_task_name ::=
    $fwrite | $fwriteb | $fwriteh | $fwriteo
variable_format_string_output_task ::=
    $sformat ( output_reg, format, list_of_arguments );

```

Syntax 17-6 Syntax for formatting data tasks

The syntax for the string output system tasks is

```

$fwrite ( output_reg, list_of_arguments );
$sformat ( output_reg, format_string, list_of_arguments );
length = $sformat ( output_reg, format_string, list_of_arguments );

```

The **\$fwrite** family of tasks are based on the **\$fwrite** family of tasks, and accept the same type of arguments as the tasks upon which they are based, with one exception: The first parameter to **\$fwrite** shall be a reg variable to which the resulting string shall be written, instead of a variable specifying the file to which to write the resulting string.

The variable *output_reg* is assigned using the Verilog's string assignment to variable rules, as specified in 4.2.3.

The system task `$sformat` is similar to the system task `$swrite`, with a one major difference.

Unlike the display and write family of output system tasks, `$sformat` always interprets its second argument, and only its second argument as a format string. This format argument can be a static string, such as "data is %d", or can be a reg variable whose content is interpreted as the format string. No other arguments are interpreted as format strings. `$sformat` supports all the format specifiers supported by `$display`, as documented in 17.1.1.2.

The remaining arguments to `$sformat` are processed using any format specifiers in the *format_string*, until all such format specifiers are used up. If not enough arguments are supplied for the format specifiers, or too many are supplied, then the application shall issue a warning, and continue execution. The application, if possible, can statically determine a mismatch in format specifiers and number of arguments, and issue a compile time error message.

NOTE If the *format_string* is a reg, it might not be possible to determine its value at compile time.

The variable *output_reg* is assigned using the Verilog *s* string assignment to variable rules, as specified in 4.2.3.

17.2.4 Reading data from a file

Files opened using file descriptors can be read from only if they were opened with either the *r* or *r+* type values. See 17.2.1 for more information about opening files.

17.2.4.1 Reading a character at a time

```
c = $fgetc ( fd );
```

Read a byte from the file specified by *fd*. If an error occurs reading from the file, then *c* is set to EOF (−1). Define the width of the reg to be wider than 8 bits so that a return value from `$fgetc` of EOF (−1) can be differentiated from the character code 0xFF. Applications can call `$ferror` to determine the cause of the most recent error (see 17.2.7).

```
code = $ungetc ( c, fd );
```

Insert the character specified by *c* into the buffer specified by file descriptor *fd*. The character *c* shall be returned by the next `$fgetc` call on that file descriptor. The file itself is unchanged. Note that the features of the underlying implementation of fileio on the host system limits the number of characters that can be pushed back onto a stream. Note also that operations like `$fseek` might erase any pushed back characters. If an error occurs pushing a character onto a file descriptor, then *code* is set to EOF. Otherwise *code* is set to zero. Applications can call `$ferror` to determine the cause of the most recent error (see 17.2.7).

17.2.4.2 Reading a line at a time

```
integer code = $fgets ( str, fd );
```

Read characters from the file specified by *fd* into the reg *str* until either *str* is filled, or a newline character is read and transferred to *str*, or an end-of-file condition is encountered. If *str* is not an integral number of bytes in length, the most significant partial byte is not used in order to determine the size.

If an error occurs reading from the file, then *code* is set to zero. Otherwise the number of characters read is returned in *code*. Applications can call `$ferror` to determine the cause of the most recent error (see below).

17.2.4.3 Reading formatted data

```
integer code = $fscanf ( fd, format, args );  
integer code = $sscanf ( str, format, args );
```

\$fscanf reads from the files specified by the file descriptor `fd`.

\$sscanf reads from the reg `str`.

Both functions read characters, interpret them according to a format, and store the results. Both expect as arguments a control string, `format`, and a set of arguments specifying where to place the results. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are ignored.

If an argument is too small to hold the converted input, then in general, the least significant bits are transferred. Arguments of any length that is supported by Verilog can be used. However if the destination is a **real** or **realtime** then the value `+Inf` (or `-Inf`) is transferred. The format can be a string constant or a reg containing a string constant. The string contains conversion specifications, which direct the conversion of input into the arguments. The control string can contain

- a) White-space characters (blanks, tabs, new-lines, or form-feeds) that, except in one case described below, cause input to be read up to the next non-white-space character.
- b) An ordinary character (not `%`) that must match the next character of the input stream.
- c) Conversion specifications consisting of the character `%`, an optional assignment suppression character `*`, a decimal digit string that specifies an optional numerical maximum field width, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable specified in the corresponding argument unless assignment suppression was indicated by the character `*`; in this case no argument shall be supplied.

The suppression of assignment provides a way of describing an input field that is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the maximum field width, if one is specified, is exhausted. For all descriptors except the character `c`, white space leading an input field is ignored.

- `%` A single `%` is expected in the input at this point; no assignment is done.
- `b` Matches a binary number, consisting of a sequence from the set `0,1,X,x,Z,z,?` and `_`.
- `o` Matches a octal number, consisting of a sequence of characters from the set `0,1,2,3,4,5,6,7,X,x,Z,z,?` and `_`.
- `d` Matches an optionally signed decimal number, consisting of the optional sign from the set `+` or `-`, followed by a sequence of characters from the set `0,1,2,3,4,5,6,7,8,9` and `_`, or a single value from the set `X,x,Z,z,?`.
- `h` or `x` Matches a hexadecimal number, consisting of a sequence of characters from the set `0,1,2,3,4,5,6,7,8,9,a,A,b,B,c,C,d,D,e,E,f,F,x,X,z,Z,?` and `_`.
- `f`, `e` or `g` Matches a floating point number. The format of a floating point number is an optional sign (either `+` or `-`), followed by a string of digits from the set `0,1,2,3,4,5,6,7,8,9` optionally containing a decimal point character (`.`), then an optional exponent part including `e` or `E` followed by an optional sign, followed by a string of digits from the set `0,1,2,3,4,5,6,7,8,9`.
- `v` Matches a net signal strength, consisting of three character sequence as specified in 17.1.1.5. This conversion is not extremely useful, as strength values are really only usefully assigned to nets and **\$fscanf** can only assign values to regs (if assigned to regs, the values are converted to the 4 value equivalent).

t Matches a floating point number. The format of a floating point number is an optional sign (either + or -), followed by a string of digits from the set 0,1,2,3,4,5,6,7,8,9 optionally containing a decimal point character (.), then an optional exponent part including e or E followed by an optional sign, followed by a string of digits from the set 0,1,2,3,4,5,6,7,8,9. The value matched is then scaled and rounded according to the current time scale as set by **\$timeformat**. For example, if the timescale is 'timescale 1ns/100ps and the time format is **\$timeformat(-3,2," ms",10);**, then a value read with **\$sscanf("10.345", "%t", t)** would return 10350000.0.

c Matches a single character, whose 8 bit ASCII value is returned.

s Matches a string, which is a sequence of non white space characters.

u Matches unformatted (binary) data. The application shall transfer sufficient data from the input to fill the target reg. Typically the data is obtained from a matching **\$fwrite ("%u",data)**, or from an external application written in another programming language such as C, Perl or FORTRAN.

The application shall transfer the 2 value binary data from the input stream to the destination reg, expanding the data to the four value format. This escape sequence can be used with any of the existing input system tasks, although **\$fscanf** should be the preferred one to use. As the input data can not represent x or z, it is not possible to obtain an x or z in the result reg. This formatting specifier is intended to be used to support transferring data to and from external programs that have no concept of x and z.

Applications that require preservation of x and z are encouraged to use the %z I/O format specification.

The data shall be read from the file in the native endian format of the underlying system (i.e., in the same endian order as if the PLI was used, and the C language **read(2)** system call was used).

For POSIX applications: It might be necessary to open files for unformatted I/O with the "rb", "rb+" or "r+b" specifiers, to avoid the systems implementation of I/O altering patterns in the unformatted stream that match special characters.

z The formatting specification %z (or %Z) is defined for reading data without formatting (binary values). The application shall transfer the 4 value binary representation of the specified data from the input stream to the destination reg. This escape sequence can be used with any of the existing input system tasks, although **\$fscanf** should be the preferred one to use.

This formatting specifier is intended to be used to support transferring data to and from external programs that recognize and support the concept of x and z. Applications that do not require the preservation of x and z are encouraged to use the %u I/O format specification.

The data shall be read from the file in the native endian format of the underlying system (i.e., in the same endian order as if the PLI was used, and the data were in a **s_vpi_vecval** structure (See 27.14, Figure 27-8), and the C language **read(2)** system call was used to read the data from disk).

For POSIX applications: It might be necessary to open files for unformatted I/O with the "rb", "rb+" or "r+b" specifiers, to avoid the systems implementation of I/O altering patterns in the unformatted stream that match special characters.

m Returns the current hierarchical path as a string. Does not read data from the input file or str argument. If an invalid conversion character follows the %, the results of the operation are implementation dependent.

If the format string, or the **str** argument to **\$sscanf** contains unknown bits (x or z) then the system task shall return EOF.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable.

The number of successfully matched and assigned input items is returned in `code`; this number can be 0 in the event of an early matching failure between an input character and the control string. If the input ends before the first matching failure or conversion, EOF is returned. Applications can call `$ferror` to determine the cause of the most recent error (see below).

17.2.4.4 Reading binary data

```
integer code = $fread( myreg, fd);
integer code = $fread( mem, fd);
integer code = $fread( mem, fd, start);
integer code = $fread( mem, fd, start, count);
integer code = $fread( mem, fd, , count);
```

Read a binary data from the file specified by `fd` into the reg `myreg` or the memory `mem`.

`start` is an optional argument. If present, `start` shall be used as the address of the first element in the memory to be loaded. If not present the lowest numbered location in the memory shall be used.

`count` is an optional argument. If present, `count` shall be the maximum number of locations in `mem` that shall be loaded. If not supplied the memory shall be filled with what data is available.

`start` and `count` are ignored if `$fread` is loading a reg.

If no addressing information is specified within the system task, and no address specifications appear within the data file, then the default start address is the lowest address given in the declaration of the memory. Consecutive words are loaded towards the highest address until either the memory is full or the data file is completely read. If the start address is specified in the task without the finish address, then loading starts at the specified start address and continues towards the highest address given in the declaration of the memory.

`start` is the address in the memory. For `start = 12` and the memory `up[10:20]`, the first data would be loaded at `up[12]`. For the memory `down[20:10]`, the first location loaded would be `down[12]`, then `down[13]`.

The data in the file shall be read byte by byte to fulfill the request. An 8-bit wide memory is loaded using one byte per memory word, while a 9-bit wide memory is loaded using 2 bytes per memory word. The data is read from the file in a big endian manner; the first byte read is used to fill the most significant location in the memory element. If the memory width is not evenly divisible by 8 (8, 16, 24, 32), not all data in the file is loaded into memory because of truncation.

The data loaded from the file is taken as "two value" data. A bit set in the data is interpreted as a 1, and bit not set is interpreted as a 0. It is not possible to read a value of `x` or `z` using `$fread`.

If an error occurs reading from the file, then `code` is set to zero. Otherwise the number of characters read is returned in `code`. Applications can call `$ferror` to determine the cause of the most recent error (see 17.2.7).

Note that there is not a "binary" mode and a "ASCII" mode; one can freely intermingle binary and formatted read commands from the same file.

17.2.5 File positioning

```
integer pos = $ftell ( fd );
```

Returns in **pos** the offset from the beginning of the file of the current byte of the file **fd** which shall be read or written by a subsequent operation on that file descriptor.

This value can be used by subsequent **\$fseek** calls to reposition the file to this point. Note that any repositioning shall cancel any **\$ungetc** operations. If an error occurs EOF is returned. Applications can call **\$ferror** to determine the cause of the most recent error (see 17.2.7).

```
code = $fseek ( fd, offset, operation );  
code = $rewind ( fd );
```

Sets the position of the next input or output operation on the file specified by **fd**. The new position is at the signed distance offset bytes from the beginning, from the current position, or from the end of the file, according to an operation value of 0, 1 and 2 as follows:

- 0 set position equal to offset bytes
- 1 set position to current location plus offset
- 2 set position to EOF plus offset

\$rewind is equivalent to **\$fseek (fd, 0, 0);**

Repositioning the current file position with **\$fseek** or **\$rewind** shall cancel any **\$ungetc** operations.

\$fseek() allows the file position indicator to be set beyond the end of the existing data in the file. If data is later written at this point, subsequent reads of data in the gap shall return zero until data is actually written into the gap. **\$fseek**, by itself, does not extend the size of the file.

When a file is opened for append (that is, when **type** is "a", or "a+"), it is impossible to overwrite information already in the file. **\$fseek** can be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output.

If an error occurs repositioning the file, then **code** is set to -1. Otherwise **code** is set to 0. Applications can call **\$ferror** to determine the cause of the most recent error (see 17.2.7).

17.2.6 Flushing output

```
$fflush ( mcd );  
$fflush ( fd );  
$fflush ( );
```

Writes any buffered output to the file(s) specified by **mcd**, the file specified by **fd** or if **\$fflush** is invoked with no arguments, writes any buffered output to all open files.

17.2.7 I/O error status

Should any error be detected by one of the fileio routines, an error code is returned. Often this is sufficient for normal operation; (i.e., if the opening of a optional configuration file fails, the application typically would simply continue using default values.) However sometimes it is useful to obtain more information about the error for correct application operation. In this case the **\$ferror** function can be used:

```
integer errno = $ferror ( fd, str );
```

A string description of type of error encountered by the most recent file I/O operation is written into `str` which should be at least 640 bits wide. The integral value of the error code is returned in `errno`. If the most recent operation did not result in an error, then the value returned shall be zero, and the reg `str` shall be cleared.

17.2.8 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 17-7 Syntax for memory load system tasks

Two system tasks **\$readmemb** and **\$readmemh** read and load data from a specified text file into a specified memory. Either task can be executed at any time during simulation. The text file to be read shall contain only the following:

- White space (spaces, 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 `@` character (`@`) followed by a hexadecimal number as follows:

```
@hh...h
```

Both uppercase and lowercase digits are allowed in the number. No white space is allowed between the `@` and the number. As many address specifications as needed within the data file can be used. When the system task encounters an address specification, it loads subsequent data starting at that memory address.

If no addressing information is specified within the system task, and no address specifications appear within the data file, then the default start address shall be the left-hand address given in the declaration of the memory. Consecutive words shall be 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 shall start at the specified start address and shall continue 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 shall begin at the start address and shall continue 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 shall be issued if the number of data words in the file differs from the number of words in the range implied by the start through finish addresses.

Example:

```
reg [7:0] mem[1:256];
```

Given this declaration, each of the following statements load data into mem in a different manner:

```
initial $readmemh("mem.data", mem);
initial $readmemh("mem.data", mem, 16);
initial $readmemh("mem.data", mem, 128, 1);
```

The first statement loads up the memory at simulation time 0 starting at the memory address 1. The second statement begins loading at address 16 and continue on towards address 256. For the third and final statement, loading begins 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.

17.2.9 Loading timing data from an SDF file

The syntax for the **\$sdf_annotate** system task is shown in Syntax 17-8.

```
sdf_annotate_task ::=
    $sdf_annotate("sdf_file" [, [ module_instance ] [, [ "config_file" ]
        [, [ "log_file" ] [, [ "mtm_spec" ]
        [, [ "scale_factors" ] [, [ "scale_type" ] ] ] ] ] ] ] );
```

Syntax 17-8 Syntax for \$sdf_annotate system task

The **\$sdf_annotate** system task reads timing data from an SDF file into a specified region of the design.

sdf_file is a character string, or a reg containing a character string naming the file to be opened.

module_instance is an optional argument specifying the scope to which to annotate the information in the SDF file. The SDF annotator uses the hierarchy level of the specified instance for running the annotation. Array indices are permitted. If the *module_instance* not specified, the SDF Annotator uses the module containing the call to the **\$sdf_annotate** system task as the *module_instance* for annotation.

config_file is an optional character string argument providing the name of a configuration file. Information in this file can be used to provide detailed control over many aspects of annotation.

log_file is an optional character string argument providing the name of the log file used during SDF annotation. Each individual annotation of timing data from the SDF file results in an entry in the log file.

mtm_spec is an optional character string argument specifying which member of the min/typ/max triples shall be annotated. The legal values for this string are described in Table 73. This overrides any MTM_SPEC keywords in the configuration file.

Table 73—mtm spec argument

Keyword	Description
MAXIMUM	Annotate the maximum value
MINIMUM	Annotate the minimum value
TOOL_CONTROL (default)	Annotate the value as selected by the simulator
TYPICAL	Annotate the typical value

scale_factors is an optional character string argument specifying the scale factors to be used while annotating timing values. For example, "1.6:1.4:1.2" causes minimum values to be multiplied by 1.6, typical values by 1.4, and maximum values by 1.2. The default values are 1.0:1.0:1.0. The *scale_factors* argument overrides any SCALE_FACTORS keywords in the configuration file.

scale_type is an optional character string argument specifying how the scale factors should be applied to the min/typ/max triples. The legal values for this string are shown in Table 74. This overrides any SCALE_TYPE keywords in the configuration file.

Table 74—scale type argument

Keyword	Description
FROM_MAXIMUM	Apply scale factors to maximum value
FROM_MINIMUM	Apply scale factors to minimum value
FROM_MTM (default)	Apply scale factors to min/typ/max values
FROM_TYPICAL	Apply scale factors to typical value

17.3 Timescale system tasks

The following system tasks display and set timescale information:

- a) **\$sprinttimescale**
- b) **\$timeformat**

17.3.1 \$sprinttimescale

The **\$sprinttimescale** system task displays the time unit and precision for a particular module. The syntax for the system task is shown in Syntax 17-9.

```
printtimescale_task ::=  
    $printtimescale [ ( hierarchical_identifier ) ] ;
```

Syntax 17-9 Syntax for \$printtimescale

This system task can be specified with or without an argument.

- When no argument is specified, **\$printtimescale** displays the time unit and precision of the module that is the current scope.
- When an argument is specified, **\$printtimescale** displays the time unit and precision of the module passed to it.

The timescale information shall appear in the following format:

```
Time scale of (module_name) is unit / precision
```

Example:

```
'timescale 1 ms / 1 us  
module a_dat;  
initial  
    $printtimescale(b_dat.c1);  
endmodule  
  
'timescale 10 fs / 1 fs  
module b_dat;  
    c_dat c1 ();  
endmodule  
  
'timescale 1 ns / 1 ns  
module c_dat;  
    .  
    .  
    .  
endmodule
```

In this example, module **a_dat** invokes the **\$printtimescale** system task to display timescale information about another module **c_dat**, which is instantiated in module **b_dat**.

The information about **c_dat** shall be displayed in the following format:

```
Time scale of (b_dat.c1) is 1ns / 1ns
```

17.3.2 \$timeformat

The syntax for **\$timeformat** system task is shown in Syntax 17-10.

```
timeformat_task ::=
    $timeformat [ ( units_number , precision_number , suffix_string , minimum_field_width ) ] ;
```

Syntax 17-10 Syntax for \$timeformat

The **\$timeformat** system task performs the following two functions:

- It specifies how the %t format specification reports time information for the **\$write**, **\$display**, **\$strobe**, **\$monitor**, **\$fwrite**, **\$fdisplay**, **\$fstrobe**, and **\$fmonitor** group of system tasks.
- It specifies the time unit for delays entered interactively.

The units number argument shall be an integer in the range from 0 to -15. This argument represents the time unit as shown in Table 75.

Table 75—\$timeformat units_number arguments

Unit number	Time unit	Unit number	Time unit
0	1 s	-8	10 ns
-1	100 ms	-9	1 ns
-2	10 ms	-10	100 ps
-3	1 ms	-11	10 ps
-4	100 us	-12	1 ps
-5	10 us	-13	100 fs
-6	1 us	-14	10 fs
-7	100 ns	-15	1 fs

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 76.

Table 76—\$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.

17.4 Simulation control system tasks

There are two simulation control system tasks:

- a) **\$finish**
- b) **\$stop**

17.4.1 \$finish

Syntax 17-11 shows the syntax for **\$finish** system task.

```
finish_task ::=
    $finish [ ( n ) ] ;
```

Syntax 17-11 Syntax for \$finish

The **\$finish** system task simply makes the simulator exit and pass control back to the host operating system. If an expression is supplied to this task, then its value (0, 1, or 2) determines the diagnostic messages that are printed before the prompt is issued. If no argument is supplied, then a value of 1 is taken as the default.

Table 77—Diagnostics for \$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

17.4.2 \$stop

The syntax for **\$stop** system task is shown in Syntax 17-12.

```
stop_task ::=
    $stop [ ( n ) ] ;
```

Syntax 17-12 Syntax for \$stop

The **\$stop** system task causes simulation to be suspended. This task takes an optional expression argument (0, 1, or 2) that determines what type of diagnostic message is printed. The amount of diagnostic messages output increases with the value of the optional argument passed to **\$stop**.

17.5 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. The syntax for PLA modeling system task is shown in Syntax 17-13.

```
pla_system_task ::=
    $array_type$logic$format ( memory_type , input_terms , output_terms );
array_type ::=
    sync | async
logic ::=
    and | or | nand | nor
format ::=
    array | plane
input_terms ::=
    expression
output_terms ::=
    variable_lvalue
```

Syntax 17-13 Syntax for PLA modeling system task

NOTE The input terms can be nets or variables whereas the output terms shall only be variables.

The PLA syntax allows for the system tasks as shown in Table 78.

Table 78—PLA modeling system tasks

\$async\$and\$array	\$sync\$and\$array	\$async\$and\$plane	\$sync\$and\$plane
\$async\$nand\$array	\$sync\$nand\$array	\$async\$nand\$plane	\$sync\$nand\$plane
\$async\$or\$array	\$sync\$or\$array	\$async\$or\$plane	\$sync\$or\$plane
\$async\$nor\$array	\$sync\$nor\$array	\$async\$nor\$plane	\$sync\$nor\$plane

17.5.1 Array types

The modeling of both synchronous and asynchronous arrays is provided by the PLA system tasks. The synchronous forms control the time at which the logic array shall be evaluated and the outputs shall be updated. For the asynchronous forms, the evaluations are automatically performed whenever an input term changes value or any word in the personality memory is changed.

For both the synchronous and asynchronous forms, the output terms are updated without any delay.

Examples:

An example of an asynchronous system call is as follows:

```
wire      a1, a2, a3, a4, a5, a6, a7;
reg       b1, b2, b3;
wire [1:7] awire;
reg  [1:3] breg;

$async$and$array(mem, {a1, a2, a3, a4, a5, a6, a7}, {b1, b2, b3});
or
$async$and$array(mem, awire, breg);
```

An example of a synchronous system call is as follows:

```
$sync$or$plane(mem, {a1, a2, a3, a4, a5, a6, a7}, {b1, b2, b3});
```

17.5.2 Array logic types

The logic arrays are modeled with and, or, nand, and nor logic planes. This applies to all array types and formats.

Examples:

An example of a nor plane system call is as follows:

```
$async$nor$plane(mem, {a1, a2, a3, a4, a5, a6, a7}, {b1, b2, b3});
```

An example of a nand plane system call is as follows:

```
$sync$nand$plane(mem, {a1, a2, a3, a4, a5, a6, a7}, {b1, b2, b3});
```

17.5.3 Logic array personality declaration and loading

The logic array personality is declared as an array of regs that is as wide as the number of input terms and as deep as the number of output terms.

The personality of the logic array is normally loaded into the memory from a text data file using the system tasks **\$readmemb** or **\$readmemh**. Alternatively, the personality data can be written directly into the memory using the procedural assignment statements. PLA personalities can be changed dynamically at any time during simulation simply by changing the contents of the memory. The new personality shall be reflected on the outputs of the logic array at the next evaluation.

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.

17.5.4 Logic array personality formats

Two separate personality formats are supported by the Verilog HDL and are differentiated by using either an array system call or a plane system call. The array system call allows for a 1 or 0 in the memory that has been declared. A 1 means take the input value and a 0 means do not take the input value.

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
    $asyncland$array(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
end
endmodule
```

Where the file `array.dat` contains the binary data for the PLA personality:

```
1100000
0011100
0000111
```

A synchronous version of this example has the following description:

```

module sync_array(a1,a2,a3,a4,a5,a6,a7,b1,b2,b3,clk);
input a1, a2, a3, a4, a5, a6, a7, clk;
output b1, b2, b3;
reg [1:7] mem[1:3]; // memory declaration
reg b1, b2, b3;
initial begin
    // setup the personality
    $readmemb("array.dat", mem);
    // setup a synchronous logic array to be evaluated
    // when a positive edge on the clock occurs
    forever @(posedge clk)
        $asyncland$array(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
end
endmodule

```

Example 2 An example of the usage of the plane format tasks follows. The logical function of this PLA is shown first, followed by the PLA personality in the new format, the Verilog HDL description using the **\$asyncland\$plane** system task, and finally the result of running the simulation.

The logical function of the PLA is as follows:

```

b[1] = a[1] & ~a[2];
b[2] = a[3];
b[3] = ~a[1] & ~a[3];
b[4] = 1;

```

The PLA personality is as follows:

```

3'b10?
3'b??1
3'b0?0
3'b???

```

```

module pla;
`define rows 4
`define cols 3
reg [1:`cols] a, mem[1:`rows];
reg [1:`rows] b;
initial begin
    // PLA system call
    $async$and$plane(mem,a[1:3],b[1:4]);
    mem[1] = 3'b10?;
    mem[2] = 3'b??1;
    mem[3] = 3'b0?0;
    mem[4] = 3'b???;
    // stimulus and display
    #10 a = 3'b111;
    #10 $displayb(a, " -> ", b);
    #10 a = 3'b000;
    #10 $displayb(a, " -> ", b);
    #10 a = 3'bxxx;
    #10 $displayb(a, " -> ", b);
    #10 a = 3'b101;
    #10 $displayb(a, " -> ", b);
end
endmodule

```

The output is as follows:

```

111 -> 0101
000 -> 0011
xxx -> xxx1
101 -> 1101

```

17.6 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) ;

```

17.6.1 \$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 79.

Table 79—Types of queues of \$q_type values

q_type 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 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 79.

17.6.2 \$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 79.

17.6.3 \$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 79.

17.6.4 \$q_full

The **\$q_full** system function checks whether there is room for another entry on a queue. It returns 0 when the queue is not full and 1 when the queue is full.

17.6.5 \$q_exam

The **\$q_exam** system task provides statistical information about activity at the queue **q_id**. It returns a value in **q_stat_value** depending on the information requested in **q_stat_code**. The values of **q_stat_code** and the corresponding information returned in **q_stat_value** are described in Table 80.

Table 80—Parameter values for \$q_exam system task

Value requested in q_stat_code	Information received back from q_stat_value
1	Current queue length
2	Mean interarrival time
3	Maximum queue length
4	Shortest wait time ever
5	Longest wait time for jobs still in the queue
6	Average wait time in the queue

17.6.6 Status codes

All of the queue management tasks and functions return an output status parameter. The status parameter values and corresponding information are described in Table 81.

Table 81 — Status parameter values

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

17.7 Simulation time system functions

The following system functions provide access to current simulation time:

\$time **\$stime** **\$realtime**

17.7.1 \$time

The syntax for **\$time** system function is shown in Syntax 17-14.

time_function ::= \$time

Syntax 17-14 Syntax for \$time

The **\$time** system function returns an integer that is a 64-bit time, scaled to the timescale unit of the module that invoked it.

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:

- 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.
- 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.

17.7.2 \$stime

The syntax for `$stime` system function is shown in Syntax 17-15.

```

stime_function ::=
    $stime

```

Syntax 17-15 Syntax for `$stime`

The `$stime` system function returns an unsigned integer that is a 32-bit time, scaled to the timescale unit of the module that invoked it. If the actual simulation time does not fit in 32 bits, the low order 32 bits of the current simulation time are returned.

17.7.3 \$realtime

The syntax for `$realtime` system function is shown in Syntax 17-16.

```

realtime_function ::= $realtime

```

Syntax 17-16 Syntax for `$realtime`

The **\$realtime** system function returns a real number time that, like **\$time**, is scaled to the time unit of the module that invoked it.

Example:

```
'timescale 10 ns / 1 ns
module test;
reg set;
parameter p = 1.55;
initial begin
    $monitor($realtime, , "set=", set);
    #p set = 0;
    #p set = 1;
end
endmodule

// The output from this example is as follows:
// 0 set=x
// 1.6 set=0
// 3.2 set=1
```

In this example, the event times in the reg **set** are multiples of 10 ns because 10 ns is the time unit of the module. They are real numbers because **\$realtime** returns a real number.

17.8 Conversion functions

The following functions handle **real** values:

integer	\$rtoi (real_val) ;
real	\$itor (int_val) ;
[63:0]	\$realtobits (real_val) ;
real	\$bitstoreal (bit_val) ;

\$rtoi converts real values to integers by truncating the real value (for example, 123.45 becomes 123)

\$itor converts integers to real values (for example, 123 becomes 123.0)

\$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

```

See 4.5 for a description of **\$signed** and **\$unsigned**.

17.9 Probabilistic distribution functions

There are a set of random number generators that return integer values distributed according to standard probabilistic functions.

17.9.1 \$random function

The syntax for the system function **\$random** is shown in Syntax 17-17.

```

random_function ::=
    $random [ ( seed ) ] ;

```

Syntax 17-17 Syntax for \$random

The system function **\$random** provides a mechanism for generating random numbers. The function returns a new 32-bit random number each time it is called. The random number is a signed integer; it can be positive or negative. For further information on probabilistic random number generators, see 17.9.2.

The **seed** parameter controls the numbers that **\$random** returns such that different seeds generate different random streams. The **seed** parameter shall be either a reg, 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;
```

17.9.2 \$dist_ functions

```
dist_functions ::=  
    $dist_uniform ( seed , start , end ) ;  
    $dist_normal ( seed , mean , standard_deviation ) ;  
    $dist_exponential ( seed , mean ) ;  
    $dist_poisson ( seed , mean ) ;  
    $dist_chi_square ( seed , degree_of_freedom ) ;  
    $dist_t ( seed , degree_of_freedom ) ;  
    $dist_erlang ( seed , k_stage , mean ) ;
```

Syntax 17-18 Syntax for 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 shall 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 ensures 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 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 spread the returned values over a wider range.

17.9.3 Algorithm for probabilistic distribution functions

Table 82 shows the Verilog probabilistic distribution functions listed with their corresponding C functions.

Table 82— Verilog to C function cross-listing

Verilog function	C function
\$dist_uniform	rtl_dist_uniform
\$dist_normal	rtl_dist_normal
\$dist_exponential	rtl_dist_exponential
\$dist_poisson	rtl_dist_poisson
\$dist_chi_square	rtl_dist_chi_square
\$dist_t	rtl_dist_t
\$dist_erlang	rtl_dist_erlang
\$random	rtl_dist_uniform(seed, LONG_MIN, LONG_MAX)

The algorithm for these functions is defined by the following C code.

```

/*
 * Algorithm for probabilistic distribution functions.
 *
 * IEEE Std 1364-2000 Verilog Hardware Description Language (HDL).
 */

#include <limits.h>

static double uniform( long *seed, long start, long end );
static double normal( long *seed, long mean, long deviation);
static double exponential( long *seed, long mean);
static long poisson( long *seed, long mean);
static double chi_square( long *seed, long deg_of_free);
static double t( long *seed, long deg_of_free);
static double erlangian( long *seed, long k, long mean);

long
rtl_dist_chi_square( seed, df )
    long *seed;
    long df;
{
    double r;
    long i;

    if(df>0)
    {
        r=chi_square(seed,df);
        if(r>=0)
        {
            i=(long)(r+0.5);

```

```

    }
    else
    {
        r = -r;
        i=(long)(r+0.5);
        i = -i;
    }
}
else
{
    print_error("WARNING: Chi_square distribution must have
positive
                                degree of freedom\n");
    i=0;
}
return (i);
}

long
rtl_dist_erlang( seed, k, mean )
    long *seed;
    long k, mean;
{
    double r;
    long i;

    if(k>0)
    {
        r=erlangian(seed,k,mean);
        if(r>=0)
        {
            i=(long)(r+0.5);
        }
        else
        {
            r = -r;
            i=(long)(r+0.5);
            i = -i;
        }
    }
    else
    {
        print_error("WARNING: k-stage erlangian distribution must have
positive k\n");
        i=0;
    }

    return (i);
}

long
rtl_dist_exponential( seed, mean )
    long *seed;

```

```

        long mean;
    {
        double r;
        long i;

        if(mean>0)
        {
            r=exponential(seed,mean);
            if(r>=0)

                {
                    i=(long)(r+0.5);
                }
            else
                {
                    r = -r;
                    i=(long)(r+0.5);
                    i = -i;
                }
        }
        else
        {
            print_error("WARNING: Exponential distribution must have a
                        positive mean\n");
            i=0;
        }

        return (i);
    }

long
rtl_dist_normal( seed, mean, sd )
    long *seed;
    long mean, sd;
{
    double r;
    long i;
    r=normal(seed,mean,sd);
    if(r>=0)
    {
        i=(long)(r+0.5);
    }
    else
    {
        r = -r;
        i=(long)(r+0.5);
        i = -i;
    }

    return (i);
}

long

```

```
rtl_dist_poisson( seed, mean )
    long *seed;
    long mean;
{
    long i;

    if(mean>0)
    {
        i=poisson(seed,mean);
    }
    else
    {
        print_error("WARNING: Poisson distribution must have a positive
                    mean\n");
        i=0;
    }
    return (i);
}

long
rtl_dist_t( seed, df )
    long *seed;
    long df;
{
    double r;
    long i;

    if(df>0)
    {
        r=t(seed,df);
        if(r>=0)
        {
            i=(long)(r+0.5);
        }
        else
        {
            r = -r;
            i=(long)(r+0.5);
            i = -i;
        }
    }
    else
    {
        print_error("WARNING: t distribution must have positive degree
                    of freedom\n");
        i=0;
    }
    return (i);
}

long
rtl_dist_uniform(seed, start, end)
    long *seed;
    long start, end;
{
    double r;
```

```

long i;

if (start >= end) return(start);

if (end != LONG_MAX)
{
    end++;
    r = uniform( seed, start, end );
    if (r >= 0)

        {
            i = (long) r;
        }
    else
    {
        i = (long) (r-1);
    }
    if (i<start) i = start;
    if (i>=end) i = end-1;
}
else if (start!=LONG_MIN)
{
    start--;
    r = uniform( seed, start, end) + 1.0;
    if (r>=0)
    {
        i = (long) r;
    }
    else
    {
        i = (long) (r-1);
    }
    if (i<=start) i = start+1;
    if (i>end) i = end;
}
else
{
    r=(uniform(seed,start,end)+2147483648.0)/
4294967295.0);
    r = r*4294967296.0-2147483648.0;
    if (r>=0)
    {
        i = (long) r;
    }
    else
    {
        i = (long) (r-1);
    }
}

return (i);
}

static double
uniform( seed, start, end )

```



```

        long *seed, start, end;
    {
        union u_s
        {
            float s;
            unsigned stemp;
        } u;

        double d = 0.00000011920928955078125;
        double a,b,c;

        if ((*seed) == 0)
            *seed = 259341593;

        if (start >= end)
        {
            a = 0.0;
            b = 2147483647.0;
        }
        else
        {
            a = (double) start;
            b = (double) end;
        }
        *seed = 69069 * (*seed) + 1;
        u.stemp = *seed;

        /*
         * This relies on IEEE floating point format
         */
        u.stemp = (u.stemp >> 9) | 0x3f800000;

        c = (double) u.s;
        c = c+(c*d);
        c = ((b - a) * (c - 1.0)) + a;

        return (c);
    }

```

```

static double
normal(seed,mean,deviation)
long *seed,mean,deviation;
{
    double v1,v2,s;
    double log(), sqrt();

    s = 1.0;
    while((s >= 1.0) || (s == 0.0))
    {
        v1 = uniform(seed,-1,1);
        v2 = uniform(seed,-1,1);
        s = v1 * v1 + v2 * v2;
    }
}

```

```

    s = v1 * sqrt(-2.0 * log(s) / s);
    v1 = (double) deviation;
    v2 = (double) mean;
    return(s * v1 + v2);
}

```

```

static double
exponential(seed,mean)
long *seed,mean;
{
    double log(),n;
    n = uniform(seed,0,1);
    if(n != 0)
    {
        n = -log(n) * mean;
    }
    return(n);
}

```

```

static long
poisson(seed,mean)
long *seed,mean;
{
    long n;
    double p,q;
    double exp();

    n = 0;
    q = -(double)mean;
    p = exp(q);
    q = uniform(seed,0,1);
    while(p < q)
    {
        n++;
        q = uniform(seed,0,1) * q;
    }
    return(n);
}

```

```

static double
chi_square(seed,deg_of_free)
long *seed,deg_of_free;
{
    double x;
    long k;
    if(deg_of_free % 2)
    {
        x = normal(seed,0,1);
        x = x * x;
    }
    else
    {
        x = 0.0;
    }
    double log(),n;

```

```
n = uniform(seed,0,1);
if(n != 0)
{
    n = -log(n) * mean;
}
return(n);
}

static double
t(seed,deg_of_free)
long *seed,deg_of_free;

{
    double sqrt(),x;
    double chi2 = chi_square(seed,deg_of_free);
    double div = chi2 / (double)deg_of_free;
    double root = sqrt(div);
    x = normal(seed,0,1) / root;
    return(x);
}

static double
erlangian(seed,k,mean)
long *seed,k,mean;
{
    double x,log(),a,b;
    long i;

    x=1.0;
    for(i=1;i<=k;i++)

    {
        x = x * uniform(seed,0,1);
    }
    a=(double)mean;
    b=(double)k;
    x= -a*log(x)/b;
    return(x);
}
```

17.10 Command line input

An alternative to reading a file to obtain information for use in the simulation is specifying information with the command to invoke the simulator. This information is in the form of a optional argument provided to the simulation. These arguments are visually distinguished from other simulator arguments by the starting with the plus (+) character.

These arguments, referred to below as *plusargs*, are accessible through the system functions described in 17.10.1 and 17.10.2.

17.10.1 \$test\$plusargs (string)

This system function searches the list of plusargs for the provided string. The plusargs present on the command line are searched in the order provided. If the prefix of one of the supplied plusargs matches all characters in the provided string, a non-zero integer is returned. If no plusarg from the command line matches the string provided, the integer value zero (0) is returned.

Examples:

Run simulator with command: +HELLO

The Verilog code is:

```
initial begin
    if ($test$plusargs("HELLO")) $display("Hello argument found.")
    if ($test$plusargs("HE")) $display("The HE subset string is
detected.");
    if ($test$plusargs("H")) $display("Argument starting with H
found.");
    if ($test$plusargs("HELLO_HERE")) $display("Long argument.");
    if ($test$plusargs("HI")) $display("Simple greeting.");
    if ($test$plusargs("LO")) $display("Does not match.");
end
```

This would produce the following output:

```
Hello argument found.
The HE subset string is detected.
Argument starting with H found.
```

17.10.2 \$value\$plusargs (user_string, variable)

This system function searches the list of plusargs (like the **\$test\$plusargs** system function) for a user specified plusarg string. The string is specified in the first argument to the system function as either a string or a register which is interpreted as a string. If the string is found, the remainder of the string is converted to the type specified in the *user_string* and the resulting value stored in the variable provided. If a string is found, the function returns a non-zero integer. If no string is found matching, the function returns the integer value zero and the variable provided is not modified. No warnings shall be generated when the function returns zero (0).

The *user_string* shall be of the form: "*plusarg_string format_string*". The format strings are the same as the **\$display** system tasks. These are the only valid ones (upper and lower case as well as a leading 0 forms are valid):

%d	decimal conversion
%o	octal conversion
%h	hexadecimal conversion
%b	binary conversion
%e	real exponential conversion
%f	real decimal conversion
%g	real decimal or exponential conversion
%s	string (no conversion)

The first string, from the list of *plusargs* provided to the simulator, which matches the *plusarg_string* portion of the *user_string* specified shall be the *plusarg* string available for conversion. The remainder string of the matching *plusarg* (the remainder is the part of the *plusarg* string after the portion which matches the users

plusarg_string) shall be converted from a string into the format indicated by the format string and stored in the variable provided. If there is no remaining string, the value stored into the variable shall either be a zero (0) or an empty string value.

If the size of the variable is larger than the value after conversion, the value stored is zero (0) padded to the width of the variable. If the variable can not contain the value after conversion, the value shall be truncated. If the value is negative, the value shall be considered larger than the variable provided. If characters exist in the string available for conversion, which are illegal for the specified conversion, the variable shall be written with the value 'bx'.

Examples:

```
+FINISH=10000 +TESTNAME=this_test +FREQ+5.6666 +FREQUENCY +TEST12

// Get clock to terminate simulation if specified.
real frequency;
reg [8*32:1] testname;
integer stop_clock;
if ($value$plusargs("FINISH=%d", stop_clock))
begin
repeat (stop_clock) @(posedge clk);
$finish;
end
// Get testname from plusarg.
if ($value$plusargs("TESTNAME=%s", testname))
begin
$display("Running test %0s.", testname);
startTest();
end

// Get frequency from command line; set default if not specified.
if (!$value$plusargs("FREQ+%0F", frequency))
frequency = 8.33333; // 166MHz;

forever
begin
#frequency clk = 0;
#frequency clk = 1;
end

reg [64*8:1] pstring;
pstring = "+TEST%d";
if ($value$plusargs(pstring, test[31:0]))
begin
$display("Running test number %0d.", test);
startTest();
end
```

This code would have the following effects:

- The variable *test* would get the value 'd12.
- The variable *stop_clock* obtains the value 10000.
- The variable *testname* obtains the value *this_test*.
- The variable *frequency* obtains the value 5.6666; note the final *plusarg* +FREQUENCY does not affect the value of the variable *frequency*.

The output is:

```
Running test this_test.  
Running test number 12.
```

Withdrawing
IECNORM.COM: Click to view the full PDF of IEC 61691-4:2004

18. Value change dump (VCD) files

A *value change dump (VCD) file* contains information about value changes on selected variables in the design stored by value change dump system tasks. Two types of VCD files exist:

- Four state: to represent variable changes in 0, 1, x, and Z with no strength information.
- Extended: to represent variable changes in all states and strength information.

This clause describes how to generate both types of VCD files and their format.

18.1 Creating the four state value change dump file

The steps involved in creating the four state VCD file are listed below and illustrated in Figure 51.

- Insert the VCD system tasks in the Verilog source file to define the dump file name and to specify the variables to be dumped.
- Run the simulation.

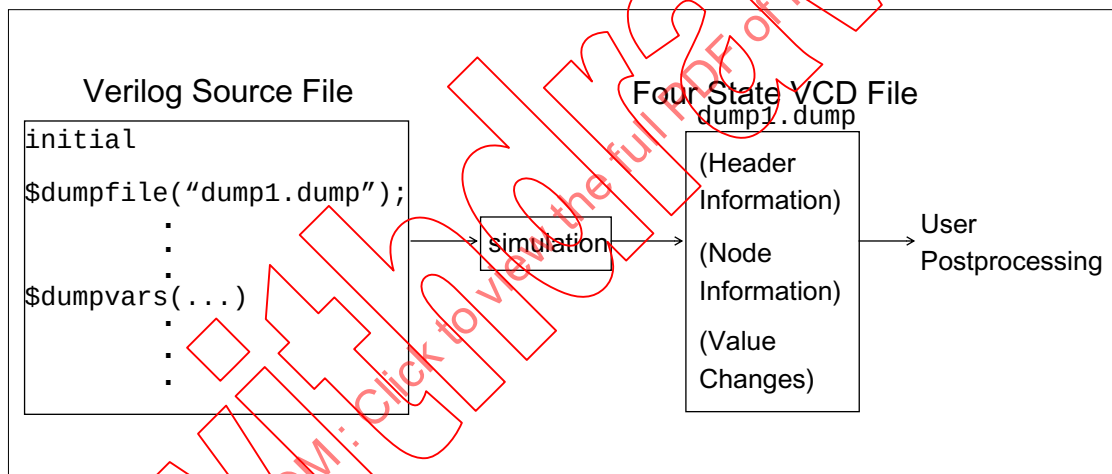


Figure 51 — Creating the four state VCD file

A VCD file is an ASCII file which contains header information, variable definitions, and the value changes for all variables specified in the task calls.

Several system tasks can be inserted in the source description to create and control the VCD file.

18.1.1 Specifying 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 given in Syntax 18-1.

```

dumpfile_task ::=
    $dumpfile ( filename ) ;

```

Syntax 18-1 Syntax for \$dumpfile task

The *filename* syntax is given in Syntax 18-2.

```

filename ::=
    literal_string
    | variable
    | expression

```

Syntax 18-2 Syntax for filename

The *filename* is optional and defaults to the literal string `dump.vcd` if not specified.

Example:

```

initial $dumpfile ("module1.dump") ;

```

18.1.2 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 can be invoked as often as desired throughout the model (for example, within various blocks), but the execution of all the **\$dumpvars** tasks shall be at the same simulation time.

The **\$dumpvars** task can be used with or without arguments. The syntax for the **\$dumpvars** task is given in Syntax 18-3.

```

dumpvars_task ::= (Not in the Annex A BNF)
    $dumpvars ;
    | $dumpvars ( levels [ , list_of_modules_or_variables ] ) ;
list_of_modules_or_variables ::= (Not in the Annex A BNF)
    module_or_variable { , module_or_variable }
module_or_variable ::=
    module_identifier
    | variable_identifier

```

Syntax 18-3 Syntax for \$dumpvars task

When invoked with no arguments, **\$dumpvars** dumps all the variables in the model to the VCD file.

When the **\$dumpvars** task is specified with arguments, the first argument indicates how many *levels* of the hierarchy below each specified module instance to dump to the VCD file. Subsequent arguments specify which scopes of the model to dump to the VCD file. These arguments can specify entire modules or individual variables within a module.

Setting the first argument to 0 causes a dump of all variables in the specified module and in all module instances below the specified module. The argument 0 applies only to subsequent arguments which 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 shall dump all variables in the module **top** and in all module instances below module **top** in the hierarchy.

Example 3 This example shows how the **\$dumpvars** task can specify both modules and individual variables:

```
$dumpvars (0, top.mod1, top.mod2.net1);
```

This call shall dump all variables in module **mod1** and in all module instances below **mod1**, along with variable **net1** in module **mod2**. The argument 0 applies only to the module instance **top.mod1** and not to the individual variable **top.mod2.net1**.

18.1.3 Stopping and resuming the dump (\$dumpoff/\$dumpon)

Executing the **\$dumpvars** task causes the value change dumping to start at the end of the current simulation time unit. To suspend the dump, the **\$dumpoff** task can be invoked. To resume the dump, the **\$dumpon** task can be invoked. The syntax of these two tasks is given in Syntax 18-4.

```
dumpoff_task ::=  
    $dumpoff ;  
dumpon_task ::=  
    $dumpon ;
```

Syntax 18-4 Syntax for \$dumpoff and \$dumpon tasks

When the **\$dumpoff** task is executed, a checkpoint is made in which every selected variable is dumped as an X value. When the **\$dumpon** task is later executed, each variable is dumped with its value at that time. In the interval between **\$dumpoff** and **\$dumpon**, no value changes are dumped.

The **\$dumpoff** and **\$dumpon** tasks provide the mechanism to control the simulation period during which the dump shall take place.

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).

18.1.4 Generating a checkpoint (**\$dumpall**)

The **\$dumpall** task creates a checkpoint in the VCD file which shows the current value of all selected variables. The syntax is given in Syntax 18-5.

```

dumpall_task ::=
  $dumpall ;

```

Syntax 18-5 Syntax for \$dumpall task

When dumping is enabled, the value change dumper records the values of the variables which change during each time increment. Values of variables which do not change during a time increment are not dumped.

18.1.5 Limiting the size of the dump file (**\$dumplimit**)

The **\$dumplimit** task can be used to set the size of the VCD file. The syntax for this task is given in Syntax 18-6.

```

dumplimit_task ::=
  $dumplimit ( filesize ) ;

```

Syntax 18-6 Syntax for \$dumplimit task

The *filesize* argument which 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 the dump limit was reached.

18.1.6 Reading the dump file during simulation (\$dumpflush)

The **\$dumpflush** task can be used to empty the VCD file buffer of the operating system to ensure all the data in that buffer is stored in the VCD file. After executing a **\$dumpflush** task, dumping is resumed as before so no value changes are lost. The syntax for the task is given in Syntax 18-7.

```
dumpflush_task ::=  
    $dumpflush ;
```

Syntax 18-7 Syntax for \$dumpflush task

A common application is to call **\$dumpflush** to update the dump file so an application program can read the VCD file during a simulation.

Examples:

Example 1 This example shows how the **\$dumpflush** task can 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

```

18.2 Format of the four state VCD file

The dump file is structured in a free format. White space is used to separate commands and to make the file easily readable by a text editor.

18.2.1 Syntax of the four state VCD file

The syntax of the four state VCD file is given in Syntax 18-8.

```

value_change_dump_definitions ::=
    { declaration_command } { simulation_command }
declaration_command ::=
    declaration_keyword
    [ command_text ]
    $end
simulation_command ::=
    simulation_keyword { value_change } $end
    | $comment [ comment_text ] $end
    | simulation_time
    | value_change
declaration_keyword ::=
    $comment | $date | $enddefinitions | $scope | $timescale | $supscope
    | $var | $version
simulation_keyword ::=
    $dumpall | $dumpoff | $dumpon | $dumpvars
simulation_time ::=
    # decimal_number
value_change ::=
    scalar_value_change
    | vector_value_change
scalar_value_change ::=
    value identifier_code
value ::=
    0 | 1 | x | X | z | Z
vector_value_change ::=
    b binary_number identifier_code
    | B binary_number identifier_code
    | r real_number identifier_code
    | R real_number identifier_code
identifier_code ::=
    { ASCII character }

```

Syntax 18-8 Syntax of the output four state VCD file

The VCD file starts with header information giving the date, the version number of the simulator used for the simulation, and the timescale used. Next, the file contains definitions of the scope and type of variables being dumped, followed by the actual value changes at each simulation time increment. Only the variables which 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 which 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 which 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.

18.2.2 Formats of variable values

Variables can be either scalars or vectors. Each type is dumped in its own format. Dumps of value changes to scalar variables shall not have any white space between the value and the identifier code.

Dumps of value changes to vectors shall not have any white space between the base letter and the value digits, but they shall have one white space between the value digits and the identifier code.

The output format for each value is right-justified. Vector values appear in the shortest form possible: redundant bit values which result from left-extending values to fill a particular vector size are eliminated.

The rules for left-extending vector values are given in Table 83.

Table 83— Rules for left-extending vector values

When the value is	VCD left-extends with
1	0
0	0
Z	Z
X	X

Table 84 shows how the VCD can shorten values.

Table 84— How the VCD can shorten values

The binary value	Extends to fill a 4-bit reg as	Appears in the VCD file as
10	0010	b10
X10	XX10	bX10
ZX0	ZZX0	bZX0
0X10	0X10	b0X10

Events are dumped in the same format as scalars; for example, $1*%$. For events, however, the value (1 in this example) is irrelevant. Only the identifier code ($*%$ in this example) is significant. It appears in the VCD file as a marker to indicate the event was triggered during the time step.

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

18.2.3 Description of keyword commands

The general information in the VCD file is presented as a series of sections surrounded by keywords. Keyword commands provide a means of inserting information in the VCD file. Keyword commands can be inserted either by the dumper or manually.

This sub clause deals with the keyword commands given in Table 85.

Table 85—Keyword commands

Declaration keywords		Simulation keywords
\$comment	\$timescale	\$dumpall
\$date	\$upscope	\$dumpoff
\$enddefinitions	\$var	\$dumpon
\$scope	\$version	\$dumpvars

18.2.3.1 \$comment

The **\$comment** section provides a means of inserting a comment in the VCD file. The syntax for the section is given in Syntax 18-9.

```
vcd_declaration_comment ::=  
$comment comment_text $end
```

Syntax 18-9 Syntax for \$comment section

Examples:

```
$comment This is a single-line comment      $end  
$comment This is a  
multiple-line comment  
$end
```

18.2.3.2 \$date

The **\$date** section indicates the date on which the VCD file was generated. The syntax for the section is given in Syntax 18-10.

```
vcd_declaration_date ::=
    $date date_text $end
```

Syntax 18-10 Syntax for \$date section

Example:

```
$date
    June 25, 1989 09:24:35
$end
```

18.2.3.3 \$enddefinitions

The **\$enddefinitions** section marks the end of the header information and definitions. The syntax for the section is given in Syntax 18-11.

```
vcd_declaration_enddefinitions ::=
    $enddefinitions $end
```

Syntax 18-11 Syntax for \$enddefinitions section

18.2.3.4 \$scope

The **\$scope** section defines the scope of the variables being dumped. The syntax for the section is given in Syntax 18-12.

```
vcd_declaration_scope ::=
    $scope scope_type scope_identifier $end
scope_type ::=
    begin
    fork
    function
    module
    task
```

Syntax 18-12 Syntax for \$scope section

The scope type indicates one of the following scopes:

<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
```

18.2.3.5 \$timescale

The **\$timescale** keyword specifies what timescale was used for the simulation. The syntax for the keyword is given in Syntax 18-13.

```
vcd_declaration_timescale ::=
    $timescale time_number time_unit $end
time_number ::=
    1 | 10 | 100
time_unit ::=
    s | ms | us | ns | ps | fs
```

Syntax 18-13 Syntax for \$timescale

Example:

```
$timescale 10 ns $end
```

18.2.3.6 \$upscope

The **\$upscope** section indicates a change of scope to the next higher level in the design hierarchy. The syntax for the section is given in Syntax 18-14.

```
vcd_declaration_upscope ::=
    $upscope $end
```

Syntax 18-14 Syntax for \$upscope section

18.2.3.7 \$version

The **\$version** section indicates which version of the VCD writer was used to produce the VCD file and the **\$dumpfile** system task used to create the file. If a variable or an expression was used to specify the *filename* within **\$dumpfile**, the unevaluated variable or expression literal shall appear in the **\$version** string. The syntax for the **\$version** section is given in Syntax 18-15.

```
vcd_declaration_version ::=
    $version version_text system_task $end
```

Syntax 18-15 Syntax for \$version section

Example:

```
$version
    VERILOG-SIMULATOR 1.0a
$dumpfile("dump1.dump")
$end
```

18.2.3.8 \$var

The **\$var** section prints the names and identifier codes of the variables being dumped. The syntax for the section is given in Syntax 18-16.

```
vcd_declaration_vars ::=
    $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
```

Syntax 18-16 Syntax for \$var section

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.

- The msb index indicates the most significant index; the lsb index indicates the least significant index.
- More than one reference name can be mapped to the same identifier code. For example, net10 and net15 can be interconnected in the circuit and therefore have the same identifier code.
- The individual bits of vector nets can be dumped individually.
- The identifier is the name of the variable being dumped in the model.

Example:

```
$var
    integer 32 (2 index
$end
```

18.2.3.9 \$dumpall

The **\$dumpall** keyword specifies current values of all variables dumped. The syntax for the keyword is given in Syntax 18-17.

```
vcd_simulation_dumpall ::=
    $dumpall { value_changes } $end
```

Syntax 18-17 Syntax for \$dumpall keyword

Example:

```
$dumpall 1*@ x*# 0*$ bx (k $end
```

18.2.3.10 \$dumpoff

The **\$dumpoff** keyword indicates all variables dumped with X values. The syntax for the keyword is given in Syntax 18-18.

```
vcd_simulation_dumpoff ::=
    $dumpoff { value_changes } $end
```

Syntax 18-18 Syntax for \$dumpoff keyword

Example:

```
$dumpoff x*@ x*# x*$ bx (k $end
```

18.2.3.11 \$dumpon

The **\$dumpon** keyword indicates resumption of dumping and lists current values of all variables dumped. The syntax for the keyword is given in Syntax 18-19.

```
vcd_simulation_dumpon ::=
    $dumpon { value_changes } $end
```

Syntax 18-19 Syntax for \$dumpon keyword

Example:

```
$dumpon x*@ 0*# x*$ b1 (k $end
```

18.2.3.12 \$dumpvars

The section beginning with **\$dumpvars** keyword lists initial values of all variables dumped. The syntax for the keyword is given in Syntax 18-20.

```
vcd_simulation_dumpvars ::=  
  $dumpvars { value_changes } $end
```

Syntax 18-20 Syntax for \$dumpvars keyword

Example:

```
$dumpvars    x*@    z*$    b0    (k    $end
```

18.2.4 Four state VCD file format example

The following example illustrates the format of the four state VCD file.

```
$date June 26, 1989 10:05:41
$end
$version VERILOG-SIMULATOR 1.0a
$end
$timescale 1 ns
$end
$scope module top $end
$scope module m1 $end
$var trireg 1 *@ net1 $end
$var trireg 1 *# net2 $end
$var trireg 1 *$ net3 $end
$upscope $end
$scope task t1 $end
$var reg 32 (k accumulator[31:0]) $end
$var integer 32 {2 index $end
$upscope $end
$upscope $end
$enddefinitions $end
$comment
```

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 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*\$

(Continued in right column)

18.3 Creating the extended value change dump file

The steps involved in creating the extended VCD file are listed below and illustrated in Figure 52.

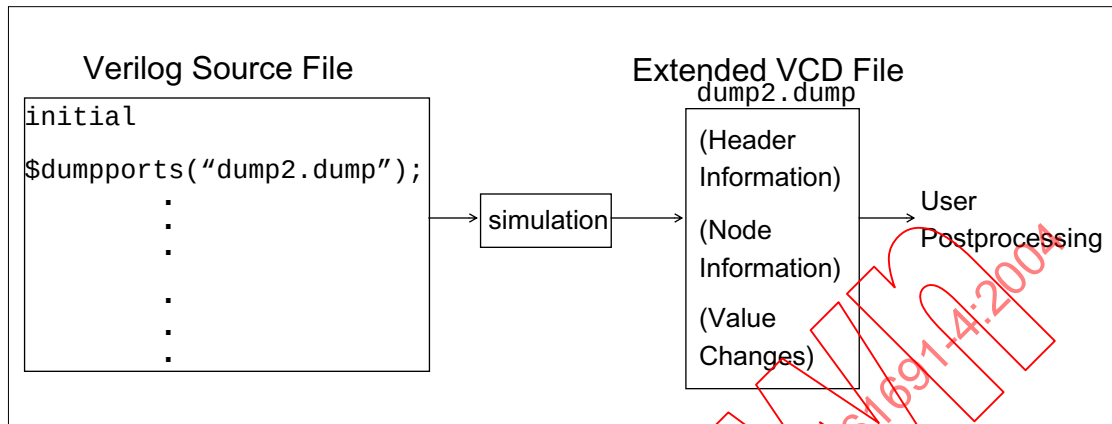


Figure 52—Creating the extended VCD file

- Insert the extended VCD system tasks in the Verilog source file to define the dump file name and to specify the variables to be dumped.
- Run the simulation.

The four state VCD file rules and syntax apply to the extended VCD file unless otherwise stated in this section.

18.3.1 Specifying the dumpfile name and the ports to be dumped (\$dumpports)

The **\$dumpports** task shall be used to specify the name of the VCD file and the ports to be dumped. The syntax for the task is given in Syntax 18-21.

```

dumpports_task ::=
  $dumpports ( scope_list , file_pathname ) ;
scope_list ::=
  module_identifier { , module_identifier }
file_pathname ::=
  literal_string
  | variable
  | expression
  
```

Syntax 18-21 Syntax for \$dumpports task

Where the arguments are optional and are defined as:

scope_list one or more module identifiers. Only modules are allowed (not variables). If more than one *module_identifier* is specified, they shall be separated by a comma. Pathnames to modules are allowed, using the period hierarchy separator. Literal strings are not allowed for the *module_identifier*.

If no *scope_list* value is provided, the scope shall be the module from which **\$dumpports** is called.

file_pathname can be a double quoted pathname (literal string), a reg type variable, or an expression which denotes the file which shall contain the port VCD information. If no *file_pathname* is provided, the file shall be written to the current working directory with the name *dumpports.vcd*. If that file already exists, it shall be silently overwritten. All file writing checks shall be made by the simulator (write rights, correct pathname, etc.) and appropriate errors or warnings issued.

The following rules apply to the use of the **\$dumpports** system task:

- All the ports in the model from the point of the **\$dumpports** call are considered primary I/O pins and shall be included in the VCD file. However, any ports which exist in instantiations below *scope_list* are not dumped.
- If no arguments are specified for the task, **\$dumpports**; and **\$dumpports()** are allowed. In both of these cases, the default values for the arguments shall be used.
- If the first argument is null, a comma shall be used before specifying the second argument in the argument list.
- Each scope specified in the *scope_list* shall be unique. If multiple calls to **\$dumpports** are specified, the *scope_list* values in these calls shall also be unique.
- The **\$dumpports** task can be used in source code which also contains the **\$dumpvars** task.
- When **\$dumpports** executes, the associated value change dumping shall start at the end of the current simulation time unit.
- The **\$dumpports** task can be invoked multiple times throughout the model, but the execution of all **\$dumpports** tasks shall be at the same simulation time. Specifying the same *file_pathname* multiple times is not allowed.

18.3.2 Stopping and resuming the dump (\$dumpportsoff/\$dumpportson)

The **\$dumpportsoff** and **\$dumpportson** system tasks provide a means to control the simulation period for dumping port values. The syntax for these system tasks is given in Syntax 18-22.

```

dumpportsoff_task ::=
    $dumpportsoff ( file_pathname ) ;
dumpportson_task ::=
    $dumpportson ( file_pathname ) ;
file_pathname ::=
    literal_string
    | variable
    | expression
    
```

Syntax 18-22 Syntax for **\$dumpportsoff** and **\$dumpportson** system tasks

The *file_pathname* argument can be a double quoted pathname (literal string), a reg type variable, or an expression which denotes the *file_pathname* specified in the **\$dumpports** system task.

\$dumpportsoff. When this task is executed, a checkpoint is made in the *file_pathname* where each specified port is dumped with an X value. Port values are no longer dumped from that simulation time forward. If *file_pathname* is not specified, all dumping to files opened by **\$dumpports** calls shall be suspended.

\$dumpportson. When this task is executed, all ports specified by the associated **\$dumpports** call shall have their values dumped. This system task is typically used to resume dumping after the execution of **\$dumpportsoff**. If *file_pathname* is not specified, dumping shall resume for all files specified by **\$dumpports** calls, if dumping to those files was stopped.

If **\$dumpportson** is executed while ports are already being dumped to *file_pathname*, the system task is ignored. If **\$dumpportsoff** is executed while port dumping is already suspended for *file_pathname*, the system task is ignored.

18.3.3 Generating a checkpoint (\$dumpportsall)

The **\$dumpportsall** system task creates a checkpoint in the VCD file which shows the value of all selected ports at that time in the simulation, regardless of whether the port values have changed since the last timestep. The syntax for this system task is given in Syntax 18-23.

```
dumpportsall_task ::=
    $dumpportsall ( file_pathname ) ;
file_pathname ::=
    literal_string
    | variable
    | expression
```

Syntax 18-23 Syntax for \$dumpportsall system task

The *file_pathname* argument can be a double quoted pathname (literal string), a reg type variable, or an expression which denotes the *file_pathname* specified in the **\$dumpports** system task.

If the *file_pathname* is not specified, checkpointing occurs for all files opened by calls to **\$dumpports**.

18.3.4 Limiting the size of the dump file (\$dumpportslimit)

The **\$dumpportslimit** system task allows control of the VCD file size. The syntax for this system task is given in Syntax 18-24.

```
dumpportslimit_task ::=
    $dumpportslimit ( filesize , file_pathname ) ;
file_size ::=
    integer
file_pathname ::=
    literal_string
    | variable
    | expression
```

Syntax 18-24 Syntax for \$dumpportslimit system task

The *filesize* argument is required and it specifies the maximum size in bytes for the associated *file_pathname*. When this *filesize* is reached, the dumping stops and a comment is inserted into *file_pathname* indicating the size limit was attained.

The *file_pathname* argument can be a double quoted pathname (literal string), a reg type variable, or an expression which denotes the *file_pathname* specified in the **\$dumpports** system task.

If the *file_pathname* is not specified, the *filesize* limit applies to all files opened for dumping due to calls to **\$dumpports**.

18.3.5 Reading the dump file during simulation (\$dumpportsflush)

To facilitate performance, simulators often buffer VCD output and write to the file at intervals, instead of line by line. The **\$dumpportsflush** system task writes all port values to the associated file, clearing a simulator's VCD buffer.

The syntax for this system task is given in Syntax 18-25.

```
dumpportsflush_task ::=  
    $dumpportsflush ( file_pathname ) ;  
file_pathname ::=  
    literal_string  
    | variable  
    | expression
```

Syntax 18-25 Syntax for \$dumpportsflush system task

The *file_pathname* argument can be a double quoted pathname (literal string), a reg type variable, or an expression which denotes the *file_pathname* specified in the **\$dumpports** system task.

If the *file_pathname* is not specified, the VCD buffers shall be flushed for all files opened by calls to **\$dumpports**.

18.3.6 Description of keyword commands

The general information in the extended VCD file is presented as a series of sections surrounded by keywords. Keyword commands provide a means of inserting information in the extended VCD file. Keyword commands can be inserted either by the dumper or manually. Extended VCD provides one additional keyword command to that of the four state VCD.

18.3.6.1 \$vcdclose

The **\$vcdclose** keyword indicates the final simulation time at the time the extended VCD file is closed. This allows accurate recording of the end simulation time, regardless of the state of signal changes, in order to assist parsers which require this information. The syntax for the keyword is given in Syntax 18-26.

```
vcdclose_task ::=
    $vcdclose final_simulation_time Send
```

Syntax 18-26 Syntax for \$vcdclose keyword

Example:

```
$vcdclose #13000 Send
```

18.3.7 General rules for extended VCD system tasks

For each extended VCD system task, the following rules apply:

- If a *file_pathname* is specified which does not match a *file_pathname* specified in a **\$dumpports** call, the control task shall be ignored.
- If no arguments are specified for the tasks which have only optional arguments, the system task name can be used with no arguments or the name followed by () can be specified. For example: **\$dumpportsflush**; or **\$dumpportsflush()**. In both of these cases, the default actions for the arguments shall be executed.

18.4 Format of the extended VCD file

The format of the extended VCD file is similar to that of the four state VCD file, as it is also structured in a free format. White space is used to separate commands and to make the file easily readable by a text editor.

18.4.1 Syntax of the extended VCD file

The syntax of the extended VCD file is given in Syntax 18-27. A four state VCD construct name which matches an extended VCD construct shall be considered equivalent, except if preceded by an *.

```

value_change_dump_definitions ::= {declaration_command} {simulation_command}
declaration_command ::= declaration_keyword [command_text] $end
simulation_command ::= (Not in the Annex A BNF)
                        simulation_keyword { value_change } $end
                        | $comment [comment_text] $end
                        | simulation_time
                        | value_change
* declaration_keyword ::=
    $comment | $date | $enddefinitions | $scope | $timescale | $supscope | $var
    | $vcdclose | $version
command_text ::=
    comment_text | close_text | date_section | scope_section | timescale_section
    | var_section | version_section
* simulation_keyword ::= $dumpports | $dumpportsoff | $dumpportson |
    $dumpportsall
simulation_time ::= #decimal_number
value_change ::= value identifier_code
value ::= pport_value 0_strength_component 1_strength_component
port_value ::= input_value | output_value | unknown_direction_value
input_value ::= D | U | N | Z | d | u
output_value ::= L | H | X | T | I | h
unknown_direction_value ::= 0 | 1 | ? | F | A | a | B | b | C | c | f
strength_component ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
* identifier_code ::= <{integer}
comment_text ::= {ASCII_character}
close_text ::= final_simulation_time
date_section ::= date_text
date_text ::= day month date time year
scope_section ::= scope_type scope_identifier
* scope_type ::= module
timescale_section ::= number time_unit
number ::= 1 | 10 | 100
time_unit ::= fs | ps | ns | us | ms | s
var_section ::= var_type size identifier_code reference
* var_type ::= port
* size ::= 1 | vector_index
vector_index ::= [ msb_index : lsb_index ]
index ::= decimal_number
* reference ::= port_identifier
identifier ::= {printable_ASCII_character}
version_section ::= version_text
* version_text ::= version_identifier {dumpports_command}
dumpports_command ::=
    $dumpports (scope_identifier , string_literal
    | variable
    | expression )

```

Syntax 18-27 Syntax of the output extended VCD file

The extended VCD file starts with header information giving the date, the version number of the simulator used for the simulation, and the timescale used. Next, the file contains definitions of the scope of the ports being dumped, followed by the actual value changes at each simulation time increment. Only the ports which change value during a time increment are listed.

The simulation time recorded in the extended VCD file is the absolute value of the simulation time for the changes in port values which follow.

Value changes for all ports are specified in binary format by 0, 1, x, or Z values and include strength information.

A real number is dumped using a `%.16g printf()` format. This preserves the precision of that number by outputting all 53 bits in the mantissa of a 64-bit *IEEE Std 754-1985* [B1] double-precision number. Application programs can read a real number using a `%g` format to `scanf()`.

NOTES:

1) The extended VCD format does not support a mechanism to dump *part* of a vector. For example, bits 8 to 15 (8:15) of a 16-bit vector cannot be dumped in VCD file; instead, the entire vector (0:15) has to be dumped. In addition, expressions, such as `a + b`, cannot be dumped in the VCD file.

2) Data in the extended VCD file is case sensitive.

18.4.2 Extended VCD node information

The node information section (also referred to as the variable definitions section) is affected by the **\$dump-ports** task as Syntax 18-28 shows.

```
$var var_type size < identifier_code reference $end
var_type ::=
    port
size ::=
    1
    | vector_index
vector_index ::=
    [msb_index : lsb_index]
index ::=
    decimal_number
identifier_code ::=
    integer
reference ::=
    port_identifier
```

Syntax 18-28 Syntax of extended VCD node information

The constructs are defined as:

var_type	the keyword port . No other keyword is allowed.
size	a decimal number indicating the number of bits in the port. If the port is a single bit, the value shall be 1. If the port is a bus, the actual index is printed. The <i>msb</i> indicates the most significant index; <i>lsb</i> the least significant index.
identifier_code	an integer preceded by < which starts at zero and ascends in one unit increments for each port, in the order found in the module declaration.
reference	identifier indicating the port name.

Example:

```

module test_device(count_out, carry, data, reset)
output count_out, carry ;
input [0:3] data;
input reset;
. . .
initial
    begin
        $dumpports(testbench.DUT, "testoutput.vcd");
    . . .
end

```

This example produces the following node information in the VCD file:

```

$scope module testbench.DUT $end
$var port 1 <0 count_out $end
$var port 1 <1 carry $end
$var port [0:3] <2 data $end
$var port 1 <3 reset $end
$upscope $end

```

At least one space shall separate each syntactical element. However, the formatting of the information is the choice of the simulator vendor. All four state VCD syntax rules for the *vector_index* apply.

If the *vector_index* appears in the port declaration, this shall be the index dumped. If the *vector_index* is not in the port declaration, the *vector_index* in the net or reg declaration matching the port name shall be dumped. If no *vector_index* is found, the port is considered scalar (1 bit wide).

Concatenated ports shall appear in the extended VCD file as separate entries.

Example:

```

module addbit ({A, b}, ci, sum, co);
    input A, b, ci;
    output sum, co;
. . .

```

The VCD file output looks like:

```
$scope module addbit $end
$var port 1 <0 A $end
$var port 1 <1 b $end
$var port 1 <2 ci $end
$enddefinitions $end
. . .
```

18.4.3 Value changes

The value change section of the VCD file is also affected by **\$dumpports**, as Syntax 18-29 shows.

p port_value	0_strength_component	1_strength_component	identifier_code

Syntax 18-29 Syntax of value change section

Where the constructs are defined as:

p	key character which indicates a port. There is no space between the p and the <i>port_value</i> .
<i>port_value</i>	state character (described below).
<i>0_strength_component</i>	one of the 8 Verilog strengths which indicates the strength0 specification for the port.
<i>1_strength_component</i>	one of the 8 Verilog strengths which indicates the strength1 specification for the port.

The Verilog strength values are (append keyword with 0 or 1 as appropriate for the strength component):

0	highz
1	small
2	medium
3	weak
4	large
5	pull
6	strong
7	supply

identifier_code the integer preceded by the < character as defined in the **\$var** construct for the port.

18.4.3.1 State characters

The following state information is listed in terms of input values from a test fixture, the output values of the device under test (DUT), and the states representing unknown direction:

INPUT (TESTFIXTURE)

D	low
U	high
N	unknown
Z	three-state
d	low (two or more drivers active)
u	high (two or more drivers active)

OUTPUT (DUT)

L	low
H	high
X	unknown (don't care)
T	three-state
l	low (two or more drivers active)
h	high (two or more drivers active)

UNKNOWN DIRECTION

0	low (both input and output are active with 0 value)
1	high (both input and output are active with 1 value)
?	unknown
F	three-state (input and output unconnected)
A	unknown (input 0 and output 1)
a	unknown (input 0 and output X)
B	unknown (input 1 and output 0)
b	unknown (input 1 and output X)
C	unknown (input X and output 0)
c	unknown (input X and output 1)
f	unknown (input and output three-stated)

18.4.3.2 Drivers

Where drivers are considered only in terms of primitives, continuous assignments, and procedural continuous assignments. Value 0/1 means both input and output are active with value 0/1. 0 and 1 are conflict states. The following rules apply to conflicts:

- If both input and output are driving the same value with the same range of strength, then this is a conflict. The resolved value is 0/1 and the strength is the stronger of the two.
- If the input is driving a strong strength (range) and the output is driving a weak strength (range), the resolved value is 0/1 and the strength is the strength of the input.
- If the input is driving a weak strength (range) and the output is driving a strong strength (range), then the resolved value is 1/h and the strength is the strength of the output.

Where range is:

- Strength supply 7 to 5 (large) - strong strength
- Strength 4 to 1 - weak strength

18.4.4 Extended VCD file format example

The following example illustrates the format of the extended VCD file.

A module declaration:

```

module adder(data0, data1, data2, data3, carry, as, rdn, reset,
test,
        write);
    inout data0, data1, data2, data3;
    output carry;
    input as, rdn, reset, test, write;
    . . .

```

And the resulting VCD fragment:

```
$scope module testbench.adder_instance $end
$var port 1 <0 data0 $end
$var port 1 <1 data1 $end
$var port 1 <2 data2 $end
$var port 1 <3 data3 $end
$var port 1 <4 carry $end
$var port 1 <5 as $end
$var port 1 <6 rdn $end
$var port 1 <7 reset $end
$var port 1 <8 test $end
$var port 1 <9 write $end
$supscope $end
$enddefinitions $end
```

```
#0
$dumpports
pX 6 6 <0
pX 6 6 <1
pX 6 6 <2
pX 6 6 <3
pX 6 6 <4
pN 6 6 <5
pN 6 6 <6
pU 0 6 <7
pD 6 0 <8
pN 6 6 <9
$end
#180
pH 0 6 <4
#200000
pD 6 0 <5
pU 0 6 <6
pD 6 0 <9
#200500
pf 0 0 <0
pf 0 0 <1
pf 0 0 <2
pf 0 0 <3
```


19. 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 clause describes the following compiler directives:

‘celldefine	[19.1]
‘default_nettype	[19.2]
‘define	[19.3]
‘else	[19.4]
‘elsif	[19.4]
‘endcelldefine	[19.1]
‘endif	[19.4]
‘ifdef	[19.4]
‘ifndef	[19.4]
‘include	[19.5]
‘line	[19.7]
‘nounconnected_drive	[19.9]
‘resetall	[19.6]
‘timescale	[19.8]
‘unconnected_drive	[19.9]
‘undef	[19.3]

19.1 ‘celldefine and ‘endcelldefine

The directives ‘**celldefine** and ‘**endcelldefine** tag modules as cell modules. Cells are used by certain PLI routines for applications, such as delay calculations. It is advisable to pair each ‘**celldefine** with an ‘**endcelldefine**. More than one of these pairs may appear in a single source description.

These directives may appear anywhere in the source description, but it is recommended that the directives be specified outside the module definition.

The ‘**resetall** directive includes the effects of a ‘**endcelldefine** directive.

19.2 ‘default_nettype

The directive ‘**default_nettype** controls the net type created for implicit net declarations (see 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 19-1 contains the syntax of the directive.

```
default_nettype_compiler_directive ::=  
    ‘default_nettype net_type  
net_type ::= wire | tri | tri0 | wand | triand | wor | trior | trireg | none
```

Syntax 19-1 Syntax for default nettype compiler directive

When no **'default_nettype** directive is present or if the **'resetall** directive is specified, implicit nets are of type **wire**. When the **'default_nettype** is set to **none**, all nets must be explicitly declared. If a net is not explicitly declared, an error is generated.

19.3 'define and 'undef

A text macro substitution facility has been provided so that meaningful names can be used to represent commonly used pieces of text. For example, in the situation where a constant number is repetitively used throughout a description, a text macro would be useful in that only one place in the source description would need to be altered if the value of the constant needed to be changed.

The text macro facility is not affected by the compiler directive **'resetall**.

19.3.1 'define

The directive **'define** creates a macro for text substitution. This directive can be used both inside and outside module definitions. After a text macro is defined, it can be used in the source description by using the **(`)** character, followed by the macro name. The compiler shall substitute the text of the macro for the string **`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 19-2.

```
text_macro_definition ::=
    'define text_macro_name macro_text
text_macro_name ::=
    text_macro_identifier [ ( list_of_formal_arguments ) ]
list_of_formal_arguments ::=
    formal_argument_identifier { , formal_argument_identifier }
text_macro_identifier ::= (From Annex A - A.9.3)
    simple_identifier
```

Syntax 19-2 Syntax for text macro definition

The macro text can be any arbitrary text specified on the same line as the text macro name. If more than one line is necessary to specify the text, the newline shall be preceded by a backslash (****). The first newline not preceded by a backslash shall end the macro text. The newline preceded by a backslash shall be replaced in the expanded macro with a newline (but without the preceding backslash character).

When formal arguments are used to define a text macro, the scope of the formal argument shall extend up to the end of the macro text. A formal argument can be used in the macro text in the same manner as an identifier.

If a one-line comment (that is, a comment specified with the characters **//**) is included in the text, then the comment shall not become part of the substituted text. The macro text can be blank, in which case the text macro is defined to be empty, and no text is substituted when the macro is used.

The syntax for using a text macro is given in Syntax 19-3.

```

text_macro_usage ::=
    'text_macro_identifier [ ( list_of_actual_arguments ) ]
list_of_actual_arguments ::=
    actual_argument { , actual_argument }
actual_argument ::=
    expression

```

Syntax 19-3 Syntax for text macro usage

For a macro without arguments, the text shall be substituted as is for every occurrence of ``text_macro_name`. However, a text macro with one or more arguments shall be expanded by substituting each formal argument with the expression used as the actual argument in the macro usage.

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 macro name shall be a simple identifier.

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.

3) The macro text can contain usages of other text macros. Such usages shall be substituted after the original macro is substituted, not when it is defined. It shall be an error for a macro to expand directly or indirectly to text containing another usage of itself (a recursive macro).

19.3.2 'undef

The directive **'undef** shall undefine a previously defined text macro. An attempt to undefine a text macro that was not previously defined using a **'define** compiler directive can result in a warning. The syntax for **'undef** compiler directive is given in Syntax 19-4.

```
undefine_compiler_directive ::=  
  'undef text_macro_identifier
```

Syntax 19-4 Syntax for undef compiler directive

An undefined text macro has no value, just as if it had never been defined.

19.4 'ifdef, 'else, 'elsif, 'endif, 'ifndef

These conditional compilation compiler directives are used to include optionally lines of a Verilog HDL source description during compilation. The **'ifdef** compiler directive checks for the definition of a `text_macro_name`. If the `text_macro_name` is defined, then the lines following the **'ifdef** directive are included. If the `text_macro_name` is not defined and an **'else** directive exists, then this source is compiled. The **'ifndef** compiler directive checks for the definition of a `text_macro_name`. If the `text_macro_name` is not defined, then the lines following the **'ifndef** directive are included. If the `text_macro_name` is defined and an **'else** directive exists, then this source is compiled.

If the **'elsif** directive exists (instead of the **'else**) the compiler checks for the definition of the `text_macro_name`. If the name exists the lines following the **'elsif** directive are included. The **'elsif** directive is equivalent to the compiler directive sequence **'else 'ifdef ... 'endif**. This directive does not need a corresponding **'endif** directive. This directive must be preceded by an **'ifdef** or **'ifndef** directive.

These directives may appear anywhere in the source description.

Situations where the **'ifdef**, **'else**, **'elsif**, **'endif**, and **'ifndef** compiler directives may be useful include:

- Selecting different representations of a module such as behavioral, structural, or switch level
- Choosing different timing or structural information
- Selecting different stimulus for a given run

The **'ifdef**, **'else**, **'elsif**, **'endif**, and **'ifndef** compiler directives have the syntax shown in Syntax 19-5.

```
conditional_compilation_directive ::=
    ifdef_directive
  | ifndef_directive
ifdef_directive ::=
    'ifdef text_macro_identifier
    ifdef_group_of_lines
    { 'elsif text_macro_identifier elsif_group_of_lines }
    [ 'else else_group_of_lines ]
    'endif
ifndef_directive ::=
    'ifndef text_macro_identifier
    ifndef_group_of_lines
    { 'elsif text_macro_identifier elsif_group_of_lines }
    [ 'else else_group_of_lines ]
    'endif
```

Syntax 19-5 Syntax for conditional compilation directives

The `text_macro_identifier` is a Verilog HDL *simple_identifier*. The `ifdef_group_of_lines`, `ifndef_group_of_lines`, `elsif_group_of_lines` and the `else_group_of_lines` are parts of a Verilog HDL source description. The **'else** and **'elsif** compiler directives and all of the groups of lines are optional.

The **'ifdef**, **'else**, **'elsif**, and **'endif** compiler directives work together in the following manner:

- When an **'ifdef** is encountered, the `ifdef` text macro identifier is tested to see if it is defined as a text macro name using **'define** within the Verilog HDL source description.
- If the `ifdef` text macro identifier is defined, the `ifdef` group of lines is compiled as part of the description and if there are **'else** or **'elsif** compiler directives, these compiler directives and corresponding groups of lines are ignored.
- If the `ifdef` text macro identifier has not been defined, the `ifdef` group of lines is ignored.
- If there is an **'elsif** compiler directive, the `elsif` text macro identifier is tested to see if it is defined as a text macro name using **'define** within the Verilog HDL source description.
- If the `elsif` text macro identifier is defined, the `elsif` group of lines is compiled as part of the description and if there are other **'elsif** or **'else** compiler directives, the other **'elsif** or **'else** directives and corresponding groups of lines are ignored.
- If the first `elsif` text macro identifier has not been defined, the first `elsif` group of lines is ignored.
- If there are multiple **'elsif** compiler directives, they are evaluated like the first **'elsif** compiler directive in the order they are written in the Verilog HDL source description.
- If there is an **'else** compiler directive, the `else` group of lines is compiled as part of the description.

Although the names of compiler directives are contained in the same name space as text macro names, the names of compiler directives are considered not to be defined by **'ifdef**, **'ifndef**, and **'elsif**.

The **'ifndef**, **'else**, **'elsif**, and **'endif** compiler directives work together in the following manner:

- When an **'ifndef** is encountered, the **ifndef** text macro identifier is tested to see if it is defined as a text macro name using **define** within the Verilog HDL source description.
- If the **ifndef** text macro identifier is not defined, the **ifndef** group of lines is compiled as part of the description and if there are **'else** or **'elsif** compiler directives, these compiler directives and corresponding groups of lines are ignored.
- If the **ifndef** text macro identifier is defined, the **ifndef** group of lines is ignored.
- If there is an **'elsif** compiler directive, the **elsif** text macro identifier is tested to see if it is defined as a text macro name using **define** within the Verilog HDL source description.
- If the **elsif** text macro identifier is defined, the **elsif** group of lines is compiled as part of the description and if there are other **'elsif** or **'else** compiler directives, the other **'elsif** or **'else** directives and corresponding groups of lines are ignored.
- If the first **elsif** text macro identifier has not been defined, the first **elsif** group of lines is ignored.
- If there are multiple **'elsif** compiler directives, they are evaluated like the first **'elsif** compiler directive in the order they are written in the Verilog HDL source description.
- If there is an **'else** compiler directive, the **else** group of lines is compiled as part of the description.

Although the names of compiler directives are contained in the same name space as text macro names, the names of compiler directives are considered not to be defined by **'ifdef**, **'ifndef**, and **'elsif**.

Nesting of **'ifdef**, **'ifndef**, **'else**, **'elsif**, and **'endif** compiler directives shall be permitted.

NOTE 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.

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

```

Example 3 The following example shows usage of chained nested conditional compilation directives.

```

module test:
    'ifdef first_block
        'ifdef second_nest
            initial $display("first_block is defined");
        'else
            initial $display("first_block and second_nest defined");
        'endif
    'elsif second_block
        initial $display("second_block defined, first_block is not");
    'else
        'ifdef last_result
            initial $display("first_block, second_block, last_result
                not defined.");
        'elsif real_last
            initial $display("first_block, second_block not defined,
                last_result and real_last defined.");
        'else
            initial $display("Only last_result defined!");
        'endif
    'endif
endmodule

```

19.5 'include

The file inclusion (**'include**) compiler directive is used to insert the entire contents of a source file in another file during compilation. The result is as though the contents of the included source file appear in place of the **'include** compiler directive. The **'include** compiler directive can be used to include global or commonly used definitions and tasks without encapsulating repeated code within module boundaries.

Advantages of using the **'include** compiler directive include the following:

- Providing an integral part of configuration management
- Improving the organization of Verilog HDL source descriptions
- Facilitating the maintenance of Verilog HDL source descriptions

The syntax for the **'include** compiler directive is given in Syntax 19-6.

```
include_compiler_directive ::=  
    'include "filename"
```

Syntax 19-6 Syntax for include compiler directive

The compiler directive **'include** can be specified anywhere within the Verilog HDL description. The *file-name* 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.

19.6 'resetall

When **'resetall** compiler directive is encountered during compilation, all compiler directives are set to the default values. This is useful for ensuring that only 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.

19.7 'line

The compiler is expected to maintain the current line and the filename of the file being compiled. The line number (**'line**) compiler directive is used to reset the current line number and filename of the current file to the line number and filename presented. This can be used to reflect the location in an original file; if the actual source file has been modified by addition or reduction of lines. After specifying the new line number or file name, the compiler can correctly refer to the original source file location. For example error messages, source code debugging, etc. can direct the user to the actual original line.

The syntax for the **'line** compiler directive is given in Syntax 19-7.

<pre>line_compiler_directive ::= 'line number "filename" level</pre>

Syntax 19-7 Syntax for line compiler directive

The directive can be specified anywhere within the Verilog HDL source description. The number parameter is the new line number of the next line. The filename parameter is the new name of the file. The filename can be a full or relative path name. The level parameter indicates whether an include file has been entered (value is 1), an include file is exited (value is 2), or neither has been done (value is 0).

The results of this directive are not affected by the compiler directive **'resetall**. As the compiler processes the remainder of the file and new files, the line number shall be incremented as each line is read and the filename shall be updated to the new current file being processed. When beginning to read include files, the current line and filename shall be stored for restoration at the termination of the include file. The updated line number and filename information shall be available for PLI access. The mechanism of library searching is not affected by the effects of the **'line** compiler directive.

19.8 'timescale

This directive specifies the time unit and time precision of the modules that follow it. The time unit is the unit of measurement for time values such as the simulation time and delay values.

To use modules with different time units in the same design, the following timescale constructs are useful:

- The **'timescale** compiler directive to specify the unit of measurement for time and precision of time in the modules in the design
- The **\$printrtimescale** system task to display the time unit and precision of a module
- The **\$time** and **\$realtime** system functions, the **\$timeformat** system task, and the **%t** format specification to specify how time information is reported

The **'timescale** compiler directive specifies the unit of measurement for time and delay values and the degree of accuracy for delays in all modules that follow this directive until another **'timescale** compiler directive is read. If there is no **'timescale** specified or it has been reset by a **'resetall** directive, the time unit and precision are simulator specific. It shall be an error if some modules have a **'timescale** specified and others do not.

The syntax for the **'timescale** directive is given in Syntax 19-8.

```
timescale_compiler_directive ::=
    'timescale time_unit / time_precision
```

Syntax 19-8 Syntax for timescale compiler directive

The `time_unit` argument specifies the unit of measurement for times and delays.

The `time_precision` argument specifies how delay values are rounded before being used in simulation. The values used are accurate to within the unit of time specified here, even if there is a smaller `time_precision` argument elsewhere in the design. The smallest `time_precision` argument of all the **'timescale** compiler directives in the design determines the precision of the time unit of the simulation.

The `time_precision` argument shall be at least as precise as the `time_unit` argument; it cannot specify a longer unit of time than `time_unit`.

The integers in these arguments specify an order of magnitude for the size of the value; the valid integers are 1, 10, and 100. The character strings represent units of measurement; the valid character strings are **s**, **ms**, **us**, **ns**, **ps**, and **fs**.

The units of measurement specified by these character strings are given in Table 86.

Table 86—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 ns because the `time_unit` argument is 10 ns. 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.

19.9 'unconnected_drive and 'nounconnected_drive

All unconnected input ports of a module appearing between the directives **'unconnected_drive** and **'nounconnected_drive** are pulled up or pulled down instead of the normal default.

The directive **'unconnected_drive** takes one of two arguments **pull1** or **pull0**. When **pull1** is specified, all unconnected input ports are automatically pulled up. When **pull0** is specified, unconnected ports are pulled down. These directives shall be specified in pairs, and outside of the module declarations.

The **'resetall** directive includes the effects of a **'endcelldefine** directive.

20. PLI overview

20.1 PLI purpose and history (informative)

Clause 20 through Clause 27 and Annex E through Annex G describe the C language procedural interface standard and interface mechanisms that are part of the Verilog HDL. This procedural interface, known as the 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 the Verilog PLI that has been in use since the mid-1980s. This standard comprises three primary generations of the Verilog PLI.

- 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.

20.2 User-defined system task or function names

A user-defined system 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

20.3 User-defined system task or function types

The type of a user-defined system task or function determines how a PLI application is called from the Verilog HDL source code. The types are:

- A user *task* can be used in the same places a Verilog HDL task can be used (refer to 10.2). A user-defined system task can read and modify the arguments of the task, but does not return any value.
- A user *function* can be used in the same places a Verilog HDL function can be used (refer to 10.3). A user-defined system function can read and modify the arguments of the function, and shall return a scalar or vector value. The bit width of the return value shall be determined by a user-supplied *size* application (see 21.1.1).
- A user *real-function* can be used in the same places a Verilog HDL function can be used (refer to 10.3). A user-defined system real-function can read and modify the arguments of the function, and will return a double-precision floating point value.

20.4 Overriding built-in system task and function names

Clause 17 defines a number of built-in system tasks and functions that are part of the Verilog language. In addition, software products can include other built-in system tasks and functions specific to the product. These built-in system task 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 override the built-in system task/function, replacing its functionality with that of the user-provided C application. For example, a user could write a random number generator as a PLI application and then associate the application with the name **\$random**, thereby overriding the built-in **\$random** function with the user's application.

Verilog timing checks, such as **\$setup**, are not system tasks, and cannot be overridden.

The system functions **\$signed** and **\$unsigned** can be overridden. These system functions are unique in the Verilog HDL, in that the return width is based on the width of their argument. If overridden, the PLI version shall have the same return width for all instances of the system function. The PLI return width is defined by the PLI *size* routine.

20.5 User-supplied PLI applications

User-supplied PLI applications are C language functions that utilize the library of PLI C functions to access and interact dynamically with Verilog HDL software implementations as the Verilog HDL source code is executed.

These PLI applications are not independent C programs. They are C functions, 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.

20.6 PLI interface mechanism

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 provides two interface mechanisms:

- The TF and ACC interface mechanism is an older interface, which can be used to associate PLI applications which use routines from the ACC and TF function libraries. This interface mechanism is described in Clause 21.
- The VPI interface mechanism is a newer interface, which can be used to associate PLI applications which use routines from the VPI function libraries. This interface mechanism is described in Clause 26.

Instances of system tasks and functions which are defined using the TF and ACC interface mechanism can only be accessed using the TF and ACC function libraries. Instances of system tasks and functions which are defined using the VPI interface mechanism can only be accessed using the VPI function library.

20.7 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*). 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, TF routines and VPI routines for information on specific routines that work with task/function arguments.

20.8 PLI include files

The libraries of PLI functions are defined in C include files, which are a normative part of the 1364 standard. These files also define constants, structures, and other data used by the library of PLI routines and the interface mechanisms. The files are `acc_user.h` (listed in Annex E), `veriusr.h` (listed in Annex F) and `vpi_user.h` (listed in Annex G).

- PLI applications that use the TF routines shall include the file `veriusr.h`.
- PLI applications that use the ACC routines shall include the file `acc_user.h`.
- PLI applications that use the VPI routines shall include the file `vpi_user.h`.

20.9 PLI Memory Restrictions

Memory allocated by the PLI routines is not to be modified by the user, with the exception of the value storage returned by the PLI routines `tf_exprinfo()` and `tf_nodeinfo()`, as defined in 25.15 and 25.35.

21. 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 Clause 26 and Clause 27.

21.1 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 for TF and ACC routines provides five classes of user-supplied PLI applications: *checktf* applications, *size tf* applications, *calltf* applications, *misctf* applications, and *consumer* applications. The *size tf*, *checktf*, *calltf*, and *misctf* routines are called during specific periods during processing. The purpose of each of the PLI application classes is explained in the following subsections.

21.1.1 The size tf class of PLI applications

A *size tf* 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 value shall be. The *size tf* application may be called early in the process, prior to a complete instantiation of the design. As a result, access to objects may be limited at this time. Each *size tf* function shall be called at most once. It shall be called if its associated system function appears in the design. The value returned by the *size tf* function shall be the number of bits that the *calltf* routine shall provide as the return value for the system function. If no *size tf* application is specified for a user-defined system function, the function shall return 32-bits. The *size tf* application shall not be called for user-defined system tasks or real-functions.

21.1.2 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. The *checktf* shall be called at the earliest possible time after all

simulation data structures required by the PLI are available. Generally this means after the design is fully instantiated, but no simulation events have occurred. By the time the checktf application is called, all PLI routines can be used without concern for the state of the process, with the exception of setting the return value of user defined system functions. The return value of a user defined system function can only be set during a calltf application.

21.1.3 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*.

21.1.4 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 tasks. 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 87 and Table 88 list the reasons for which the *misctf* application can be called.

For most reasons, the *misctf* routine will not be called until the instance of the system task has been executed (at which point the *calltf* routine is called). The following reasons are exceptions, and will be called for each instance of the system task in the design regardless of whether or not it has been executed:

- reason_endofcompile
- reason_save
- reason_startofsave
- reason_restart
- reason_endofreset
- reason_reset

21.1.5 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.

21.2 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.

For the TF and ACC interface mechanism, 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.

21.3 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 subsections.

21.3.1 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.

21.3.2 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 `veruser.h`. The reason constant that is passed is based on the class of the PLI application, as follows:

- The `calltf` application is passed the reason constant `reason_calltf`.
- The `checktf` application is passed the reason constant `reason_checktf`.
- The `sizetf` application is passed the reason constant `reason_sizetf`.
- The `misctf` application is passed one of the constants listed in Table 87. Software implementations can define additional reason constants to be passed to the `misctf` application. Table 88 lists some common reason constants that can be available in some software implementations.

Table 87—(normative) Predefined `misctf` reason constants

Integer constant	Reason
<code>reason_endofcompile</code>	end of Verilog source compilation/start of execution
<code>reason_paramvc</code>	a change of value on a user-defined system task or function argument
<code>reason_synch</code>	end of a time step flagged by <code>tf_synchronize()</code>
<code>reason_rosynch</code>	end of a time step flagged by <code>tf_rosynchronize()</code>
<code>reason_reactivate</code>	a simulation event scheduled by <code>tf_setdelay()</code>
<code>reason_finish</code>	the <code>\$finish()</code> built-in system task executed

Table 88—(informative) Additional misctf reason constants

Integer constant	Reason
reason_paramdrc	a value change on the driver of a user-defined system task or function argument
reason_force	execution of a procedural force or procedural continuous assignment on any net, reg, integer variable, time variable or real variable
reason_release	execution of a procedural release or procedural deassign on any net, reg, integer variable, time variable or real variable
reason_disable	execution of a procedural disable statement
reason_interactive	execution of the \$stop() built-in system task
reason_scope	execution of the \$scope() built-in system task
reason_startofsave	start of execution of the \$save() built-in system task
reason_save	completion of execution of the \$save() built-in system task
reason_restart	execution of the \$restart() built-in system task
reason_reset	start of execution of the \$reset() built-in system task
reason_endofreset	completion of execution of the \$reset() built-in system task

21.3.3 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**. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

22. Using ACC routines

This clause presents a general discussion of how and why to use PLI ACC routines. Clause 23 defines the ACC routine syntax, listed in alphabetical order.

22.1 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 or data paths
- Intermodule paths
- Top-level modules
- Primitive instances
- Primitive terminals
- Nets
- Reg variables
- 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
- Reg variable logic values
- Net logic values (force/release only)
- Integer, real and time variable values
- Sequential UDP logic values

22.2 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 22.4.2 for a discussion of handle routines and 22.4.3 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");
```

22.3 Using ACC routines

22.3.1 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 B.

22.3.2 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.

22.3.3 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.

22.4 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 subclause contains a summary list of each major category. The ACC routines sorted by the types of objects they work with are listed in 22.5. Clause 23 presents an alphabetical list of all ACC routines, with their functions, syntax, and usage.

22.4.1 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 89—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 parameter or specparam as a double
acc_fetch_attribute_int()	Get the value of a Verilog parameter or specparam as an integer
acc_fetch_attribute_str()	Get the value of a Verilog parameter or specparam as a string
acc_fetch_defname()	Get the definition name of a module or primitive
acc_fetch_delay_mode()	Get the delay mode of a module instance
acc_fetch_delays()	Get the existing delays for a primitive, module path, timing check, intermodule path, or module input port
acc_fetch_direction()	Get the direction of a module port or primitive terminal
acc_fetch_edge()	Get the edge specifier of a 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_itfarg()	Get the value of an instance of a system task/function argument as a double
acc_fetch_itfarg_int()	Get the value of an instance of a system task/function argument as an integer
acc_fetch_itfarg_str()	Get the value of an instance of a system task/function argument as a string
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_pulsezero()	Get the current pulse handling values of a module path, intermodule path or module input port
acc_fetch_range()	Get the range of a vector
acc_fetch_size()	Get the bit size of a vector or port
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, reg, integer variable, time variable or real variable

22.4.2 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 90—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
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
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 a terminal of a primitive based on the terminal index
acc_handle_tfarg()	Get the handle to the object named in a system task/function argument
acc_handle_tfinst()	Get the handle to the current instance of a system task/function

22.4.3 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 91 summarizes how next routines shall find each target object associated with a given reference object.

Table 91 — 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

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**.

NOTE Most next routines can return objects in an arbitrary order. However, certain next routines shall return objects in a defined order, as noted in the description of the routine in Clause 23.

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();

    /*get handle for module*/
    module_handle = acc_handle_tfarg(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 92—List of next routines

ACC routine	Description
acc_next()	Get handles to all objects of a set of types
acc_next_bit()	Get handles to all bits of a port or vector
acc_next_cell()	Get handles to all cell modules in the current hierarchy and below
acc_next_cell_load()	Get handles to all cell loads on a net
acc_next_child()	Get handles to all module instances within a module
acc_next_driver()	Get handles to all primitive terminals that drive a net
acc_next_hiconn()	Get handles to all nets connected hierarchically higher to a module port
acc_next_input()	Get handles to all input terminals of a module path or data path
acc_next_load()	* Get handles to all primitive terminals driven by a net
acc_next_loconn()	Get handles to all nets connected hierarchically lower to a module port
acc_next_modpath()	Get handles to all paths in a module
acc_next_net()	Get handles to all nets in a module
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

22.4.4 Modify routines

Modify routines shall alter the values of a variety of objects in the design hierarchy. Table 93 lists the types of values that shall be modified for particular objects.

Table 93— 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	Variable data types Net data types Sequential UDPs
Pulse handling values	Module paths Intermodule paths Module input ports

Table 94— List of modify routines

ACC routine	Description
acc_append_delays()	Add delays to existing delays on primitives, module paths, timing checks, intermodule paths, and module input ports
acc_append_pulsere()	Add to existing pulse control values of module paths, intermodule paths and module input ports
acc_replace_delays()	Replace existing delays on primitives, module paths, timing checks, intermodule paths and module input ports
acc_replace_pulsere()	Replace existing values on pulse control values of module paths, intermodule paths and module input ports
acc_set_pulsere()	Set the pulse control values for a module path, intermodule path or module input port as a percentage of the delay
acc_set_value()	Set and propagate a logic value onto a reg, integer variable, time variable, real variable or sequential UDP; continuously assign/deassign a reg or variable; force/release a net or reg or variable

More details on using the **acc_append_delays()** and **acc_replace_delays()** ACC routines are provided in 22.8.

22.4.5 Miscellaneous routines

Miscellaneous routines shall perform a variety of operations, such as initializing and configuring the ACC routine environment.

Table 95—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_typelist()	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

22.4.6 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 22.10.

Table 96—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

22.5 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)
- Regs (scalars, vectors, and bit- or part-selects of vectors)
- Integer variables (and bit- or part-selects of integers)
- Real and time variables
- Named events
- Parameters
- Specparams
- Timing checks
- Timing check terminals
- User-Defined system task/function arguments

The following tables summarize the operations that can be performed for each of the above object types.

22.5.1 ACC routines that operate on module instances

Table 97—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()
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()

22.5.2 ACC routines that operate on module ports

Table 98—Operations on module ports

To	Use
Obtain handles for ports of a module instance	acc_next_port()
Obtain handles for output ports of a module instance	acc_next_portout()
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()
Read Module Input Port Delay (MIPD)	acc_fetch_delays()
Append to existing MIPD	acc_append_delays()
Replace existing MIPD	acc_replace_delays()

22.5.3 ACC routines that operate on bits of a port

Table 99—Operations on bits of a port

To	Use
Obtain handles for bits of a module port	acc_next_bit()
Obtain a handle to the port from a port bit	acc_handle_parent()
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 Module Input Port Delay (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()

22.5.4 ACC routines that operate on module paths or data paths

Table 100—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 net	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()
Append to existing path pulse control values	acc_append_pulsere()
Replace existing path pulse control values	acc_replace_pulsere()
Specify path pulse control values	acc_set_pulsere()
Free memory allocated by acc_next_input() or acc_next_output()	acc_release_object()

22.5.5 ACC routines that operate on intermodule paths

Table 101—Operations on intermodule paths

To	Use
Obtain a handle for an intermodule path	<code>acc_handle_path()</code>
Get the fulltype of an intermodule path	<code>acc_fetch_fulltype()</code>
Read intermodule path delays	<code>acc_fetch_delays()</code>
Modify intermodule path delays	<code>acc_replace_delays()</code>
Read intermodule path pulse control values	<code>acc_fetch_pulsere()</code>
Append to existing intermodule path pulse control values	<code>acc_append_pulsere()</code>
Replace existing intermodule path pulse control values	<code>acc_replace_pulsere()</code>
Specify intermodule path pulse control values	<code>acc_set_pulsere()</code>

22.5.6 ACC routines that operate on top-level modules

Table 102—Operations on top-level modules

To	Use
Obtain handles for top-level modules in a design	<code>acc_next_topmod()</code> <code>acc_next_child()</code>
Get the module name	<code>acc_fetch_name()</code> <code>acc_fetch_fullname()</code> <code>acc_fetch_defname()</code>

22.5.7 ACC routines that operate on primitive instances

Table 103—Operations on primitive instances

To	Use
Obtain handles for primitive instances within a module instance	<code>acc_next_primitive()</code>
Obtain a handle to the parent (the module that contains the primitive)	<code>acc_handle_parent()</code>
Get the instance name	<code>acc_fetch_name()</code>
Get the full hierarchical name	<code>acc_fetch_fullname()</code>
Get the definition name	<code>acc_fetch_defname()</code>
Get the primitive <i>fulltype</i>	<code>acc_fetch_fulltype()</code>
Read delays	<code>acc_fetch_delays()</code>
Append to existing primitive delays	<code>acc_append_delays()</code>
Replace existing primitive delays	<code>acc_replace_delays()</code>

22.5.8 ACC routines that operate on primitive terminals

Table 104—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()

22.5.9 ACC routines that operate on nets

Table 105—Operations on nets

To	Use
Obtain handles for nets within a module instance	acc_next_net()
Obtain handles for nets within a module instance	acc_next()
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()
Get the net vector size	acc_fetch_size()
Get the msb and lsb vector range	acc_fetch_range()
Get the net fulltype	acc_fetch_fulltype()
Get the net logic or strength value	acc_fetch_value()
Force or release the net logic value	acc_set_value()
Add VCL value change callback monitors	acc_vcl_add()
Delete VCL value change callback monitors	acc_vcl_delete()

22.5.10 ACC routines that operate on reg types

Table 106—Operations on reg types

To	Use
Obtain handles to regs within a given scope	acc_next()
Obtain handles to bits of a vector reg	acc_next_bit()
Obtain a handle to the parent (module instance containing the reg)	acc_handle_parent()
Obtain handles to load terminals of the reg	acc_next_load()
Determine if reg is a scalar or a vector	acc_object_of_type()
Get the reg name	acc_fetch_name()
Get the full hierarchical name	acc_fetch_fullname()
Get the reg size	acc_fetch_size()
Get the msb and lsb vector range	acc_fetch_range()
Get the reg value	acc_fetch_value()
Set the reg value	acc_set_value()
Add VCL value change callback monitors	acc_vcl_add()
Delete VCL value change callback monitors	acc_vcl_delete()

22.5.11 ACC routines that operate on integer, real, and time variables

Table 107—Operations on integer, real, and time variables

To	Use
Obtain handles to variables within a given scope	acc_next()
Obtain a handle to the parent (module instance containing the variable)	acc_handle_parent()
Get the variable name	acc_fetch_name()
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()

22.5.12 ACC routines that operate on named events

Table 108—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()

22.5.13 ACC routines that operate on parameters and specparams

Table 109—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()

22.5.14 ACC routines that operate on timing checks

Table 110—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 handles to all timing check terminals	acc_next_input()
Free memory allocated by acc_next_input()	acc_release_object()
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()

22.5.15 ACC routines that operate on timing check terminals

Table 111—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()

22.5.16 ACC routines that operate on user-defined system task/function arguments

Table 112—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()

22.6 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 113 lists all type and fulltype constants that shall be supported by ACC routines, listed alphabetically by the type name.

Table 113—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
accFunction	accFunction	Object is a Verilog HDL function
accIntegerVar	accIntegerVar	Object is declared as an integer data type
accModPath	accModPath	Object is a module path
accModule	accModuleInstance	Object is a module instance
	accCellInstance	Object is a module instance that has been defined as a cell
	accTopModule	Object is a top-level module
accNamedEvent	accNamedEvent	Object is declared as an event data type

Table 113—List of all predefined type and fulltype constants (continued)

type constant	fulltype constant	Description
accNet	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	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
accPathTerminal	accPathInput	Object is an input terminal of a module path
	accPathOutput	Object is an output terminal of a module path
accPort	accConcatPort	Object is a module port concatenation
	accScalarPort	Object is a scalar module port
	accBitSelectPort	Object is a bit-select of a module port (e.g.: module (.a[1], .a[0], ...); input [1:0] a;
	accPartSelectPort	Object is a part-select of a module port (e.g.: module (.a[3:2], .a[1:0], ...); input [3:0] a;
	accVectorPort	Object is a vector module port
accPortBit	accPortBit	Object is a bit of a module port

Table 113—List of all predefined type and fulltype constants (continued)

type constant	fulltype constant	Description
accPrimitive	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
accSpecparam	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

Table 113—List of all predefined type and fulltype constants (continued)

type constant	fulltype constant	Description
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	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	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)

22.7 Error handling

When an ACC routine detects an error, it shall perform the following operations:

- Set the global error flag **acc_error_flag** to non-zero
- Display an error message at run time to the output channel of the software product which invoked the PLI application
- Return an exception value

When an ACC routine is called, it automatically resets **acc_error_flag** to 0.

22.7.1 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".

22.7.2 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".

22.7.3 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 53.

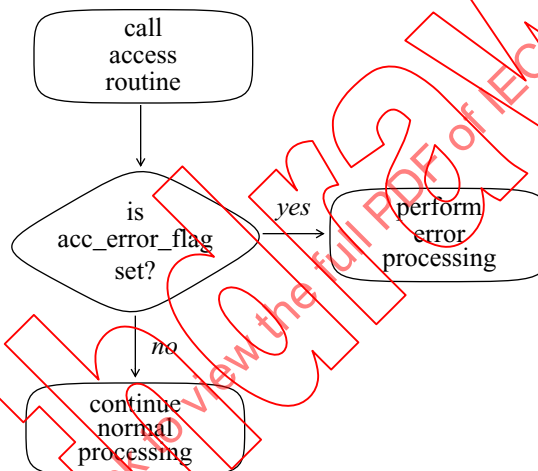


Figure 53—Using acc_error_flag to detect errors

22.7.4 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" );

    /* 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();
}
```

22.7.5 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 114—Exception values returned by ACC routines on errors

When routine returns	The exception value shall be
PLI_INT32	0
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`.

22.8 Reading and writing delay values

This section 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 Clause 23 for the complete syntax of each of these routines.

22.8.1 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 115 lists the objects that can have delays and the number of different delays for each object.

Table 115—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 paths Intermodule paths Module ports Module port bits	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
Timing checks	1	One delay for: timing limit

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, intermodule path and module input port with 1 delay might have one value or three values, and a module path, intermodule path and module input port with 12 delays can have 12 delay values or 36 delay values.

22.8.2 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 115.

How these configuration parameters are used is presented in 22.8.3. Refer to 23.6 for details on using `acc_configure()`.

Table 116—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 three delay values, representing minimum, typical, maximum, respectively. The default shall be "false".
accToHiZDelay	When set to "average", "max" or "min", the delay modify ACC routines shall calculate the toZ delay for 3-state primitives, or for path and input port objects when accPathDelayCount is set to 2. When set to "from_user", the toZ delay shall not be calculated. The default is "from_user". 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, intermodule path and module input port delays. Shall be set to "1", "2", "3", "6", or "12". The default shall be "6".

22.8.3 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 subsections discuss how these factors affect the number of arguments for delay ACC routines.

22.8.3.1 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 117.

Table 117—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	accToHiZDelay set to "min", "max", or "average"	2 arguments: rise, fall transitions (toZ delay is calculated; see Section 22.8.3.3)
	accToHiZDelay set to "from_user"	3 arguments: rise, fall, toZ transitions

Table 117—Number of delay arguments in single delay mode (continued)

Object handle type	Configuration parameters	Number and order of delay arguments
Module paths Intermodule paths Module ports Module port bits	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

22.8.3.2 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 placed into or read from the array shall be determined by the type of object handle passed to the delay ACC routine, as shown in Table 118.

Table 118—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		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 at least 9 elements shall be declared, even if toZ delays are not used by the object)

Table 118—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 Intermodule paths Module ports Module port bits	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 118—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

22.8.3.3 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 119.

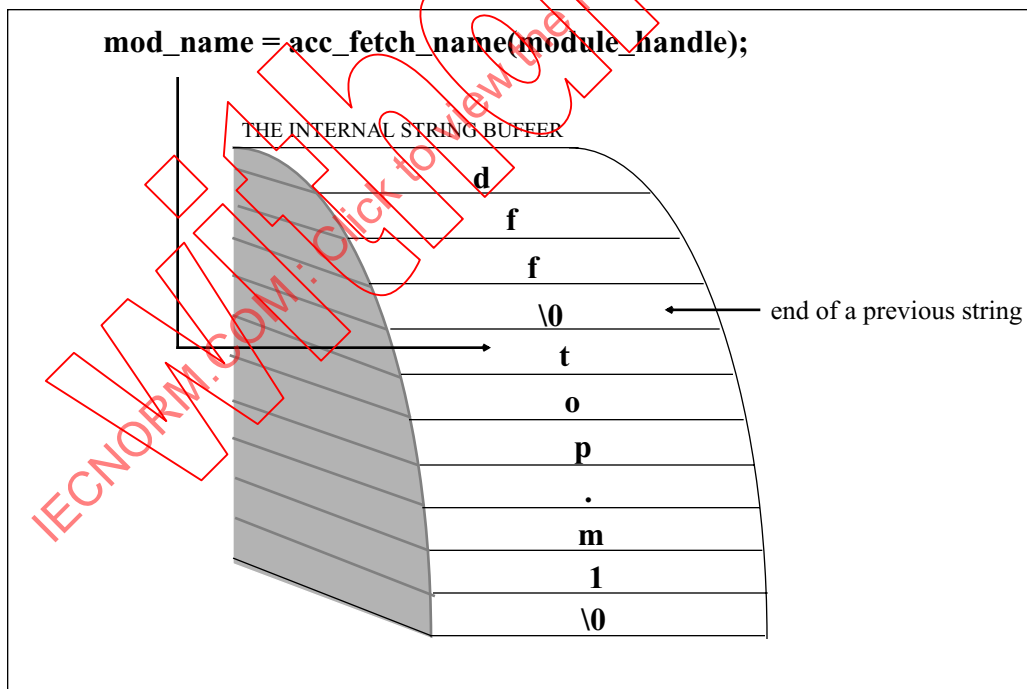
Table 119—Configuring accToHiZDelay to determine the toZ delay

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

22.9 String handling

22.9.1 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 54. 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 54—How ACC routines store strings in the internal buffer**

22.9.2 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 55.

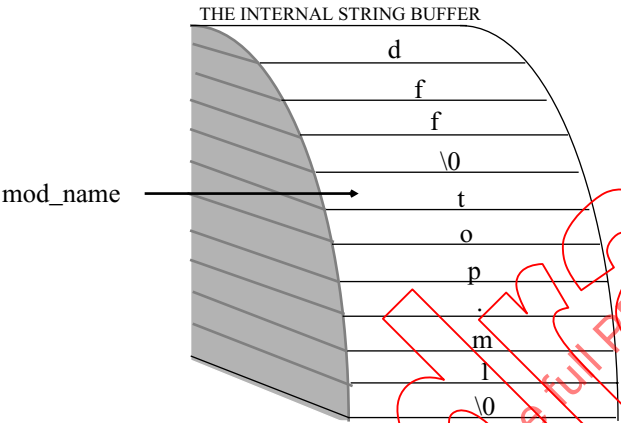
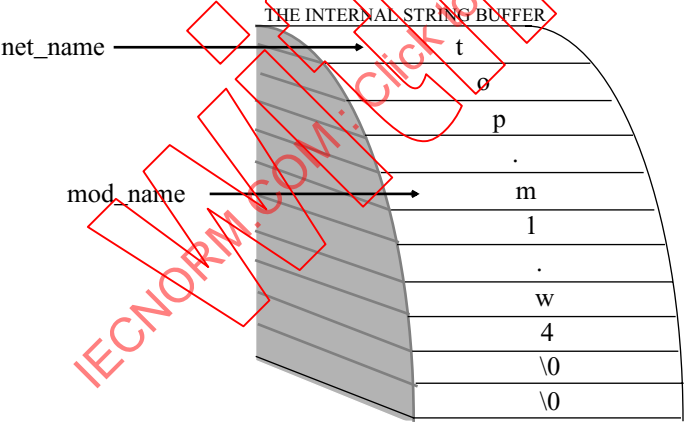
<i>Action:</i>	<i>Results:</i>
mod_name = acc_fetch_fullname(module_handle); 	mod_name points to the string "top.m1". The string happens to be stored near the end of the buffer.
net_name = acc_fetch_fullname(net_handle); 	acc_fetch_fullname() cannot place the next string at the end of the buffer. Therefore, a buffer reset occurs. net_name points to the string "top.m1.w4" 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 "m1.w4".

Figure 55—Buffer reset causes data in the string buffer to be overwritten

22.9.2.1 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()**.

22.9.3 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.

22.9.4 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"
```

```
id display_cells_in_module(mod)
ndle mod;
```

```
handle cell;
char *mod_name;
PLI_BYTE8 *temp;
```

```
/* save the module name in local buffer mod_name */
temp = acc_fetch_fullname(mod);
mod_name = (char*)malloc((strlen((char*)temp)+1) * sizeof(PLI_BYTE8));
```

storage the size of the full
module name is allocated

```
strcpy(mod_name, (char*)temp);
```

`strcpy` saves the full module
name in `mod_name`

```
cell = null;
while (cell = acc_next_cell(mod, cell) )
    io_printf( "%s %s\n", mod_name, acc_fetch_name( cell ) );

free(mod_name);
```

22.10 Using VCL ACC routines

The VCL routines add or delete value change monitors on a specified object. If a value change monitor is placed on an 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. 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 23.97 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 22.10.1.

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 56.

The *user_data* argument shall be a **PLI_BYTE8** 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 allocate memory for a user-defined structure with several values that need to be passed to the consumer routine. A pointer to the memory for the user-defined structure is then passed as the *user_data* argument. Note that the *user_data* argument is defined as a **PLI_BYTE8** pointer; therefore, any other data type should be cast to a **PLI_BYTE8** 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()**.

22.10.1 VCL objects

The VCL shall monitor value changes for the following objects:

- Scalar variables and bit-selects of vector variables
- Scalar nets, unexpanded vector nets, and bit-selects of expanded vector nets
- Integer, real, and time variables
- Module ports
- Primitive output or inout terminals
- Named events

Note Adding a value change link to a module port is equivalent to adding a value change link to the *loconn* of the port. The *vc_reason* returned shall be based on the *loconn* of the port.

22.10.2 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 56.


```
typedef struct t_vc_record
{
    PLI_INT32 vc_reason;
    PLI_INT32 vc_hightime;
    PLI_INT32 vc_lowtime;
    PLI_BYTE8 *user_data;
    union
    {
        PLI_UBYTE8 logic_value;
        double real_value;
        handle vector_handle;
        s_strengths strengths_s;
    } out_value;
} s_vc_record, *p_vc_record;
```

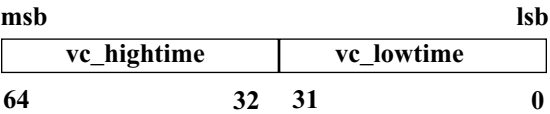
Figure 56—The VCL s_vc_record structure

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 120.

Table 120—Predefined vc_reason constants

Predefined vc_reason 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 reg changed logic value.
vregister_value_change	A vector reg or part-select of a vector reg 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	A named event occurred.

The *vc_hightime* and *vc_lowtime* fields of *vc_record* shall be 32-bit integers that shall contain the simulation time in the simulator's time units during which the change occurred, as follows:



The *user_data* field of *vc_record* shall be a *PLI_BYTE8* 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 121.

Table 121 – Predefined out_value constants

If vc_reason is	The out_value shall be a type of	Description
logic_value_change	PLI_UBYTE8	A predefined constant, from the following: vcI0 vcI1 vcIX vcLx vcIZ vcLz
strength_value_change	s_strengths structure	A structure with logic and strength, as shown in Figure 57
vector_value_change	handle	A handle to a vector net or part-select of a vector net
sregister_value_change	PLI_UBYTE8	A predefined constant, from the following: vcI0 vcI1 vcIX vcLx vcIZ vcLz
vregister_value_change	handle	A handle to a vector reg or part-select of a vector reg
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**, the s_strengths structure fields of the out_value field of vc_record shall contain the value. This structure shall contain three fields, as shown in Figure 57.

```
typedef struct t_strengths
{
    PLI_UBYTE8 logic_value;
    PLI_UBYTE8 strength1;
    PLI_UBYTE8 strength2;
} s_strengths, *p_strengths;
```

Figure 57 – The VCL s_strengths structure

The values of the s_strengths structure fields are defined in Table 122.

Table 122 – Predefined out_value constants

s_strengths field	C data type	Description
logic_value	PLI_UBYTE8	A predefined constant, from the following: vcI0 vcI1 vcIX vcLx vcIZ vcLz
strength1 strength2	PLI_UBYTE8	A predefined constant, from the following: vcISupply vcIWeak vcIStrong vcIMedium vcIPull vcISmall vcILarge vcIHighZ

The *strength1* and *strength2* fields of the *s_strengths* structure can represent

- a) A known strength when *strength1* and *strength2* contain the same value, the signal strength shall be that value.
- b) An ambiguous strength with a known *logic_value* when *strength1* and *strength2* contain different values and the *logic_value* contains either **vc10** or **vc11**, 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.
- c) An ambiguous strength with an unknown *logic_value* when *strength1* and *strength2* contain different values and the *logic_value* contains **vc1X**, 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.

22.10.3 Effects 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 23.6 on **acc_configure()** for a list of routines that are affected by configuration parameters.

22.10.4 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 Clause 21.

```
$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.

```
PLI_INT32 netmon_checktf()  
{  
    int i;  
    PLI_INT32 arg_cnt = tf_nump();  
  
    /* initialize the environment for access routines */  
    acc_initialize();  
  
    /* check number and type of task/function arguments */  
    if (arg_cnt == 0)  
        tf_error("$net_monitor[_off] must have at least one argument");  
    else  
        for (i = 1; i <= arg_cnt; i++)  
            if (acc_fetch_type(acc_handle_tfarg(i)) != accNet) {  
                tf_error("$net_monitor[_off] arg %d is not a net type",i);  
            }  
  
    acc_close();  
    return(0);  
}
```

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`.

```
PLI_INT32 netmon_calltf(data)
PLI_INT32 data;
{
    handle net;
    PLI_INT32 netmon_consumer();
    PLI_INT32 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();

    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)
            {
                /* add a VCL monitor; pass net pointer as user data argument */
                acc_vcl_add(net, netmon_consumer, (PLI_BYTE8*)net,
                           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, (PLI_BYTE8*)net, vcl_verilog);
            }
            break;
    }
    acc_close();
}
```

The consumer routine, `netmon_consumer`, 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. In this example, it is assumed that `$net_monitor` is associated with a data value of 0, and `$net_monitor_off` is associated with a data value of 1. Refer to 21.3.1 for a description of associating data values.

```
PLI_INT32 netmon_consumer(vc_record)
p_vc_record vc_record; /* record type passed to consumer routine */
{
    PLI_BYTE8 net_value;
    char      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,
                    acc_fetch_name((handle) 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,
                    acc_fetch_name((handle) vc_record->user_data),
                    acc_fetch_value(vector_value, "%b", NULL) );
            break;
        }
    }
}
```

23. ACC routine definitions

This clause 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 Clause 23, Clause 25, and Clause 27.

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.

23.1 acc_append_delays()

acc_append_delays() for single delay values (accMinTypMaxDelays set to false)			
Synopsis:	Add delays to existing delay on primitives, module paths, intermodule paths, timing checks, and module input ports.		
Syntax:			
Primitives	acc_append_delays(object_handle, rise_delay, fall_delay, z_delay)		
Module paths Intermodule paths Ports or port bits	acc_append_delays(object_handle, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12)		
Timing checks	acc_append_delays(object_handle, limit)		
Type		Description	
Returns:	PLI_INT32	1 if successful; 0 if an error occurred	
Type		Name	Description
Arguments:	handle	object_handle	Handle of a primitive, module path, intermodule path, timing check, module input port or bit of a module input port
	double	rise_delay fall_delay	Rise and fall transition delay for 2-state primitives, 3-state primitives
Conditional	double	z_delay	If accToOffZDelay is set to from_user : turn-off (to Z) transition delay for 3-state primitives
	double	d1	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 1 : delay for all transitions If accPathDelayCount is set to 2 or 3 : rise transition delay If accPathDelayCount is set to 6 or 12 : 0->1 transition delay
Conditional	double	d2	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 2 or 3 : fall transition delay If accPathDelayCount is set to 6 or 12 : 1->0 transition delay
Conditional	double	d3	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 3 : turn-off transition delay If accPathDelayCount is set to 6 or 12 : 0->Z transition delay
Conditional	double	d4 d5 d6	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 6 or 12 : d4 is Z->1 transition delay d5 is 1->Z transition delay d6 is Z->0 transition delay
Conditional	double	d7 d8 d9 d10 d11 d12	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 12 : d7 is 0->X transition delay d8 is X->1 transition delay d9 is 1->X transition delay d10 is X->0 transition delay d11 is X->Z transition delay d12 is Z->X transition delay
	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, intermodule 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:	PLI_INT32	1 if successful; 0 if an error is encountered	
Type		Name	Description
Arguments:	handle	object_handle	Handle of a primitive, module path, intermodule 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 22.8)
Related routines:	Use acc_fetch_delays() to retrieve an object's delay values Use acc_replace_delays() to replace an object's delay values Use acc_configure() to set accPathDelayCount and accMinTypMaxDelays		

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 section shall apply.

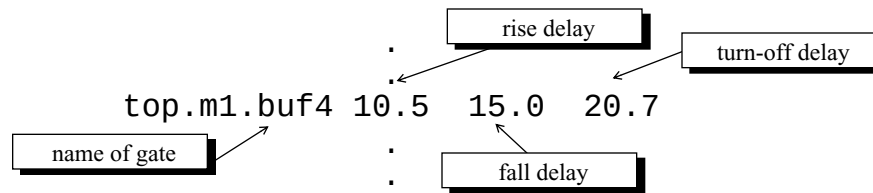
When **accMinTypMaxDelays** is set to true, **acc_append_delays()** shall pass one or more sets of minimum:typical:maximum delays contained in an array, rather than single delays passed as individual arguments. For this min:typ:max delay mode, the second syntax table in this section shall apply.

The number of delay values appended by **acc_append_delays()** shall be determined by the type of object and the setting of configuration parameters. Refer to 22.8 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.

When altering the delay via **acc_append_delays()** the value of the reject/error region will not be affected unless they exceed the value of the delay. If the reject/error limits exceed the delay they will be truncated down to the new delay limit.

The example shown in Figure 58 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 in Figure 58.



```
#include <stdio.h>
#include "acc_user.h"
```

PLI_INT32 write_gate_delays()

```
{
    FILE      *infile;
    PLI_BYTE8  full_gate_name[NAME_SIZE];
    double     rise, fall, toz;
    handle     gate_handle;

    /*initialize the environment for ACC routines*/
    acc_initialize();

    /*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 58—Using acc_append_delays() in single delay value mode

The example shown in Figure 59 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:

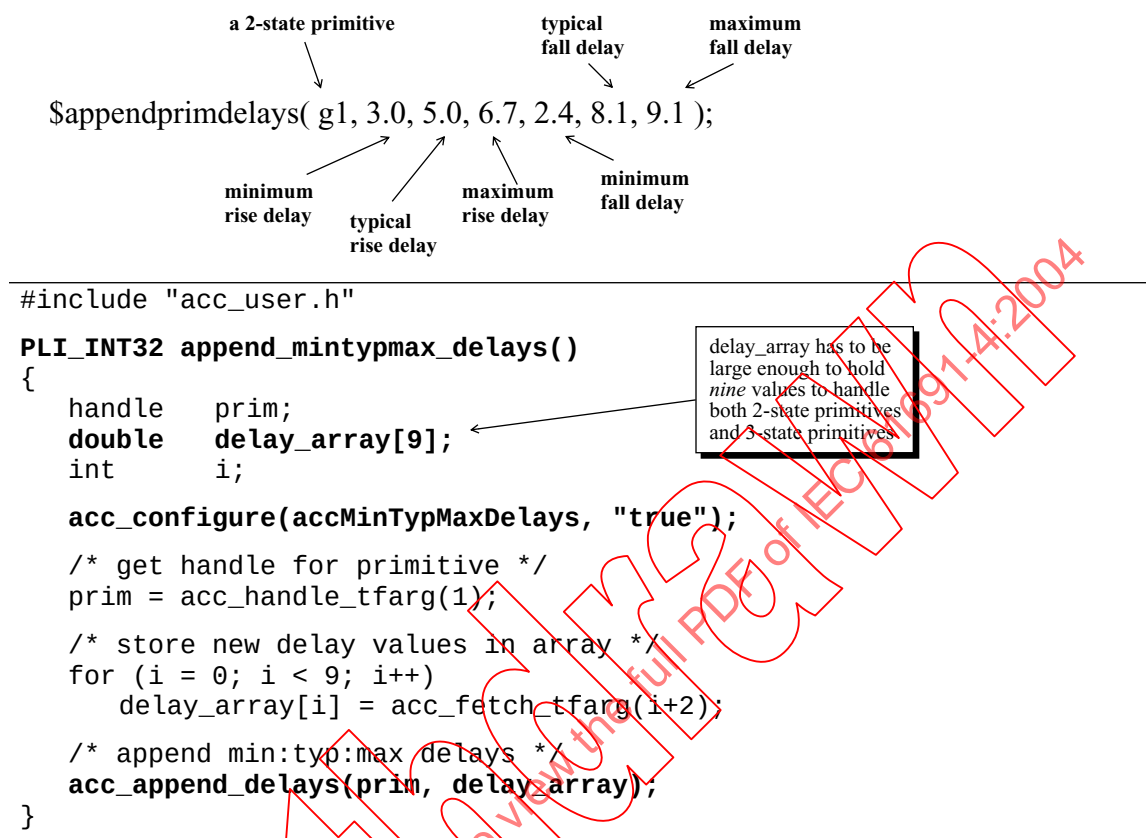


Figure 59—Using `acc_append_delays()` in min:typ:max mode

23.2 acc_append_pulsere()

acc_append_pulsere()			
Synopsis:	Add delays to existing pulse handling <i>reject_limit</i> and <i>e_limit</i> for a module path, intermodule path or module input port.		
Syntax:	acc_append_pulsere(object, 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:	PLI_INT32	1 if successful; 0 if an error is encountered	
Type		Name	Description
Arguments:	handle	object	Handle of module path, intermodule path or module input port
	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 Use acc_configure() to set accPathDelayCount		

The ACC routine **acc_append_pulsere()** shall add to an existing pulse handling *reject_limit* value and *e_limit* value for a module path, intermodule path and module input port. The *reject_limit* and *e_limit* values are used to control how pulses are propagated through paths.

A *pulse* is defined as two transitions that occur in a shorter period of time than the 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.
- 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.
- A pulse that is greater than or equal to the *e_limit* shall propagate.

Table 123 illustrates the relationship between the *reject_limit* and the *e_limit*.

Table 123—Pulse control example

When	The pulse 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 delay.

If any of the limits do not meet the above rules, they shall be truncated.

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 124.

Table 124—How the value of `accPathDelayCount` affects `acc_append_pulsere()`

When <code>accPathDelayCount</code> is	<code>acc_append_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_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 object handle.

23.3 acc_close()

acc_close()		
Synopsis:	Free internal memory used by ACC routines; reset all configuration parameters to default values.	
Syntax:	acc_close()	
Returns:	Type	Description
	void	No return
Arguments:	Type	Name
	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 60 presents a C language routine that calls **acc_close()** before exiting.

```
#include "acc_user.h"
void show_versions()
{
    /*initialize environment for ACC routines*/
    acc_initialize();

    /*show version of ACC routines and simulator */
    io_printf("Running %s with %s\n",acc_version(),acc_product_version()
);
    acc_close();
}
```

Figure 60—Using acc_close()

23.4 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	
Type		Name	Description
Arguments:	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
	PLI_INT32 *	number_of_objects	Integer pointer where the count of objects collected shall be written
Related routines:	All acc_next_routines except acc_next_topmod() 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 61 presents a C language routine that uses **acc_collect()** to collect and display all nets in a module.

```
#include "acc_user.h"

PLI_INT32 display_nets()
{
    handle    *list_of_nets, module_handle;
    PLI_INT32  net_count, i;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 61—Using acc_collect()

23.5 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	
	PLI_INT32	true if handles refer to the same object; false if different objects	
Arguments:	Type	Name	Description
	handle	handle1	Handle to any object
	handle	handle2	Handle to any object

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 62 uses **acc_compare_handles()** to determine if a primitive drives the specified output of a scalar port of a module.

```
#include "acc_user.h"
PLI_INT32 prim_drives_scalar_port(prim, mod, port_num)
handle    prim, mod;
PLI_INT32 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 62—Using acc_compare_handles()

23.6 acc_configure()

acc_configure()			
Synopsis:	Set parameters that control the operation of various ACC routines.		
Syntax:	acc_configure(configuration_parameter, configuration_value)		
Returns:	Type	Description	
	PLI_INT32	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 accPathDelayCount accPathDelimStr accToHiZDelay
Related routines:	quoted string	configuration_value	One of a fixed set of string values for the configuration_parameter
	For accDefaultAttr0 acc_fetch_attribute() acc_fetch_attribute_int() acc_fetch_attribute_str() For accDisplayErrors all ACC routines For accDisplayWarnings all ACC routines For accEnableArgs acc_handle_modpath() acc_handle_tcnk() 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() acc_replace_pulsere()	For accPathDelimStr acc_fetch_attribute() acc_fetch_attribute_int() acc_fetch_attribute_str() 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 125 through 134 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 125—accDefaultAttr0 configuration parameter

accDefaultAttr0	Set of values	Effect	Default
	"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 126—accDevelopmentVersion configuration parameter

accDevelopmentVersion	Set of values	Effect	Default
	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	None (can be used to document which version of ACC routines was used to develop a PLI application)	Current version of ACC routines

Table 127—accDisplayErrors configuration parameter

accDisplayErrors	Set of values	Effect	Default
	"true"	ACC routines shall display error messages	"true"
	"false"	ACC routines shall not display error messages	

Table 128—accDisplayWarnings configuration parameter

accDisplayWarnings	Set of values	Effect	Default
	"true"	ACC routines shall display warning messages	"false"
	"false"	ACC routines shall not display warning messages	

Table 129—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 130—accMapToMipd configuration parameter

	Set of values	Effect	Default
accMapToMipd	"max"	acc_replace_delays() and acc_append_delays() shall map the longest intermodule path delay to the MIPD	"max"
	"min"	acc_replace_delays() and acc_append_delays() shall map the shortest intermodule path delay to the MIPD	
	"latest"	acc_replace_delays() and acc_append_delays() shall map the last intermodule path delay to the MIPD	

Table 131—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 132—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 133—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 134—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 63 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"

PLI_INT32 display_load_capacitance()
{
    handle    module_handle, port_handle, net_handle;
    double    cap_val;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 = %1f\n",
                    acc_fetch_index(port_handle), cap_val);
    }
    acc_close();
}
```

default_value argument is dropped

Figure 63—Using **acc_configure()** to set **accDefaultAttr0**

The example shown in Figure 64 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"

PLI_INT32 get_path()
{
    handle    path_handle, mod_handle, src_handle, dst_handle;

    /*initialize the environment for ACC routines*/
    acc_initialize();

    /*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 64— Using **acc_configure()** to set **accEnableArgs**

The example shown in Figure 65 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"

PLI_INT32 set_path_delays()
{
    handle    mod_handle;
    handle    path_handle;
    double    rise_delay, fall_delay, max_delay;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 65—Using **acc_configure()** to set **accPathDelayCount**

The example shown in Figure 66 shows how **accToHiZDelay** is used to direct **acc_replace_delays()** to derive the turn-off delay for a Z-state primitive automatically as the smaller of its rise and fall delays.

```
#include "acc_user.h"

PLI_INT32 set_buf_delays()
{
    handle    primitive_handle;
    handle    path_handle;
    double    added_rise, added_fall;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 66— Using acc_configure() to set accToHiZDelay

23.7 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>		
Returns:	Type	Description	
	PLI_INT32	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 67 uses **acc_count()** to count the number of nets in a module.

```
,
#include "acc_user.h"
PLI_INT32 count_nets()
{
    handle    module_handle;
    PLI_INT32 number_of_nets;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 67—Using acc_count()

23.8 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
	PLI_INT32	Number of command-line arguments
Arguments:	Type	Name
	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 68 uses **acc_fetch_argc()** to determine the number of invocation arguments used.

```
#include "acc_user.h"
#include <string.h> /* string.h is implementation dependent */

PLI_BYTE8* my_scan_plusargs(str)
    PLI_BYTE8 *str;
{
    PLI_INT32    i;
    int          length = strlen(str);
    PLI_BYTE8    *curStr;
    PLI_BYTE8    **argv = acc_fetch_argv();

    for (i = acc_fetch_argc()-1; i>0; i--)
    {
        curStr = argv[i];
        if ((curStr[0] == '+') && (!strcmp(curStr+1, str, length)))
        {
            PLI_BYTE8 *retVal;

            length = strlen(&curStr[length]) + 1;
            retVal = (PLI_BYTE8 *)malloc(sizeof(PLI_BYTE8) * length);
            strcpy(retVal, &curStr[length]);
            return(retVal);
        }
    }
    return(null);
}
```

Figure 68—Using acc_fetch_argc()

23.9 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:	PLI_BYTE8 **	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 format of the argv array is that each pointer in the array shall point to a NULL terminated character array which contains the string located on the tool's invocation command line. There shall be argc entries in the argv array. The value in entry zero shall be the tool's name.

The argument following a -f argument shall contain a pointer to a NULL terminated array of pointers to characters. This new array shall contain the parsed contents of the file. The value in entry zero shall contain the name of the file. The remaining entries shall contain pointers to NULL terminated character arrays containing the different options in the file. The last entry in this array shall be a NULL. If one of the options is a -f then the next pointer shall behave the same as described above.

The example shown in Figure 69 uses **acc_fetch_argv()** to retrieve the invocation arguments used.

```
#include "acc_user.h"
#include <string.h> /* string.h is implementation dependent */

PLI_BYTE8* my_scan_plusargs(str)
    PLI_BYTE8 *str;
{
    PLI_INT32    i;
    int          length = strlen(str);
    PLI_BYTE8    *curStr;
    PLI_BYTE8 **argv = acc_fetch_argv();

    for (i = acc_fetch_argc()-1; i>0; i--)
    {
        curStr = argv[i];
        if ((curStr[0] == '+') && (!strncmp(curStr+1, str, length)))
        {
            PLI_BYTE8 *retVal;

            length = strlen(&curStr[length]) + 1;
            retVal = (PLI_BYTE8 *)malloc(sizeof(PLI_BYTE8) * length);
            strcpy(retVal, &curStr[length]);
            return(retVal);
        }
    }
    return(null);
}
```

Figure 69—Using acc_fetch_argv()

23.10 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:	acc_fetch_attribute(object_handle, attribute_string, default_value)		
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 PLI_BYTE8 *	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 acc_fetch_attribute_int() to get an attribute value as an integer 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()** 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 135. 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 135—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 136 show how to name module paths using different delimiter strings.

Table 136—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

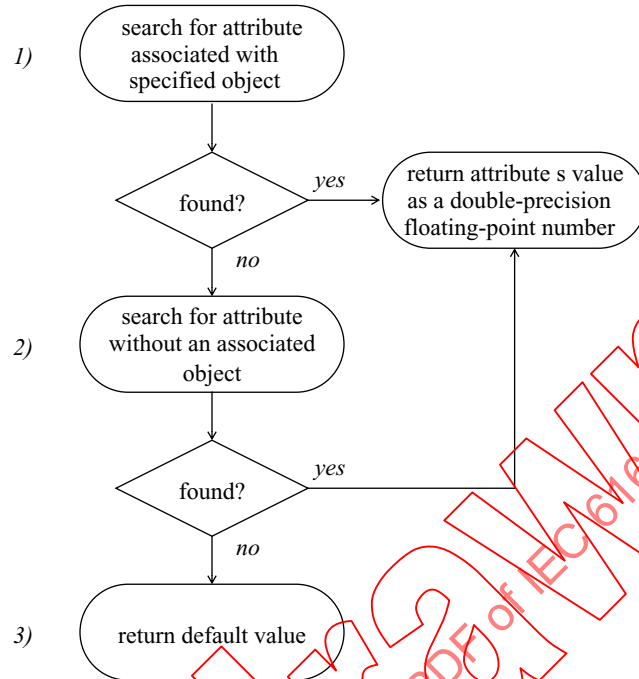
The following example shows an attribute name for a particular module path object:

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 137.

Table 137—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 70 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"
PLI_INT32 display_load_capacitance()
{
    handle    module_handle, port_handle, net_handle;
    double    cap_val;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 = %1f\n",
                    acc_fetch_index(port_handle), cap_val);
    }
    acc_close();
}
```

Figure 70—Using acc_fetch_attribute()

23.11 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:	acc_fetch_attribute_int(object_handle, attribute_string, default_value)		
Returns:	Type	Description	
	PLI_INT32	Value of the parameter or specparam	
Arguments:	Type	Name	Description
	handle	object_handle	Handle of a named object
	quoted string or PLI_BYTE8 *	attribute_string	Literal string or character string pointer with the <i>attribute</i> portion of the parameter or specparam declaration
	Optional	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 23.10 for a description of attribute naming and how attribute values are fetched.

23.12 acc_fetch_attribute_str()

acc_fetch_attribute_str()			
Synopsis:	Get the 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)		
		Type	Description
Returns:	PLI_BYTE8 *	Value of the parameter or specparam	
		Type	Name
Arguments:	handle	object_handle	Handle of a named object
	quoted string or PLI_BYTE8 *	attribute_string	Literal string or character string pointer with the <i>attribute</i> portion of the parameter or specparam declaration
Optional	quoted string or PLI_BYTE8 *	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 22.9 for explanation of strings in ACC routines.

Any parameter or specparam can be an attribute. Refer to 23.10 for a description of attribute naming and how attribute values are fetched.

23.13 acc_fetch_defname()

acc_fetch_defname()		
Synopsis:	Get the definition name of a module instance or primitive instance.	
Syntax:	acc_fetch_defname(object_handle)	
	Type	Description
Returns:	PLI_BYTE8 *	Pointer to a character string containing the definition name
	Type	Name
Arguments:	handle	object_handle
		Description
		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 22.9 for explanation of strings in ACC routines.

The example shown in Figure 71 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"

void 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 71—Using acc_fetch_defname()

23.14 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	
	PLI_INT32	A predefined integer constant representing the delay mode of the module instance: accDelayModeNone accDelayModeZero accDelayModeUnit accDelayModePath accDelayModeDistrib accDelayModeMTM	
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 138.

Table 138—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 72 uses **acc_fetch_delay_mode()** to retrieve the delay mode of all children of a specified module.

```

#include "acc_user.h"
PLI_INT32 display_delay_mode()
{
    handle    mod, child;

    /*reset environment for ACC routines*/
    acc_initialize();

    /*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 72—Using acc_fetch_delay_mode()

23.15 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	acc_fetch_delays(object_handle, rise_delay, fall_delay, z_delay)		
Module paths Intermodule paths Ports or port bits	acc_fetch_delays(object_handle, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12)		
Timing checks	acc_fetch_delays(object_check_handle, limit)		
Type		Description	
Returns:	PLI_INT32	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, bit of a module input port, or intermodule path
	double *	rise_delay fall_delay	Rise and fall delay for 2-state primitive or 3-state primitive
Conditional	double *	z_delay	Turn-off (to Z) transition delay for 3-state primitives
	double *	d1	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 1 : delay for all transitions If accPathDelayCount is set to 2 or 3 : rise transition delay If accPathDelayCount is set to 6 or 12 : 0->1 transition delay
Conditional	double *	d2	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 2 or 3 : fall transition delay If accPathDelayCount is set to 6 or 12 : 1->0 transition delay
Conditional	double *	d3	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 3 : turn-off transition delay If accPathDelayCount is set to 6 or 12 : 0->Z transition delay
Conditional	double *	d4 d5 d6	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 6 or 12 : d4 is Z->1 transition delay d5 is 1->Z transition delay d6 is Z->0 transition delay
Conditional	double *	d7 d8 d9 d10 d11 d12	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 12 : d7 is 0->X transition delay d8 is X->1 transition delay d9 is 1->X transition delay d10 is X->0 transition delay d11 is X->Z transition delay d12 is Z->X transition delay
	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_fetch_delays(object_handle, array_ptr),		
Type		Description	
Returns:	PLI_INT32	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 Section 22.8)

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 section 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 section shall apply.

The number of 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 22.8 for a description of how the number of delay values is 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 73 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"

void display_path_delays()
{
    handle    mod_handle;
    handle    path_handle;
    double    rise_delay, fall_delay, toz_delay;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 73—Using acc_fetch_delays() in single delay mode

The example shown in Figure 74 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.

```

#include "acc_user.h"
void 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 wire
path that represents the
connection from an output
(or inout) port to an input
(or inout) port

acc_fetch_delays places the
following values in delay_array:

delay_array[0] =	} min:typ:max rise delay
delay_array[1] =	
delay_array[2] =	
delay_array[3] =	} min:typ:max fall delay
delay_array[4] =	
delay_array[5] =	
delay_array[6] =	} min:typ:max turn-off delay
delay_array[7] =	
delay_array[8] =	

Figure 74—Using acc_fetch_delays() in min:typ:max delay mode

23.16 acc_fetch_direction()

acc_fetch_direction()			
Synopsis:	Get the direction of a port or terminal.		
Syntax:	acc_fetch_direction(object_handle)		
Returns:	Type	Description	
	PLI_INT32	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 139.

Table 139—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 75 presents a C language application that uses **acc_fetch_direction()** to determine whether or not a port is an input.

```
#include "acc_user.h"

int is_port_input(port_handle)
handle port_handle;
{
    PLI_INT32 direction;

    direction = acc_fetch_direction(port_handle);
    if (direction == accInput || direction == accInout)
        return(true);
    else
        return(false);
}
```

Figure 75—Using acc_fetch_direction()

23.17 acc_fetch_edge()

acc_fetch_edge()			
Synopsis:	Get the edge specifier of a module path or timing check terminal.		
Syntax:	acc_fetch_edge(pathio_handle)		
Returns:	Type	Description	
	PLI_INT32	A predefined integer constant representing the edge specifier of a path input or output terminal: accNoedge accEdge01 accEdge1 accPosedge accEdge10 accEdge1x accNegedge accEdge0x accEdge0	
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 or timing check terminal.

Table 140 lists the predefined edge specifiers as they are specified in `acc_user.h`.

Table 140—Edge specifiers constants

Edge type	Defined constant	Binary value
None	accNoedge	0
Positive edge (0→1, 0→x, x→1)	accPosedge	00001101
Negative edge (1→0, 1→x, x→0)	accNegedge	00110010
0→1 edge	accEdge01	00000001
1→0 edge	accEdge10	00000010
0→x edge	accEdge0x	00000100
x→1 edge	accEdge1	00001000
1→x edge	accEdge1x	00010000
x→0 edge	accEdge0	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 76 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".

```
PLI_BYTE8 *acc_fetch_edge_str(pathio)
handle pathio;
{
    PLI_INT32 edge = acc_fetch_edge(pathio);
    static PLI_BYTE8 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 76—Using acc_fetch_edge()

23.18 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
	PLI_BYTE8 *	Character pointer to a string containing the full hierarchical name of the object
Arguments:	Type	Name
	handle	object_handle
Description		
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 84, 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**.

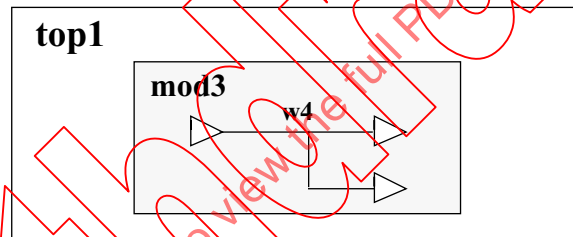


Figure 77—A design hierarchy; the fullname of net w4 is “top1.mod3.w4”

Table 141 lists the objects in a Verilog HDL description for which **acc_fetch_fullname()** shall return a name.

Table 141—Named objects supported by acc_fetch_fullname()

Modules	Integer, time and real variables
Module ports	Named events
Module paths	Parameters
Data paths	Specparams
Primitives	Named blocks
Nets	Verilog HDL tasks
Regs or Variables	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 142—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 22.9).

In the example shown in Figure 78, 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"

PLI_INT32 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 78—Using **acc_fetch_fullname()**

23.19 acc_fetch_fulltype()

acc_fetch_fulltype()			
Synopsis:	Get the fulltype of an object.		
Syntax:	acc_fetch_fulltype(object_handle)		
Returns:	Type	Description	
	PLI_INT32	A predefined integer constant from the list shown in 22.6	
Arguments:	Type	Name	Description
	handle	object_handle	Handle of the object
Related routines:	Use acc_fetch_type() to get the general type classification of an object		
	Use acc_fetch_type_str() to get the fulltype as a 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 113 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 143 illustrate the difference between the type of an object and the fulltype of the same object for selected objects.

Table 143—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	accFchk	accSetup
An and gate primitive	accPrimitive	accAndGate
A sequential UDP	accPrimitive	accSeqPrim

The example shown in Figure 79 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"

PLI_INT32 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");
        }
    }
}
```

Figure 79—Using `acc_fetch_fulltype()` to display the fulltypes of timing checks

The example shown in Figure 80 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"

PLI_INT32 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 80—Using **acc_fetch_fulltype() to display the fulltypes of primitives**

23.20 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	
	PLI_INT32	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 144 shows how indices shall be derived.

Table 144—Deriving indices

For	Indices are
<i>Terminals:</i> nand g1(out, in1, in2);	0 for terminal out 1 for terminal in1 2 for terminal in2
<i>Implicit ports:</i> module A(q, a, b);	0 for port q 1 for port a 2 for port b
<i>Explicit ports:</i> 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 81 presents a C language application that uses **acc_fetch_index()** to find and display the input ports of a module.

```
#include "acc_user.h"

PLI_INT32 display_inputs(module_handle)
handle module_handle;
{
    handle    port_handle;
    PLI_INT32 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 81 -- Using acc_fetch_index()

23.21 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)		
	Type	Description	
Returns:	PLI_INT32	1 if successful; 0 if an error is encountered	
	Type	Name	Description
Arguments:	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 82.

```
typedef struct t_location
{
    PLI_INT32 line_no;
    PLI_BYTE8 *filename;
} s_location, *p_location;
```

Figure 82—s_location data structure

filename field is a character pointer.

line_no field is a nonzero positive integer.

Table 141 lists the objects that shall be supported by **acc_fetch_location()**.

Table 145—Objects supported by acc_fetch_location()

Object type	Location returned
Modules	Module instantiation line
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
Reg variables	Definition line
Integer, time and real variables	Definition line

Table 145—Objects supported by `acc_fetch_location()` (continued)

Object type	Location returned
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 22.9 for an explanation of strings in ACC routines.

The example shown in Figure 83 uses **`acc_fetch_location()`** to print the file name and line number for an object.

```

PLI_INT32 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 83—Using `acc_fetch_location()`

23.22 acc_fetch_name()

acc_fetch_name()		
Synopsis:	Get the instance name of any named object or module path.	
Syntax:	acc_fetch_name(object_handle)	
Returns:	Type	Description
	PLI_BYTE8 *	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 acc_fetch_fullname() to get the full hierarchical name of the object Use acc_fetch_defname() to get the definition name of the object Use acc_configure(accPathDelimStr...) 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 84. In this example, the name of the net is **w4**.

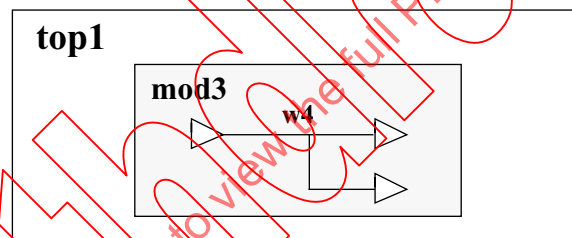


Figure 84—A design hierarchy; the name of net w4 is “w4”

The return value for this routine is placed in the ACC internal string buffer. See 22.9 for an explanation of strings in ACC routines.

Table 141 lists the objects in a Verilog HDL description for which **acc_fetch_name()** shall return a name.

Table 146—Named objects supported by acc_fetch_name()

Modules	Integer, time and real variables
Module ports	Named events
Module paths	Parameters
Data paths	Specparams
Primitives	Named blocks
Nets	Verilog HDL tasks
Regs or Variables	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 147 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 147—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 following example uses **acc_fetch_name()** to display the names of top-level modules.

```
#include "acc_user.h"
PLI_INT32 show_top_mods()
{
    handle    module_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 85—Using *acc_fetch_name()*

23.23 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	
	PLI_INT32	A predefined integer constant representing the data type of a parameter: accIntParam accIntegerParam accRealParam accStringParam	
Arguments:	Type	Name	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 86 uses **acc_fetch_paramtype()** to display the values of all parameters within a module.

```
#include "acc_user.h"
PLI_INT32 print_parameter_values()
{
    handle module_handle, param_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();

    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 86—Using acc_fetch_paramtype()

23.24 acc_fetch_paramval()

acc_fetch_paramval()			
Synopsis:	Get the value of a parameter or specparam.		
Syntax:	acc_fetch_paramval(parameter_handle)		
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 acc_fetch_paramtype() to retrieve the data type of a parameter Use acc_next_parameter() to scan all parameters within a module Use acc_next_specparam() 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 87.

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 148. 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 22.9 for explanation of strings in ACC routines.

Table 148—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): int_val= (int) acc_fetch_paramval(...);
String	Cast the return value to a character pointer using the C language cast operators (char*)(int): str_ptr= (char*)(int) acc_fetch_paramval(...);

The example shown in Figure 87 presents a C language application, `print_parameter_values`, that uses `acc_fetch_paramtype()` to display the values of all parameters within a module.

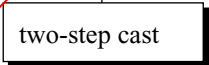
```
#include "acc_user.h"

PLI_INT32 print_parameter_values()
{
    handle    module_handle;
    handle    param_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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();
}
```



The diagram illustrates a 'two-step cast' where the return value of `acc_fetch_paramval` is first cast to `(int)` and then to `(char*)` to be used in the `io_printf` statement for string parameters.

Figure 87—Using `acc_fetch_paramval()`

23.25 acc_fetch_polarity()

acc_fetch_polarity()			
Synopsis:	Get the polarity of a path.		
Syntax:	acc_fetch_polarity(path_handle)		
Returns:	Type	Description	
	PLI_INT32	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 149.

Table 149—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 88 takes a path argument and returns the string corresponding to its polarity.

```

PLI_BYTE8 *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 88—Using acc_fetch_polarity()

23.26 acc_fetch_precision()

acc_fetch_precision()		
Synopsis:	Get the smallest time precision argument specified in all `timescale compiler directives in a given design.	
Syntax:	acc_fetch_precision()	
Returns:	Type	Description
	PLI_INT32	An integer value that represents a time precision
Arguments:	Type	Name
	None	
Related routines:	Use acc_fetch_timescale_info() 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 150.

Table 150— 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 s
-5	10 s
-6	1 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).

23.27 acc_fetch_pulsere()

acc_fetch_pulsere()			
Synopsis:	Get current pulse handling <i>reject_limit</i> and <i>e_limit</i> for a module path, intermodule path or module input port.		
Syntax:	acc_fetch_pulsere(object, 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:	PLI_INT32	1 if successful; 0 if an error is encountered	
Arguments:	Type	Name	Description
	handle	object	Handle of module path, intermodule path or module input port
	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 Use acc_configure() to set accPathDelayCount		

The ACC routine **acc_fetch_pulsere()** shall obtain the current values controlling how pulses are propagated through a module path, intermodule path or module input port.

A *pulse* is defined as two transitions that occur in a shorter period of time than the 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
- 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
- A pulse that is greater than or equal to the *e_limit* shall propagate
- Table 151 illustrates the relationship between the *reject_limit* and the *e_limit*.

Table 151—Pulse control example

When	The pulse 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 152.

Table 152—How the `accPathDelayCount` affects `acc_fetch_pulsere()`

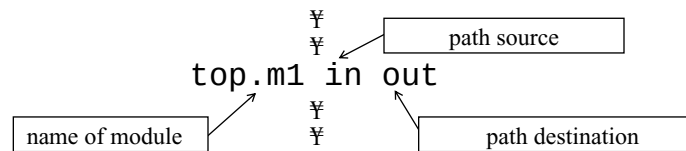
When <code>accPathDelayCount</code> is	<code>acc_fetch_pulsere()</code> shall retrieve
1	One pair of <code>reject_limit</code> and <code>e_limit</code> values: one pair for all transitions, <code>r1</code> and <code>e1</code>
2	Two pairs of <code>reject_limit</code> and <code>e_limit</code> values: one 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_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 object handle.

The example shown in Figure 89 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
PLI_INT32 get_pulsevals()
{
    FILE      *infile;
    PLI_BYTE8 mod_name[NAME_SIZE];
    PLI_BYTE8 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 89—Using acc_fetch_pulsere()

23.28 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:	PLI_INT32	Zero if successful; nonzero upon error	
Arguments:	Type	Name	Description
	handle	vector_handle	Handle to a vector net or reg
	PLI_INT32 *	msb	Pointer to an integer variable to hold the most significant bit of vector_handle
	PLI_INT32 *	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 90 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>].

```
PLI_INT32 display_vector_nets()
{
    handle    mod = acc_handle_tfarg(1);
    handle    net;
    PLI_INT32 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 90—Using acc_fetch_range()

23.29 acc_fetch_size()

acc_fetch_size()			
Synopsis:	Get the bit size of a net, reg, integer, time, real or port.		
Syntax:	acc_fetch_size(object_handle)		
Returns:	Type	Description	
	PLI_INT32	Number of bits in the net, reg, integer, time, real or port	
Arguments:	Type	Name	Description
	handle	object_handle	Handle to a net, reg, integer, time, real or port, or a bit-select or part select thereof

The ACC routine **acc_fetch_size()** shall return the number of bits of a net, reg, integer, time, real or port.

The example shown in Figure 91 uses **acc_fetch_size()** to display the size of a vector net.

```
#include "acc_user.h"

PLI_INT32 display_vector_size()
{
    handle    net_handle;
    PLI_INT32 size_in_bits;

    /* reset environment for ACC routines */
    acc_initialize();

    /*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 91 —Using acc_fetch_size()

23.30 **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:	acc_fetch_tfarg (argument_number) acc_fetch_itfarg (argument_number, tfinst)		
Type		Description	
Returns:	double	The value of the task/function argument, returned as a double-precision number	
Type		Name	Description
Arguments:	PLI_INT32	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 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, regs, 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 routine **acc_fetch_tfarg()** returns values as type double. The routines **acc_fetch_tfarg_int()** and **acc_fetch_tfarg_str()** return values as integers or string pointers, respectively. The value returned by **acc_fetch_tfarg()** can also be converted to integers or character pointers using the C language *cast* mechanism, as shown in Table 153. 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 22.9 for explanation of strings in ACC routines.

Table 153—Casting acc_fetch_tfarg() return values

To convert to	Follow these steps
Integer	Cast the return value to the integer data type using the C language cast operator (PLI_INT32): <code>int_val= (PLI_INT32) acc_fetch_tfarg(...);</code>
String	Cast the return value to a character pointer using the C language cast operators (char*)(int)): <code>str_ptr= (char*)(int) acc_fetch_tfarg(...);</code>

The example shown in Figure 92 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 "veriusers.h"

PLI_INT32 display_arg_value()
{
    PLI_INT32    arg_type;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*check type of argument*/
    io_printf("Argument value is ");

    switch(tf_typep(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 92—Using acc_fetch_tfarg(), acc_fetch_tfarg_int(), and acc_fetch_tfarg_str()

23.31 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:	PLI_INT32	The value of the task/function argument, returned as an integer number	
Type		Name	Description
Arguments:	PLI_INT32	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, regs, 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.

If a user-defined task/function argument is a real value, the value is cast to a `PLI_INT32` and returned as an integer. If the task/function argument is a string value, the string is copied into the ACC string buffer and the pointer to the string is cast to the type `PLI_INT32` and returned as an integer.

Refer to Figure 92 for an example of using **acc_fetch_tfarg_int()**.

23.32 `acc_fetch_tfarg_str()`, `acc_fetch_itfarg_str()`

<code>acc_fetch_tfarg_str()</code> , <code>acc_fetch_itfarg_str()</code>			
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:	<code>acc_fetch_tfarg_str(argument_number)</code> <code>acc_fetch_itfarg_str(argument_number, tfinst)</code>		
Type		Description	
Returns:	PLI_BYTE8 *	The value of the task/function argument, returned as a pointer to a character string	
Type		Name	Description
Arguments:	PLI_INT32	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 <code>acc_fetch_tfarg()</code> or <code>acc_fetch_itfarg()</code> to get the task/function argument value as a double Use <code>acc_fetch_tfarg_int()</code> or <code>acc_fetch_itfarg_int()</code> to get the task/function argument value as an integer Use <code>acc_handle_tfinst()</code> 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 of 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 22.9 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, regs, 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.

If a user-defined task/function argument is a value, each 8 bits of the value are converted into its equivalent ASCII character.

Refer to Figure 92 for an example of using **`acc_fetch_tfarg_str()`**.

23.33 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 154.

Table 154—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 82.

```
typedef struct t_timescale_info
{
    PLI_INT16 unit;
    PLI_INT16 precision;
} s_timescale_info, *p_timescale_info;
```

Figure 93—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 155.

Table 155—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
```


23.34 acc_fetch_type()

acc_fetch_type()			
Synopsis:	Get the type of an object.		
Syntax:	acc_fetch_type(object_handle)		
Returns:	Type	Description	
	PLI_INT32	A predefined integer constant from the list shown in 22.6	
Arguments:	Type	Name	Description
	handle	object_handle	Handle of the object
Related routines:	Use acc_fetch_fulltype() to get the full type classification of an object Use acc_fetch_type_str() to get the type as a 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 Table 113 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 143 illustrates the difference between the *type* of an object and the *fulltype* of the same object.

Table 156—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	accFchk	accSetup
An and gate primitive	accPrimitive	accAndGate
A sequential UDP	accPrimitive	accSeqPrim

The example shown in Figure 94 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 23.19).

```
#include "acc_user.h"

PLI_INT32 display_object_type()
{
    handle object_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();

    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 94—Using acc_fetch_type()

23.35 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:	PLI_BYTE8 *	Pointer to a character string		
		Type	Name	Description
Arguments:	PLI_INT32	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 22.9 for explanation of strings in ACC routines.

In the example shown in Figure 95, 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"
PLI_INT32 display_object_type(object)
handle object;
{
    PLI_INT32 type = acc_fetch_type(object);

    io_printf("Object %s is of type %s \n",
        acc_fetch_fullname(object),
        acc_fetch_type_str(type));
}
```

Figure 95—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.

23.36 acc_fetch_value()

acc_fetch_value()			
Synopsis:	Get the logic or strength value of a net, reg, or variable.		
Syntax:	acc_fetch_value(object_handle, format_string, value)		
		Type	Description
Returns:	PLI_BYTE8 *	Pointer to a character string	
		Type	Name
Arguments:	handle	object_handle	Handle of the object
	quoted string or PLI_BYTE8 *	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 in which the value of the object is returned when the format string is %% (should be set to NULL when not used)
Related routines:	Use acc_fetch_size() to determine how many 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, reg, and integer, time and real variables; **acc_fetch_value()** shall return *strength* values for scalar nets and scalar regs 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 157.

Table 157—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 be returned
%d	decimal	
%h	hexadecimal	
%o	octal	
%v	strength	
%%	s_acc_value structure	Value shall be retrieved and placed in a structure variable pointed to by the optional <i>value</i> argument

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, and unknown and high-impedance values shall be obtained. Note that strings are placed in a temporary buffer, and they should be preserved if not used immediately. Refer to 22.9 for details on preserving strings.

The %v format shall return a three character string containing the strength code of a scalar net. Refer to 17.1.1.5 for the strength representations.

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
{
    PLI_INT32 format;
    union
    {
        PLI_BYTE8      *str;
        PLI_INT32      scalar;
        PLI_INT32      integer;
        double          real;
        p_acc_vecval    vector;
    } value;
} s_setval_value, *p_setval_value, s_acc_value, *p_acc_value;
```

Figure 96—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 158.
- The structure variable has to be passed as the third argument to **acc_fetch_value()**.
- The function return value from **acc_fetch_value()** should be ignored.

Table 158—Format constants for the s_acc_value structure

Format constant	acc_fetch_value() shall return the value to the s_acc_value union field	Description
accBinStrVal	str	value is retrieved in the same format as %b
accOctStrVal	str	value is retrieved in the same format as %o
accDecStrVal	str	value is retrieved in the same format as %d
accHexStrVal	str	value is retrieved in the same format as %h
accStringVal	str	value is converted to a string, see Section 2.6 for a description of Verilog strings
accScalarVal	scalar	value is retrieved as one of the constants: acc0 , acc1 , accZ or accX
accIntVal	integer	value is retrieved as a C integer
accRealVal	real	value is retrieved as a C double
accVectorVal	vector	value is represented as aval/bval pairs stored in an array of s_acc_vecval structures

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 96. 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 159.

```
typedef struct t_acc_vecval
{
    PLI_INT32 aval;
    PLI_INT32 bval;
} s_acc_vecval, *p_acc_vecval;
```

Figure 97—s_acc_vecval structure

Table 159—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 99 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 98 uses **acc_fetch_value()** to retrieve the logic values of all nets in a module as strings.

```
#include "acc_user.h"
PLI_INT32 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 98—Using `acc_fetch_value()` to retrieve the logic values as strings

The example in Figure 99 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"

PLI_INT32 my_fetch_value()
{
    handle          reg = acc_handle_tfarg(1);
    PLI_INT32        size = ((acc_fetch_size(reg) - 1) / 32) + 1;
    s_acc_value      value;
    int              index1, min_size;
    static PLI_BYTE8 table[4] = {'0', '1', 'z', 'x'};
    static PLI_BYTE8 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--)
    {
        int index2;
        PLI_INT32 abits = value.value.vector[index1].aval;
        PLI_INT32 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 99—Using acc_fetch_value() to retrieve values into a data structure

23.37 acc_free()

acc_free()			
Synopsis:	Frees memory allocated by acc_collect() .		
Syntax:	acc_free (handle_array_pointer)		
Returns:	Type	Description	
	void	No return	
Arguments:	Type	Name	Description
	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 100 uses **acc_free()** to deallocate memory allocated by **acc_collect()** to collect handles to all nets in a module.

```
#include "acc_user.h"

PLI_INT32 display_nets()
{
    handle      *list_of_nets, module_handle;
    PLI_INT32    net_count, i;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 100—Using acc_free()

23.38 acc_handle_by_name()

acc_handle_by_name()			
Synopsis:	Get the handle to any named object based on its name and scope.		
Syntax:	acc_handle_by_name(object_name, scope_handle)		
Returns:	Type	Description	
	handle	A handle to the specified object	
Arguments:	Type	Name	Description
	quoted string or PLI_BYTE8 *	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 acc_handle_object() 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 160.

Table 160—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	Search for the <i>object_name</i> in the module containing the current system task or function

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. Object searching shall conform to rules in 12.4 on hierarchical name referencing.

Table 161 lists the objects in a Verilog HDL description for which **acc_handle_by_name()** shall return a handle.

Table 161—Named objects supported by acc_handle_by_name()

Modules	Parameters
Primitives	Specparams
Nets	Named blocks
Regs	Verilog HDL tasks
Integer, time and real 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 101 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"

PLI_INT32 is_net_in_module(module_handle, net_name)
handle module_handle;
PLI_BYTE8 *net_name;
{
    handle 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 101—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);
```

23.39 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:	acc_handle_calling_mod_m		
Returns:	Type	Description	
	handle	Handle to a module	
Arguments:	Type	Name	Description
	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.

23.40 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 102 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.

```
int is_path_conditional(path)
{
    if (acc_handle_condition(path) )
        return(TRUE);
    else
        return(FALSE);
}

int is_level_sensitive(path)
{
    int 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 102—Using acc_handle_condition()

23.41 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(terminal_handle)		
Returns:	Type	Description	
	handle	Handle of a net	
Arguments:	Type	Name	Description
	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 103 displays the net connected to the output terminal of a gate.

```
#include "acc_user.h"

PLI_INT32 display_driven_net()
{
    handle gate_handle, terminal_handle, net_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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();
}
```

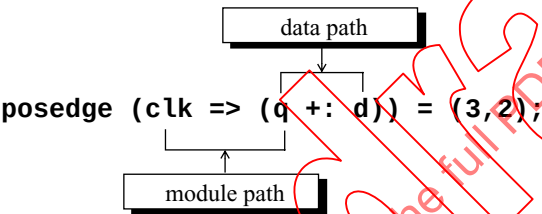
Figure 103—Using acc_handle_conn()

23.42 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 104 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.

```
PLI_INT32 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 104—Using acc_handle_datapath()

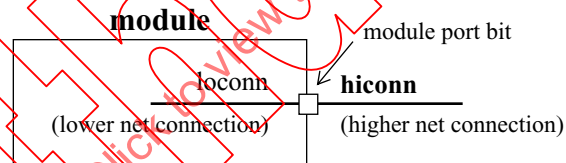
23.43 acc_handle_hiconn()

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)		
Returns:	Type	Description	
	handle	Handle of a net	
Arguments:	Type	Name	Description
	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 105 uses **acc_handle_hiconn()** and **acc_handle_loconn()** to display the higher and lower connections of a module port.

```
PLI_INT32 display_port_info(mod, index)
handle    mod;
PLI_INT32 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 105—Using `acc_handle_hiconn()` and `acc_handle_loconn()`

23.44 acc_handle_interactive_scope()

acc_handle_interactive_scope()			
Synopsis:	Get a handle to the current interactive scope of the 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_fetch_type() or acc_fetch_fulltype() to determine the scope type returned Use acc_set_interactive_scope() 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

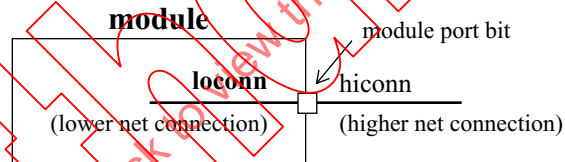
23.45 acc_handle_loconn()

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)		
Returns:	Type	Description	
	handle	Handle of a net	
Arguments:	Type	Name	Description
	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 23.43 for an example of using **acc_handle_loconn()**.

23.46 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 PLI_BYTE8 *	source_name	Literal string or character string pointer with the name of a net connected to a module path source
	quoted string or PLI_BYTE8 *	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 if one can be found. If a module path cannot be found the return value shall be **null**, the **acc_error_flag** will not be set. If any of the input args are improper a **null** shall be returned and the **acc_error_flag** will be set.

Table 162—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

```
module path
(in *> out) = 1.8;
posedge (clk => (q +: d) ) = (3,2);
module path
```

Diagram illustrating the path resolution process:

- A box labeled **module name** points to the text **top.mod1 in**.
- A box labeled **path source** points to the text **out**.
- A box labeled **path destination** points to the text **out**.
- Four vertical arrows connect **path source** and **path destination**.
- A red 'X' mark is placed over the **path destination** box.

```
#include <stdio.h>
#include "acc_user.h"

#define NAME_SIZE 256

PLI_INT32 get_paths()
{
    FILE      *infile;
    PLI_BYTE8 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();

    /* 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 (path_handle)
            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();
}
```

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

23.47 acc_handle_notifier()

acc_handle_notifier()			
Synopsis:	Get the notifier reg 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 reg associated with a timing check.

The example shown in Figure 117 uses **acc_handle_notifier()** to display the name of a notifier associated with a timing check.

23.48 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 PLI_BYTE8 *	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 163.

Table 163—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 Section 12.6

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 141 lists the objects in a Verilog HDL description for which **acc_handle_object()** shall return a handle.

Table 164—Named objects

Modules	Named events
Module ports	Parameters
Data paths	Specparams
Primitives	Named blocks
Nets	Verilog HDL tasks
Regs	Verilog HDL functions
Integer, time and real variables	

23.49 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, port 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 a port bit shall be the port that contains the bit.
- 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 108 uses **acc_handle_parent()** to determine which terminals of a primitive drive a net.

```
#include "acc_user.h"

PLI_INT32 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 108—Using acc_handle_parent()

23.50 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)		
Returns:		Type	Description
		handle	Handle of the intermodule path
Arguments:	Type	Name	Description
	handle	port_output_handle	Handle to one of the following: ¥ 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: ¥ 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 handle 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 109 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"
PLI_INT32 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 109—Using acc_handle_path()

23.51 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 110 uses **acc_handle_pathin()** to find the net connected to the input of a path.

```
#include "acc_user.h"  
PLI_INT32 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 110—Using acc_handle_pathin()

23.52 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 111 uses **acc_handle_pathout()** to find the net connected to the output of a path.

```

#include "acc_user.h"

PLI_INT32 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 111—Using acc_handle_pathout()

23.53 acc_handle_port()

acc_handle_port()			
Synopsis:	Get a handle for a module port, based on the position of the port.		
Syntax:	acc_handle_port(module_handle, port_index)		
Returns:	Type	Description	
	handle	Handle to a module port	
Arguments:	Type	Name	Description
	handle	module_handle	Handle of a module
	PLI_INT32	port_index	An integer index of the desired port
Related routines:	Use acc_next_port() 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 165 shows how port indices are derived.

Table 165—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 112 uses **acc_handle_port()** to identify whether a particular module port is an output.

```
#include "acc_user.h"

int is_port_output(module_handle,port_index)
handle      module_handle;
PLI_INT32   port_index;
{
    handle      port_handle;
    PLI_INT32   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 112—Using acc_handle_port()

23.54 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 113 uses **acc_handle_scope()** to display the scope that contains an object.

```
PLI_INT32 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 113—Using acc_handle_scope()

23.55 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 114 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"

PLI_INT32 display_simulated_nets()
{
    handle    mod_handle;
    handle    simulated_net_handle;
    handle    net_handle;

    /*reset environment for ACC routines*/
    acc_initialize();

    /*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 114—Using acc_handle_simulated_net()

23.56 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>		
Returns:		Type	Description
		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: accHold accSetup accNochange accSkew accPeriod accWidth accRecovery
	quoted string or PLI_BYTE8 *	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: accNegedge accNoedge accPosedge or a list of the following constants, separated by +: accEdge01 accEdge0x accEdgex1 or a list of the following constants, separated by +: accEdge10 accEdge1x accEdgex0
	Conditional quoted string or PLI_BYTE8 *	second_arg_conn_name	Name of the net connected to second timing check argument (depends on type of timing check)
	Conditional integer constant	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 handle	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 handle	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 166 shows how the number of arguments for **acc_handle_tchk()** is determined.

Table 166—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 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 167. The routine shall treat transitions to or from a logic **Z** as transitions to or from a logic **X**.

Table 167—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 168.

Table 168—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 115 shows a call to **acc_handle_tchk()** that accesses a \$width timing check containing edge-control specifiers.

This ACC routine call	Accesses this timing check
acc_handle_tchk(cell_handle, accWidth, "clk", accEdge10+accEdgex0);	\$width(edge[10,x0]clk, limit);
edge sum models edge-control specifier	

Figure 115—Edge sums model edge-control specifiers

The example shown in Figure 116 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"

PLI_INT32 get_ps_tchks()
{
    handle    module_handle, port_handle, net_handle, cell_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 116—Using acc_handle_tchk()

23.57 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 117 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"

PLI_INT32 show_check_nets()
{
    handle      module_handle, cell_handle;
    handle      tchk_handle, tchkarg1_handle, tchkarg2_handle,
notifier_handle;
    PLI_INT32   tchk_type, counter;

    /* initialize environment for ACC routines */
    acc_initialize();

    /* 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);
                io_printf(" data net is %s\n",
                    acc_fetch_name(acc_handle_conn(tchkarg1_handle) ) );
                tchkarg2_handle = acc_handle_tchkarg2(tchk_handle);
                io_printf(" reference net is %s\n",
                    acc_fetch_name(acc_handle_conn(tchkarg2_handle) ) );
                notifier_handle = acc_handle_notifier(tchk_handle);
                if (notifier_handle != null)
                    io_printf(" notifier reg is %s\n",
                        acc_fetch_name(acc_handle_conn(notifier_handle) ) );
                else
                    io_printf(" no notifier reg\n");
            }
        }
    }
    acc_close();
}

```

**Figure 117—Using acc_handle_tchkarg1(),
acc_handle_tchkarg2() and acc_handle_notifier()**

23.58 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)		
Returns:	Type	Description	
	handle	Handle to 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_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 117 for an example of using **acc_handle_tchkarg2()**.

23.59 acc_handle_terminal()

acc_handle_terminal()			
Synopsis:	Get a handle for a primitive terminal based on the position of the primitive terminal.		
Syntax:	acc_handle_terminal(primitive_handle, terminal_index)		
Returns:	Type	Description	
	handle	Handle of a primitive terminal	
Arguments:	Type	Name	Description
	handle	primitive_handle	Handle of a primitive
	PLI_INT32	terminal_index	Integer index of the desired terminal
Related routines	Use acc_handle_conn() 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 169 shows how terminal indices are derived.

Table 169—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 118 uses **acc_handle_terminal()** to identify the name of a net connected to a primitive terminal.

```
#include "acc_user.h"

PLI_INT32 print_terminal_net(gate_handle, term_index)
handle gate_handle;
PLI_INT32 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 118—Using acc_handle_terminal()

23.60 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:	acc_handle_tfarg(argument_number) acc_handle_itfarg(argument_number, instance_handle)		
Returns:	Type	Description	
	handle	Handle to an object	
Arguments:	Type	Name	Description
	PLI_INT32	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:

- A module instance
- A primitive instance
- A net, reg, integer variable, time variable, or real variable
- A legal bit select of a net, reg, integer variable or time variable

Table 170—How acc_handle_tfarg() operates

When	acc_handle_tfarg() shall
The system task or function argument is an unquoted Verilog HDL identifier	Return a handle to the object
The system task or function argument is a quoted string name of any object	Function similar to acc_handle_object() by searching for an object matching the string and, if found, returning a handle to the object. The object shall be searched for in the following order: a) The current PLI scope [as set by acc_set_scope()] b) The scope of the system task/function

The example shown in Figure 119 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"

PLI_INT32 new_timing()
{
    handle    gate_handle;
    double    new_rise, new_fall;

    /*initialize and configure ACC routines*/
    acc_initialize();
    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 119—Using **acc_handle_tfarg()**

23.61 acc_handle_tfinst()

acc_handle_tfinst()			
Synopsis:	Get a handle to the current user-defined system task or function call.		
Syntax:	acc_handle_tfinst()		
Returns:	Type	Description	
	handle	Handle of a user-defined system task or function	
Arguments:	Type	Name	Description
	None		
Related routines:	Use acc_fetch_type() or acc_fetch_fulltype() 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.

23.62 acc_initialize()

acc_initialize()		
Synopsis:	Initializes the environment for ACC routines.	
Syntax:	acc_initialize()	
Returns:	Type	Description
	PLI_INT32	1 if successful; 0 if an error is encountered
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 120 uses **acc_initialize()** to initialize the environment for ACC routines.

```
#include "acc_user.h"
PLI_INT32 append_mintypmax_delays()
{
    handle    prim;
    double    delay_array[9];
    int       i;

    /* initialize environment for ACC routines */
    acc_initialize();

    /* configure ACC routine environment */
    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);

    /* close the environment for ACC routines */
    acc_close();
}
```

Figure 120—Using acc_initialize()

23.63 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>		
Returns:	Type	Description	
	handle	Handle of the object found	
Arguments:	Type	Name	Description
	static PLI_INT32 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 <code>null</code>

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 171. 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 PLI_INT32 net_reg_list[3] = {accNet, accRegister, 0};
```

When this array is passed to **acc_next()**, the ACC routine shall return handles to nets and regs 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 171.

Table 171 – 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
	Reg	accRegister
	Time variable	accTimeVar
	Parameter	accParameter
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	accTriage
Parameter <i>fulltypes</i>	Integer parameters	accIntegerParam
	Real parameters	accRealParam
	String parameters	accStringParam

Table 171 — 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 121 uses **`acc_next()`** to find all nets and regs in a module. The application then displays the names of these nets and reg.

```
#include "acc_user.h"

PLI_INT32 display_nets_and_registers()
{
    static PLI_INT32 net_reg_list[3] = {accNet, accRegister, 0};
    handle          mod_handle, obj_handle;

    /*reset environment for ACC routines*/
    acc_initialize();

    /*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) );

    acc_close();
}
```

Figure 121—Using acc_next()

23.64 acc_next_bit()

acc_next_bit()			
Synopsis:	Get handles to bits in a port or expanded vector.		
Syntax:	acc_next_bit(reference_handle, bit_handle)		
Returns:	Type	Description	
	handle	Handle of a port bit, vector bit or path terminal bit	
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a port, expanded vector or path terminal
	handle	bit_handle	Handle of the previous bit found; initially null
Related routines:	Use acc_next_port() to return the next port of a module Use acc_handle_port() to return the handle for a module port Use acc_object_of_type() to determine if a vector is expanded		

The ACC routine **acc_next_bit()** shall obtain handles to the bits of a vector port, an expanded vector, or a path terminal.

An *expanded vector* is a 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(vector_handle, accExpandedVector) )
    while (bit_handle = acc_next_bit(vector_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 122 uses **acc_next_bit()** to display the lower connection of each bit of a port.

```
#include "acc_user.h"
PLI_INT32 display_port_bits(module_handle, port_number)
handle      module_handle;
PLI_INT32   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 122—Using acc_next_bit() with module ports

The example shown in Figure 123 uses **acc_next_bit()** to assign a VCL monitor flag to each bit of a vector net.

```
#include "acc_user.h"
PLI_INT32 monitor_bits()
{
    handle    bit_handle, net_handle, mod_handle;

    /* reset environment for ACC routines */
    acc_initialize();

    /* 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 123—Using acc_next_bit() with a vector net

23.65 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)		
Returns:	Type	Description	
	handle	Handle of a cell module	
Arguments:	Type	Name	Description
	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 124 uses **acc_next_cell()** to list all cell instances at or below a given hierarchy scope.

```
#include "acc_user.h"
PLI_INT32 list_cells()
{
    handle    module_handle, cell_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 124—Using acc_next_cell()

23.66 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 a scalar net or bit select of a vector 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 125 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.

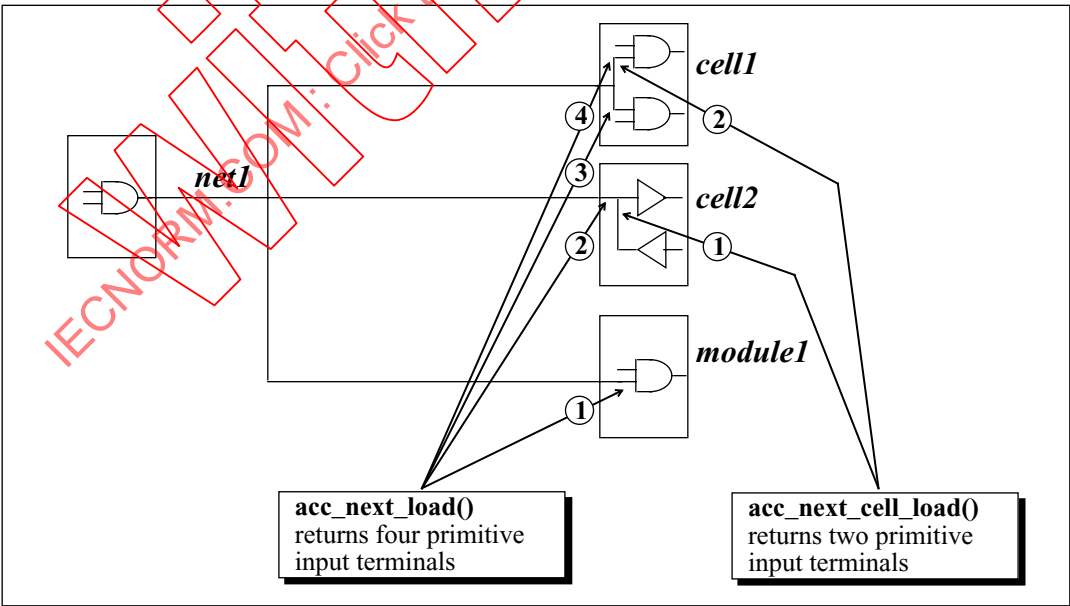


Figure 125— The difference between acc_next_load() and acc_next_cell_load()

The example shown in Figure 126 uses **acc_next_cell_load()** to find all cell loads on a net.

```
#include "acc_user.h"

PLI_INT32 get_cell_loads()
{
    handle    net_handle;
    handle    load_handle, load_net_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 126—Using **acc_next_cell_load()**

23.67 acc_next_child()

acc_next_child()			
Synopsis:	Get handles for children of a module.		
Syntax:	acc_next_child(reference_handle, child_handle)		
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 172.

Table 172—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 127 shows the use of **acc_next_child()** to display the names of all modules instantiated within a module.

```
#include "acc_user.h"
PLI_INT32 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 127—Using acc_next_child()

23.68 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 scalar net or bit select of a vector 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 128 uses **acc_next_driver()** to determine which terminals of a primitive drive a net.

```
#include "acc_user.h"

PLI_INT32 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) );

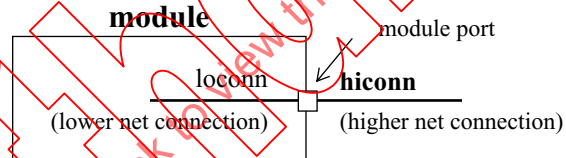
    /*get primitive that owns each terminal that drives the net*/
    driver_handle = null;
    while (driver_handle = acc_next_driver(net_handle, driver_handle) )
    {
        primitive_handle = acc_handle_parent(driver_handle);
        io_printf("    %s\n",
                  acc_fetch_fullname(primitive_handle) );
    }
}
```

Figure 128—Using acc_next_driver()

23.69 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	reference_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 129 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"

PLI_INT32 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) );
}
```

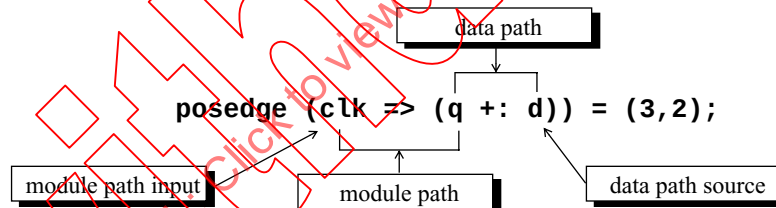
Figure 129—Using `acc_next_hiconn()` and `acc_next_loconn()`

23.70 acc_next_input()

acc_next_input()			
Synopsis:	Get handles to input path terminals of a module path, source terminals of a data path, or the terminals of a timing check.		
Syntax:	acc_next_input (reference_handle, terminal_handle)		
Returns:	Type	Description	
	handle	Handle of a module path terminal, a data path terminal, or a timing check terminal	
Arguments:	Type	Name	Description
	handle	reference_handle	Handle to a module path, data path or timing check
	handle	terminal_handle	Handle of the previous terminal found; initially null
Related routines:	Use acc_handle_conn() to get the net attached to the path terminal Use acc_release_object() to free memory allocated by acc_next_input()		

The ACC routine **acc_next_input()** shall return handles to the input path terminals of a module path, the source terminals of a data path or the timing check terminals of a timing check. 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 130 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.

```

int is_net_on_path_input(net, path)
handle net; /* scalar net or bit-select of vector net */
handle path;
{
    handle pterm_in, pterm_conn, bit;

    /* scan path input terminals */
    pterm_in = null;
    while (pterm_in = acc_next_input(path, pterm_in) )
    {
        /* retrieve net connected to path terminal */
        pterm_conn = acc_handle_conn (pterm_in);

        bit = null;
        if (acc_object_of_type (pterm_conn, accExpandedVector) )
        {
            bit = null;
            while (bit = acc_next_bit (pterm_conn, bit) )
                if (acc_compare_handles (bit, net) )
                    return (true);
        }
        else
            if (acc_compare_handles(bit, net) )
                return (true);
    }
    return (false);
}

```

Figure 130—Using acc_next_input()

23.71 acc_next_load()

acc_next_load()			
Synopsis:	Get handles to primitive terminals driven by a net.		
Syntax:	acc_next_load(reference_handle, load_handle)		
Returns:	Type	Description	
	handle	Handle of a primitive terminal	
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a scalar net or bit select of a vector net
	handle	load_handle	Handle of the previous load found; initially null
Related routines:	Use acc_next_cell_load() 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 131 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.

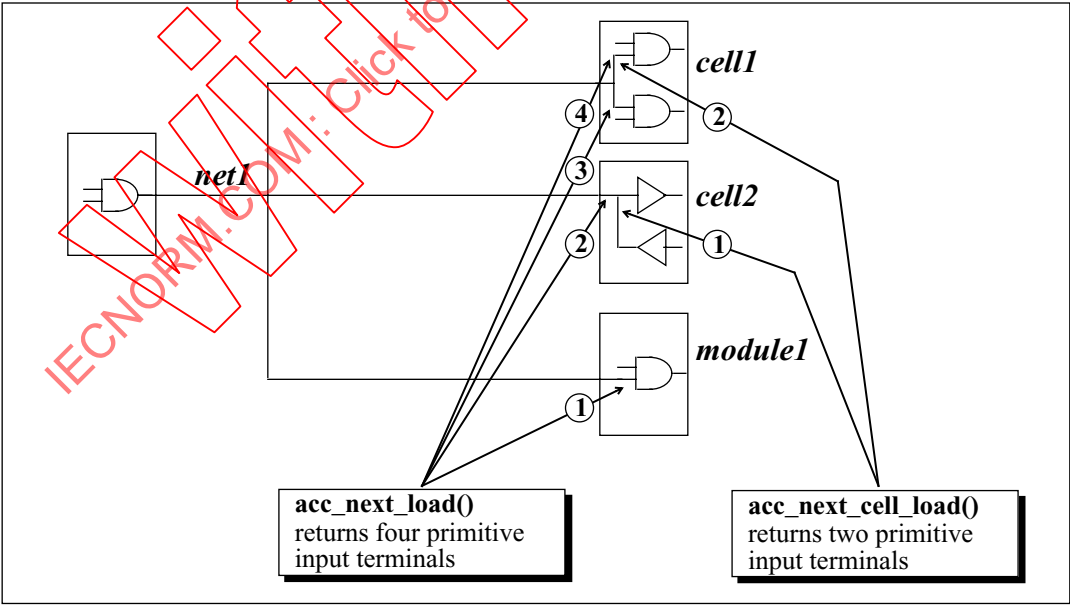


Figure 131 — The difference between acc_next_load() and acc_next_cell_load()

The example shown in Figure 132 uses **acc_next_load()** to find all terminals driven by a net.

```
#include "acc_user.h"

PLI_INT32 get_loads()
{
    handle    net_handle, load_handle, load_net_handle;

    /*initialize the environment for ACC routines*/
    acc_initialize();

    /*get handle for net*/
    net_handle = acc_handle_tfarg(1);
    io_printf("Net %s is driven by:\n", acc_fetch_fullname(net_handle) );

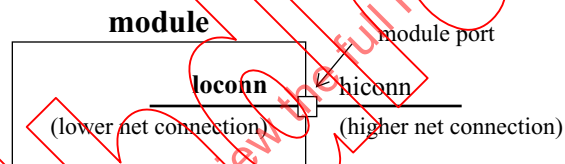
    /*get primitive that owns each terminal driven by the net*/
    load_handle = null;
    while (load_handle = acc_next_load(net_handle, load_handle) )
    {
        load_net_handle = acc_handle_conn(load_handle);
        io_printf("    %s ",
                  acc_fetch_fullname(load_net_handle) );
    }
    acc_close();
}
```

Figure 132—Using acc_next_load()

23.72 acc_next_loconn()

acc_next_loconn()			
Synopsis:	Get handles to hierarchically lower net connections to a port of a module.		
Syntax:	acc_next_loconn(reference_handle, net_handle)		
Returns:	Type	Description	
	handle	Handle of a net	
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a port
	handle	net_handle	Handle of the previous net found; initially null
Related routines:	Use acc_handle_loconn() to get a handle to hierarchically lower connection of a specific port bit Use acc_next_hiconn() 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:



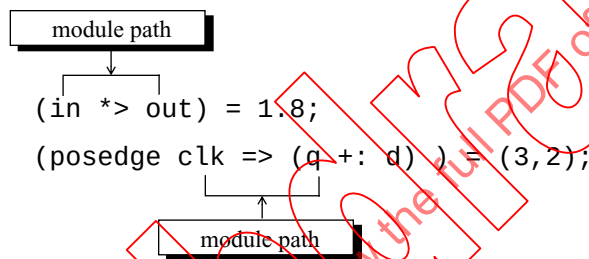
When the reference handle passed to **acc_next_loconn()** is a vector port, the routine shall return the loconn nets bit-by-bit, starting with the msb (leftmost bit) and ending with the lsb (rightmost bit).

Refer to Figure 129 for an example of using **acc_next_loconn()**.

23.73 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 133 uses **acc_next_modpath()** to list the nets connected to all module paths in a module.

```

#include "acc_user.h"
PLI_INT32 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 133—Using acc_next_modpath()

23.74 acc_next_net()

acc_next_net()			
Synopsis:	Get handles to nets in a module.		
Syntax:	acc_next_net(reference_handle, net_handle)		
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 acc_object_of_type() to determine if a net is scalar or vector, expanded or unexpanded Use acc_next_bit() 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 134 uses **acc_next_net()** to display the names of all nets in a module.

```
#include "acc_user.h"

PLI_INT32 display_net_names()
{
    handle    mod_handle, net_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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();
}
```

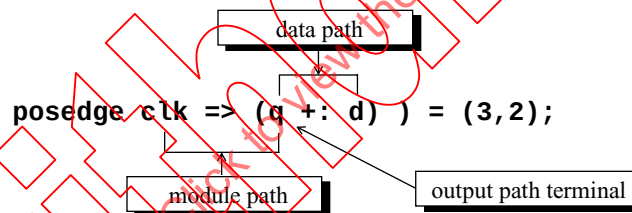
Figure 134—Using acc_next_net()

23.75 acc_next_output()

acc_next_output()			
Synopsis:	Get handles to output path terminals of a module path or data path.		
Syntax:	acc_next_output(reference_handle, terminal_handle)		
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 acc_handle_conn() to get the net attached to the path terminal Use acc_release_object() to free memory allocated by acc_next_output()		

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 135 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.

```
int is_net_on_path_output(net, path)
handle net; /* scalar net or bit-select of vector net */
handle path;
{
    handle pterm_out, pterm_conn, bit;

    /* scan path output terminals */
    pterm_out = null;
    while (pterm_out = acc_next_output(path, pterm_out) )
    {
        /* retrieve net connected to path terminal */
        pterm_conn = acc_handle_conn (pterm_out);

        if (acc_object_of_type (pterm_conn, accExpandedVector) )
        {
            bit = null;
            while (bit = acc_next_bit (pterm_conn, bit) )
                if (acc_compare_handles (bit, net) )
                    return (true);
        }
        else
            if (acc_compare_handles (pterm_conn, net) )
                return (true);
    }
    return (false);
}
```

Figure 135—Using acc_next_output()

23.76 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 scope
	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 scope. This handle can be passed to **acc_fetch_paramtype()** and **acc_fetch_paramval()** to retrieve the data type and value of the parameter. A scope is a module, task, function, or named block.

The example shown in Figure 136 uses **acc_next_parameter()** to scan for all parameters in a module.

```
#include "acc_user.h"
PLI_INT32 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", (PLI_INT32)acc_fetch_paramval(param_handle) );
                break;
            case accStringParam:
                io_printf("%s\n",
                    (char*)(int)acc_fetch_paramval(param_handle) );
                break;
        }
    }
}
```

Figure 136—Using acc_next_parameter()

23.77 acc_next_port()

acc_next_port()			
Synopsis:	Gets handles to the ports of a module, or to ports which are connected to a given net or reg.		
Syntax:	acc_next_port(reference, port_handle)		
Returns:	Type	Description	
	handle	Handle of a module port	
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a module, net, reg or variable
	handle	object_handle	Handle of the previous port found, initially null
Related routines:	Use acc_fetch_direction() to determine the direction of a port Use acc_next_portout() 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 173.

Table 173—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, reg or variable	All ports connected to the net, reg or variable within the scope of the net, reg or variable

The example shown in Figure 137 uses **acc_next_port()** to find and display the input ports of a module.

```
#include "acc_user.h"

PLI_INT32 display_inputs(module_handle)
handle module_handle;
{
    handle    port_handle;
    PLI_INT32 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 137—Using acc_next_port() with a module handle

The example shown in Figure 138 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.

```
PLI_INT32 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 138—Using acc_next_port() with a net handle

23.78 acc_next_portout()

acc_next_portout()			
Synopsis:	Get handles to output or inout ports of a module.		
Syntax:	acc_next_portout(reference_handle, port_handle)		
Returns:	Type	Description	
	handle	Handle of a module port	
Arguments:	Type	Name	Description
	handle	reference_handle	Handle of a module
	handle	port_handle	Handle of the previous port found, initially null
Related routines:	Use acc_fetch_direction() to determine the direction of a port Use acc_next_port() 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 139 uses **acc_next_portout()** to find the output and inout ports of a module.

```
#include "acc_user.h"

PLI_INT32 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 139—Using acc_next_portout()

23.79 acc_next_primitive()

acc_next_primitive()			
Synopsis:	Get handles to gates, switches, or user-defined primitives (UDPs) within a module.		
Syntax:	acc_next_primitive(reference_handle, primitive_handle)		
Returns:	Type	Description	
	handle	Handle of a primitive	
Arguments:	Type	Name	Description
	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 user-defined primitives within a module.

The example shown in Figure 140 uses **acc_next_primitive()** to display the definition names of all primitives in a module.

```
#include "acc_user.h"

PLI_INT32 get_primitive_definitions()
{
    handle module_handle, prim_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 140—Using acc_next_primitive()

23.80 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 one 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

23.81 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	Name	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 141 uses **acc_next_specparam()** to scan for all specparams in a module.

```
#include "acc_user.h"
PLI_INT32 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 141—Using acc_next_specparam()

23.82 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 reg 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 timing check, and to read or modify timing check values.

The example shown in Figure 142 uses **acc_next_tchk()** to display information about setup timing checks.

```

#include "acc_user.h"
PLI_INT32 show_setup_check_nets()
{
    handle          mod_handle, cell_handle;
    handle          tchk_handle, tchkarg1_handle, tchkarg2_handle;
    PLI_INT32        tchk_type, counter;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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);
                tchkarg2_handle = acc_handle_tchkarg2(tchk_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 142—Using acc_next_tchk()

23.83 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 143 uses **acc_next_terminal()** together with **acc_handle_conn()** to retrieve all nets connected to a primitive.

```
#include "acc_user.h"

PLI_INT32 display_terminals()
{
    handle    prim_handle, term_handle;

    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 143—Using acc_next_terminal()

23.84 acc_next_topmod()

acc_next_topmod()			
Synopsis:	Get handles to top-level modules.		
Syntax:	acc_next_topmod(module_handle)		
Returns:	Type	Description	
	handle	Handle of a top-level module	
Arguments:	Type	Name	Description
	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 144 uses **acc_next_topmod()** to display the names of all top-level modules.

```
#include "acc_user.h"
PLI_INT32 show_top_modules()
{
    handle module_handle;
    /*initialize environment for ACC routines*/
    acc_initialize();
    /*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 144—Using acc_next_topmod()

23.85 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:	acc_object_in_typelist(object_handle, object_type_array)		
Returns:	Type	Description	
	PLI_INT32	true if the <i>type</i> , <i>fulltype</i> , or <i>property</i> 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 acc_object_of_type() 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 PLI_INT32  
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 113.

The special property constants supported by **acc_object_in_typelist()** are listed in Table 174.

The example shown in Figure 145 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"

PLI_INT32 display_wired_nets()
{
    static PLI_INT32    wired_nets[5]={accWand,accWor,accTriand,accTrior,0};
    handle              net_handle;

    /*reset environment for ACC routines*/
    acc_initialize();

    /*get handle for net*/
    net_handle = acc_handle_tfarg(1);

    /*if a wired logic net, display its name*/
    if (acc_object_in_typedlist(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));

    acc_close();
}
```

Figure 145—Using acc_object_in_typedlist()

23.86 acc_object_of_type()

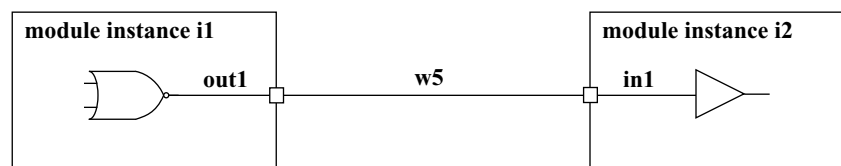
acc_object_of_type()			
Synopsis:	Determine whether an object fits a specified type or fulltype, or special property.		
Syntax:	acc_object_of_type(object_handle, object_type)		
Returns:	Type	Description	
	PLI_INT32	true if the type, fulltype, or property of an object matches the object_type argument false if there is no match	
Arguments:	Type	Name	Description
	handle	object_handle	Handle of an object
	PLI_INT32	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 113. The special property constants supported by **acc_object_of_type()** are listed in Table 174.

Table 174—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.

acc_object_of_type() with an **accScope** type constant will return true if the reference object is a Verilog scope. A scope is a module, task, function or named block.

acc_object_of_type() with an **accModPathHasIfnone** type constant will return true if the reference object is a Verilog module path, and there is an ifnone condition specified for the path.

The example shown in Figure 146 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"

PLI_INT32 display_collapsed_nets()
{
    handle    mod_handle;
    handle    net_handle;
    handle    simulated_net_handle;

    /*reset environment for ACC routines*/
    acc_initialize();

    /*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 146—Using acc_object_of_type()

23.87 acc_product_type()

acc_product_type()		
Synopsis:	Get the software product type that is calling the PLI application.	
Syntax:	acc_product_type()	
Returns:	Type	Description
	PLI_INT32	A predefined integer constant representing the software product type
Arguments:	Type	Name
	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 175.

Table 175—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 147 uses **acc_product_type()** to identify and display the product type being used.

```
#include "acc_user.h"
PLI_INT32 show_application()
{
    /* reset environment for ACC routines */
    acc_initialize();

    /* 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 147—Using acc_product_type()

23.88 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()	
Returns:	Type	Description
	PLI_BYTE8 *	Pointer to a character string
Arguments:	Type	Name
	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 22.9 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 148 uses **acc_product_version()** to identify the version of the software product that is linked to ACC routines.

```
#include "acc_user.h"

PLI_INT32 show_versions()
{
    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 148—Using acc_product_version()

23.89 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	
	PLI_INT32	0 if successful; 1 if an error is encountered	
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 149 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.

```

PLI_INT32 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 149—Using acc_release_object()

23.90 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	acc_replace_delays(object_handle, rise_delay, fall_delay, z_delay)		
Module paths Intermodule paths Ports or port bits	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)		
Returns:		Type	Description
		PLI_INT32	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, bit of a module input port, or intermodule path
	double	rise_delay fall_delay	Rise and fall delay for 2-state primitives or 3-state primitives
Conditional	double	z_delay	If accToHIZDelay is set to from_user : turn-off (to Z) transition delay for 3-state primitives
	double	d1	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 1 : delay for all transitions If accPathDelayCount is set to 2 or 3 : rise transition delay If accPathDelayCount is set to 6 or 12 : 0->1 transition delay
Conditional	double	d2	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 2 or 3 : fall transition delay If accPathDelayCount is set to 6 or 12 : 1->0 transition delay
Conditional	double	d3	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 3 : turn-off transition delay If accPathDelayCount is set to 6 or 12 : 0->Z transition delay
Conditional	double	d4 d5 d6	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 6 or 12 : d4 is Z->1 transition delay d5 is 1->Z transition delay d6 is Z->0 transition delay
Conditional	double	d7 d8 d9 d10 d11 d12	For module/intermodule paths and input ports/port bits: If accPathDelayCount is set to 12 : d7 is 0->X transition delay d8 is X->1 transition delay d9 is 1->X transition delay d10 is X->0 transition delay d11 is X->Z transition delay d12 is Z->X transition delay
	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:	acc_append_delays(object_handle, array_ptr)		
Type		Description	
Returns:	PLI_INT32	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 Section 22.8)

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 section shall apply.

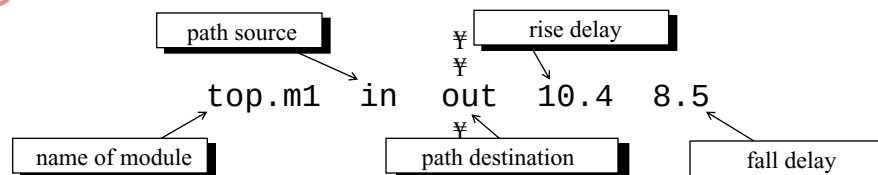
When **accMinTypMaxDelays** is set to **true**, **acc_replace_delays()** shall pass one or more sets of minimum:typical:maximum delays contained in an array, rather than single delays passed as individual arguments. For this min:typ:max delay mode, the second syntax table in this section shall apply.

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 22.8 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**.

When altering the delay via **acc_replace_delays()**, the value of the reject/error region will not be affected unless the limits exceed the value of the delay. If the reject/error limits exceed the delay they will be truncated down to the new delay limit.

The example shown in Figure 150 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"

#define NAME_SIZE 256
PLI_INT32 write_path_delays()
{
    FILE          *infile;
    PLI_BYTE8      full_module_name[NAME_SIZE];
    PLI_BYTE8      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 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 150—Using `acc_replace_delays()` in single delay mode

The example shown in Figure 151 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 );
```

scope

scale factor for minimum delay

scale factor for typical delay

scale factor for maximum delay

```

#include "acc_user.h"
#include "veriusers.h"

PLI_INT32 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(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:
            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;
            acc_replace_delays(prim, da); ← scale delays
        }
    }
    acc_close();
}

```

array has to hold three sets of *min:typ:max* values for rise, fall, and turn-off delays

argument #1: Scope
argument #2: Scale factor for minimum delay
argument #3: Scale factor for typical delay
argument #4: Scale factor for maximum delay

fetch *min:typ:max* delays and store in array **da** as follows:

da[0]	typical rise delay
da[1]	
da[2]	
da[3]	typical fall delay
da[4]	
da[5]	
da[6]	typical turn-off delay
da[7]	
da[8]	

scale delays

replace *min:typ:max* delays with scaled values

Figure 151—Using `acc_replace_delays()` in *min:typ:max* delays mode

23.91 acc_replace_pulsere()

acc_replace_pulsere()			
Synopsis:	Replace existing pulse handling <i>reject_limit</i> and <i>e_limit</i> for a module path, intermodule path or module input port.		
Syntax:	acc_replace_pulsere(object,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:	PLI_INT32	1 if successful; 0 if an error is encountered	
Arguments:	Type	Name	Description
	handle	object	Handle of module path, intermodule path or module input port
	double	r1...r12	reject_limit values; the number of arguments is determined by accPathDelayCount
Related routines:	double	e1...e12	e_limit values; the number of arguments is determined by accPathDelayCount
	Use acc_fetch_pulsere() to get current pulse handling values		
	Use acc_append_pulsere() to append existing pulse handling values		
Related routines:	Use acc_set_pulsere() to set pulse handling values as a percentage of the path delay		
	Use acc_configure() to set accPathDelayCount		

The ACC routine **acc_replace_pulsere()** shall replace existing pulse handling *reject_limit* and *e_limit* values for a module path, intermodule path and module input port. The *reject_limit* and *e_limit* values are used to control how *pulses* are propagated through paths.

A pulse is defined as two transitions that occur in a shorter period of time than the 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
- 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
- A pulse that is greater than or equal to the *e_limit* shall propagate
- The example in Table 176 illustrates the relationship between the *reject_limit* and the *e_limit*.

Table 176—Pulse control example

When	The pulse 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 delay.

If any of the limits do not meet the above rules, they will be truncated.

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 177.

Table 177—How the value of `accPathDelayCount` affects `acc_replace_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, r1 and e1
2	Two pairs of <code>reject_limit</code> and <code>e_limit</code> values: one pair for rise transitions, r1 and e1 one pair for fall transitions, r2 and e2
3	Three pairs of <code>reject_limit</code> and <code>e_limit</code> 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 <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, 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 <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, 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_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 object handle.

The example shown in Figure 152 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

PLI_INT32 replace_halfpulsevals()
{
    FILE      *infile;
    PLI_BYTE8  mod_name[NAME_SIZE];
    PLI_BYTE8  pathin_name[NAME_SIZE], pathout_name[NAME_SIZE];
    handle     mod, path;
    double     rise_reject_limit=0.0, rise_e_limit=0.0,
               fall_reject_limit=0.0, fall_e_limit=0.0;

    /*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 152—Using **acc_replace_pulsere()**

23.92 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 22.9 for more information on the character string buffer.

23.93 acc_set_interactive_scope()

acc_set_interactive_scope()			
Synopsis:	Set the interactive scope of a software tool.		
Syntax:	acc_set_interactive_scope(scope, callback_flag)		
Returns:	Type	Description	
	handle	Handle of a Verilog hierarchy scope	
Arguments:	Type	Name	Description
	handle	scope	Handle to the scope which will be the new interactive scope
	PLI_INT32	callback_flag	If set to TRUE, then the miscf routines shall be called with reason reason_scope immediately. If set to FALSE, then the miscf routines are not called
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

23.94 acc_set_pulsere()

acc_set_pulsere()			
Synopsis:	Set the pulse handling values for a module path, intermodule path or module input port as a percentage of the delay.		
Syntax:	acc_set_pulsere(object, reject_percentage, e_percentage)		
Type		Description	
Returns:	PLI_INT32	Always returns 0	
Arguments:	Type	Name	Description
	handle	object	Handle of a module path, intermodule path or module input port
	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, intermodule path or module input port, specified as a percentage multiplier of the delay.

A pulse is defined as two transitions that occur in a shorter period of time than the 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
- 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
- A pulse that is greater than or equal to the *e_percentage* shall propagate
- The example in Table 178 illustrates the relationship between the *reject_percentage* and the *e_percentage*.

Table 178—Pulse control example

Given a path with a delay of 5.0	
When	A pulse 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 153 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"

PLI_INT32 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 153—Using acc_set_pulsere()

23.95 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:	PLI_BYTE8 *	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 PLI_BYTE8 *	module_name	Quoted string or pointer to a character string with the name of a module instance (optional: used when accEnableArgs is set and <i>module_handle</i> is <i>null</i>)
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 179.

Table 179—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 <i>null</i>	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 <i>null</i>	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 <i>null</i>	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 154 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"

PLI_INT32 is_net_in_module(module_handle, net_name)
handle      module_handle;
PLI_BYTE8   *net_name;
{
    handle    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 154—Using acc_set_scope()

23.96 acc_set_value()

acc_set_value()			
Synopsis:	Set and propagate a value on a reg, variable, user-defined system function or a sequential UDP; procedurally assign a reg or variable; force a reg, variable, or net.		
Syntax:	acc_set_value(object_handle, value_p, delay_p)		
Type		Description	
Returns:	PLI_INT32	Zero if no errors; nonzero if an error occurred	
Arguments:		Type	Description
		handle	object_handle
		p_setval_value	value_p
		p_setval_delay	delay_p
Related routines:		Use acc_fetch_value() to retrieve a logic value Use acc_fetch_size() to get the number of bits in a vector	

The ACC routine **acc_set_value()** shall set and propagate a value onto a reg, integer variable, time variable, real variable, or a sequential UDP. The routine shall also perform procedural assign/deassign or procedural force/release functions.

The **acc_set_value()** routine shall also return the value of a system function by passing a handle to the user-defined system function as the object handle. This should only occur during execution of the calltf routine for the system function. Attempts to use **acc_set_value()** with a handle to the system function when the calltf routine is not active shall be ignored. Should the calltf routine for a user defined system function fail to put a value during its execution, the default value of 0 shall be applied.

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 155 (note that this structure is also used with the **acc_fetch_value()** routine).

The *format* field in the s_setval_value structure shall indicate the value type. The format shall be a pre-defined integer constant, listed in Table 180.

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
{
    PLI_INT32 format;
    union
    {
        PLI_BYTE8      *str;
        PLI_INT32       scalar;
        PLI_INT32       integer;
        double          real;
        p_acc_vecval    vector;
    } value;
} s_setval_value, *p_setval_value, s_acc_value, *p_acc_value;
```

Figure 155—The s_setval_value structure used by acc_set_value()

Table 180—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 regs, any ASCII string; for real-valued objects, any string that represents a real number
accBinStrVal	A base 2 representation as a string
accOctStrVal	A base 8 representation as a string
accDecStrVal	A base 10 representation as a string
accHexStrVal	A base 16 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 156.

```
typedef struct t_acc_vecval
{
    PLI_INT32 aval;
    PLI_INT32 bval;
} s_acc_vecval, *p_acc_vecval;
```

Figure 156—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 the 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 181.

Table 181—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 157.

The *time* field in the `s_setval_delay` structure shall indicate the delay that shall take place before a reg value assignment. The time field shall be of type `s_acc_time` structure, as shown in Figure 158.

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 182 and Table 184.

```
typedef struct t_setval_delay
{
    s_acc_time time;
    PLI_INT32 model;
} s_setval_delay; *p_setval_delay;
```

Figure 157—The `s_setval_delay` structure for `acc_set_value()`

Table 182—Predefined delay constants for the model field of `s_setval_delay`

Integer constant	Delay model	Description
<code>accNoDelay</code>	No delay	Sets a reg, variable or sequential UDP to the indicated value with no delay; other events scheduled for the object are not affected
<code>accInertialDelay</code>	Inertial delay	Sets a reg or variable to the indicated value after the specified delay; all scheduled events on the object are removed before this event is scheduled
<code>accTransportDelay</code>	Modified transport delay	Sets a reg or variable to the indicated value after the specified delay; all scheduled events on the object for times later than this event are removed
<code>accPureTransportDelay</code>	Pure transport delay	Sets a reg or variable 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 158.

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 183.

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
{
    PLI_INT32 type;
    PLI_INT32 low,
              high;
    double    real;
} s_acc_time, *p_acc_time;
```

Figure 158—The *s_acc_time* structure for *acc_set_value()*

Table 183—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 reg or variable or to deassign the reg or variable. 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 a value onto a reg, variable or net, or to release the reg, variable 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 184 shall be used.

Table 184—Predefined assign/force constants for the model field of `s_setval_delay`

Integer constant	Description
accAssignFlag	Assigns a reg or variable 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 an assigned reg or variable; 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 reg, variable or net; other events scheduled for the object are overridden. Same functionality as the Verilog HDL procedural force keyword.
accReleaseFlag	Releases a forced reg, variable 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 159 uses **acc_set_value()** to set and propagate a value onto a reg. 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 reg, `r1`:

```

$my_set_value(r1, "x011", 2.4);

PLI_INT32 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 159—Using `acc_set_value()`

23.97 **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)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	handle	object_handle	Handle to an object to be monitored (such as a reg or net)
	C routine pointer	consumer_routine	Unquoted name of the C routine to be called when the object changes value
	PLI_BYTE8 *	user_data	User-defined data that is passed back to the consumer routine when the object changes value
	PLI_INT32	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 regs and bit-selects of vector regs
- Scalar nets, unexpanded vector nets, and bit-selects of expanded vector nets
- Integer real and time variables
- Module ports
- Primitive output or inout terminals
- Named events

NOTE Adding a value change link to a module port is equivalent to adding a value change link to the loconn of the port. The *vc_reason* returned shall be based on the loconn of the port.

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 22.10 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 `PLI_BYTE8*`.

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 185.

Table 185—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()**.

NOTE It is not recommended that multiple VCL flags be added with the same object, consumer and *user_data*. If multiple flags with the same values are added, then each call to **acc_vcl_delete()** with those values shall delete one flag; the order of deletion is indeterminate.

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 22.10 for an example of using **acc_vcl_add()**.

23.98 acc_vcl_delete()

acc_vcl_delete()			
Synopsis:	Removes a VCL callback monitor.		
Syntax:	acc_vcl_delete(object_handle, consumer_routine, user_data, vcl_flag)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	handle	object_handle	Handle to the object to be monitored specified in the call to acc_vcl_add()
	C routine pointer	consumer_routine	Unquoted name of the C routine specified in the call to acc_vcl_add()
	PLI_BYTE8 *	user_data	User-defined data specified in the call to acc_vcl_add()
Related routines:	PLI_INT32	vcl_flag	Predefined integer constant; vcl_verilog
	Use acc_vcl_add() 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 22.10 for an example of using **acc_vcl_delete()**.

23.99 acc_version()

acc_version()		
Synopsis:	Get a pointer to a character string that indicates version number of the ACC routine software.	
Syntax:	<code>acc_version()</code>	
Returns:	Type	Description
	PLI_BYTE8 *	Character string pointer
Arguments:	Type	Name
	None	
Related routines:	Use <code>acc_product_version()</code> to get the version of the software product in use	
	Use <code>acc_product_type()</code> 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 22.9 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 PLI version of ACC routines, **acc_version()** might 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 160 uses **acc_version()** to identify the version of ACC routines linked to the application.

```
#include "acc_user.h"

PLI_INT32 show_versions()
{
    /*initialize environment for ACC routines*/
    acc_initialize();

    /*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 160—Using acc_version()

24. Using TF routines

This clause 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.

24.1 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.

24.2 TF routine system task/function arguments

The number of arguments passed to a system task shall be returned by **tf_nump()**. A type for each argument shall be returned by **tf_typep()** and is primarily used to determine if an argument is writable.

An argument shall be considered *read-only* if, in the Verilog HDL source description, the argument 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.

Arguments shall be considered *writable* from the PLI if the arguments 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.

24.3 Reading and writing system task/function argument values

User-defined system task and function argument 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.

24.3.1 Reading and writing 2-state parameter argument values

To access the 2-state (logic 0 and 1) value of a system task/function argument of size less than or equal to 32 bits, the routine **tf_getp()** can be used. To set the 2-state value of an argument of size less than or equal to 32 bits, **tf_putp()** can be used. If the argument is 33–64 bits **tf_getlongp()** and **tf_putlongp()** can be used. For arguments of type real, **tf_getrealp()** and **tf_putrealp()** can be used. Logic X and Z bits in the argument value shall be interpreted as 0.

24.3.2 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_strdelputp()**, **tf_strlongdelputp()**, and **tf_strrealdelputp()** can be used to write 4-state values to writable arguments. 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 an argument encoded in an **s_vecval** structure. After **tf_exprinfo()** has been called once for an argument, the pointer to the **s_vecval** structure can be saved. The argument 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.

24.3.3 Reading and writing strength values

Strength values on scalar net arguments can be accessed with the routine **tf_nodeinfo()**.

24.3.4 Reading and writing to memories

Memory array values can be accessed with the routine **tf_nodeinfo()**. This routine returns a 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.

24.3.5 Reading and writing string values

The routine **tf_getstringp()** shall return the string representation of a string constant or a vector argument. There is no direct method to write string values using TF routines, but it can be accomplished by writing 8-bit ASCII character values to 8-bit reg elements in a vector reg using the **tf_exprinfo()** value structure.

24.3.6 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()**, **tf_putlongp()** and **tf_putrealp()** with an argument value of 0. It is illegal to schedule the return value of a system function at a future simulation time. The routines **tf_strdelputp()**, **tf_strlongdelputp()**, and **tf_strrealdelputp()** cannot be used to return the value of a system function. Should the calltf routine for a user defined system function fail to put a value during its execution, the default value of 0 shall be applied.

NOTE calling put routines to TF argument 0 (return of a function) shall only return a value in a calltf application, when the call to the function is active. The action of the put routine shall be ignored when the function is not active.

24.3.7 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 argument.

The following examples illustrate what cautions should be taken.

If the second argument of a system task/function instance is of type **tf_readwritereal**, meaning the argument is declared as a real variable in the Verilog HDL source description, the following **tf_put** routines shall produce valid results:

```
PLI_INT32 i = 5;
tf_putp(2, i); /* write an integer value to 2nd argument */
```

This example sets the second task/function argument 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 argument */
```

This example sets the second task/function argument to 5.7.

The following routines, however, shall produce invalid results for the following reasons:

```
PLI_INT32 i = 5;
tf_putrealp(2, i); /* invalid result */
```

The statement **PLI_INT32 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`.

24.4 Value change detection

Value changes on system task/function arguments can be detected by enabling asynchronous callbacks with **tf_asynchon()**. The callbacks can be disabled with **tf_asynchoff()**. When argument change callbacks are enabled with **tf_asynchon()**, whenever an argument changes value, the misctf 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**. The value change can be examined immediately, or a second callback can be requested later in the same time step (as described in 24.6). By setting a second callback at the end of the time step, an application can process all argument 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 arguments that changed in a time step.

24.5 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_longdelay()**, **tf_scale_realdelay()**, **tf_unscale_longdelay()**, and **tf_unscale_realdelay()** can be used to convert between scaled delays and internal simulation time.

24.6 Simulation synchronization

There are TF routines that allow synchronized calling of the misctf application associated with a user-defined system task or function. The misctf 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 a 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 misctf application can propagate new values 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 system task/function arguments during a **reason_rosynch** callback (the 'ro' indicates read-only). Placing a callback for **tf_rosynchronize()** during a callback for **reason_rosynch** will result in another **reason_rosynch** callback occurring during the same time slice.

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.

24.7 Instances of user-defined tasks 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 arguments of one instance can be manipulated from another task or function instance.

24.8 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 a Verilog HDL task or function, a named block, or a module instance, shall be returned by **tf_spname()**.

24.9 Saving information from one system TF call to the next

The TF routines **tf_setworkarea()** and **tf_getworkarea()** provide a special storage *work area* that can be used for:

- Saving data during one call to a PLI application that can be retrieved in 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.

24.10 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 output channel of the software product which invoked the PLI application 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.

24.11 Stopping and finishing

The routines **tf_dostop()** and **tf_dofinish()** are the PLI equivalents to the built-in system tasks **\$stop** and **\$finish**.

25. TF routine definitions

This clause defines the PLI TF routines, explaining their function, syntax, and usage. The routines are listed in alphabetical order. See Clause 23 for conventions that are used in the definitions of the PLI routines.

Withdrawing
IECNORM.COM: Click to view the full PDF of IEC 61691-4:2004

25.1 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
	PLI_INT32	mcd	An integer multi-channel descriptor value representing one or more open files
	quoted string or PLI_BYTE8 *	format	A quoted character string or pointer to a character string that controls the message to be written
	(optional)	arg1...arg12	1 to 12 optional arguments of 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 the output channel of the software product which invoked the PLI application and to the Verilog product log file		

The TF routine **io_mcdprintf()** shall write a formatted message to one or more open files, as described by the multi-channel descriptor mcd. This routine uses the descriptors created by the **\$fopen** system task or the VPI routine **vpi_mcd_open()**. See 17.2.1 for the functional description of \$fopen, and 27.25 for the description of vpi_mcd_open().

The format strings shall use the same format as the C routine fprintf().

The maximum number of arguments that can be used in the format control string is 12.

25.2 io_printf()

io_printf()			
Synopsis:	Print a formatted message to the output channel of the software product which invoked the PLI application and to the log file of the product.		
Syntax:	io_printf(format, arg1, ...arg12)		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments: (optional)	quoted string or PLI_BYTE8 *	format	A quoted character string or pointer to a character string that controls the message to be written
		arg1...arg12	1 to 12 optional arguments of 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 output channel of the software product which invoked the PLI application 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.

25.3 mc_scan_plusargs()

mc_scan_plusargs()			
Synopsis:	Scan software product invocation command line for plus (+) options.		
Syntax:	mc_scan_plusargs(startarg)		
	Type	Description	
Returns:	PLI_BYTE8 *	Pointer to a string with the result of the search	
	Type	Name	Description
Arguments:	quoted string or PLI_BYTE8 *	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 the 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

25.4 tf_add_long()

tf_add_long()			
Synopsis:	Add two 64-bit integers.		
Syntax:	tf_add_long(aof_low1, aof_high1, low2, high2)		
Returns:	Type	Description	
	PLI_INT32	Always returns 0	
Arguments:	Type	Name	Description
	PLI_INT32 *	aof_low1	Pointer to least significant 32 bits of first operand
	PLI_INT32 *	aof_high1	Pointer to most significant 32 bits of first operand
	PLI_INT32	low2	Least significant 32 bits of second operand
	PLI_INT32	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 161 shows the high and low 32 bits of two 64-bit integers and how **tf_add_long()** shall add them.

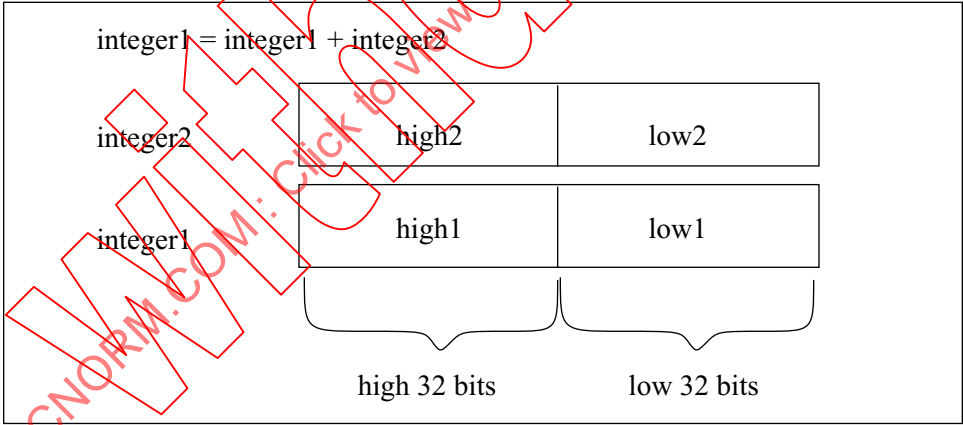


Figure 161 – Adding with tf_add_long()

25.5 tf_asynchoff(), tf_iasynchoff()

tf_asynchoff(), tf_iasynchoff()			
Synopsis:	Disable asynchronous calling of the misctf application.		
Syntax:	tf_asynchoff() tf_iasynchoff(instance_p)		
Type		Description	
Returns:	PLI_INT32	Always returns 0	
Type		Name	Description
Arguments:	PLI_BYTE8 *	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 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_asynchon()** or **tf_iasynchon()**.

25.6 tf_asynchon(), tf_iasynchon()

tf_asynchon(), tf_iasynchon()			
Synopsis:	Enable asynchronous calling of the miscf application for system task/function argument value changes.		
Syntax:	tf_asynchon() tf_iasynchon(instance_p)		
Type		Description	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
Type		Name	Description
Arguments:	PLI_BYTE8 *	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 miscf application Use tf_getpchange() or tf_igetpchange() to get the index number of the argument 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 miscf user application to be called asynchronously whenever a system task/function argument 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 miscf in the PLI interface mechanism shall be called with a reason of **reason_paramvc** each time any task/function argument changes value or strength. The index number of the argument that changed is passed to the miscf 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 Section 24.6). By setting a second callback at the end of the time step, an application can process all argument value changes within 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 arguments that changed in a time step.

Task/function argument index numbering shall proceed from left to right, and the left-most argument shall be number 1.

25.7 tf_clearalldelays(), tf_iclearalldelays()

tf_clearalldelays(), tf_iclearalldelays()			
Synopsis:	Clear all scheduled reactivations by tf_setdelay() or tf_isetdelay().		
Syntax:	tf_clearalldelays() tf_iclearalldelays(instance_p)		
Type		Description	
Returns:	PLI_INT32	Always returns 1	
Type		Name	Description
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_setdelay() or tf_isetdelay() to schedule a reactivation Use tf_getinstance() 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.

25.8 tf_compare_long()

tf_compare_long()			
Synopsis:	Compare two 64-bit integer values.		
Syntax:	tf_compare_long(low1, high1, low2, high2)		
Type		Description	
Returns:	PLI_INT32	An integer flag indicating the result of the comparison	
Arguments:	Type	Name	Description
	PLI_UINT32	low1	Least significant 32 bits of first operand
	PLI_UINT32	high1	Most significant 32 bits of first operand
	PLI_UINT32	low2	Least significant 32 bits of second operand
	PLI_UINT32	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 186.

Table 186—Return values for tf_compare_long()

When	tf_compare_long() shall return
operand1 < operand2	-1
operand1 = operand2	0
operand1 > operand 2	1

25.9 tf_copypvc_flag(), tf_icopypvc_flag()

tf_copypvc_flag(), tf_icopypvc_flag()			
Synopsis:	Copy system task/function argument value change flags.		
Syntax:	tf_copypvc_flag(narg) tf_icopypvc_flag(narg, instance_p)		
Type		Description	
Returns:	PLI_INT32	The value of the pvc flag	
Type		Name	Description
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument, or -1
	PLI_BYTE8 *	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 argument 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 *narg* is the index number of an argument in the current instance or a specific instance of a user-defined system task or function. Task/function argument indexing shall proceed from left to right, with the left-most argument being number 1. If *narg* is **-1**, then all argument pvc flags shall be copied and the logical OR of all saved flags returned.

Argument Value Change (pvc) flags shall be used to indicate whether a particular user-defined system task or function argument has changed value. Each argument 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.

25.10 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:	PLI_INT32 *	aof_low1	Pointer to least significant 32 bits of first operand
	PLI_INT32 *	aof_high1	Pointer to most significant 32 bits of first operand
	PLI_INT32	low2	Least significant 32 bits of second operand
	PLI_INT32	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 162 shows the high and low 32 bits of two 64-bit integers and how **tf_divide_long()** shall divide them.

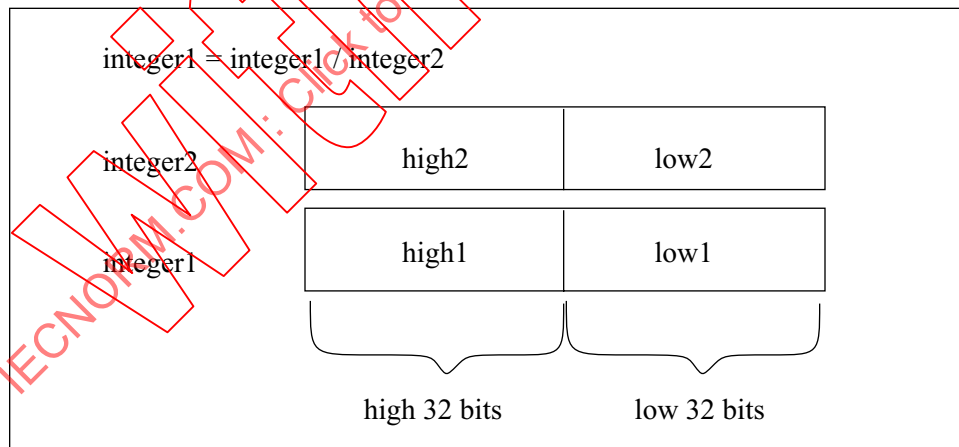


Figure 162—Dividing with tf_divide_long()

25.11 tf_dofinish()

tf_dofinish()			
Synopsis:	Exit software product execution.		
Syntax:	tf_dofinish()		
	Type	Description	
Returns:	PLI_INT32	Always returns 0	
	Type	Name	Description
Arguments:	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.

25.12 tf_dostop()

tf_dostop()		
Synopsis:	Cause software product to enter interactive mode.	
Syntax:	tf_dostop()	
	Type	Description
Returns:	PLI_INT32	Always returns 0
	Type	Name
Arguments:	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.

25.13 `tf_error()`

<code>tf_error()</code>			
Synopsis:	Report an error message.		
Syntax:	<code>tf_error(format, arg1, ...arg5)</code>		
Type		Description	
Returns:	PLI_INT32	Always returns 0	
Type		Name	Description
Arguments: (optional)	quoted string or PLI_BYTE8 *	format	A quoted character string or pointer to a character string that controls the message to be written
		arg1...arg5	One to five optional arguments of the format control string; the type of each argument should be consistent with how it is used in the format string
Related routines:	Use <code>tf_message()</code> to write error messages with additional format control Use <code>tf_warning()</code> to write a warning message Use <code>io_printf()</code> or <code>io_mcdprintf()</code> 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 output channel of the software product which invoked the PLI application 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.

25.14 tf_evaluatep(), tf_ievaluatep()

tf_evaluatep(), tf_ievaluatep()			
Synopsis:	Evaluate a system task/function argument expression.		
Syntax:	tf_evaluatep(narg) tf_ievaluatep(narg, instance_p)		
Type		Description	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
Arguments:	Type	Name	Description
	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	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 argument 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 *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

25.15 tf_exprinfo(), tf_iexprinfo()

tf_exprinfo(), tf_iexprinfo()			
Synopsis:	Get system task/function argument expression information.		
Syntax:	tf_exprinfo(narg, exprinfo_p) tf_iexprinfo(narg, exprinfo_p, instance_p)		
Returns:		Type	Description
		struct t_tfexprinfo *	Pointer to a structure containing the value of the second argument if successful; 0 if an error occurred
Arguments:	Type		Name
	Name		Description
	PLI_INT32	narg	Index number of the user-defined system task or function argument
	struct t_tfexprinfo *	exprinfo_p	Pointer to a variable declared as a t_tfexprinfo structure type
	PLI_BYTE8 *	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 arguments 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 argument 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 *narg* is out of range, or if some other error is found, then 0 shall be returned. The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

The `s_tfexprinfo` structure is defined in `veriuser.h` and is listed in Figure 163.

```
typedef struct t_tfexprinfo
{
    PLI_INT16 expr_type;
    PLI_INT16 padding;
    struct t_vecval *expr_value_p;
    double    real_value;
    PLI_BYTE8 *expr_string;
    PLI_INT32 expr_ngroups;
    PLI_INT32 expr_vec_size;
    PLI_INT32 expr_sign;
    PLI_INT32 expr_lhs_select;
    PLI_INT32 expr_rhs_select;
} s_tfexprinfo, *p_tfexprinfo;
```

Figure 163—The `s_tfexprinfo` structure definition

The *expr_type* of the `s_tfexprinfo` structure shall indicate the Verilog HDL data type of the argument, and it shall be one of the predefined constants as given in Table 187 and defined in `veriuser.h`.

Table 187—Predefined constants used with `tf_exprinfo()`

Predefined constant	Description
tf_nullparam	For null or non-existent arguments
tf_string	For string arguments
tf_readonly	For net, net bit, net part select and constant integer arguments
tf_readonlyreal	For constant real number arguments
tf_readwrite	For reg, integer and time variable arguments
tf_readwritereal	For real variable arguments
tf_rwbselect	For bit-select of reg, integer and time variable arguments
tf_rwpartselect	For part-select of reg, integer and time variable arguments
tf_rwmemselect	For memory word arguments

If the expression type is **tf_readonly**, **tf_readwrite**, **tf_rwbselect**, **tf_rwpartselect**, or **tf_rwmemselect**, 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 `veriuser.h` and is listed in Figure 164.

```
typedef struct t_vecval
{
    PLI_INT32 avalbits;
    PLI_INT32 bvalbits;
} s_vecval, *p_vecval;
```

Figure 164—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 argument. 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 188.

Table 188—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_rwpselect**, or **tf_rwmselect**, the *expr_ngroups* of the *s_tfexprinfo* structure shall indicate the number of groups for the argument 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_rwpselect**, or **tf_rwmselect**, 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.

The *expr_lhs_select* and *expr_rhs_select* fields shall contain the select information about the object if it is a reg bit-select, net bit-select, part-select, variable array word-select, or memory word-select.

25.16 tf_getcstringp(), tf_igetcstringp()

tf_getcstringp(), tf_igetcstringp()			
Synopsis:	Get system task/function argument value as a string.		
Syntax:	tf_getcstringp(narg) tf_igetcstringp(narg, instance_p)		
Type		Description	
Returns:	PLI_BYTE8 *	Pointer to a character string	
Type		Name	Description
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	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 an argument value as a 32-bit integer Use tf_getlongp() or tf_igetlongp() to get an argument value as a 64-bit integer Use tf_getrealp() or tf_igetrealp() to get an argument value as a double Use tf_strgetp() or tf_istrgetp() to get an argument 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 argument in the current instance or a specific instance of a user-defined system task or function. If the argument identified by *narg* is a literal string, reg, integer variable, time 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 argument identified by *narg* is null or if *narg* is out of range, then a null shall be returned. If the argument identified by *narg* is a real variable or an expression that evaluates to a real value, then **tf_getcstringp()** and **tf_igetcstringp()** shall return NULL.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

25.17 tf_getinstance()

tf_getinstance()		
Synopsis:	Get a pointer to the current instance of a user-defined system task or function.	
Syntax:	tf_getinstance()	
	Type	Description
Returns:	PLI_BYTE8 *	Pointer to a system task or function instance
	Type	Name
Arguments:	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.

25.18 tf_getlongp(), tf_igetlongp()

tf_getlongp(), tf_igetlongp()			
Synopsis:	Get system task/function argument value as a 64-bit integer.		
Syntax:	tf_getlongp(aof_highvalue, narg) tf_igetlongp(aof_highvalue, narg, instance_p)		
Type		Description	
Returns:	PLI_INT32	Least significant (right-most) 32 bits of the argument value	
Type		Name	Description
Arguments:	PLI_INT32 *	aof_highvalue	Pointer to most significant (left-most) 32 bits of the argument value
	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	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 an argument value as a 32-bit integer Use tf_getrealp() or tf_igetrealp() to get an argument value as a double Use tf_getcstringp() or tf_igetcstringp() to get an argument value as a string Use tf_strgetp() or tf_istrgetp() to get an argument 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 argument specified by *narg* in the current instance or a specific instance of a user-defined system task or function. If *narg* is out of range or the argument is **NULL**, then 0 shall be returned. Logic X and Z bits in the argument value shall be interpreted as 0.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

25.19 `tf_getlongtime()`, `tf_igetlongtime()`

<code>tf_getlongtime()</code>, <code>tf_igetlongtime()</code>			
Synopsis:	Get current simulation time as a 64-bit integer.		
Syntax:	<code>tf_getlongtime(aof_hightime)</code> <code>tf_igetlongtime(aof_hightime, instance_p)</code>		
	Type	Description	
Returns:	PLI_INT32	Least significant (right-most) 32 bits of simulation time	
	Type	Name	Description
Arguments:	PLI_INT32 *	<code>aof_hightime</code>	Pointer to most significant (left-most) 32 bits of simulation time
	PLI_BYTE8 *	<code>instance_p</code>	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use <code>tf_gettime()</code> to get the simulation time as a 32-bit integer Use <code>tf_str_gettime()</code> to get the simulation time as a character string Use <code>tf_getinstance()</code> 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.

25.20 tf_getnextlongtime()

tf_getnextlongtime()			
Synopsis:	Get next time at which a simulation event is scheduled.		
Syntax:	tf_getnextlongtime(aof_lowtime, aof_hightime)		
Type		Description	
Returns:	PLI_INT32	Integer value representing the meaning of the next event time obtained	
Type		Name	Description
Arguments:	PLI_INT32 *	aof_lowtime	Pointer to least significant (right-most) 32 bits of simulation time
	PLI_INT32 *	aof_hightime	Pointer to most significant (left-most) 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 189 summarizes the functions of **tf_getnextlongtime()**.

Table 189—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.

25.21 `tf_getp()`, `tf_igetp()`

<code>tf_getp()</code> , <code>tf_igetp()</code>			
Synopsis:	Get a system task/function argument value as an integer or character string pointer.		
Syntax:	<code>tf_getp(narg)</code> <code>tf_igetp(narg, instance_p)</code>		
Type		Description	
Returns:	PLI_INT32	Integer value of an argument or character string pointer of argument string value	
Type		Name	Description
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use <code>tf_getlongp()</code> or <code>tf_igetlongp()</code> to Get an argument value as a 64-bit integer Use <code>tf_getrealp()</code> or <code>tf_igetrealp()</code> to get an argument value as a double Use <code>tf_getcstringp()</code> or <code>tf_igetcstringp()</code> to get an argument value as a string Use <code>tf_strgetp()</code> or <code>tf_istrgetp()</code> to get an argument value as a formatted string Use <code>tf_getinstance()</code> 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 argument specified by *narg* in the current instance or a specific instance of a user-defined system task or function. If the value of the argument is an integer or a real number, the routines shall return an integer value. If the argument is a literal string, then the routines shall return a pointer to a C type string (a string terminated by a '\0' character). If *narg* is out of range or the argument is null, then 0 shall be returned. Logic X and Z bits in the argument value shall be interpreted as 0.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

The routines **`tf_getp()`** and **`tf_getrealp()`** differ in the value returned, as shown by the following example.

If the fourth argument in the user-defined system task or function has a value of 9.6 (a real value), then

```
PLI_INT32 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 PLI_INT32 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.

25.22 tf_getpchange(), tf_igetpchange()

tf_getpchange(), tf_igetpchange()			
Synopsis:	Get the index number of the next system task/function argument that changed value.		
Syntax:	tf_getpchange(narg) tf_igetpchange(narg, instance_p)		
Type		Description	
Returns:	PLI_INT32	Index number of the argument that changed	
Arguments:	Type	Name	Description
	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	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_imovepvc_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 argument with a number greater than *narg* that changed value for the current instance or for a specific instance of a user-defined system task or function. The *narg* argument shall be 0 the first time this routine is called within a given user routine invocation. The routines shall return the argument number if there is a change in an argument with a number greater than *narg*, and they shall return 0 if there are no changes in arguments greater than *narg* 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 *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

PVC flags shall indicate whether a particular user-defined system task or function argument has changed value. Each argument 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.

25.23 tf_getrealp(), tf_igetrealp()

tf_getrealp(), tf_igetrealp()			
Synopsis:	Get a system task/function argument value as a double-precision value.		
Syntax:	tf_getrealp(narg) tf_igetrealp(narg, instance_p)		
Type		Description	
Returns:	double	Double-precision value of an argument	
Type		Name	Description
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	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 an argument value as a 32-bit integer Use tf_getlongp() or tf_igetlongp() to get an argument value as a 64-bit integer Use tf_getcstringp() or tf_igetcstringp() to get an argument value as a string Use tf_strgetp() or tf_istrgetp() to get an argument 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 argument specified by *narg* in the current instance or a specific instance of a user-defined system task or function. If *narg* is out of range or the argument is null, then 0 shall be returned. Logic X and Z bits in the argument value shall be interpreted as 0.

The routines **tf_getrealp()** and **tf_igetrealp()** shall return 0.0 if the value being read is a literal string. Therefore, before calling these routines, **tf_typep()** or **tf_itypep()** should be called to check the type of the argument.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

25.24 tf_gettime(), tf_igettime()

tf_gettime(), tf_igettime()			
Synopsis:	Get the current simulation time in double-precision format.		
Syntax:	<pre>tf_gettime() tf_igettime(instance_p)</pre>		
Type		Description	
Returns:	double	Current simulation time	
Type		Name	Description
Arguments:	PLI_BYTE8 *	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_gettime()** and **tf_igettime()** 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.

25.25 tf_gettime(), tf_igettime()

tf_gettime(), tf_igettime()			
Synopsis:	Get the current simulation time as a 32-bit integer.		
Syntax:	tf_gettime() tf_igettime(instance_p)		
Type		Description	
Returns:	PLI_INT32	Least significant 32 bits of simulation time	
Type		Name	Description
Arguments:	PLI_BYTE8 *	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_getrealtime() 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.

25.26 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:	PLI_INT32	An integer value that represents a time precision	
Type		Name	Description
Arguments:	PLI_BYTE8 *	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 190.

Table 190—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 s
-5	10 s
-6	1 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.

25.27 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:	PLI_INT32	An integer value that represents a time unit	
Type		Name	Description
Arguments:	PLI_BYTE8 *	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 191.

Table 191—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.

25.28 tf_getworkarea(), tf_igetworkarea()

tf_getworkarea(), tf_igetworkarea()			
Synopsis:	Get work area pointer.		
Syntax:	tf_getworkarea() tf_igetworkarea(instance_p)		
Type		Description	
Returns:	PLI_BYTE8 *	Pointer to a work area shared by all routines for a specific task/function instance	
Type		Name	Description
Arguments:	PLI_BYTE8 *	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()**.

25.29 tf_long_to_real()

tf_long_to_real()			
Synopsis:	Convert a 64-bit integer to a real number.		
Syntax:	tf_long_to_real(low, high, aof_real)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	PLI_INT32	low	Least significant (right-most) 32 bits of a 64-bit integer
	PLI_INT32	high	Most significant (left-most) 32 bits of a 64-bit integer
	double *	aof_real	Pointer to a double-precision variable
Related routines:	Use tf_real_to_long() to convert a real number to a 64-bit integer. Use tf_longtime_tostr() to convert a 64-bit integer to a character string.		

The TF routine **tf_long_to_real()** shall convert a 64-bit 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.

25.30 tf_longtime_tostr()

tf_longtime_tostr()			
Synopsis:	Convert 64-bit integer time value to a character string.		
Syntax:	tf_longtime_tostr(lowtime, hightime)		
Type		Description	
Returns:	PLI_BYTE8 *	Pointer to a character string representing the simulation time value	
Arguments:	Type	Name	Description
	PLI_INT32	lowtime	Least significant (right-most) 32 bits of simulation time
	PLI_INT32	hightime	Most significant (left-most) 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.

25.31 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)		
	Type	Description	
Returns:	PLI_INT32	Always returns 0	
	Type	Name	Description
Arguments:	PLI_INT32	level	A predefined constant indicating the severity level of the error
	quoted string or PLI_BYTE8 *	facility	A quoted character string or pointer to a character string used in the output message
	quoted string or PLI_BYTE8 *	code	A quoted character string or pointer to a character string used in the output message
	quoted string or PLI_BYTE8 *	message	A quoted character string or pointer to a character string that controls the message to be written
	(optional)	arg1...arg5	One to five optional arguments of 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 output channel of the software product which invoked the PLI application 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.

25.32 tf_mipname(), tf_imipname()

tf_mipname(), tf_imipname()			
Synopsis:	Get the hierarchical module instance path name as a string.		
Syntax:	tf_mipname() tf_imipname(instance_p)		
Type		Description	
Returns:	PLI_BYTE8 *	Pointer to a string containing the hierarchical path name	
Type		Name	Description
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_spname() or tf_ispname() to get the scope path name Use tf_getinstance() 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.

25.33 tf_movepvc_flag(), tf_imovepvc_flag()

tf_movepvc_flag(), tf_imovepvc_flag()			
Synopsis:	Move system task/function argument value change flags.		
Syntax:	tf_movepvc_flag(narg) tf_imovepvc_flag(narg, instance_p)		
Type		Description	
Returns:	PLI_INT32	The value of the pvc flag	
Arguments:	Type	Name	Description
	PLI_INT32	narg	Index number of the user-defined system task or function argument, or -1
	PLI_BYTE8 *	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 argument 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 *narg* shall be the index number of an argument in a specific instance of a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1. If *narg* is **-1**, then all argument 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 argument has changed value. Each argument 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.

25.34 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:	PLI_INT32 *	aof_low1	Pointer to least significant 32 bits of first operand
	PLI_INT32 *	aof_high1	Pointer to most significant 32 bits of first operand
	PLI_INT32	low2	Least significant 32 bits of second operand
	PLI_INT32	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 multiply 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 165 shows the high and low 32 bits of two 64-bit integers and how **tf_multiply_long()** shall multiply them.

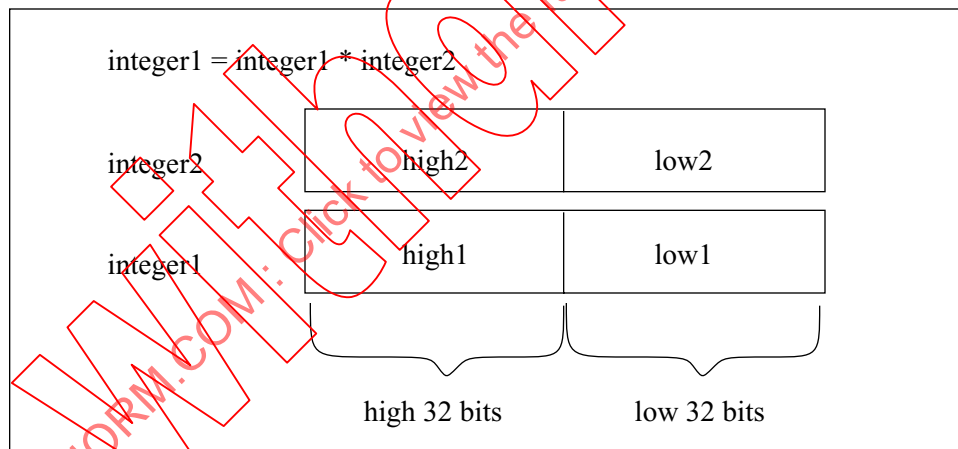


Figure 165—Multiplying with tf_multiply_long()

25.35 tf_nodeinfo(), tf_inodeinfo()

tf_nodeinfo(), tf_inodeinfo()			
Synopsis:	Get system task/function argument node information.		
Syntax:	tf_nodeinfo(narg, nodeinfo_p) tf_inodeinfo(narg, nodeinfo_p, instance_p)		
Type		Description	
Returns:	struct t_tfnodeinfo *	The value of the second argument if successful; 0 if an error occurred	
Arguments:	Type	Name	Description
	PLI_INT32	narg	Index number of the user-defined system task or function argument
	struct t_tfnodeinfo *	nodeinfo_p	Pointer to a variable declared as the t_tfnodeinfo structure type
	PLI_BYTE8 *	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 arguments 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 argument 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 arguments that are of the types described in Table 192. 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 *narg* is out of range, or if some other error is found, then 0 shall be returned.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

The **tf_nodeinfo()** and **tf_inodeinfo()** routines shall support at least the following Verilog data types as a system task or system function argument:

- scalar and vector regs
- scalar and vector nets
- integer, time and real variables
- word select of a one-dimensional reg, integer or time array
- null argument

The `s_tfnodeinfo` structure is defined in `veriusers.h` and is listed in Figure 166.

```

typedef struct t_tfnodeinfo
{
    PLI_INT16 node_type;
    PLI_INT16 padding;
    union
    {
        struct t_vecval *vecval_p;
        struct t_strengthval *strengthval_p;
        PLI_BYTE8 *memoryval_p;
        double *real_val_p;
    } node_value;
    PLI_BYTE8 *node_symbol;
    PLI_INT32 node_ngroups;
    PLI_INT32 node_vec_size;
    PLI_INT32 node_sign;
    PLI_INT32 node_ms_index;
    PLI_INT32 node_ls_index;
    PLI_INT32 node_mem_size;
    PLI_INT32 node_lhs_element;
    PLI_INT32 node_rhs_element;
    PLI_INT32 *node_handle;
} s_tfnodeinfo, *p_tfnodeinfo;

```

Figure 166—The s_tfnodeinfo structure definition

The following paragraphs define the fields of the s_tfnodeinfo structure.

The *node_type* field of the s_tfnodeinfo structure shall indicate the Verilog HDL data type of the argument, and is one of the predefined constants as given in Table 192 and defined in `veriusr.h`.

Table 192—Predefined constants for node_type

Predefined constant	Description
tf_null_node	Not a writable argument
tf_reg_node	Argument references a reg variable
tf_integer_node	Argument references an integer variable
tf_real_node	Argument references a real variable
tf_time_node	Argument references a time variable
tf_netvector_node	Argument references a vector net
tf_netscalar_node	Argument references a scalar net
tf_memory_node	Argument 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 argument. The union member accessed shall depend on the *node_type*. The union members are given in Table 193.

Table 193—How the `node_value` union is used

When the <code>node_type</code> is	The union member used is
<code>tf_reg_node</code> , <code>tf_integer_node</code> , <code>tf_time_node</code> , or <code>tf_netvector_node</code>	<code>vecval_p</code>
<code>tf_real_node</code>	<code>real_val_p</code>
<code>tf_netscalar_node</code>	<code>strengthval_p</code>
<code>tf_memory_node</code>	<code>memoryval_p</code>

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 `veruser.h` and is listed in Figure 167.

```
typedef struct t_vecval
{
    PLI_INT32 avalbits;
    PLI_INT32 bvalbits;
} s_vecval, *p_vecval;
```

Figure 167—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.vecval_p` array. For 33 bits to 64 bits, two groups shall be in the array, and so on. The number of groups shall also be 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 argument. The lsb in the value shall be represented by the lsb in the `avalbits` and `bvalbits` components, and so on. The bit coding shall be as given in Table 194.

Table 194—`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 168.

```
typedef struct t_strengthval
{
    PLI_INT32 strength0;
    PLI_INT32 strength1;
} s_strengthval, *p_strengthval;
```

Figure 168—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 169.

```
struct
{
    PLI_BYTE8 avalbits[node_ngroups];
    PLI_BYTE8 bvalbits[node_ngroups];
} memval[node_mem_size];
```

Figure 169—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 `PLI_BYTE8` type. The memory element with the lowest number address in the Verilog array 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 argument.

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 argument *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 *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.

If the *node_type* field of the `s_tfnodeinfo` structure is **tf_reg_node** or **tf_netvector_node**, then the *node_value.node_ms_element* and *node_value.node_ls_element* fields shall contain the msb and lsb of the given vector.

If the *node_type* field of the `s_tfnodeinfo` structure is **tf_reg_node** or **tf_netvector_node**, and the argument is a part-select, then the *node_value.node_rhs_index* and *node_value.node_lhs_index* fields shall contain the msb and lsb of the given part-select.

The field *node_handle* is not used.

25.36 tf_nump(), tf_inump()

tf_nump(), tf_inump()			
Synopsis:	Get number of task or function arguments.		
Syntax:	tf_nump() tf_inump(instance_p)		
Type		Description	
Returns:	PLI_INT32	The number of arguments	
Type		Name	Description
Arguments:	PLI_BYTE8 *	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 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.

Note: null arguments are counted. Therefore, \$foo() returns a count of 1 and \$foo(.) returns a count of 2. The routine **tf_typep()** returns a type of **tf_nullparam** for a null argument.

25.37 **tf_propagatep()**, **tf_ipropagatep()**

tf_propagatep(), tf_ipropagatep()			
Synopsis:	Propagate a system task/function argument value.		
Syntax:	tf_propagatep (narg) tf_ipropagatep (narg, instance_p)		
Type		Description	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
Arguments:	Type	Name	Description
	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	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 an argument expression value. Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function.		

The TF routines **tf_propagatep()** and **tf_ipropagatep()** shall write a value to an argument node of the current instance or a specific instance of a user-defined system task or function, and then propagate the value to any loads that read the value of the node.

In order to write values back into a Verilog software product data structure using **tf_propagatep()** 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 argument types except memories.

25.38 tf_putlongp(), tf_iputlongp()

tf_putlongp(), tf_iputlongp()			
Synopsis:	Write a 64-bit integer value to a system task/function argument or function return.		
Syntax:	tf_putlongp(narg, lowvalue, highvalue) tf_iputlongp(narg, lowvalue, highvalue, instance_p)		
Type		Description	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
Arguments:	Type	Name	Description
	PLI_INT32	narg	Index number of the user-defined system task or function argument or 0 to return a function value
	PLI_INT32	lowvalue	Least significant (right-most) 32 bits of value
	PLI_INT32	highvalue	Most significant (left-most) 32 bits of value
Related routines:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
	Use tf_putp() or tf_iputp() to put an argument value as a 32-bit integer		
	Use tf_putrealp() or tf_iputrealp() to put an argument 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 argument specified by *narg* of the current instance or a specific instance of a user-defined system task or function. If *narg* is 0, **tf_putlongp()** and **tf_iputlongp()** shall write the value as the return of a user-defined system function. If *narg* is out of range or the argument cannot be written to, then the routines shall do nothing. Should the calltf routine for a user defined system function fail to put a value during its execution, the default value of 0 shall be applied.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

The data type of the values to be written should be consistent with the type of put routine and the type of the argument to which the value shall be written. Refer to 24.3 for more details on proper data type selection with put routines.

NOTE calling put routines to TF argument 0 (return of a function) shall only return a value in a calltf application, when the call to the function is active. The action of the put routine shall be ignored when the function is not active.

25.39 tf_putp(), tf_iputp()

tf_putp(), tf_iputp()			
Synopsis:	Put an integer value to a system task/function argument or function return.		
Syntax:	tf_putp(narg, value) tf_iputp(narg, value, instance_p)		
Type		Description	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
Arguments:	Type	Name	Description
	PLI_INT32	narg	Index number of the user-defined system task or function argument or 0 to return a function value
	PLI_INT32	value	An integer value
	PLI_BYTE8 *	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 an argument value as a 64-bit integer Use tf_putrealp() or tf_iputrealp() to put an argument 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 argument specified by *narg* of the current instance or a specific instance of a user-defined system task or function. If *narg* is 0, **tf_putp()** or **tf_iputp()** shall write the value as the return of a user-defined system function. If *narg* is out of range or the argument cannot be written to, then the routines shall do nothing. Should the calltf routine for a user defined system function fail to put a value during its execution, the default value of 0 shall be applied.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

The data type of the value to be written should be consistent with the type of put routine and the type of the argument to which the value shall be written. Refer to Section 24.3 for more details on proper data type selection with put routines.

NOTE Calling put routines to TF argument 0 (return of a function) shall only return a value in a calltf application, when the call to the function is active. The action of the put routine shall be ignored when the function is not active.

25.40 tf_putrealp(), tf_iputrealp()

tf_putrealp(), tf_iputrealp()			
Synopsis:	Write a real value to a system task/function argument or function return.		
Syntax:	tf_putrealp(narg, value) tf_iputrealp(narg, value, instance_p)		
Type		Description	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
Arguments:	Type	Name	Description
	PLI_INT32	narg	Index number of the user-defined system task or function argument or 0 to return a function value
	double	value	A double-precision value
	PLI_BYTE8 *	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 an argument value as a 32-bit integer Use tf_putlongp() or tf_iputlongp() to put an argument 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 argument specified by *narg* of the current instance or a specific instance of a user-defined system task or function. If *narg* is 0, **tf_putrealp()** and **tf_iputrealp()** shall write the value as the return of a user-defined system function. If *narg* is out of range or the argument cannot be written to, then the routines shall do nothing. Should the calltf routine for a user defined system function fail to put a value during its execution, the default value of 0.0 shall be applied.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

The data type of the value to be written should be consistent with the type of put routine and the type of the argument to which the value shall be written. Refer to 24.3 for more details on proper data type selection with put routines.

NOTE calling put routines to TF argument 0 (return of a function) shall only return a value in a calltf application, when the call to the function is active. The action of the put routine shall be ignored when the function is not active.

25.41 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
		PLI_INT32	Nonzero if successful; zero if an error occurred
Arguments:	Type	Name	Description
	PLI_BYTE8 *	blockptr	Pointer to block of saved data
	PLI_INT32	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 misctf application when the misctf 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.

25.42 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_low, aof_high)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	double	realvalue	Value to be converted
	PLI_INT32 *	aof_low	Pointer to an integer variable for storing the least significant (right-most) 32 bits of the converted value
	PLI_INT32 *	aof_high	Pointer to an integer variable for storing the most significant (left-most) 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 64-bit integer. The converted value shall be returned in the variables pointed to by *aof_low* and *aof_high*.

25.43 tf_rosynchronize(), tf_irosynchronize()

tf_rosynchronize(), tf_irosynchronize()			
Synopsis:	Synchronize to end of simulation time step.		
Syntax:	tf_rosynchronize() tf_irosynchronize(instance_p)		
Type		Description	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
Type		Name	Description
Arguments:	PLI_BYTE8 *	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 routines **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()**.

25.44 tf_scale_longdelay()

tf_scale_longdelay()			
Synopsis:	Convert a 64-bit integer delay to the timescale of the module instance.		
Syntax:	tf_scale_longdelay(instance_p, delay_lo, delay_hi, aof_delay_lo, aof_delay_hi)		
Type		Description	
Returns:	void		
Arguments:	Type	Name	Description
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
	PLI_INT32	delay_lo	Least significant (right-most) 32 bits of the delay to be converted
	PLI_INT32	delay_hi	Most significant (left-most) 32 bits of the delay to be converted
	PLI_INT32 *	aof_delay_lo	Pointer to a variable to store the least significant (right-most) 32 bits of the conversion result
	PLI_INT32 *	aof_delay_hi	Pointer to a variable to store the most significant (left-most) 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 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.

25.45 tf_scale_realdelay()

tf_scale_realdelay()			
Synopsis:	Convert a double-precision floating-point delay to the timescale of the module instance.		
Syntax:	tf_scale_realdelay(instance_p, realdelay, aof_realdelay)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	PLI_BYTE8 *	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.

25.46 tf_setdelay(), tf_isetdelay()

tf_setdelay(), tf_isetdelay()			
Synopsis:	Activate the misctf application at a particular simulation time.		
Syntax:	tf_setdelay(delay) tf_isetdelay(delay, instance_p)		
Type		Description	
Returns:	PLI_INT32	1 if successful; 0 if an error occurred	
Arguments:	Type	Name	Description
	PLI_INT32	delay	32-bit integer delay time
	PLI_BYTE8 *	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 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_setdelay()** and **tf_isetdelay()** routines can be called several times with different delays, and several reactivations shall be scheduled. Multiple calls to **tf_setdelay()** and **tf_isetdelay()** for the same time step are permitted and shall result in multiple calls to the misctf application for that time step.

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.

25.47 **tf_setlongdelay()**, **tf_isetlongdelay()**

tf_setlongdelay(), tf_isetlongdelay()			
Synopsis:	Activate the misctf application at a particular simulation time.		
Syntax:	tf_setlongdelay (lowdelay, highdelay) tf_isetlongdelay (lowdelay, highdelay, instance_p)		
Type		Description	
Returns:	PLI_INT32	1 if successful; 0 if an error occurred	
Type		Name	Description
Arguments:	PLI_INT32	lowdelay	Least significant (right-most) 32 bits of the delay time to reactivation
	PLI_INT32	highdelay	Most significant (left-most) 32 bits of the delay time to reactivation
	PLI_BYTE8 *	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 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 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. Multiple calls to **tf_setlongdelay()** and **tf_isetlongdelay()** for the same time step are permitted and shall result in multiple calls to the misctf application for that time step.

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.

25.48 tf_setrealdelay(), tf_isetrealdelay()

tf_setrealdelay(), tf_isetrealdelay()			
Synopsis:	Activate the <i>misctf</i> application at a particular simulation time.		
Syntax:	tf_setrealdelay(realdelay) tf_isetrealdelay(realdelay, instance_p)		
Returns:		Type	Description
		PLI_INT32	1 if successful; 0 if an error occurred
Arguments:	Type	Name	Description
	double	realdelay	Double-precision delay time to reactivation
	PLI_BYTE8 *	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. Multiple calls to **tf_setrealdelay()** and **tf_isetrealdelay()** for the same time step are permitted and shall result in multiple calls to the *misctf* application for that time step.

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.

25.49 tf_setworkarea(), tf_isetworkarea()

tf_setworkarea(), tf_isetworkarea()			
Synopsis:	Store user data pointer in work area.		
Syntax:	tf_setworkarea(workarea) tf_isetworkarea(workarea, instance_p)		
Type		Description	
Returns:	PLI_INT32	Always returns 0	
Type		Name	Description
Arguments:	PLI_BYTE8 *	workarea	Pointer to user data
	PLI_BYTE8 *	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 one type of PLI application to another, such as from a checktf application to a calltf application

Note that the workarea pointer is a *PLI_BYTE8 ** type. If the memory allocated for the user data is of some other type, it should be cast to *PLI_BYTE8 **.

25.50 **tf_sizep()**, **tf_isizep()**

tf_sizep(), tf_isizep()			
Synopsis:	Get the bit length of a system task/function argument.		
Syntax:	<pre>tf_sizep(narg) tf_isizep(narg, instance_p)</pre>		
Type		Description	
Returns:	PLI_INT32	The number of bits of the system task/function argument	
Arguments:	Type	Name	Description
	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	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 argument in the current instance or a specific instance of a user-defined system task or function.

If the specified argument is a literal string, **tf_sizep()** and **tf_isizep()** shall return the string length.

If the specified argument is real or if an error is detected, **tf_sizep()** and **tf_isizep()** shall return 0.

The argument *narg* shall be the index number of an ARGUMENT in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

25.51 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:	PLI_BYTE8 *	Pointer to a character string with the hierarchical path name	
Type		Name	Description
Arguments:	PLI_BYTE8 *	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.

25.52 tf_strdelputp(), tf_istrdelputp()

tf_strdelputp(), tf_istrdelputp()			
Synopsis:	Write a value to a system task/function argument from string value specification, using a 32-bit integer delay.		
Syntax:	tf_strdelputp(narg, bitlength, format, value_p, delay, delaytype) tf_istrdelputp(narg, bitlength, format, value_p, delay, delaytype, instance_p)		
Type		Description	
Returns:	PLI_INT32	1 if successful; 0 if an error is detected	
Arguments:	Type	Name	Description
	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_INT32	bitlength	Number of bits the value represents
	PLI_INT32	format	A character in single quotes representing the radix (base) of the value
	quoted string or PLI_BYTE8 *	value_p	Quoted character string or pointer to a character string with the value to be written
	PLI_INT32	delay	Integer value representing the time delay before the value should be written to the argument
	PLI_INT32	delaytype	Integer code representing the delay mode for applying the value
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	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 argument of the current instance or a specific instance of a user-defined system task or function. The actual change to the argument shall be scheduled as an event on the argument in the Verilog model at a future simulation time. An argument value of 0 (system function return) shall be illegal.

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 195.

Table 195—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 argument, 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 reg or variable. The *delaytype* shall be one of integer values shown in Table 196.

Table 196—delaytype codes

delaytype code	Definition	Description
0	Inertial delay	All scheduled events on the output argument 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 argument in the Verilog model are removed before scheduling a new event
2	Pure transport delay	No scheduled events on the output argument 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

25.53 tf_strgetp(), tf_istrgetp()

tf_strgetp(), tf_istrgetp()			
Synopsis:	Get formatted system task/function argument values.		
Syntax:	tf_strgetp(narg, format) tf_istrgetp(narg, format, instance_p)		
Type		Description	
Returns:	PLI_BYTE8 *	Pointer to a character string with the argument value	
Arguments:	Type	Name	Description
	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_INT32	format	Character in single quotes controlling the return value format
	PLI_BYTE8 *	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 an argument value as a 32-bit integer Use tf_getlongp() or tf_igetlongp() to get an argument value as a 64-bit integer Use tf_getrealp() or tf_igetrealp() to get an argument value as a double Use tf_getcstringp() or tf_igetcstringp() to get an argument 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 argument 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 197.

Table 197—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 argument 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.

25.54 tf_strgettime()

tf_strgettime()		
Synopsis:	Get the current simulation time as a string.	
Syntax:	tf_strgettime()	
	Type	Description
Returns:	PLI_BYTE8 *	Pointer to a character string with the simulation time
	Type	Name
Arguments:		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_gettime() 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.

Time shall be expressed in simulation time units, which is the smallest time precision used by all modules in a design.

25.55 tf_strlongdelpu(), tf_istrlongdelpu()

tf_strlongdelpu(), tf_istrlongdelpu()			
Synopsis:	Write a value to a system task/function argument from string value specification, using a 64-bit integer delay.		
Syntax:	tf_strlongdelpu(narg, bitlength, format, value_p, lowdelay, highdelay, delaytype) tf_istrlongdelpu(narg, bitlength, format, value_p, lowdelay, highdelay, delaytype, instance_p)		
Type		Description	
Returns:	PLI_INT32	1 if successful; 0 if an error is detected	
Arguments:	Type	Name	Description
	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_INT32	bitlength	Number of bits the value represents
	PLI_INT32	format	A character in single quotes representing the radix (base) of the value
	quoted string or PLI_BYTE8 *	value_p	Quoted character string or pointer to a character string with the value to be written
	PLI_INT32	lowdelay	Least significant (right-most) 32 bits of delay before the value is be written to the argument
	PLI_INT32	highdelay	Most significant (left-most) 32 bits of delay before the value is be written to the argument
	PLI_INT32	delaytype	Integer code representing the delay mode for applying the value
PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function	
Related routines:	Use tf_strdelpu() or tf_istrdelpu() for 32-bit integer delays Use tf_strrealdelpu() or tf_istrrealdelpu() 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_strlongdelpu()** and **tf_istrlongdelpu()** shall write a string value to the specified argument of the current instance or a specific instance of a user-defined system task or function. The actual change to the argument shall be scheduled as an event on the argument in the Verilog model at a future simulation time. An argument value of 0 (system function return) shall be illegal.

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 198.

Table 198—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 argument, 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 reg or variable. The *delaytype* shall be one of integer values shown in Table 199.

Table 199—delaytype codes

delaytype code	Definition	Description
0	Inertial delay	All scheduled events on the output argument 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 argument in the Verilog model are removed before scheduling a new event
2	Pure transport delay	No scheduled events on the output argument 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

tf_strrealdelputp(), tf_istrrealdelputp()			
Synopsis:	Write a value to a system task/function argument from string value specification, using a real number delay.		
Syntax:	tf_strrealdelputp(narg, bitlength, format, value_p, realdelay, delaytype) tf_istrrealdelputp(narg, bitlength, format, value_p, realdelay, delaytype, instance_p)		
	Type	Description	
Returns:	PLI_INT32	1 if successful; 0 if an error is detected	
	Type	Name	Description
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_INT32	bitlength	Number of bits the value represents
	PLI_INT32	format	A character in single quotes representing the radix (base) of the value
	quoted string or PLI_BYTE8 *	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 argument
	PLI_INT32	delaytype	Integer code representing the delay mode for applying the value
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	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 *bitlength* argument shall define the value size in bits.

Table 200—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 argument, 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 reg or variable. The *delaytype* shall be one of integer values shown in Table 201.

Table 201—delaytype codes

delaytype code	Definition	Description
0	Inertial delay	All scheduled events on the output argument 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 argument in the Verilog model are removed before scheduling a new event
2	Pure transport delay	No scheduled events on the output argument 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

25.57 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:	PLI_INT32	Always returns 0	
Arguments:	Type	Name	Description
	PLI_INT32 *	aof_low1	Pointer to least significant 32 bits of first operand
	PLI_INT32 *	aof_high1	Pointer to most significant 32 bits of first operand
	PLI_INT32	low2	Least significant 32 bits of second operand
	PLI_INT32	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 170 shows the high and low 32 bits of two 64-bit integers and how **tf_subtract_long()** shall subtract them.

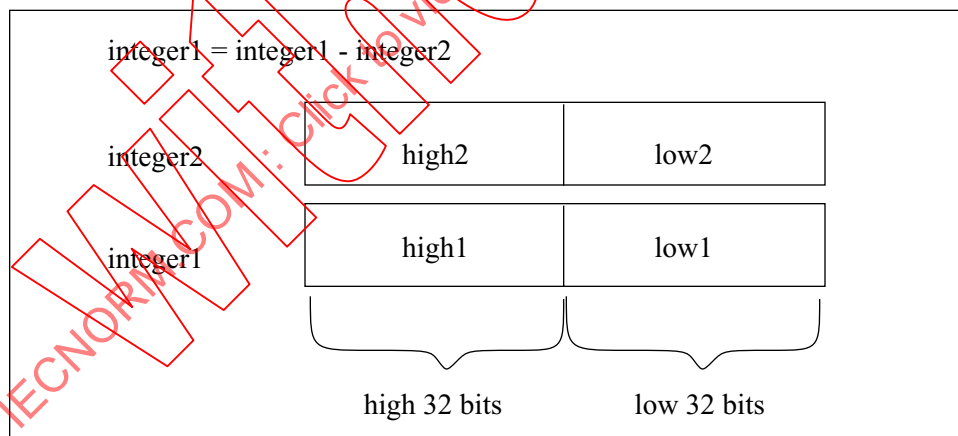


Figure 170—Subtracting with tf_subtract_long()

The example program fragment shown in Figure 171 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 between the two **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.

```
{
    PLI_INT32 currlow, currhight;
    PLI_INT32 relalow, relahigh;

    currlow = tf_getlongtime(&currhight);
    io_printf("At time %s: ", tf_longtime_tostr(currlow, currhight));
    if(tf_getnextlongtime(&relalow, &relahight) == 0)
    {
        tf_subtract_long(&relalow, &relahight, currlow, currhight);
        io_printf("relative time to next event is %s",
            tf_longtime_tostr(relalow, relahight));
    }
    else
        printf("there are no future events");
}
```

Figure 171—Using `tf_subtract_long()`

25.58 tf_synchronize(), tf_isynchronize()

tf_synchronize(), tf_isynchronize()			
Synopsis:	Synchronize to end of simulation time step.		
Syntax:	tf_synchronize() tf_isynchronize(instance_p)		
Type		Description	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
Type		Name	Description
Arguments:	PLI_BYTE8 *	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		

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()**.

25.59 tf_testpvc_flag(), tf_itestpvc_flag()

tf_testpvc_flag(), tf_itestpvc_flag()			
Synopsis:	Test system task/function argument value change flags.		
Syntax:	tf_testpvc_flag(narg) tf_itestpvc_flag(narg, instance_p)		
Returns:	Type	Description	
	PLI_INT32	The value of the saved pvc flag	
Arguments:	Type	Name	Description
	PLI_INT32	narg	Index number of the user-defined system task or function argument, or -1
	PLI_BYTE8 *	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 argument 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 *narg* shall be the index number of an argument in a specific instance of a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1. If *narg* is **-1**, then all argument 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 argument has changed value. Each argument 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.

25.60 tf_text()

tf_text()			
Synopsis:	Store error message information.		
Syntax:	<code>tf_text(message, arg1, ...arg5)</code>		
Type		Description	
Returns:	PLI_INT32	Always returns 0	
Type		Name	Description
Arguments: (optional)	quoted string or PLI_BYTE8 *	message	A quoted character string or pointer to a character string with a message to be stored
		arg1...arg5	One to five optional arguments of the format control string; the type of each argument should be consistent with how it is used in the message string
Related routines:	Use <code>tf_message()</code> 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
```

25.61 tf_typep(), tf_itypep()

tf_typep(). tf_itypep()			
Synopsis:	Get a system task/function argument type.		
Syntax:	tf_typep(narg) tf_itypep(narg, instance_p)		
Type		Description	
Returns:	PLI_INT32	A predefined integer constant representing the Verilog HDL data type for the argument	
Type		Name	Description
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	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 an argument 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 202.

Table 202—Predefined tf_typep() constants

Predefined constant	Description
tf_nullparam	The argument is a <i>null</i> expression (where no text has been given as the argument), or <i>narg</i> is out of range
tf_string	The argument is a literal string
tf_readonly	The argument is a expression with a value that can be read but not written
tf_readwrite	The argument is a expression with a value that can be read and written
tf_readonlyreal	The argument is a real number expression with a value that can be read but not written
tf_readwritereal	The argument 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 *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

25.62 tf_unscale_longdelay()

tf_unscale_longdelay()			
Synopsis:	Convert a delay from internal simulation time units to the timescale of a particular module.		
Syntax:	tf_unscale_longdelay (<i>instance_p</i> , <i>delay_lo</i> , <i>delay_hi</i> , <i>aof_delay_lo</i> , <i>aof_delay_hi</i>)		
Type		Description	
Returns:	void		
Arguments:	Type	Name	Description
	PLI_BYTE8 *	<i>instance_p</i>	Pointer to a specific instance of a user-defined system task or function
	PLI_INT32	<i>delay_lo</i>	Least significant (right-most) 32 bits of the delay to be converted
	PLI_INT32	<i>delay_hi</i>	Most significant (left-most) 32 bits of the delay to be converted
	PLI_INT32 *	<i>aof_delay_lo</i>	Pointer to a variable to store the least significant (right-most) 32 bits of the conversion result
	PLI_INT32 *	<i>aof_delay_hi</i>	Pointer to a variable to store the most significant (left-most) 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 timescale of the module instance 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 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.

25.63 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
	PLI_BYTE8 *	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 timescale of the module instance 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.

25.64 tf_warning()

tf_warning()			
Synopsis:	Report a warning message.		
Syntax:	tf_warning(format, arg1, ...arg5)		
Type		Description	
Returns:	PLI_INT32	Always returns 0	
Type		Name	Description
Arguments: (optional)	quoted string or PLI_BYTE8 *	format	A quoted character string or pointer to a character string that controls the message to be written
		arg1...arg5	One to five optional arguments of 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 an error 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 output channel of the software product which invoked the PLI application and the output log file of the product.

The **tf_warning()** routine shall not abort parsing or compilation of Verilog HDL source code.

25.65 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	
	PLI_INT32	Nonzero value if successful, zero if an error is encountered	
Arguments:	Type	Name	Description
	PLI_BYTE8 *	blockptr	Pointer to the first byte of the block of data to be saved
	PLI_INT32	blocklen	Number of bytes are to be saved
Related routines:	Use tf_read_restart() 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 misctf application when misctf 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_restart()** as were written with **tf_write_save()**.

26. Using VPI routines

Clause 26 and Clause 27 specify the Verilog Procedural Interface (VPI) for the Verilog HDL. This clause describes how the VPI routines are used, and Section 27 defines each of the routines in alphabetical order.

26.1 VPI system tasks and functions

User defined system tasks and functions are created using the routine **vpi_register_systf()** (see 27.34). The registration of system tasks must occur prior to elaboration or the resolution of references.

The intended use model would be to place a reference to a routine within the **vlog_startup_routines[]** array. This routine would register all user defined system tasks and functions when it is called.

VPI system tasks have *compiletf*, *sizebf*, and *calltf* routines which have the same use model as the corresponding *checktf*, *sizebf* and *calltf* routines in the TF interface mechanism for user defined system tasks and functions (refer to Clause 21). The functionality provided in the TF interface mechanism for the *misctf* routine is supported via a set of callbacks, which can be registered using **vpi_register_cb()**.

26.2 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

26.2.1 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.

26.2.2 VPI access to Verilog HDL objects and simulation objects

Accessible Verilog HDL objects and simulation objects and their relationships and properties are described using data model diagrams. These diagrams are presented in 26.6. The data model diagrams indicate the routines and constants that are required to access and manipulate objects within an application environment. An associated set of routines to access these objects is defined in Clause 27.

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 Table 211.

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.

26.2.3 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.

26.2.4 Function availability

Certain features of the VPI interface must occur early in the execution of a tool. In order to allow this process to occur in an orderly manner, some functionality must be restricted in these early stages. Specifically, when the routines within the **vlog_startup_routines[]** array are executed, there is very little functionality available. Only two routines can be called at this time:

- **vpi_register_systf()**
- **vpi_register_cb()**

In addition, the **vpi_register_cb()** routine can only be called for the following reasons:

- **cbEndOfCompile**
- **cbStartOfSimulation**
- **cbEndOfSimulation**
- **cbUnresolvedSystf**
- **cbError**
- **cbPLIError**

Refer to 27.34 for a further explanation of the use of the **vlog_startup_routines[]** array.

The next earliest phase is when the **sizetf** routines are called for the user defined system functions. At this phase, no additional access is permitted. After the **sizetf** routines are called, the routines registered for reason **cbEndOfCompile** are called. At this point, and continuing until the tool has finished execution, all functionality is available.

26.2.5 Traversing expressions

The VPI routines provide access to any expression which can be written in the HDL. Dealing with these expressions can be complex, since very complex expressions can be written in the HDL. Expressions with