Verilog-AMS
Language Reference Manual

VAMS-2023

Feb 14, 2024

Copyright© 2024 Accellera Systems Initiative. All rights reserved.

Accellera Systems Initiative Inc., 8698 Elk Grove Blvd. Suite 1, #114, Elk Grove, CA 95624, USA

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

Notices

Accellera Systems Initiative (Accellera) standards documents are developed within Accellera by its Technical Committee. Accellera develops its standards through a consensus development process, approved by its members and board of directors, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are members of Accellera and serve without compensation. While Accellera administers the process and establishes rules to promote fairness in the consensus development process, Accellera does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

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

Accellera does not warrant or represent the accuracy or content of the material contained herein, and expressly dis- claims any express or implied warranty, including any implied warranty of merchantability or suitability for a specific purpose, or that the use of the material contained herein is free from patent infringement. Accellera standards documents are supplied "**AS IS.**"

The existence of an Accellera 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 an Accellera standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change due to developments in the state of the art and comments received from users of the standard. Every Accellera standard is subjected to review periodically for revision and update. Users are cautioned to check to determine that they have the latest edition of any Accellera standard.

In publishing and making this document available, Accellera is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is Accellera undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other Accellera 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 Accellera, Accellera will initiate action to prepare appropriate responses. Since Accellera 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, Accellera and the members of its Technical Committee are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Comments for revision of Accellera standards are welcome from any interested party, regardless of membership affiliation with Accellera. 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:

Accellera Systems Initiative Inc.
8698 Elk Grove Blvd.
Suite 1, #114
Elk Grove, CA 95624
USA

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

Accellera is the sole entity that may authorize the use of Accellera-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein.

Authorization to reuse portions of any Accellera standard for any purpose other than internal or personal use must be granted by Accellera, provided that permission is obtained from and any required fee is paid to Accellera. To arrange for authorization please contact Lynn Garibaldi, Accellera, 8698 Elk Grove Blvd, Suite 1, #114, Elk Grove, CA, 95624, , e-mail lynn@accellera.org. Permission to copy portions of an Accellera standard for educational or classroom use can also be obtained from Accellera.

Suggestions for improvements to the Verilog-AMS Language Reference Manual are welcome. They should be sent to the Verilog-AMS e-mail reflector

sv-ams@lists.accellera.org

The following people contributed to the creation, editing, and review of this document.
Peter Grove, Renesas Electronics, *Chair*
David Miller, NXP Semiconductor, *Technical Editor, Secretary*

Boon Chong Ang, Intel Corporation
Adnan Assar, Siemens EDA
Lakshmanan Balasubramanian, Texas Instruments
Luis Humberto Rezende Barbosa, Cadence Design Systems, Inc.
Shalom Bresticker, Accellera
Jerry Chang, Texas Instruments
Shekar Chetput, Cadence Design Systems, Inc.
Geoffrey Coram, Analog Devices
Dave Cronauer, Synopsys
Hardik Parekh, NXP Semiconductor
Bob Pau, Qualcomm
Rock Shi, Allegro Microsystems
Aaron Spratt, Cadence Design Systems, Inc.
Evgeny Vlasov, Synopsys
Mina Zaki, Siemens EDA

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The following people have made contributions to previous versions of this document.

Ramana Aisola
Andre Baguenier
Kenneth Bakalar
Jim Barby
Martin Barnasconi
Graham Bell
William Bell
Xavier Bestel
Kevin Cameron
James Cavanaugh
Srikanth Chandrasekaran
Ed Chang
Chandrashekar Chetput
Joe Daniels
Jonathan David
Al Davis
Raphael Dorado
John Downey
Dan FitzPatrick
Bob Floyd
Paul Floyd
Vassilios Gerousis
Ian Getreu
Kim Hailey
Steve Hamm
Graham Helwig
William Hobson
Junwei Hou
Robert Hughes
Dick Klaassen
Marq Kole
Abhi Kolpekwar
Ken Kundert
Laurent Lemaitre
Top Lertpanyavit
Oskar Leuthold
S. Peter Liebmann
Scott Little
Colin McAndrew
Steve Meyer
Marek Mierzwinski
Ira Miller
Michael Mirmak
John Moore
Scott Morrison
Arpad Muranyi
Patrick O'Halloran
Don O'Riordan
Jeroen Paasschens
Rick Poore
Farzin Rasteh
Tom Reeder
Steffen Rochel
Jon Sanders
David Sharrit
John Shields
James Spoto
Stuart Sutherland
Prasanna Tamhankar
George Tipple
Richard Trihy
Yatin Trivedi
Boris Troyanovsky
Alessandro Valerio
Martin Vlach
Don Webber
Frank Weiler
Ian Wilson
Ilya Yusim
Alex Zamfirescu
Amir Zarkesh
David Zweidinger

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Contents

1. Verilog-AMS introduction 1
1.1 Overview 1
1.2 Mixed-signal language features 1
1.3 Systems 2
1.3.1 Conservative systems 2
1.3.2 Kirchhoff's Laws 3
1.3.3 Natures, disciplines, and nets 4
1.3.4 Signal-flow systems 5
1.3.5 Mixed conservative/signal flow systems 6
1.4 Conventions used in this document 8
1.5 Contents 8
2. Lexical conventions 11
2.1 Overview 11
2.2 Lexical tokens 11
2.3 White space 11
2.4 Comments 11
2.5 Operators 11
2.6 Numbers 12
2.6.1 Integer constants 13
2.6.2 Real constants 15
2.7 String literals 16
2.8 Identifiers, keywords, and system names 17
2.8.1 Escaped identifiers 17
2.8.2 Keywords 17
2.8.3 System tasks and functions 17
2.8.4 Compiler directives 18
2.9 Attributes 19
2.9.1 Syntax 20
2.9.2 Standard attributes 23
3. Data types 24
3.1 Overview 24
3.2 Integer and real data types 24
3.2.1 Output variables 25
3.3 String data type 25
3.4 Parameters 27
3.4.1 Type specification 28
3.4.2 Value range specification 29
3.4.3 Parameter units and descriptions 30
3.4.4 Parameter arrays 30
3.4.5 Local parameters 30
3.4.6 String parameters 30
3.4.7 Parameter aliases 31
3.4.8 Multidimensional parameter array examples 32
3.5 Genvars 33
3.6 Net_discipline 34
3.6.1 Natures 34
3.6.2 Disciplines 37
3.6.3 Net_discipline declaration 41
3.6.4 Ground declaration 43
3.6.5 Implicit nets 43
3.7 Real net declarations 44

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

3.8 Default discipline ... 45
3.9 Disciplines of primitives ... 45
3.10 Discipline precedence ... 45
3.11 Net compatibility ... 45
3.11.1 Discipline and Nature Compatibility ... 46
3.12 Branches ... 48
3.12.1 Port Branches ... 49
3.13 Namespace ... 50
3.13.1 Nature and discipline ... 50
3.13.2 Access functions ... 50
3.13.3 Net ... 50
3.13.4 Branch ... 50
4. Expressions ... 51
4.1 Overview ... 51
4.2 Operators ... 51
4.2.1 Operators with real operands ... 52
4.2.2 Operator precedence ... 53
4.2.3 Expression evaluation order ... 54
4.2.4 Arithmetic operators ... 55
4.2.5 Relational operators ... 56
4.2.6 Case equality operators ... 56
4.2.7 Logical equality operators ... 56
4.2.8 Logical operators ... 57
4.2.9 Bitwise operators ... 57
4.2.10 Reduction operators ... 58
4.2.11 Shift operators ... 58
4.2.12 Conditional operator ... 59
4.2.13 Concatenations ... 59
4.2.14 Assignment patterns ... 60
4.3 Built-in mathematical functions ... 61
4.3.1 Standard mathematical functions ... 61
4.3.2 Transcendental functions ... 62
4.4 Signal access functions ... 63
4.5 Analog operators ... 64
4.5.1 Vector or array arguments to analog operators ... 65
4.5.2 Analog operators and equations ... 65
4.5.3 Time derivative operator ... 66
4.5.4 Time integral operator ... 66
4.5.5 Circular integrator operator ... 68
4.5.6 Derivative operator ... 69
4.5.7 Absolute delay operator ... 71
4.5.8 Transition filter ... 72
4.5.9 Slew filter ... 78
4.5.10 last_crossing function ... 79
4.5.11 Laplace transform filters ... 80
4.5.12 Z-transform filters ... 82
4.5.13 Limited exponential ... 84
4.5.14 Constant versus dynamic arguments ... 85
4.5.15 Restrictions on analog operators ... 86
4.6 Analysis dependent functions ... 86
4.6.1 Analysis ... 87
4.6.2 DC analysis ... 88
4.6.3 AC stimulus ... 88
4.6.4 Noise ... 88

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

4.7 User-defined functions ... 92
4.7.1 Defining an analog user-defined function ... 92
4.7.2 Returning a value from an analog user-defined function ... 94
4.7.3 Calling an analog user-defined function ... 95
5. Analog behavior ... 97
5.1 Overview ... 97
5.2 Analog procedural block ... 97
5.2.1 Analog initial block ... 97
5.3 Block statements ... 98
5.3.1 Sequential blocks ... 98
5.3.2 Block names ... 98
5.4 Analog signals ... 99
5.4.1 Access functions ... 99
5.4.2 Probes and sources ... 100
5.4.3 Accessing flow through a port ... 101
5.4.4 Unassigned sources ... 102
5.5 Accessing net and branch signals and attributes ... 102
5.5.1 Accessing net and branch signals ... 102
5.5.2 Signal access for vector branches ... 104
5.5.3 Accessing attributes ... 105
5.5.4 Creating unnamed branches using hierarchical net references ... 106
5.5.5 Accessing nets and branch signals hierarchically ... 106
5.6 Contribution statements ... 107
5.6.1 Direct branch contribution statements ... 107
5.6.2 Examples ... 111
5.6.3 Resistor and conductor ... 111
5.6.4 RLC circuits ... 112
5.6.5 Switch branches ... 112
5.6.6 Implicit Contributions ... 113
5.6.7 Indirect branch contribution statements ... 114
5.6.8 Contributing hierarchically ... 116
5.7 Analog procedural assignments ... 117
5.8 Analog conditional statements ... 118
5.8.1 if-else-if statement ... 118
5.8.2 Examples ... 119
5.8.3 Case statement ... 119
5.8.4 Restrictions on conditional statements ... 120
5.9 Looping statements ... 120
5.9.1 Repeat and while statements ... 120
5.9.2 For statements ... 121
5.9.3 Analog For Statements ... 121
5.10 Analog event control statements ... 122
5.10.1 Event OR operator ... 124
5.10.2 Global events ... 124
5.10.3 Monitored events ... 126
5.10.4 Named events ... 132
5.10.5 Digital events in analog behavior ... 133
5.11 Jump statements ... 133
6. Hierarchical structures ... 134
6.1 Overview ... 134
6.2 Modules ... 134
6.2.1 Top-level modules and $root ... 136
6.2.2 Module instantiation ... 136

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

6.3 Overriding module parameter values ... 138
6.3.1 Defparam statement ... 138
6.3.2 Module instance parameter value assignment by order ... 140
6.3.3 Module instance parameter value assignment by name ... 140
6.3.4 Parameter dependence ... 141
6.3.5 Detecting parameter overrides ... 141
6.3.6 Hierarchical system parameters ... 141
6.4 Paramsets ... 143
6.4.1 Paramset statements ... 144
6.4.2 Paramset overloading ... 145
6.4.3 Paramset output variables ... 147
6.5 Ports ... 147
6.5.1 Port definition ... 147
6.5.2 Port declarations ... 148
6.5.3 Real valued ports ... 149
6.5.4 Connecting module ports by ordered list ... 150
6.5.5 Connecting module ports by name ... 151
6.5.6 Detecting port connections ... 152
6.5.7 Port connection rules ... 152
6.5.8 Inheriting port natures ... 152
6.6 Generate constructs ... 152
6.6.1 Loop generate constructs ... 154
6.6.2 Conditional generate constructs ... 157
6.6.3 External names for unnamed generate blocks ... 159
6.7 Hierarchical names ... 160
6.7.1 Usage of hierarchical references ... 161
6.8 Scope rules ... 162
6.9 Elaboration ... 162
6.9.1 Concatenation of analog blocks ... 162
6.9.2 Elaboration and paramsets ... 163
6.9.3 Elaboration and connectmodules ... 163
6.9.4 Order of elaboration ... 163
7. Mixed signal ... 164
7.1 Overview ... 164
7.2 Fundamentals ... 164
7.2.1 Domains ... 164
7.2.2 Contexts ... 165
7.2.3 Nets, nodes, ports, and signals ... 165
7.2.4 Mixed-signal and net disciplines ... 166
7.3 Behavioral interaction ... 166
7.3.1 Accessing discrete nets and variables from a continuous context ... 167
7.3.2 Accessing X and Z bits of a discrete net in a continuous context ... 168
7.3.3 Accessing continuous nets and variables from a discrete context ... 169
7.3.4 Detecting discrete events in a continuous context ... 170
7.3.5 Detecting continuous events in a discrete context ... 171
7.3.6 Concurrency ... 172
7.3.7 Function calls ... 173
7.4 Discipline resolution ... 173
7.4.1 Compatible discipline resolution ... 174
7.4.2 Connection of discrete-time disciplines ... 174
7.4.3 Connection of continuous-time disciplines ... 175
7.4.4 Resolution of mixed signals ... 175
7.4.5 Discipline resolution of continuous signals ... 177
7.5 Connect modules ... 178

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

7.6 Connect module descriptions ... 178
7.7 Connect specification statements ... 179
7.7.1 Connect module auto-insertion statement ... 180
7.7.2 Discipline resolution connect statement ... 180
7.7.3 Parameter passing attribute ... 182
7.7.4 connect_mode ... 182
7.8 Automatic insertion of connect modules ... 182
7.8.1 Connect module selection ... 183
7.8.2 Signal segmentation ... 185
7.8.3 connect_mode parameter ... 187
7.8.4 Rules for driver-receiver segregation and connect module selection and insertion ... 190
7.8.5 Instance names for auto-inserted instances ... 191
7.8.6 Supply sensitive connect module examples ... 192
7.9 Driver-receiver segregation ... 198
8. Scheduling semantics ... 200
8.1 Overview ... 200
8.2 Simulation initialization ... 200
8.3 Analog simulation cycle ... 201
8.3.1 Nodal analysis ... 202
8.3.2 Transient analysis ... 202
8.3.3 Convergence ... 203
8.4 Mixed-signal simulation cycle ... 204
8.4.1 Circuit initialization ... 204
8.4.2 Mixed-signal DC analysis ... 204
8.4.3 Mixed-signal transient analysis ... 205
8.4.4 The synchronization loop ... 209
8.4.5 Synchronization and communication algorithm ... 211
8.4.6 absdelta interpolated A2D events ... 212
8.4.7 Assumptions about the analog and digital algorithms ... 213
8.5 Scheduling semantics for the digital engine ... 213
8.5.1 The stratified event queue ... 214
8.5.2 The Verilog-AMS digital engine reference model ... 215
8.5.3 Scheduling implication of assignments ... 215
9. System tasks and functions ... 218
9.1 Overview ... 218
9.2 Categories of system tasks and functions ... 218
9.3 System tasks/functions executing in the context of the Analog Simulation Cycle ... 225
9.4 Display system tasks ... 225
9.4.1 Behavior of the display tasks in the analog context ... 225
9.4.2 Escape sequences for special characters ... 226
9.4.3 Format specifications ... 227
9.4.4 Hierarchical name format ... 228
9.4.5 String format ... 228
9.4.6 Behavior of the display tasks in the analog block during iterative solving ... 228
9.4.7 Extensions to the display tasks in the digital context ... 228
9.5 File input-output system tasks and functions ... 228
9.5.1 Opening and closing files ... 228
9.5.2 File output system tasks ... 230
9.5.3 Formatting data to a string ... 231
9.5.4 Reading data from a file ... 231
9.5.5 File positioning ... 233
9.5.6 Flushing output ... 234
9.5.7 I/O error status ... 234

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

9.5.8 Detecting EOF ... 234
9.5.9 Behavior of the file I/O tasks in the analog block during iterative solving ... 235
9.6 Timescale system tasks ... 235
9.7 Simulation control system tasks ... 235
9.7.1 $finish ... 235
9.7.2 $stop ... 236
9.7.3 $fatal, $error, $warning, and $info ... 236
9.8 PLA modeling system tasks ... 237
9.9 Stochastic analysis system tasks ... 237
9.10 Simulator time system functions ... 237
9.11 Conversion system functions ... 237
9.12 Command line input ... 237
9.13 Probabilistic distribution system functions ... 238
9.13.1 $random and $arandom ... 238
9.13.2 Distribution functions ... 239
9.13.3 Algorithm for probabilistic distribution ... 240
9.14 Math system functions ... 241
9.15 Analog kernel parameter system functions ... 241
9.16 Dynamic simulation probe function ... 243
9.17 Analog kernel control system tasks and functions ... 244
9.17.1 $discontinuity ... 244
9.17.2 $bound_step task ... 246
9.17.3 $limit ... 246
9.18 Hierarchical parameter system functions ... 248
9.19 Explicit binding detection system functions ... 251
9.20 Analog node alias system functions ... 252
9.21 Table based interpolation and lookup system function ... 254
9.21.1 Table data source ... 257
9.21.2 Control string ... 258
9.21.3 Example control strings ... 260
9.21.4 Interpolation algorithms ... 260
9.21.5 Example ... 261
9.22 Connectmodule driver access system functions and operator ... 262
9.22.1 $driver_count ... 262
9.22.2 $receiver_count ... 262
9.22.3 $driver_state ... 262
9.22.4 $driver_strength ... 263
9.22.5 driver_update ... 263
9.22.6 Receiver net resolution ... 264
9.22.7 Connect module example using driver access functions ... 264
9.23 Supplementary connectmodule driver access system functions ... 266
9.23.1 $driver_delay ... 266
9.23.2 $driver_next_state ... 266
9.23.3 $driver_next_strength ... 266
9.23.4 $driver_type ... 267
10. Compiler directives ... 268
10.1 Overview ... 268
10.2 `default_discipline ... 268
10.3 `default_transition ... 269
10.4 `define and `undef ... 270
10.5 Predefined macros ... 270
10.6 `begin_keywords and `end_keywords ... 271
10.7 `__FILE__ and `__LINE__ ... 273

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11. Using VPI routines ... 274
11.1 Overview ... 274
11.2 The VPI interface ... 274
11.2.1 VPI callbacks ... 274
11.2.2 VPI access to Verilog-AMS HDL objects and simulation objects ... 274
11.2.3 Error handling ... 275
11.3 VPI object classifications ... 275
11.3.1 Accessing object relationships and properties ... 276
11.3.2 Delays and values ... 277
11.4 List of VPI routines by functional category ... 277
11.5 Key to object model diagrams ... 279
11.5.1 Diagram key for objects and classes ... 280
11.5.2 Diagram key for accessing properties ... 280
11.5.3 Diagram key for traversing relationships ... 281
11.6 Object data model diagrams ... 282
11.6.1 Module ... 283
11.6.2 Nature, discipline ... 284
11.6.3 Scope, task, function, IO declaration ... 285
11.6.4 Ports ... 286
11.6.5 Nodes ... 287
11.6.6 Branches ... 288
11.6.7 Quantities ... 289
11.6.8 Nets ... 290
11.6.9 Regs ... 291
11.6.10 Variables, named event ... 292
11.6.11 Memory ... 293
11.6.12 Parameter, specparam ... 294
11.6.13 Primitive, prim term ... 295
11.6.14 UDP ... 296
11.6.15 Module path, timing check, intermodule path ... 297
11.6.16 Task and function call ... 298
11.6.17 Continuous assignment ... 299
11.6.18 Simple expressions ... 300
11.6.19 Expressions ... 301
11.6.20 Contribs ... 302
11.6.21 Process, block, statement, event statement ... 303
11.6.22 Assignment, delay control, event control, repeat control ... 304
11.6.23 If, if-else, case ... 306
11.6.24 Assign statement, deassign, force, release, disable ... 307
11.6.25 Callback, time queue ... 308
12. VPI routine definitions ... 309
12.1 Overview ... 309
12.2 vpi_chk_error() ... 309
12.3 vpi_compare_objects() ... 310
12.4 vpi_free_object() ... 311
12.5 vpi_get() ... 311
12.6 vpi_get_cb_info() ... 312
12.7 vpi_get_analog_delta() ... 313
12.8 vpi_get_analog_freq() ... 313
12.9 vpi_get_analog_time() ... 313
12.10 vpi_get_analog_value() ... 314
12.11 vpi_get_delays() ... 315
12.12 vpi_get_str() ... 318
12.13 vpi_get_analog_systf_info() ... 318

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.14 vpi_get_systf_info() ... 319
12.15 vpi_get_time() ... 320
12.16 vpi_get_value() ... 321
12.17 vpi_get_vlog_info() ... 326
12.18 vpi_get_real() ... 327
12.19 vpi_handle() ... 327
12.20 vpi_handle_by_index() ... 328
12.21 vpi_handle_by_name() ... 329
12.22 vpi_handle_multi() ... 329
12.22.1 Derivatives for analog system task/functions ... 329
12.22.2 Examples ... 330
12.23 vpi_iterate() ... 332
12.24 vpi_mcd_close() ... 333
12.25 vpi_mcd_name() ... 334
12.26 vpi_mcd_open() ... 334
12.27 vpi_mcd_printf() ... 335
12.28 vpi_printf() ... 335
12.29 vpi_put_delays() ... 336
12.30 vpi_put_value() ... 338
12.31 vpi_register_cb() ... 340
12.31.1 Simulation-event-related callbacks ... 341
12.31.2 Simulation-time-related callbacks ... 342
12.31.3 Simulator analog and related callbacks ... 343
12.31.4 Simulator action and feature related callbacks ... 343
12.32 vpi_register_analog_systf() ... 345
12.32.1 System task and function callbacks ... 345
12.32.2 Declaring derivatives for analog system task/functions ... 346
12.32.3 Examples ... 346
12.33 vpi_register_systf() ... 350
12.33.1 System task and function callbacks ... 350
12.33.2 Initializing VPI system task/function callbacks ... 351
12.34 vpi_remove_cb() ... 352
12.35 vpi_scan() ... 353
12.36 vpi_sim_control() ... 354
Annex A (normative) Formal syntax definition ... 355
A.1 Source text ... 355
A.2 Declarations ... 359
A.3 Primitive instances ... 364
A.4 Module instantiation and generate construct ... 365
A.5 UDP declaration and instantiation ... 366
A.6 Behavioral statements ... 367
A.7 Specify section ... 372
A.8 Expressions ... 376
A.9 General ... 383
A.10 Details ... 386
Annex B (normative) List of keywords ... 387
Annex C (normative) Analog language subset ... 389
C.1 Verilog-A overview ... 389
C.2 Verilog-A language features ... 389
C.3 Lexical conventions ... 389
C.4 Data types ... 390
C.5 Expressions ... 390
C.6 Analog signals ... 390

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

C.7 Analog behavior ... 390
C.8 Hierarchical structures ... 390
C.9 Mixed signal ... 391
C.10 Scheduling semantics ... 391
C.11 System tasks and functions ... 391
C.12 Compiler directives ... 391
C.13 Using VPI routines ... 391
C.14 VPI routine definitions ... 391
C.15 Analog language subset ... 391
C.16 List of keywords ... 391
C.17 Standard definitions ... 392
C.18 SPICE compatibility ... 392
C.19 Changes from previous Verilog-A LRM versions ... 392
C.20 Obsolete functionality ... 392
Annex D (normative) Standard definitions ... 393
D.1 The disciplines.vams file ... 393
D.2 The constants.vams file ... 397
D.3 The driver_access.vams file ... 399
Annex E (normative) SPICE compatibility ... 401
E.1 Introduction ... 401
E.2 Accessing Spice objects from Verilog-AMS HDL ... 402
E.3 Preferred primitive, parameter, and port names ... 404
E.4 Other issues ... 408
Annex F (normative) Discipline resolution methods ... 410
F.1 Discipline resolution ... 410
F.2 Resolution of mixed signals ... 410
Annex G (informative) Change history ... 413
G.1 Changes from previous LRM versions ... 413
G.2 Obsolete functionality ... 424
Annex H (informative) Glossary ... 426

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

1. Verilog-AMS introduction

1.1 Overview

This Verilog-AMS Hardware Description Language (HDL) language reference manual defines a behavioral language for analog and mixed-signal systems. Verilog-AMS HDL is derived from IEEE Std 1364-2005 Verilog HDL (referred to as IEEE Std 1364 Verilog from this point forward). This document is intended to cover the definition and semantics of Verilog-AMS HDL as proposed by Accellera.

Verilog-AMS HDL consists of the complete IEEE Std 1364 Verilog specification, an analog equivalent for describing analog systems (also referred to as Verilog-A as described in Annex C), and extensions to both for specifying the full Verilog-AMS HDL.

Verilog-AMS HDL lets designers of analog and mixed-signal systems and integrated circuits create and use modules which encapsulate high-level behavioral descriptions as well as structural descriptions of systems and components. The behavior of each module can be described mathematically in terms of its ports and external parameters applied to the module. The structure of each component can be described in terms of interconnected sub-components. These descriptions can be used in many disciplines such as electrical, mechanical, fluid dynamics, and thermodynamics.

For continuous systems, Verilog-AMS HDL is defined to be applicable to both electrical and non-electrical systems description. It supports *conservative* and *signal-flow* descriptions by using the concepts of *nets, nodes*, *branches*, and *ports* as terminology for these descriptions. The solution of analog behaviors which obey the laws of conservation fall within the generalized form of Kirchhoff's Potential and Flow Laws (KPL and KFL). Both of these are defined in terms of the quantities (e.g., voltage and current) associated with the analog behaviors.

Verilog-AMS HDL can also be used to describe discrete (digital) systems (per IEEE Std 1364 Verilog) and mixed-signal systems using both discrete and continuous descriptions as defined in this LRM.

1.2 Mixed-signal language features

Verilog-AMS HDL extends the features of the digital modeling language (IEEE Std 1364 Verilog) to provide a single unified language with both analog and digital semantics with backward compatibility. Below is a list of salient features of the resulting language:

- signals of both analog and digital types can be declared in the same module
- **`initial`**, **`always`**, and **`analog`** procedural blocks can appear in the same module
- both analog and digital signal values can be accessed (read operations) from any context (analog or digital) in the same module
- digital signal values can be set (write operations) from any context outside of an `analog` procedural block
- analog potentials and flows can only receive contributions (write operations) from inside an **`analog`** procedural block
- the semantics of the **`initial`** and **`always`** blocks remain the same as in IEEE Std 1364 Verilog; the semantics for the **`analog`** block are described in this manual
- the **`discipline`** declaration is extended to digital signals
- a new construct, **`connect`** statement, is added to facilitate auto-insertion of user-defined connection modules between the analog and digital domains

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

— when hierarchical connections are of mixed type (i.e., analog signal connected to digital port or digital signal connected to analog port), user-defined connection modules are automatically inserted to perform signal value conversion

1.3 Systems

A *system* is considered to be a collection of interconnected *components* which are acted upon by a stimulus and produce a response. The components themselves can also be systems, in which case a *hierarchical system* is defined. If a component does not have any subcomponents, it is considered to be a *primitive component*. Each primitive component connects to zero or more *nets*. Each net connects to a *signal* which can traverse multiple levels of the hierarchy. The behavior of each component is defined in terms of values at each net.

A *signal* is a hierarchical collection of nets which, because of port connections, are contiguous. If all the nets which make up a signal are in the discrete domain, the signal is a *digital signal*. If, on the other hand, all the nets which make up a signal are in the continuous domain, the signal is an *analog signal*. A signal which consists of nets from both domains is called a *mixed signal*.

Similarly, a *port* whose connections are both analog is an *analog port*, a port whose connections are both digital is a *digital port*, and a port whose connections are both analog and digital is a *mixed port*. The components connect to *nodes* through ports and nets to build a hierarchy, as shown in Figure 1-1.

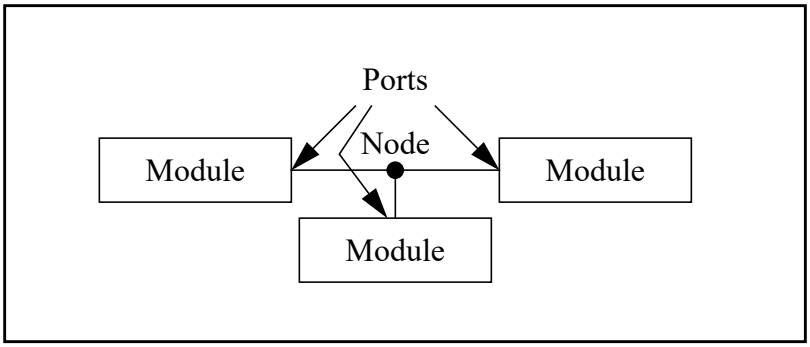


Figure 1-1: Components connect to nodes through ports

If a signal is analog or mixed, it is associated with a node (see 3.6), while a purely digital signal is not associated with a node. Regardless of the number of analog nets in an analog or mixed signal or how the analog nets in a mixed signal are interspersed with digital nets, the analog portion of an analog or mixed signal is represented by only a single node. This guarantees a mixed or analog signal has only one value which represents its potential with respect to the global reference voltage (*ground*).

In order to simulate systems, it is necessary to have a complete description of the system and all of its components. Descriptions of systems are usually given structurally. That is, the description of a system contains instances of components and how they are interconnected. Descriptions of components are given using behavior and/or structure. A behavior is a mathematical description which relates the signals at the ports of the components.

1.3.1 Conservative systems

An important characteristic of conservative systems is that there are two values associated with every node, the *potential* (also known as the *across value* or *voltage* in electrical systems) and the *flow* (the *through value* or *current* in electrical systems). The potential of the node is shared with all continuous ports and nets

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

connected to the node so all continuous ports and nets see the same potential. The flow is shared so flow from all continuous ports and nets at a node shall sum to zero (`0`). In this way, the node acts as an infinitesimal point of interconnection in which the potential is the same everywhere on the node and on which no flow can accumulate. Thus, the node embodies Kirchhoff's Potential and Flow Laws (KPL and KFL). When a component connects to a node through a conservative port or net, it can either affect, or be affected by, either the potential at the node, and/or the flow onto the node through the port or net.

With conservative systems it is also useful to define the concept of a branch. A branch is a path of flow between two nodes through a component. Every branch has an associated potential (the potential difference between the two nodes) and flow.

A behavioral description of a conservative component is constructed as a collection of interconnected branches. The constitutive equations of the component are formulated so as to relate the branch potentials and flows. Refer to 5.4.2 for further details on the probe/source approach.

1.3.1.1 Reference nodes

The potential of a single node is given with respect to a reference node. The potential of the reference node, which is called **`ground`** in electrical systems, is always zero (`0`). Any net of continuous discipline can be declared to be **`ground`**. In this case, the node associated with the net shall be the global reference node in the circuit. This is compatible with all analog disciplines and can be used to bind a port of an instantiated module to the reference node.

1.3.1.2 Reference directions

The reference directions for a generic branch are shown in Figure 1-2.

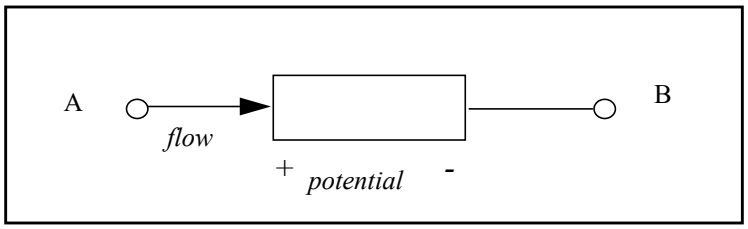


Figure 1-2: Reference directions

The *reference direction* for a potential is indicated by the plus and minus symbols near each port. Given the chosen reference direction, the branch potential is positive whenever the potential of the port marked with a plus sign (`A`) is larger than the potential of the port marked with a minus sign (`B`). Similarly, the flow is positive whenever it moves in the direction of the arrow (in this case from + to −).

Verilog-AMS HDL uses associated reference directions. A positive flow enters a branch through the port marked with the plus sign and exits the branch through the port marked with the minus sign.

1.3.2 Kirchhoff’s Laws

In formulating continuous system equations, Verilog-AMS HDL uses two sets of relationships. The first are the constitutive relationships which describe the behavior of each component. Constitutive relationships can be kept inside the simulator as built-in primitives or they can be provided by Verilog-AMS HDL module definitions.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The second set of relationships, interconnection relationships, describe the structure of the network. Interconnection relationships, which contain information on how the components are connected to each other, are only a function of the system topology. They are independent of the nature of the components.

A Verilog-AMS HDL simulator uses Kirchhoff's Laws to define the relationships between the nodes and the branches. Kirchhoff's Laws are typically associated with electrical circuits that relate voltages and currents. However, by generalizing the concepts of voltages and currents to potentials and flows, Kirchhoff's Laws can be used to formulate interconnection relationships for any type of system.

Kirchhoff's Laws provide the following properties relating the quantities present on nodes and branches, as shown in Figure 1-3.

— Kirchhoff's Flow Law (KFL)
The algebraic sum of all flows out of a node at any instant is zero (0).

— Kirchhoff's Potential Law (KPL)
The algebraic sum of all the branch potentials around a loop at any instant is zero (0).

These laws imply a node is infinitely small; so there is negligible difference in potential between any two points on the node and a negligible accumulation of flow.

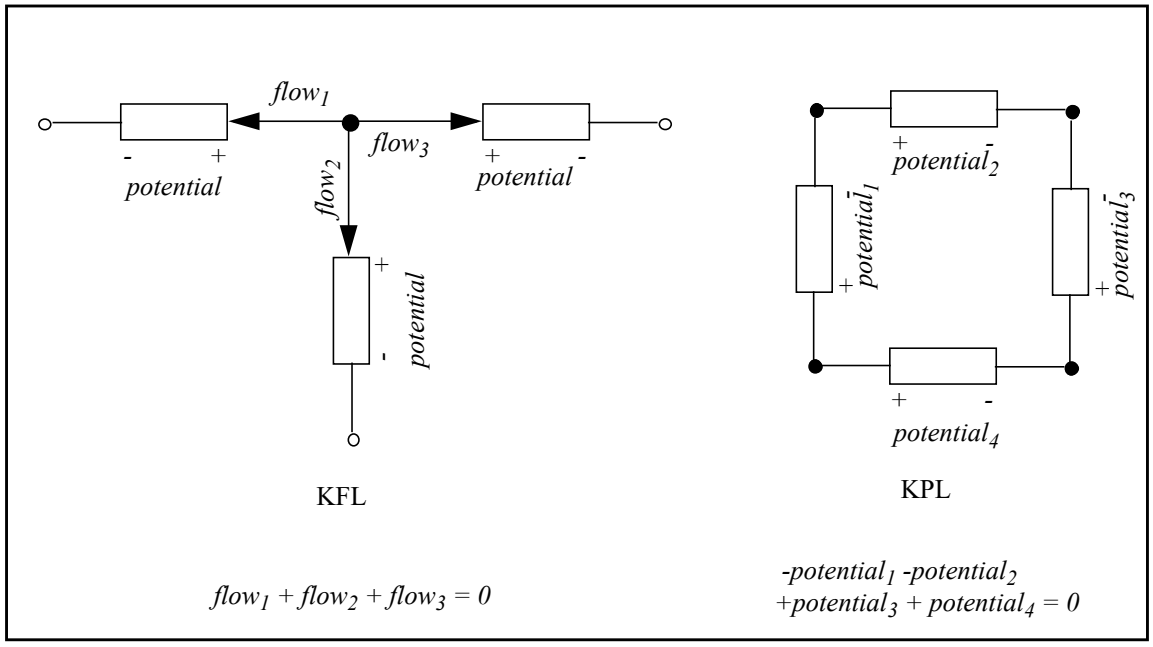


Figure 1-3: Kirchhoff's Flow Law (KFL) and Potential Law (KPL)

1.3.3 Natures, disciplines, and nets

Verilog-AMS HDL allows definition of nets based on disciplines. The disciplines associate potential and flow natures for conservative systems or either only potential or only flow nature for signal-flow systems. The natures are a collection of attributes, including user-defined attributes, which describes the units (meter, gram, newton, etc.), absolute tolerance for convergence, and the names of potential and flow access functions.

The disciplines and natures can be shared by many nets. The compatibility rules help enforce the legal operations between nets of different disciplines.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

1.3.4 Signal-flow systems

A discipline may specify two nature bindings, **`potential`** and **`flow`**, or it may specify only a single binding, either **`potential`** or **`flow`**. Disciplines with two natures are know as *conservative disciplines* because nodes which are bound to them exhibit Kirchhoff's Flow Law, and thus, conserve charge (in the electrical case). A discipline with only a potential nature or only a flow nature is known as a *signal flow discipline*.

As a result of port connections of analog nets, a single node may be bound to a number of nets of different disciplines. If a node is bound only to disciplines which have potential nature only, current contributions to that node are not legal. Flow for such a node is not defined. Conversely, if a node is bound only to disciplines which have flow nature only, potential contributions to that node are not legal. Potential for such a node is not defined.

1.3.4.1 Potential signal-flow systems

Potential signal flow models may be written so potentials of module outputs are purely functions of potentials at the inputs without taking flow into account.

The following example is a level shifting voltage follower:

```
module shiftPlus5(in, out);
input in;
output out;
voltage in, out;  //voltage is a signal flow
                  //discipline compatible with
                  //electrical, but having a
                  //potential nature only
analog begin
   V(out) <+ 5.0 + V(in);
end
endmodule
```

If a number of such modules were cascaded in series, it would not be necessary to conserve charge (i.e., sum the flows) at any intervening node.

If, on the other hand, the output of this device were bound to a node of a conservative discipline (e.g., `electrical`), then the output of the device would appear to be a controlled voltage source to ground at that node. In that case, the flow (i.e., current) through the source would contribute to charge conservation at the node. If the input of this device were bound to a node of a conservative discipline then the input would act as a voltage probe to ground. Thus, when a net of signal flow discipline with potential nature only is bound to a conservative node, contributions made to that net behave as voltage sources to ground.

Nets of potential signal flow disciplines in modules may only be bound to **`input`** or **`output`** ports of the module, not to **`inout`** ports. In that case, potential contributions may not be made to **`input`** ports.

1.3.4.2 Flow signal-flow systems

Flow signal-flow models may be written so flows of module outputs are purely functions of flows at the inputs without taking potential into account.

The following example is a current mirror:

```
module currmir(in, out);
   input in;
   output out;
   current in, out;  // current is a signal flow
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
                    // discipline compatible with
                    // electrical, but having a
                    // flow nature only

   analog begin
      I(out) <+ -I(in);
   end
endmodule
```

If a number of such modules were cascaded in series, it would not be necessary to conserve charge (i.e., sum the potentials) at any loop of branches.

However, if the output of this device were bound to a node of a conservative discipline (e.g., electrical), then the output of the device would appear to be a controlled current source flowing out of that node. In that case, the potential (i.e., voltage) across the source would contribute to charge conservation at the node. If the input of this device were bound to a node of a conservative discipline then the input would act as a current probe inbound from that node. Thus, when a net of signal flow discipline with flow nature only is bound to a conservative node, contributions made to that net behave as current sources.

Nets of flow signal-flow disciplines in modules may only be bound to input or output ports of the module, not to inout ports. Flow contributions may not be made to input ports in this case.

1.3.5 Mixed conservative/signal flow systems

When practicing the top-down design style, it is extremely useful to mix conservative and signal-flow components in the same system. Users typically use signal-flow models early in the design cycle when the system is described in abstract terms, and gradually convert component models to conservative form as the design progresses. Thus, it is important to be able to initially describe a component using a signal-flow model, and later convert it to a conservative model, with minimum changes. It is also important to allow conservative and signal-flow components to be arbitrarily mixed in the same system.

The approach taken is to write component descriptions using conservative semantics, except port and net declarations only require types for those values which are actually used in the description. Thus, signal-flow ports only require the type of either potential or flow to be specified, whereas conservative ports require types for both values (the potential and flow).

For example, consider a differential voltage amplifier, a differential current amplifier, and a resistor. The amplifiers are written using signal-flow ports and the resistor uses conservative ports.

```
module voltage_amplifier (out, in);
   input in;
   output out;
   voltage  out,     // Discipline voltage defined elsewhere
            in;      // with access function V()
   parameter real GAIN_V = 10.0;

   analog
      V(out) <+ GAIN_V * V(in);

endmodule
```

In this case, only the voltage on the ports are declared because only voltage is used in the body of the model.

```
module current_amplifier (out, in);
   input in;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   output out;
   current  out,     // Discipline current defined elsewhere
            in;      // with access function I()
   parameter real GAIN_I = 10.0;

   analog
      I(out) <+ GAIN_I * I(in);

endmodule
```

Here, only current is used in the body of the model, so only current need be declared at the ports.

```
module resistor (a, b);
   inout a, b;
   electrical a, b;        // access functions are V() and I()
   parameter real R = 1.0;

   analog
      V(a,b) <+ R * I(a,b);

endmodule
```

The description of the resistor relates both the voltage and current on the ports. Both are defined in the conservative discipline `electrical`.

In summary, only those signals types declared on the ports are accessible in the body of the model. Conversely, only those signals types used in the body need be declared.

This approach provides all of the power of the conservative formulation for both signal-flow and conservative ports, without forcing types to be declared for unused signals on signal-flow nets and ports. In this way, the first benefit of the traditional signal-flow formulation is provided without the restrictions.

The second benefit, that of a smaller, more efficient set of equations to solve, is provided in a manner which is hidden from the user. The simulator begins by treating all ports as being conservative, which allows the connection of signal-flow and conservative ports. This results in additional unnecessary equations for those nodes which only have signal-flow ports. This situation can be recognized by the simulator and those equations eliminated.

Thus, this approach to allowing mixed conservative/signal-flow descriptions provides the following benefits:

- Conservative components and signal-flow components can be freely mixed. In addition, signal-flow components can be converted to conservative components, and vice versa, by modifying only the component behavioral description.
- Many of the capabilities of conservative ports, such as the ability to access flow and the ability to access floating potentials, are available with signal-flow ports.
- Natures only have to be given for potentials and flows if they are accessed in a behavioral description.
- If nets and ports are used only in a structural description (only in instance statements), then no natures need be specified.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

1.4 Conventions used in this document

This document is organized into sections, each of which focuses on some specific area of the language. There are subsections within each section to discuss individual constructs and concepts. The discussion begins with an introduction and an optional rationale for the construct or the concept, followed by syntax and semantic description, followed by some examples and notes.

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

1) Lower case words, some containing embedded underscores, are used to denote syntactic categories. For example:

   module_declaration

2) Boldface red characters denote reserved keywords, operators and punctuation marks as required part of the syntax. For example:

   `module = ;`

3) Blue characters are used to denote syntax productions that are Verilog-AMS extensions to IEEE Std 1364 Verilog syntax. For example:

   connectrules_declaration ::=
   `connectrules` connectrules_identifier **`;`**
   { connectrules_item }
   `endconnectrules`

4) A vertical bar (|) that is not in boldface-red separates alternative items. For example:

   parameter_type ::= **`integer`** | **`real`** | **`realtime`** | **`time`** | **`string`**

5) Square brackets ([ ]) that are not in boldface-red enclose optional items. For example:

   name_of_module_instance ::= module_instance_identifier [range]

6) Braces ({ }) that are not in boldface-red enclose a repeated item. 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_assignments **`,`** param_assignment

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`. Reserved keywords in the main text and in examples are in a **`constant-width bold`** font.

1.5 Contents

This document contains the following clauses and annexes:

1. Verilog-AMS introduction
This clause gives the overview of analog modeling, defines basic concepts, and describes Kirchhoff's Potential and Flow Laws.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

2. Lexical conventions
This clause defines the lexical tokens used in Verilog-AMS HDL.

3. Data types
This clause describes the data types: integer, real, parameter, nature, discipline, and net, used in Verilog-AMS HDL.

4. Expressions
This clause describes expressions, mathematical functions, and time domain functions used in Verilog-AMS HDL.

5. Analog behavior
This clause describes the basic analog block and procedural language constructs available in Verilog-AMS HDL for behavioral modeling.

6. Hierarchical structures
This clause describes how to build hierarchical descriptions using Verilog-AMS HDL.

7. Mixed signal
This clause describes the mixed-signal aspects of the Verilog-AMS HDL language.

8. Scheduling semantics
This clause describes the basic simulation cycle as applicable to Verilog-AMS HDL.

9. System tasks and functions
This clause describes the system tasks and functions in Verilog-AMS HDL.

10. Compiler directives
This clause describes the compiler directives in Verilog-AMS HDL.

11. Using VPI routines
This clause describes how the VPI routines are used.

12. VPI routine definitions
This clause defines each of the VPI routines in alphabetical order.

A. (normative) Formal syntax definition
This annex describes formal syntax for all Verilog-AMS HDL constructs in Backus-Naur Form (BNF).

B. (normative) List of keywords
This annex lists all the words which are recognized in Verilog-AMS HDL as keywords.

C. (normative) Analog language subset
This annex describes the analog subset of Verilog-AMS HDL.

D. (normative) Standard definitions
This annex provides the definitions of several natures, disciplines, and constants which are useful for writing models in Verilog-AMS HDL.

E. (normative) SPICE compatibility
This annex describes the Spice compatibility with Verilog-AMS HDL.

F. (normative) Discipline resolution methods
This annex provides the semantics for two methods of resolving the discipline of undeclared interconnect.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

G. (informative) Change history

This annex provides a list of changes between various versions of the Verilog-AMS Language Reference Manual.

H. (informative) Glossary

This annex describes various terms used in this document.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

2. Lexical conventions

2.1 Overview

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

2.2 Lexical tokens

Verilog-AMS 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 is free format — that is, spaces and newlines shall not be syntactically significant other than being token separators, except escaped identifiers (see 2.8.1).

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

— white space
— comment
— operator
— number
— string
— identifier
— keyword

2.3 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, spaces and tabs shall be considered significant characters in strings (see 2.7).

2.4 Comments

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

```
comment ::=                                                    // from A.9.2
        one_line_comment
      | block_comment
one_line_comment ::= // comment_text \n
block_comment ::= /* comment_text */
comment_text ::= { Any_ASCII_character }
```

Syntax 2-1—Syntax for comments

2.5 Operators

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

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

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 which separate three operands.

2.6 Numbers

Constant numbers can be specified as integer constants (defined in 2.6.1) or real constants.

---

number ::= *// from A.8.7*
    decimal_number
  | octal_number
  | binary_number
  | hex_number
  | real_number

real_number[1] ::=
    unsigned_number . unsigned_number
  | unsigned_number [. unsigned_number] exp [sign] unsigned_number
  | unsigned_number [. unsigned_number] scale_factor

exp ::= **e** | **E**

scale_factor ::= **T** | **G** | **M** | **K** | **k** | **m** | **u** | **n** | **p** | **f** | **a**

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[1] ::= non_zero_decimal_digit { _ | decimal_digit}

unsigned_number[1] ::= decimal_digit { _ | decimal_digit }

binary_value[1] ::= binary_digit { _ | binary_digit }

octal_value[1] ::= octal_digit { _ | octal_digit }

hex_value[1] ::= hex_digit { _ | hex_digit }

decimal_base[1] ::= **'**[**s**|**S**]**d** | **'**[**s**|**S**]**D**

binary_base[1] ::= **'**[**s**|**S**]**b** | **'**[**s**|**S**]**B**

octal_base[1] ::= **'**[**s**|**S**]**o** | **'**[**s**|**S**]**O**

hex_base[1] ::= **'**[**s**|**S**]**h** | **'**[**s**|**S**]**H**

non_zero_decimal_digit ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

decimal_digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

binary_digit ::= x_digit | z_digit | **0** | **1**

octal_digit ::= x_digit | z_digit | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**

hex_digit ::=
    x_digit | z_digit | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
      | a | b | c | d | e | f | A | B | C | D | E | F
x_digit ::= x | X
z_digit ::= z | Z | ?
```

1) Embedded spaces are illegal

Syntax 2-2—Syntax for integer and real constants

2.6.1 Integer constants

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

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

The second token, a base_format, shall consist of a case insensitive letter specifying the base for the number, optionally preceded by the single character `s` (or `S`) to indicate a signed quantity, preceded by the apostrophe character. Legal base specifications are `d`, `D`, `h`, `H`, `o`, `O`, `b`, or `B`, for the bases decimal, hexadecimal, octal, and binary respectively. The apostrophe character and the base format character shall not be separated by any white space.

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

Simple decimal numbers without the size and the base format shall be treated as signed integers, whereas the numbers specified with the base format shall be treated as signed integers if the s designator is included or as unsigned integers if the base format only is used. The s designator does not affect the bit pattern specified, only its interpretation. A plus or minus operator preceding the size constant is a unary plus or minus operator. A plus or minus operator between the base format and the number is an illegal syntax. Negative numbers shall be represented in 2's complement form.

An `x` represents the unknown value in hexadecimal, octal, and binary constants. A `z` represents the high-impedance value. See 4.1 of IEEE Std 1364 Verilog 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 left-most bit in the unsigned number is an `x` or a `z`, then an `x` or a `z` shall be used to pad to the left respectively. If the size of the unsigned number is larger than the size specified for the constant, the unsigned number shall be truncated from the left.

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

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

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

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

When used in a number, the question-mark (`?`) character is a Verilog-AMS 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 of IEEE Std 1364 Verilog. The question-mark character is also used in user-defined primitive state tables. See Table 8-1 in 8.1.6 of IEEE Std 1364 Verilog.

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

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

Example 1 — Unsized literal 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 literal 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 literal constant numbers

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

Example 4 — Automatic left padding

```
reg [11:0] a, b, c, d;
initial begin
   a = 'h x;      // yields xxx
   b = 'h 3x;     // yields 03x
   c = 'h z3;     // yields zz3
   d = 'h 0z3;    // yields 0z3
end
reg [84:0] e, f, g;
   e = 'h5;       // yields {82{1'b0},3'b101}
   f = 'hx;       // yields {85{1'hx}}
   g = 'hz;       // yields {85{1'hz}}
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Example 5 — Using underscore character in literal constant numbers

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

Sized negative constant numbers and sized signed constant numbers are sign-extended when assigned to a **`reg`** data type, regardless of whether or not the **`reg`** itself is signed.

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

2.6.2 Real constants

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

Real numbers shall be specified in either decimal notation (e.g., `14.72`), in scientific notation (e.g., `39e8`, which indicates 39 multiplied by 10 to the 8^{th} power) or in scaled notation (e.g., `24.7K`, which indicates 24.7 multiplied by 10 to the third power). Real numbers expressed with a decimal point shall have at least one digit on each side of the decimal point. The underscore character is legal anywhere in a real constant except as the first character of the constant or the first character after the decimal point. The underscore character is ignored.

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
1.3u
7k
```

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
.1p
34.M
```

Table 2-1 describes each symbol and their value used in scaled notation or a real number.

Table 2-1—Scaled Symbols and notation

Symbol	Value
T	1e12
G	1e9

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 2-1—Scaled Symbols and notation *(continued)*

Symbol	Value
M	1e6
K, k	1e3
m	1e-3
u	1e-6
n	1e-9
p	1e-12
f	1e-15
a	1e-18

No space is permitted between the number and the symbol. Scale factors are not allowed to be used in defining digital delays (e.g., `#5u`).

See 4.2.1.1 for a discussion of real to integer conversion and 4.2.1.2 for a discussion of integer to real conversion.

2.7 String literals

A *string literal* is a sequence of characters enclosed by double quotes (`"`) and contained on a single line. A *string literal* used as an operand 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. The **`string`** variable data type can be used to store a *string literal* (see 3.3). Parameters of type **`string`** are treated differently and are described in 3.4.6.

Certain characters can only be used in a *string literal* when preceded by an introductory character called an *escape character*. Table 2-2 lists these characters in the right-hand column, with the escape sequence that represents the character in the left-hand column.

Table 2-2—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 ($0 \le d \le 7$) If less than three characters are used, the following character shall not be an octal digit. Implementations may issue an error if the character represented is greater than \377.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

2.8 Identifiers, keywords, and system names

An *identifier* shall be used to give an object a unique name so it can be referenced. An *identifier* shall either be a *simple identifier* or an *escaped identifier* (see 2.8.1). A *simple identifier* shall be any sequence of letters, digits, dollar signs (**$**), and the underscore characters (_).

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

Examples:

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

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

2.8.1 Escaped identifiers

Escaped identifiers shall start with the backslash character (`\`) and end with white space (space, tab, newline, or formfeed). 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 non-escaped identifier `cpu3`.

Examples:

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

2.8.2 Keywords

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

All keywords are defined in lowercase only. Annex B lists all defined Verilog-AMS HDL keywords.

2.8.3 System tasks and functions

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

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

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
analog_system_task_enable ::=                                          // from A.6.9
        analog_system_task_identifier [ ( [ analog_expression ] { , [ analog_expression ] } ) ] ;
system_task_enable ::= system_task_identifier [ ( [ expression ] { , [ expression ] } ) ] ;
analog_system_function_call ::=                                        // from A.8.2
        analog_system_function_identifier [ ( [ analog_expression ] { , [ analog_expression ] } ) ]
system_function_call ::= system_function_identifier
        [ ( expression { , expression } ) ]
system_function_identifier[1] ::= $ [ a-zA-Z0-9_$ ] { [ a-zA-Z0-9_$ ] }   // from A.9.3
system_task_identifier[1] ::= $ [ a-zA-Z0-9_$ ] { [ a-zA-Z0-9_$ ] }
```

Syntax 2-3—Syntax for system tasks and functions

The $*identifier* system task or function can be defined in five places,

- A standard set of $*identifier* system tasks and functions, as defined in Clause 17 and Clause 18 of IEEE Std 1364 Verilog.
- Additional $*identifier* system tasks and functions defined using the PLI, as described in Clause 20 of IEEE Std 1364 Verilog.
- Additional $*identifier* system tasks and functions defined in Clause 4 and Clause 9 of this standard.
- Additional $*identifier* system tasks and functions defined using the VPI as described in Clause 11 and Clause 12 of this standard.
- 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.

Examples:

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

2.8.4 Compiler directives

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

The *`identifier* compiler directive construct can be defined in three places

- A standard set of *`identifier* compiler directives defined in Clause 19 of IEEE Std 1364 Verilog.
- Additional `*identifier* compiler directives defined in Clause 10 of this standard.
- 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.

[1]The $ character in a system_function_identifier, system_task_identifier, or system_parameter_identifier shall not be followed by white_space. A system_function_identifier or system_task_identifier shall not be escaped.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Example:

```
`define WORDSIZE 8
```

2.9 Attributes

With the proliferation of tools other than simulators that use Verilog-AMS HDL as their source, a mechanism is included for specifying properties about objects, statements and groups of statements in the HDL source that can be used by various tools, including simulators, to control the operation or behavior of the tool. These properties shall be referred to as *attributes*. This section specifies the syntactic mechanism that shall be used for specifying attributes.

The syntax is found in Syntax 2-4.

---

attribute_instance ::= (* attr_spec { , attr_spec } *) *// from A.9.1*

attr_spec ::= attr_name [= constant_expression]

attr_name ::= identifier

Syntax 2-4—Syntax for attributes

An *attribute_instance* can appear in the Verilog-AMS 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-AMS function name in an expression.

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

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

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 *) // Multiple attribute instances also OK
case (foo)
   <rest_of_case_statement>
```

or

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

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

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

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

or

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

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

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

or

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

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

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

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

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

Example 6 — To attach an attribute to an operator:

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

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

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;
```

2.9.1 Syntax

The syntax for legal statements with attributes is shown in Syntax 2-5 through Syntax 2-10.

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

---

module_declaration ::= *// from A.1.2*
{ attribute_instance } module_keyword module_identifier [module_parameter_port_list]
list_of_ports ; { module_item }

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    endmodule
  | { attribute_instance } module_keyword module_identifier [ module_parameter_port_list ]
      [ list_of_port_declarations ] ; { non_port_module_item }
    endmodule
```

Syntax 2-5—Syntax for module declaration attributes

The syntax for port declaration attributes is given in Syntax 2-6.

```
port_declaration ::=                                                    // from A.1.3
      {attribute_instance} inout_declaration
    | {attribute_instance} input_declaration
    | {attribute_instance} output_declaration
```

Syntax 2-6—Syntax for port declaration attributes

The syntax for module item attributes is given in Syntax 2-7.

```
module_item ::=                                                         // from A.1.4
      port_declaration ;
    | non_port_module_item
module_or_generate_item ::=
      { attribute_instance } module_or_generate_item_declaration
    | { attribute_instance } local_parameter_declaration ;
    | { attribute_instance } parameter_override
    | { attribute_instance } continuous_assign
    | { attribute_instance } gate_instantiation
    | { attribute_instance } udp_instantiation
    | { attribute_instance } module_instantiation
    | { attribute_instance } initial_construct
    | { attribute_instance } always_construct
    | { attribute_instance } loop_generate_construct
    | { attribute_instance } conditional_generate_construct
    | { attribute_instance } analog_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
    | branch_declaration
    | analog_function_declaration
non_port_module_item ::=
      module_or_generate_item
    | generate_region
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
| specify_block
| { attribute_instance } parameter_declaration ;
| { attribute_instance } specparam_declaration
| aliasparam_declaration
```

Syntax 2-7—Syntax for module item attributes

The syntax for function port, task, and block attributes is given in Syntax 2-8.

```
function_port_list ::=                                                    // from A.2.6
        { attribute_instance } tf_input_declaration { , { attribute_instance } tf_input_declaration }
task_item_declaration ::=                                                 // from A.2.7
        block_item_declaration
      | { attribute_instance } tf_input_declaration ;
      | { attribute_instance } tf_output_declaration ;
      | { attribute_instance } tf_inout_declaration ;
task_port_item ::=
        { attribute_instance } tf_input_declaration
      | { attribute_instance } tf_output_declaration
      | { attribute_instance } tf_inout_declaration
block_item_declaration ::=                                                // from A.2.8
        { attribute_instance } reg [ discipline_identifier ] [ signed ] [ range ]
            list_of_block_variable_identifiers ;
      | { attribute_instance } integer list_of_block_variable_identifiers ;
      | { attribute_instance } time list_of_block_variable_identifiers ;
      | { attribute_instance } real list_of_block_real_identifiers ;
      | { attribute_instance } realtime list_of_block_real_identifiers ;
      | { attribute_instance } event_declaration
      | { attribute_instance } local_parameter_declaration ;
      | { attribute_instance } parameter_declaration ;
```

Syntax 2-8—Syntax for function port, task, and block attributes

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

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

Syntax 2-9—Syntax for port connection attributes

The syntax for udp attributes is given in Syntax 2-10.

```
udp_declaration ::=                                                       // from A.5.1
        { attribute_instance } primitive udp_identifier ( udp_port_list ) ;
            udp_port_declaration { udp_port_declaration }
            udp_body
        endprimitive
      | { attribute_instance } primitive udp_identifier ( udp_declaration_port_list ) ;
            udp_body
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    endprimitive
udp_output_declaration ::=                                          // from A.5.2
      { attribute_instance } output port_identifier
    | { attribute_instance } output [ discipline_identifier ] reg port_identifier [ = constant_expression ]
udp_input_declaration ::= { attribute_instance } input list_of_port_identifiers
udp_reg_declaration ::= { attribute_instance } reg [ discipline_identifier ] variable_identifier
```

Syntax 2-10—Syntax for udp attributes

2.9.2 Standard attributes

The Verilog-AMS HDL standardizes the following attributes:

- The `desc` attribute is used to generate help messages when attached to parameter, variable and net declarations within a module. The attribute must be assigned a string. See 3.4.3.
- The `units` attribute is used to describe the units of the parameter or variable which it is attached to within a module. The attribute must be assigned a string. See 3.4.3.
- The `op` attribute is used to indicate whether a parameter or variable should be included in a short report of the most useful operating-point values. The attribute must be assigned a value, which must be either "yes" or "no". If the attribute is specified with the value "no", then the parameter or variable will be omitted from the short report; otherwise, the parameter or variable will be included.
- The `multiplicity` attribute is used to describe how the value of a parameter or variable should be scaled for reporting. The attribute must be assigned one of the following string values: "multiply", "`divide`", or "`none`". If the attribute is specified with the value "`multiply`", the value for the associated parameter or variable will be multiplied by the value of **`$mfactor`** for the instance in any report of operating-point values. If the attribute is specified with the value "`divide`", the value for the associated parameter or variable will be divided by the value of **`$mfactor`** for the instance in any report of operating-point values. If the `multiplicity` attribute is not specified, or specified with the value "`none`", then no scaling will be performed. Note that this scaling only applies to operating-point value reports; it does not affect the automatic scaling detailed in 6.3.6.

Example - The following example shows how to attach standard attributes to a variable:

```
(* desc="effective resistance", units="Ohms", op="yes",
multiplicity="divide" *)
real reff;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

3. Data types

3.1 Overview

Verilog-AMS HDL supports **integer**, **genvar**, **real**, and **parameter** data types as found in IEEE Std 1364 Verilog. It includes the **string** data type defined by IEEE Std 1800 SystemVerilog. It also modifies the **parameter** data types. Plus, it extends the net data types to support a new type called **wreal** to model real value nets.

Verilog-AMS HDL introduces a new data type, called *net_discipline*, for representing analog nets and declaring *disciplines* of all nets and regs. The *disciplines* define the **domain** and the natures of **potential** and **flow** and their associated attributes for *continuous* domains.

3.2 Integer and real data types

The syntax for declaring **integer** and **real** is shown in Syntax 3-1.

```
integer_declaration ::= integer list_of_variable_identifiers ;                    // from A.2.1.3
real_declaration ::= real list_of_real_identifiers ;
list_of_real_identifiers ::= real_type { , real_type }                            // from A.2.3
list_of_variable_identifiers ::= variable_type { , variable_type }
real_type ::=                                                                     // from A.2.2.1
        real_identifier { dimension } [ = constant_assignment_pattern ]
      | real_identifier = constant_expression
variable_type ::=
        variable_identifier { dimension } [ = constant_assignment_pattern ]
      | variable_identifier = constant_expression
dimension ::= [ dimension_constant_expression : dimension_constant_expression ]   // from A.2.5
```

Syntax 3-1—Syntax for integer and real declarations

An *integer* declaration declares one or more variables of type integer. These variables can hold values ranging from -2^{31} to $2^{31}-1$. Arrays of integers can be declared using a range which defines the upper and lower indices of the array. Both indices shall be constant expressions and shall evaluate to a positive integer, a negative integer, or zero (0).

Arithmetic operations performed on integer variables produce 2's complement results.

A *real* declaration declares one or more variables of type **real**. The real variables are stored as 64-bit quantities, as described by IEEE Std 754, an IEEE standard for double precision floating point numbers.

Arrays of **parameter** can be declared using a range which defines the upper and lower indices of the array. Both indices shall be constant expressions and shall evaluate to a positive integer, a negative integer, or zero (0).

Integers are initialized at the start of a simulation depending on how they are used. Integer variables whose values are assigned in an analog context default to an initial value of zero (0). Integer variables whose values are assigned in a digital context default to an initial value of x. Real variables are initialized to zero (0) at the start of a simulation.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Examples:

```
integer a[1:64];           // an array of 64 integer values
real float;                // a variable to store real value
real gain_factor[1:30];    // array of 30 gain multipliers
                           // with floating point values
integer flag_array[0:8][0:3]; // a multidimensional integer array
real vtable[0:16][0:7][0:64]; // a multidimensional real array
```

See 4.2.1.1 for a discussion of real to integer conversion and 4.2.1.2 for a discussion of integer to real conversion.

3.2.1 Output variables

The standard attributes for descriptions and units, described in 2.9.2, have a special meaning for variables declared at module scope. Module scope variables with a description or units attribute, or both, shall be known as output variables, and Verilog-AMS simulators shall provide access to their values. SPICE-like simulators print the names, values, units, and descriptions of output variables for SPICE primitives along with voltages and currents when displaying operating-point information, and these variables are available for plotting as a function of time (or the swept variable of a dc sweep).

For example, a module for a MOS transistor with the following declaration at module scope provides the output variable `cgs`.

```
(* desc="gate-source capacitance", units="F" *)
real cgs;
```

An operating-point display from the simulator might include the following information:

```
cgs = 4.21e-15 F gate-source capacitance
```

Units and descriptions specified for block-level variables shall be ignored by the simulator, but can be used for documentation purposes.

3.3 String data type

Verilog-AMS includes the **`string`** data type from IEEE Std 1800 SystemVerilog, which is an ordered collection of characters. The length of a **`string`** variable is the number of characters in the collection. Variables of type **`string`** are dynamic as their length may vary during simulation.

IEEE Std 1364 Verilog supports string literals, but only at the lexical level. In Verilog, string literals behave like packed arrays of a width that is a multiple of 8 bits. A string literal assigned to a packed array of an integral variable of a different size is either truncated to the size of the variable or padded with zeroes to the left as necessary.

In Verilog-AMS, string literals behave exactly the same as in Verilog. However, Verilog-AMS also supports the **`string`** data type to which a string literal can be assigned. When using the **`string`** data type instead of an integral variable, strings can be of arbitrary length and no truncation occurs. Literal strings are implicitly converted to the **`string`** type when assigned to a **`string`** type or used in an expression involving **`string`** type operands.

The **`string`** variables can take on the special value `""`, which is the empty string. A **`string`** shall not contain the special character `"\0"`.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The syntax to declare a **string** is as follows:

```
string variable_name [ = initial_value ] ;
```

where *variable_name* is a valid identifier and the optional *initial_value* can be a string literal, the value `""` for an empty string, or a string type constant expression, such as a string parameter (see 3.4.6). For example:

```
parameter string default_name = "John Smith";
string myName = default_name;
```

If an initial value is not specified in the declaration, the variable is initialized to `""`, the empty string. An empty string has zero length.

Arrays and multidimensional arrays of string are also supported. For example:

```
string names[1:3] = '{"first", "middle", "last"};
string paths[0:2][0:1] =
            '{ '{"dir1", "fileA"}, '{"dir2", "fileA"},'{"dir1","fileB"}};
```

Verilog-AMS provides a set of operators that can be used to manipulate combinations of string variables and string literals. The basic operators defined on the **`string`** data type are listed in Table 3-3.

A string literal can be assigned to a **`string`** or an integral type. If their size differs, the literal is right justified and either truncated on the left or zero filled on the left, as necessary. For example:

```
reg [8*4:1] h = "hello";   // assigns to h "ello"
reg [10:0] a = "A";        // assigns to a 'b000_0100_0001
```

A **`string`** or a string literal can be assigned directly to a **`string`** variable. A **`string`** cannot be assigned to an integral type. A string literal assigned to a **`string`** variable is converted according to the following steps:

— All `"\0"` characters in the string literal are ignored (i.e., removed from the **`string`**).
— If the result of the first step is an empty string literal, the **`string`** is assigned the empty string.
— Otherwise, the **`string`** is assigned the remaining characters in the string literal.

For example:

```
string s1 = "hello\0world"; // sets s1 to "helloworld"
```

As a second example:

```
reg [15:0] r;
integer i = 1;
string b = "";
string a = {"Hi", b};
b = "Hi";          // OK
b = {5{"Hi"}};     // OK
a = {i{"Hi"}};     // OK (non constant replication)
r = {i{"Hi"}};     // invalid (non constant replication)
a = {i{b}};        // OK
a = {a,b};         // OK
a = {"Hi",b};      // OK
r = {"H",""};      // yields "H\0". "" is converted to 8'b0
b = {"H",""};      // yields "H". "" is the empty string
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 3-3—String operators

Operator	Semantics
`Str1 == Str2`	Equality. Checks whether the two strings are equal. Result is 1 if they are equal and 0 if they are not. Both strings can be of type **`string`**, or one of them can be a string literal which is implicitly converted to a **`string`** type for the comparison. If both operands are string literals, the operator is the same Verilog equality operator as for integer types.
`Str1 != Str2`	Inequality. Logical negation of ==

`Str1 < Str2`
`Str1 <= Str2`
`Str1 > Str2`
`Str1 >= Str2`
	Comparison. Relational operators return 1 if the corresponding condition is true using the lexicographical ordering of the two strings `Str1` and `Str2`. Both operands can be of type **`string`**, or one of them can be a string literal which is implicitly converted to a **`string`** type for the comparison.

`{Str1,Str2,...,Strn}`	Concatenation. Each operand can be of type **`string`** or a string literal (it shall be implicitly converted to type **`string`**). If at least one operand is of type **`string`**, then the expression evaluates to the concatenated string and is of type **`string`**. If all the operands are string literals, then the expression behaves like a Verilog concatenation of integral types; if the result is then used in an expression involving **`string`** types, it is implicitly converted to the **`string`** type.
`{multiplier{Str}}`	Replication. `Str` can be of type **`string`** or a string literal. `multiplier` must be of integral type and can be nonconstant. If `multiplier` is nonconstant or `Str` is of type **`string`**, the result is a string containing *N* concatenated copies of `Str`, where *N* is specified by `multiplier`. If `Str` is a literal and `multiplier` is constant, the expression behaves like numeric replication in Verilog (if the result is used in another expression involving string types, it is implicitly converted to the string type).

3.4 Parameters

The syntax for parameter declarations is shown in Syntax 3-2.

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

For parameters defined as arrays, the initializer shall be a *constant_assignment_pattern* expression which is a list of constant expressions containing only constant numbers and previously defined parameters using an assignment pattern (see 4.2.14) , i.e. within **`'{`** and **`}`** delimiters.

Parameters represent constants, hence it is illegal to modify their value at runtime. However, parameters can be modified at compilation time to have values which 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_instantiation* statement.

---

local_parameter_declaration ::= *// from A.2.1.1*
  `localparam` [**`signed`**] [range] list_of_param_assignments
| **`localparam`** parameter_type list_of_param_assignments

parameter_declaration ::=
  `parameter` [**`signed`**] [range] list_of_param_assignments
| **`parameter`** parameter_type list_of_param_assignments

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
specparam_declaration ::= specparam [ range ] list_of_specparam_assignments ;
parameter_type ::=
        integer | real | realtime | time | string
aliasparam_declaration ::= aliasparam parameter_identifier = parameter_identifier ;
list_of_param_assignments ::= param_assignment { , param_assignment }                    // from A.2.3
param_assignment ::=                                                                     // from A.2.4
        parameter_identifier = constant_mintypmax_expression { value_range }
      | parameter_identifier range = constant_assignment_pattern { value_range }
range ::= [ msb_constant_expression : lsb_constant_expression ]                          // from A.2.5
value_range ::=
        value_range_type ( value_range_expression : value_range_expression )
      | value_range_type ( value_range_expression : value_range_expression ]
      | value_range_type [ value_range_expression : value_range_expression )
      | value_range_type [ value_range_expression : value_range_expression ]
      | value_range_type '{ string { , string } }
      | exclude constant_expression
value_range_type :: = from | exclude
value_range_expression ::= constant_expression | -inf | inf
```

Syntax 3-2—Syntax for parameter declaration

By nature, analog behavioral specifications are characterized more extensively in terms of parameters than their digital counterparts. There are three fundamental extensions to the parameter declarations defined in IEEE Std 1364 Verilog:

- A range of permissible values can be defined for each parameter. In IEEE Std 1364 Verilog, this check had to be done in the user's model or was left as an implementation specific detail.
- Parameter arrays of basic **integer** and **real** data types can be specified.
- String parameters may be declared.

3.4.1 Type specification

The parameter declaration can contain an optional *type* specification. In this sense, the **parameter** keyword acts more as a type qualifier than a type specifier. A default value for the parameter shall be specified.

The following examples illustrate this concept:

```
parameter real slew_rate = 1e-3;
parameter integer size = 16;
```

If the *type* of a parameter is not specified, it is derived from the type of the final value assigned to the parameter, after any value overrides have been applied, as in IEEE Std 1364 Verilog. Note that the *type* of a string parameter (see 3.4.6) and any of the array parameters (see 3.4.4) is mandatory.

If the type of the parameter is specified as **integer** or **real**, and the value assigned to the parameter conflicts with the type of the parameter, the value is converted to the type of the parameter (see 4.2.1.1). No conversion shall be applied for strings; it shall be an error to assign a numeric value to a parameter declared as **string** or to assign a string value to a **real** parameter, whether that parameter was declared as **real** or had its type derived from the type of the value of the constant expression.

Example:

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
parameter real size = 10;
```

Here, `size` is coerced to `10.0`.

3.4.2 Value range specification

A parameter declaration can contain optional specifications of the permissible range of the values of a parameter. More than one range can be specified for inclusion or exclusion of values as legal values for the parameter.

Brackets, **`[`** and **`]`**, indicate inclusion of the end points in the value range. Parentheses, **`(`** and **`)`**, indicate exclusion of the end points from the value range. It is possible to include one end point and not the other using **`[ )`** and **`( ]`**. The first expression in the range shall be numerically smaller than the second expression in the range.

Examples:

```
parameter real neg_rail = -15 from [-50:0);
parameter integer pos_rail = 15 from (0:50);
parameter real gain = 1 from [1:1000];
```

Here, the default value for `neg_rail` is `-15` and it is only allowed to acquire values within the range of `-50 <= neg_rail < 0`. Similarly, the default value for parameter `pos_rail` is `15` and it is only allowed to acquire values within the range of `0 < pos_rail < 50`. And, the default value for `gain` is `1` and it is allowed to acquire values within the range of `1 <= gain <= 1000`.

The keyword **`inf`** can be used to indicate infinity. If preceded by a negative sign, it indicates negative infinity.

Example:

```
parameter real val3=0 from [0:inf) exclude (10:20) exclude (30:40];
```

A single value can be excluded from the possible valid values for a parameter.

Example:

```
parameter real res = 1.0 exclude 0;
```

Here, the value of a parameter is checked against the specified range. Range checking applies to the value of the parameter for the instance and not against the default values specified in the device. It shall be an error only if the value of the parameter is out of range during simulation.

Valid values of string parameters are indicated differently. The **`from`** keyword may be used with a list of valid string values, or the exclude keyword may be used with a list of invalid string values. In either case, the list is constructed using an assignment pattern (see 4.2.14), i.e. enclosed in braces preceded by an apostrophe, **`'{ }`**, and the items are separated by commas.

Examples:

```
parameter string transistortype = "NMOS" from '{ "NMOS", "PMOS" };
parameter string filename = "output.dat" exclude '{ "" };
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

3.4.3 Parameter units and descriptions

The standard attributes for descriptions and units, described in 2.9.2, can be used for parameters.

Example:

```
(* desc="Resistance", units="Ohms" *)
parameter real res = 1.0 from [0:inf);
```

The units and descriptions are only for documentation of the module; in particular, no dimensional analysis is performed on the units. However, it is often important for the user to know the units of a parameter, such as an angle that could be specified in radians or degrees. It should be noted that the **`timescale** directive of IEEE Std 1364 Verilog also affects units throughout the module, which can be confusing to the user.

The units and descriptions are of particular value for compact models, where the number of parameters is large and the description is not always clear from the parameter name. Simulators can use this information when generating help messages for a module; many SPICE-like simulators can generate help messages with this information for built-in primitives.

Units and descriptions specified for block-level parameters shall be ignored by the simulator, but can be used for documentation purposes.

3.4.4 Parameter arrays

Verilog-AMS HDL includes behavioral extensions which utilize arrays. It requires these arrays be initialized in their definitions and allows overriding their values, as with other parameter types.Parameter arrays have the following restrictions. Failure to follow these restrictions shall result in an error.

- A type of a parameter array shall be given in the declaration.
- An array assigned to an instance of a module to override the default value of an array parameter shall be of the exact size of the parameter array, as determined by its declaration.
- Since array range in the parameter array declaration may depend on previously-declared parameters, the array size may be changed by overriding the appropriate parameters. If the array size is changed, the parameter array shall be assigned an array of the new size from the same module as the parameter assignment that changed the parameter array size.

Example:

```
parameter real poles[0:3] = '{ 1.0, 3.198, 4.554, 2.00 };
```

3.4.5 Local parameters

IEEE Std 1364 Verilog local parameters, identified by the **localparam** keyword, are identical to parameters except that they cannot directly be modified with the **defparam** statement or by the ordered or named parameter value assignment, as described in 6.3. Local parameters can be assigned constant expressions containing parameters, which can be modified with **defparam** statements or module instance parameter value assignments.

3.4.6 String parameters

String parameters can be declared. Strings are useful for parameters that act as flags, where the correspondence between numerical values and the flag values may not be obvious. The set of allowed values for the string can be specified as a comma-separated list of strings inside curly braces. String parameters may be used with the string operators listed in Table 3-3.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Example:

```
module ebersmoll (c,b,e);
   inout c, b, e;
   electrical c, b, e;
   parameter string transistortype = "NPN" from '{ "NPN", "PNP" };
   parameter real alphaf = 0.99 from (0:inf);
   parameter real alphar = 0.5 from (0:inf);
   parameter real ies = 1.0e-17 from (0:inf);
   parameter real ics = 1.0e-17 from (0:inf);
   real sign, ifor, irev;
   analog begin
      sign = (transistortype == "NPN") ? 1.0 : -1.0;
      ifor = ies * (limexp(sign*V(b,e)/$vt)-1);
      irev = ics * (limexp(sign*V(b,c)/$vt)-1);
      I(b,e) <+ sign*(ifor - alphar * irev);
      I(b,c) <+ sign*(irev - alphaf * ifor);
   end
endmodule
```

Note how the string parameter `transistortype` associates the string `"PNP"` with a negative one (-1) value for the variable `sign`. It is common in compact modeling of transistors for the equations to be formulated for NPN or NMOS devices, and behavior of a PNP or PMOS can be described by multiplying all the voltages and currents by -1, even though the “p” denotes positively-charged carriers in the channel of the PMOS.

3.4.7 Parameter aliases

Aliases can be defined for parameters. This allows an alternate name to be used when overriding module parameter values as described in 6.3. Parameters with different names may be used for the same purpose in different simulators; some compact models accept parameter names with the letter “O” in place of the number “0.”

Parameter aliases are subject to the following rules.

- The type of an alias (**`real`**, **`integer`**, or **`string`**) shall be determined by the original parameter, as is its range of allowed values, if specified.
- The *alias_identifier* shall not occur anywhere else in the module; in particular, it shall not conflict with a different *parameter_identifier*, and the equations in the module shall reference the parameter by its original name, not the alias.
- Multiple aliases can point to the same parameter.
- When overriding parameters, it shall be an error to specify an override for a parameter by its original name and one or more aliases, or by more than one alias, regardless of how the override is done (by name or using the **`defparam`** statement).
- When the simulator generates a list of parameter values used, such as for an operating point analysis, only the original name shall appear in the list.

For example, suppose a module named `nmos2` has the following declarations in the module:

```
parameter real dtemp = 0 from [-`P_CELSIUS0:inf);
aliasparam trise = dtemp;
```

Then the following two instantiations of the module are valid:

```
nmos2 #(.trise(5)) m1(.d(d), .g(g), .s(s), .b(b));
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
nmos2 #(.dtemp(5)) m2(.d(d), .g(g), .s(s), .b(b));
```

and the value of the parameter `dtemp`, as used in the module equations for both instances, is 5.

This last instantiation is an error:

```
nmos2 #(.trise(5), .dtemp(5)) m3(.d(d), .g(g), .s(s), .b(b)); //error
```

because an override is specified for the parameter `dtemp` and its alias, even though the values are equal.

Parameter aliases may also be declared for the hierarchical parameter system functions (see 9.18) as in the example below:

```
aliasparam m = $mfactor;
```

3.4.8 Multidimensional parameter array examples

The following example demonstrates the usage of a multidimensional real array parameter and various usages of assignment patterns.

```
module test;
   electrical out[0:2];
   electrical in[0:2];
   /* Instantiate crosstalk module passing a
    * multidimensional parameter array literal
    * for channel coupling
    */
   crosstalk #(.c('{'{0.0,0.1,0.1},'{0.1,0.0,0.1},'{0.1,0.1,0.0}}))
      C1(out,in,1'b1);

   gen G1(in);
   sink S1(out);
endmodule

module crosstalk(out, in, distort_enable);
   input in[0:2];
   input distort_enable;
   output out[0:2];
   // A multidimensional real parameter array for channel coupling
   parameter real c[0:2][0:2] =
            '{'{0.0,0.2,0.2},'{0.2,0.0,0.2},'{0.2,0.2,0.0}};

   electrical in[0:2];
   electrical out[0:2];

   /* A multidimensional real variable to hold the distortion calculations
    * all elements are initialized to 0.0 using
    * an assignment pattern and replication operator
    */
   real distort[0:2][0:2] = '{ 3{ '{3{0.0}}}};

   /* multidimensional string to flag excessive distortion
    * all elements are initialized to " " using
    * an assignment pattern and replication operator
    */
   string above_0p5[0:2][0:2] = '{ 3{ '{3{" "}}}};

   real in_val[0:2];
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    integer ii, jj;
    analog begin
       // assign to variable using an assignment pattern
       in_val = '{V(in[0]),V(in[1]),V(in[2])};

       if (distort_enable) begin
          for( ii=0; ii <= 2; ii=ii+1 ) begin
             for (jj=0; jj<= 2; jj=jj+1 ) begin
                distort[ii][jj] = c[ii][jj]*in_val[jj];
                if (distort[ii][jj] > 0.1)
                   above_0p5[ii][jj] = "*";
             end
          end
       end

       V(out[0]) <+ in_val[0]     + distort[0][1] + distort[0][2];
       V(out[1]) <+ distort[1][0] + in_val[1]     + distort[1][2];
       V(out[2]) <+ distort[2][0] + distort[2][1] + in_val[2];

       @(final_step) begin
          $display("Table of distortions greater than 0.5");
          $display("#012"); // write the table header
          for( ii=0; ii <= 2; ii=ii+1 ) begin
             $write("%0d",ii); // %0d means write int in minimum width
             for (jj=0; jj<= 2; jj=jj+1 ) begin
                $write(above_0p5[ii][jj]);
             end
             $display; // print a newline
          end
       end
    end
endmodule
```

3.5 Genvars

Genvars are integer-valued variables which compose static expressions for instantiating structure behaviorally such as accessing analog signals within behavioral looping constructs. The syntax for declaring genvar variables is shown in Syntax 3-3.

---

genvar_declaration ::= *// from A.4.2*
    genvar list_of_genvar_identifiers **;**

list_of_genvar_identifiers ::=
    genvar_identifier { **,** genvar_identifier }

Syntax 3-3—Syntax for genvar declaration

The static nature of genvar variables is derived from the limitations upon the contexts in which their values can be assigned.

Examples:

```
genvar i;
analog begin
   ...
   for (i = 0; i < 8; i = i + 1) begin
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
        V(out[i]) <+ transition(value[i], td, tr);
      end
      ...
  end
```

The genvar variable `i` can only be assigned within the for-loop control. Assignments to the genvar variable `i` can consist only of expressions of static values, e.g., parameters, literals, and other genvar variables.

3.6 Net_discipline

In addition to the data types supported by IEEE Std 1364 Verilog, an additional data type, *net_discipline*, is introduced in Verilog-AMS HDL for continuous time and mixed-signal simulation. *net_discipline* is used to declare analog nets, as well as declaring the domains of digital nets and regs.

A signal can be digital, analog, or mixed, and is a hierarchical collection of nets which are contiguous (because of port connections). For analog and mixed signals, a single node is associated with all continuous net segments of the signal. The fundamental characteristic of analog and mixed signals is the values of the associated *node* are determined by the simultaneous solution of equations defined by the instances connected to the node using Kirchhoff's conservation laws. In general, a *node* represents a point of physical connections between nets of continuous-time description and it obeys conservation-law semantics.

A net is characterized by the discipline it follows. For example, all low-voltage nets have certain common characteristics, all mechanical nets have certain common characteristics, etc. Therefore, a *net* is always declared as a type of discipline. In this sense, a discipline is a user-defined type for declaring a net.

A *discipline* is characterized by the domain and the attributes defined in the *natures* for **`potential`** and **`flow`**.

3.6.1 Natures

A *nature* is a collection of attributes. In Verilog-AMS HDL, there are several pre-defined attributes. In addition, user-defined attributes can be declared and assigned constant values in a nature.

The nature declarations are at the same level as discipline and module declarations in the source text. That is, natures are declared at the top level and nature declarations do not nest inside other nature declarations, discipline declarations, or module declarations.

The syntax for defining a nature is shown in Syntax 3-4.

```
nature_declaration ::=                                          // from A.1.6
        nature nature_identifier [ : parent_nature ] [ ; ]
            { nature_item }
        endnature
parent_nature ::=
        nature_identifier
      | discipline_identifier . potential_or_flow
nature_item ::= nature_attribute
nature_attribute ::= nature_attribute_identifier = nature_attribute_expression ;
potential_or_flow ::= potential | flow                          // from A.1.7
nature_attribute_identifier ::=                                 // from A.9.3
        abstol | access | ddt_nature | idt_nature | units | identifier
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Syntax 3-4—Syntax for nature declaration

A nature shall be defined between the keywords **nature** and **endnature**. Each nature definition shall have a unique identifier as the name of the nature and shall include all the required attributes specified in 3.6.1.2.

Examples:

```
nature current;
   units = "A";
   access = I;
   idt_nature = charge;
   abstol = 1u;
endnature

nature voltage;
   units = "V";
   access = V;
   abstol = 1u;
endnature
```

3.6.1.1 Derived natures

A nature can be derived from an already declared nature. This allows the new nature to have the same attributes as the attributes of the existing nature. The new nature is called a *derived nature* and the existing nature is called a *parent nature.* If a nature is not derived from any other nature, it is called a *base nature*.

In order to derive a new nature from an existing nature, the new nature name shall be followed by a colon (**:**) and the name of the parent nature in the nature definition.

A derived nature can declare additional attributes or override attribute values of the parent nature, with certain restrictions (as outlined in 3.6.1.2) for the predefined attributes.

The attributes of the derived nature are accessed in the same manner as accessing attributes of any other nature.

Examples:

```
nature ttl_curr;
   units = "A";
   access = I;
   abstol = 1u;
endnature

// An alias
nature ttl_net_curr : ttl_curr;
endnature

nature new_curr : ttl_curr;   // derived, but different
   abstol = 1m;               // modified for this nature
   maxval = 12.3;             // new attribute for this nature
endnature
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

3.6.1.2 Attributes

Attributes define the value of certain quantities which characterize the nature. There are five predefined attributes: **abstol**, **access**, **idt_nature**, **ddt_nature**, and **units**. In addition, user-defined attributes can be defined in a nature (see 3.6.1.3). Attribute declaration assigns a constant expression to the attribute name, as shown in the example in 3.6.1.1.

abstol

The **abstol** attribute is a real value constant expression that provides a tolerance measure (metric) for convergence of potential or flow calculations. It specifies the maximum negligible value for signals associated with the nature.

This attribute is required for all base natures. It is legal for a derived nature to change **abstol**, but if left unspecified it shall inherit the **abstol** from its parent nature.

access

The **access** attribute identifies the name for the access function. When the nature is used to bind a potential, the name is used as an access function for the potential; when the nature is used to bind a flow, the name is used as an access function for the flow. The usage of access functions is described further in 4.4.

This attribute is required for all base natures. The constant expression assigned to it shall be an identifier (by name, not as a string).

It is illegal for a derived nature to change the access attribute; the derived nature always inherits the access attribute of its parent nature.

idt_nature

The **idt_nature** attribute provides a relationship between a nature and the nature representing its time integral.

idt_nature can be used to reduce the need to specify tolerances on the **idt()** operator. If this operator is applied directly on nets, the tolerance can be taken from the node, which eliminates the need to give a tolerance with the operator.

If specified, the constant expression assigned to **idt_nature** shall be the name (not a string) of a nature which is defined elsewhere. It is possible for a nature to be self-referencing with respect to its **idt_nature** attribute. In other words, the value of **idt_nature** can be the **nature** that the attribute itself is associated with.

The **idt_nature** attribute is optional; the default value is the **nature** itself. While it is possible to override the parent's value of **idt_nature** using a derived nature, the nature thus specified shall be related (share the same base nature) to the nature the parent uses for its **idt_nature**.

ddt_nature

The **ddt_nature** attribute provides a relationship between a nature and the nature representing its time derivative.

ddt_nature can be used to reduce the need to specify tolerances on the **ddt()** operator. If this operator is applied directly on nets, the tolerance can be taken from the node, eliminating the need to give a tolerance with the operator.

If specified, the constant expression assigned to **ddt_nature** shall be the name (not a string) of a nature which is defined elsewhere. It is possible for a nature to be self-referencing with respect to its **ddt_nature** attribute. In other words, the value of **ddt_nature** can be the **nature** that the attribute itself is associated with.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The **ddt_nature** attribute is optional; the default value is the **nature** itself. While it is possible to override the parent's value of **ddt_nature** using a derived nature, the nature thus specified shall be related (share the same base nature) to the nature the parent uses for its **ddt_nature**.

units

The **units** attribute provides a binding between the value of the access function and the units for that value. The **units** field is provided so simulators can annotate the continuous signals with their units and is also used in the *net compatibility rule check*.

This attribute is required for all base natures. It is illegal for a derived nature to define or change the **units**; the derived nature always inherits its parent nature **units**. If specified, the constant expression assigned to it shall be a string.

3.6.1.3 User-defined attributes

In addition to the predefined attributes listed above, a nature can specify other attributes which can be useful for analog modeling. Typical examples include certain maximum and minimum values to define a valid range.

A user-defined attribute can be declared in the same manner as any predefined attribute. The name of the attribute shall be unique in the nature being defined and the value being assigned to the attribute shall be constant.

3.6.2 Disciplines

A *discipline* description consists of specifying a **domain** type and binding any *natures* to **potential** or **flow**.

The syntax for declaring a discipline is shown in Syntax 3-5.

```
discipline_declaration ::=                                    // from A.1.7
        discipline discipline_identifier [ ; ]
            { discipline_item }
        enddiscipline
discipline_item ::=
        nature_binding
      | discipline_domain_binding
      | nature_attribute_override
nature_binding ::= potential_or_flow nature_identifier ;
potential_or_flow ::= potential | flow
discipline_domain_binding ::= domain discrete_or_continuous ;
discrete_or_continuous ::= discrete | continuous
nature_attribute_override ::= potential_or_flow . nature_attribute
```

Syntax 3-5—Syntax for discipline declaration

A *discipline* shall be defined between the keywords **discipline** and **enddiscipline**. Each discipline shall have a unique identifier as the name of the discipline.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The discipline declarations are at the same level as *nature* and *module* declarations in the source text. That is, disciplines are declared at the top level and discipline declarations do not nest inside other discipline declarations, nature declarations, or module declarations. Analog behavioral nets (nodes) must have a discipline defined for them but interconnect and digital nets do not. It is possible to set the discipline of interconnect and digital nets through discipline declaration with hierarchical references to these nets. It shall be an error to hierarchically override the discipline of a net that was explicitly declared unless it is a compatible discipline.

3.6.2.1 Nature binding

Each discipline can bind a *nature* to its **`potential`** and **`flow`**.

Only the name of the nature is specified in the discipline. The nature binding for potential is specified using the keyword **`potential`**. The nature binding for flow is specified using the keyword **`flow`**.

The access function defined in the nature bound to potential is used in the model to describe the signal-flow which obeys Kirchhoff's Potential Law (KPL). This access function is called the *potential access function*.

The access function defined in the nature bound to flow is used in the model to describe a quantity which obeys Kirchhoff's Flow Law (KFL). This access function is called the *flow access function*.

Disciplines with two natures are called *conservative disciplines* and the nets associated with conservative disciplines are called *conservative nets*. Conservative disciplines shall not have the same *nature* specified for both the **`potential`** and the **`flow`**. Disciplines with a single nature are called *signal-flow disciplines* and the nets with signal-flow disciplines are called *signal-flow nets*. A signal-flow discipline may specify either the potential or the flow nature, as shown in the following examples.

Examples:

Conservative discipline

```
discipline electrical;
   potential Voltage;
   flow Current;
enddiscipline
```

Signal-flow disciplines

```
discipline voltage;
   potential Voltage;
enddiscipline

discipline current;
   flow Current;
enddiscipline
```

Multi-disciplinary example

Disciplines in Verilog-AMS HDL allow designs of multiple disciplines to be easily defined and simulated. Disciplines can be used to allow unique tolerances based on the size of the signals and outputs displayed in the actual units of the discipline. This example shows how an application spanning multiple disciplines can be modeled in Verilog-AMS HDL. It models a DC-motor driven by a voltage source.

```
module motorckt;
   parameter real freq=100;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   electrical gnd; ground gnd;

   electrical drive;
   rotational shaft;

   motor m1 (drive, gnd, shaft);
   vsine #(.freq(freq), .ampl(1.0)) v1 (drive, gnd);

endmodule

// vp:  positive terminal [V,A]     vn:  negative terminal [V,A]
// shaft:motor shaft [rad,Nm]
// INSTANCE parameters
// Km = motor constant [Vs/rad]  Kf = flux constant [Nm/A]
// j = inertia factor [Nms^2/rad] D= drag (friction) [Nms/rad]
// Rm = motor resistance [Ohms]  Lm = motor inductance [H]
// A model of a DC motor driving a shaft
module motor(vp, vn, shaft);
   inout vp, vn, shaft;
   electrical vp, vn;
   rotational shaft;

   parameter real Km = 4.5, Kf = 6.2;
   parameter real j = 0.004, D = 0.1;
   parameter real Rm = 5.0, Lm = 0.02;

   analog begin
      V(vp, vn) <+ Km*Theta(shaft) + Rm*I(vp, vn) + ddt(Lm*I(vp, vn));
      Tau(shaft) <+ Kf*I(vp, vn) - D*Theta(shaft) - ddt(j*Theta(shaft));
   end
endmodule
```

3.6.2.2 Domain binding

Analog signal values are represented in continuous time, whereas digital signal values are represented in discrete time. The **domain** attribute of the discipline stores this property of the signal. It takes two possible values, **discrete** or **continuous.** Signals with continuous-time domains are real valued. Signals with discrete-time domains can either be binary (0, 1, X, or Z), integer or real values.

Examples:

```
discipline electrical;
   domain continuous;
   potential Voltage;
   flow Current;
enddiscipline

discipline ddiscrete;
   domain discrete;
enddiscipline
```

The **domain** attribute is optional. The default value for **domain** is **continuous** for disciplines which specify **nature** bindings. It is an error for a discipline to have a **domain** binding of **discrete** if it has **nature** bindings.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

3.6.2.3 Natureless disciplines and domainless disciplines

It is possible to define a discipline with no nature bindings. These are known as natureless disciplines (historically referred to as empty disciplines).

Such disciplines may have a **domain** binding or they may be domainless, thus allowing the domain to be determined by the connectivity of the net (see 7.4 and Annex F).

Disciplines without a **domain** binding and without a **nature** binding are known as domainless disciplines. The **domain** binding of a discipline with **nature** bindings defaults to **continuous** if not specified. A **discipline** with **nature** bindings cannot be a domainless discipline.

Example:

```
discipline natureless;
   domain continuous;
enddiscipline

discipline domainless
enddiscipline
```

Usage of domainless disciplines and continuous natureless disciplines is discouraged. Domainless and continuous natureless disciplines are provided for backward compatibility with previous versions of the Verilog-AMS and Verilog-A standards. Furthermore, domainless disciplines are deprecated and the definition of a domainless discipline may be made an error in future versions of Verilog-AMS HDL.

3.6.2.4 Discipline of nets and undeclared nets

It is possible for a module to have nets where there are no discipline declarations. If such a net appears bound only to ports in module instantiations, it may have no declaration at all or may be declared to have a net type such as **wire**, **tri**, **wand**, **wor**, etc. If it is referenced in behavioral code, then it must have a net type.

In these cases, the net shall be treated as having no discipline. If the net is referenced in behavioral code, then it shall be treated as having no discipline with a domain binding of **discrete**, otherwise it shall be treated as having no discipline and no domain binding. If a net has a wire type but is not connected to behavioral code (interconnect) and it resolved to domain **discrete** then its wire type shall be used in any net type resolution steps per IEEE Std 1364 Verilog.

The discipline and domain of all nets of a mixed or continuous signal is determined by discipline resolution if these nets do not already have a declared discipline and domain binding (see 7.4 and Annex F).

3.6.2.5 Overriding nature attributes from discipline

A discipline can override the value of the bound nature for the pre-defined attributes (except as restricted by 3.6.1.2), as shown for the flow `ttl_curr` in the example below. To do so from a discipline declaration, the bound nature and attribute needs to be defined, as shown for the **abstol** value within the discipline `ttl` in the example below. The general form is shown as the *nature_attribute_override* nonterminal in Syntax 3-5: the keyword **flow** or **potential**, then the hierarchical separator **.** and the attribute name, and, finally, set all of this equal to (=) the new value (e.g., `flow.abstol = 10u`).

Examples:

```
nature ttl_curr;
   units = "A";
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   access = I;
   abstol = 1u;
endnature

nature ttl_volt;
   units = "V";
   access = V;
   abstol = 100u;
endnature

discipline ttl;
   potential ttl_volt;
   flow ttl_curr;
   flow.abstol = 10u;
enddiscipline
```

3.6.2.6 Deriving natures from disciplines

A nature can be derived from the *nature* bound to the **potential** or **flow** in a discipline. This allows the new nature to have the same attributes as the attributes for the nature bound to the **potential** or the **flow** of the discipline.

If the nature bound to the potential or the flow of a discipline changes, the new nature shall automatically inherit the attributes for the changed nature.

In order to derive a new nature from flow or potential of a discipline, the nature declaration shall also include the discipline name followed by the hierarchical separator (**.**) and the keyword **flow** or **potential**, as shown for `ttl_net_curr` in the example below.

A nature derived from the flow or potential of a discipline can declare additional attributes or override values of the attributes already declared.

Examples:

```
nature ttl_net_curr : ttl.flow; // from the example in 3.6.2.5
endnature      // abstol = 10u as modified in ttl

nature ttl_net_volt : ttl.potential; // from the example in 3.6.2.5
   abstol = 1m;      // modified for this nature
   maxval = 12.3;    // new attribute for this nature
endnature
```

3.6.2.7 User-defined attributes

Like natures, a **discipline** can specify user-defined attributes. Discipline user-defined attributes are useful for the same reasons as nature user-defined attributes (see 3.6.1.3).

3.6.3 Net_discipline declaration

Each *net_discipline* declaration associates nets with an already declared discipline. Syntax 3-6 shows how to declare disciplines of nets and regs.

---

net_declaration ::= *// from A.2.1.3*

```
    ...
  | discipline_identifier [ range ] list_of_net_identifiers ;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    | discipline_identifier [ range ] list_of_net_decl_assignments ;
      ...
range ::= [ msb_constant_expression : lsb_constant_expression ]                    // from A.2.5
list_of_net_decl_assignments ::= net_decl_assignment { , net_decl_assignment }     // from A.2.3
list_of_net_identifiers ::= ams_net_identifier { , ams_net_identifier }
net_decl_assignment ::= ams_net_identifier = expression                             // from A.2.4
```

Syntax 3-6—Syntax for net discipline declaration

If a range is specified for a net, the net is called a *vector net*; otherwise it is called a *scalar net*. A vector net is also called a *bus*.

Examples:

```
electrical [MSB:LSB] n1; // MSB and LSB are parameters
voltage [5:0] n2, n3;
magnetic inductor;
ddiscrete [10:1] connector1;
```

Nets represent the abstraction of information about signals. As with ports, nets represent component interconnections. Nets declared in the module interface define the ports to the module (see 6.5).

A net used for modeling a conservative system shall have a discipline with both access functions (**`potential`** and **`flow`**) defined. When modeling a signal-flow system, the discipline of a net can have only **`potential`** access functions. When modeling a discrete system, the discipline of a net can only have a **`domain`** of **`discrete`** defined.

Nets declared with a natureless discipline or declared without a discipline do not have declared natures, so such nets can not be used in analog behavioral descriptions (because the access functions are not known). However, such nets can be used in structural descriptions, where they inherit the natures from the ports of the instances of modules that connect to them.

3.6.3.1 Net descriptions

Nets can be declared with a description attribute. This information can be used by the simulator to generate help messages for a module.

Example:

```
(* desc="drain terminal" *) electrical d;
```

If a net is also a module port, the description attribute may also be specified on the port declaration line (in which the net is declared as **`input`**, **`inout`**, or **`output`**). If the description attribute is specified for the same *net_identifier* in both the net discipline declaration and the port declaration, then the last attribute value shall be used and the tool can give a warning that a duplicate attribute specification has occurred.

3.6.3.2 Net Discipline Initial (Nodeset) Values

Nets with continuous disciplines are allowed to have initializers on their net discipline declarations; however, nets of non-continuous disciplines are not.

```
electrical a = 5.0;
electrical [0:4] bus = '{2.3,4.5,,6.0};
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
mechanical top.foo.w = 250.0;
```

The initializer shall be a *constant_expression* and will be used as a nodeset value for the potential of the net by the analog solver. In the case of analog buses, a constant array expression is used as an initializer. A null value in the constant array indicates that no nodeset value is being specified for this element of the bus.

If different nets of a node have conflicting initializers, then initializers on hierarchical net declarations win. If there are multiple hierarchical declarations, then the declaration on the highest level wins. If there are multiple hierarchical declarations on the highest level, then it is a race condition for which the initializer wins. If the multiple conflicting initializers are not hierarchical, then it is also a race condition for which the initializer wins.

3.6.4 Ground declaration

Each ground declaration is associated with an already declared net of continuous discipline. The node associated with the net will be the global reference node in the circuit. The net must be assigned a continuous discipline to be declared ground.

Syntax 3-7 shows the syntax used for declaring the global reference node (*ground*).

```
net_declaration ::=                                          // from A.2.1.3
      ...
    | ground [ discipline_identifier ] [ range ] list_of_net_identifiers ;
```

Syntax 3-7—Syntax for declaring ground

Examples:

```
module loadedsrc(in, out);
   input in;
   output out;
   electrical in, out;
   electrical gnd;
   ground gnd;
   parameter real srcval = 5.0;

   resistor #(.r(10K)) r1(out,gnd);
   analog begin
      V(out) <+ V(in,gnd)*2;
   end
endmodule
```

3.6.5 Implicit nets

Nets can be used in structural descriptions without being declared. In this case, the net's discipline and domain binding will be determined by discipline resolution (see 7.4 and Annex F).

Examples:

```
module top(i1, i2, o1, o2, o3);
   input i1, i2;
   output o1, o2, o3;
   electrical i1, i2, o1, o2, o3;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   // ab1, ab2, cb1, cb2 are implicit nets, not declared
   blk_a a1( i1, ab1 );
   blk_a a2( i2, ab2 );
   blk_b b1( ab1, cb1 );
   blk_b b2( ab2, cb2 );
   blk_c c1( o1, o2, o3, cb1, cb2);
endmodule
```

3.7 Real net declarations

The **wreal**, or real net data type, represents a real-valued physical connection between structural entities. A wreal net shall not store its value. A **wreal** net can be used for real-valued nets which are driven by a single driver, such as a continuous assignment. If no driver is connected to a wreal net, its value shall be zero (`0.0`). Unlike other digital nets which have an initial value of ‘z’, wreal nets shall have an initial value of zero.

wreal nets can only be connected to compatible interconnect and other wreal or real expressions. They cannot be connected to any other wires, although connection to explicitly declared 64-bit wires can be done via system tasks $realtobits and $bitstoreal. Compatible interconnect are nets of type **wire**, **tri**, and **wreal** where the IEEE Std 1364 Verilog net resolution is extended for **wreal**. When the two nets connected by a port are of net type **wreal** and **wire**/**tri**, the resulting single net will be assigned as **wreal**. Connection to other net types will result in an error.

Syntax 3-8 shows the syntax for declaring digital nets.

---

net_declaration ::= *// from A.2.1.3*

```
    ...
  | wreal [ discipline_identifier ] [ range] list_of_net_identifiers ;
  | wreal [ discipline_identifier ] [ range] list_of_net_decl_assignments ;
```

Syntax 3-8—Syntax for declaring digital nets

Examples:

```
module drv(in, out);
   input in;
   output out;
   wreal in;
   electrical out;
   analog begin
      V(out) <+ in;
   end
endmodule

module top();
   real stim;
   electrical load;
   wreal wrstim;
   assign wrstim = stim;
   drv f1(wrstim, load);
   always begin
      #1 stim = stim + 0.1;
   end
endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

3.8 Default discipline

Verilog-AMS HDL supports the **`default_discipline** compiler directive. This directive specifies a default discrete discipline to be applied to any discrete net which does not have an explicit discipline declaration as part of discipline resolution (see 7.4 and Annex F). A description and its syntax is shown in 10.2.

3.9 Disciplines of primitives

With internal simulator primitives the discipline of the `vpiLoConn` to be used in discipline resolution during a mixed-signal simulation must be known. For digital primitives the domain is discrete and thus the discipline is set via the default_discipline directive as it is for digital modules. If the discipline of digital connections (`vpiLoConn`) to a mixed net are unknown then the default_discipline must be specified (via the directive or other vendor specific method). If not specified, an error will result during discipline resolution.

For analog primitives, the discipline will be defined by the attribute port_discipline on that instance. If no attribute is found then it will acquire the discipline of other compatible continuous disciplines connected to that net segment. If no disciplines are connected to that net, then the default discipline is set to electrical. This is further described in E.3.2.2.

3.10 Discipline precedence

While a net itself can be declared only in the module to which it belongs, the discipline of the net can be specified in a number of ways.

— The discipline name can appear in the declaration of the net.

— The discipline name can be used in a declaration which makes an out of context reference to the net from another module.

Discipline conflicts can arise if more than one of these methods is applied to the same net. Discipline conflicts shall be resolved using the following order of precedence:

1) A declaration from a module other than the module to which the net belongs using an out-of-module reference, e.g.,

```
module example1;
   electrical example2.net;
endmodule
```

2) The local declaration of the net in the module to which it belongs, e.g.,

```
module example2;
   electrical net;
endmodule
```

3) Discipline resolution (see 7.4 and Annex F)

It is not legal to have two different disciplines at the same level of precedence for the same net.

3.11 Net compatibility

Certain operations can be done on nets only if the two (or more) nets are compatible. For example, if an access function has two nets as arguments, they must be compatible. The following rules shall apply to determine the compatibility of two (or more) nets:

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Discrete Domain Rule: Digital nets with the same signal value type (i.e., **`real`**, **`integer`**) are compatible with each other if their disciplines are compatible (i.e., the discipline has a discrete domain or is empty.

Signal Domain Rule: It shall be an error to connect two ports or nets of different domains unless there is a connect statement (see 7.4) defined between the disciplines of the nets or ports.

Signal Connection Rule: It shall be an error to connect two ports or nets of the same domain with incompatible disciplines.

3.11.1 Discipline and Nature Compatibility

The following rules shall apply to determine discipline compatibility:

- *Self Rule (Discipline):* A discipline is compatible with itself.
- *Natureless Discipline Rule:* A natureless discipline is compatible with all other disciplines of the same domain.
- *Domainless Discipline Rule*: A domainless discipline is compatible with all disciplines as there is no nature or domain conflict. Note that domainless disciplines are deprecated.
- *Domain Incompatibility Rule:* Disciplines with different domain attributes are incompatible.
- *Potential Incompatibility Rule:* Disciplines with incompatible potential natures are incompatible.
- *Flow Incompatibility Rule:* Disciplines with incompatible flow natures are incompatible.

The following rules shall apply to determine nature compatibility:

- S*elf Rule (Nature):* A nature is compatible with itself.
- *Non-Existent Binding Rule:* A nature is compatible with a non-existent discipline binding.
- *Base Nature Rule:* A derived nature is compatible with its base nature.
- *Derived Nature Rule:* Two natures are compatible if they are derived from the same base nature.
- *Units Value Rule:* Two natures are compatible if they have the same value for the units attribute.

The following examples illustrate these rules.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
nature Voltage;
   access = V;
   units = "V";
   abstol = 1u;
endnature

nature Current;
   access = I;
   units = "A";
   abstol = 1p;
endnature

nature highvoltage: Voltage;
   abstol = 1.0;
endnature

discipline electrical;
   potential Voltage;
   flow Current;
enddiscipline

discipline highvolt;
   potential highvoltage;
   flow Current;
enddiscipline

discipline sig_flow_v;
   potential Voltage;
enddiscipline

discipline sig_flow_i;
   flow Current;
enddiscipline

nature Position;
   access = X;
   units = "m";
   abstol = 1u;
endnature
```

```
nature Force;
   access = F;
   units = "N";
   abstol = 1n;
endnature

discipline rotational;
   potential Position;
   flow Force;
enddiscipline

discipline sig_flow_x;
   potential Position;
enddiscipline

discipline sig_flow_f;
   flow Force;
enddiscipline

discipline domainless;
enddiscipline

discipline ddiscrete;
   domain discrete;
enddiscipline

discipline natureless;
   domain continuous;
enddiscipline

discipline continuous_elec;
   domain continuous;
   potential Voltage;
   flow Current;
enddiscipline
```

The following compatibility observations can be made from the above examples:

— `Voltage` and `highvoltage` are compatible natures because they both exist and are derived from the same base natures.

— `electrical` and `highvolt` are compatible disciplines because the natures for both potential and flow exist and are derived from the same base natures.

— `electrical` and `sig_flow_v` are compatible disciplines because the nature for potential is same for both disciplines and the nature for flow does not exist in `sig_flow_v`.

— `electrical` and `rotational` are incompatible disciplines because the natures for both potential and flow are not derived from the same base natures.

— `electrical` and `sig_flow_x` are incompatible disciplines because the nature for both potentials are not derived from the same base nature.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

— The natureless discipline `natureless` is compatible with all other disciplines of the same domain (i.e continuous) because it does not have a potential or a flow nature. Without natures, there can be no conflicting natures.

— `domainless` is compatible with all other disciplines from the domainless discipline rule.

— `electrical` and `ddiscrete` are incompatible disciplines because the domains are different. A connect statement must be used to connect nets or ports of these disciplines together.

— `electrical` and `continuous_elec` are compatible disciplines because the default domain for discipline `electrical` is continuous and the specified natures for potential and flow are the same.

3.12 Branches

A *branch* is a path between two nets. If both nets are conservative, then the branch is a *conservative branch* and it defines a branch potential and a branch flow. If one net is a signal-flow net, then the branch is a *signal-flow branch* and it defines either a branch potential or a branch flow, but not both.

Each branch declaration is associated with two nets from which it derives a discipline. These nets are referred to as the *branch terminals*. Only one net need be specified, in which case the second net defaults to `ground` and the discipline for the branch is derived from the specified net. The disciplines for the specified nets shall be compatible (see 3.11).

Branches can either be explicitly or implicitly declared. Explicitly declared branches are referred to as named branches. The syntax for declaring named branches is shown in Syntax 3-9. Unnamed branches are created by applying an access function (see 4.4 and 5.4.1) to either a net or a pair of nets. If the access function is applied to a single net, then the branch is formed between that net and the global reference node (ground). If it is applied to a pair of nets, the branch is formed between the two nets. There shall be at most one unnamed branch between any two nets or between a net and implicit ground (in addition to any number of named branches).

```
branch_declaration ::=                                          // from A.2.1.3
        branch ( branch_terminal [ , branch_terminal ] ) list_of_branch_identifiers ;
      | port_branch_declaration
port_branch_declaration ::=
        branch ( < port_identifier > ) list_of_branch_identifiers ;
      | branch ( < hierarchical_port_identifier > ) list_of_branch_identifiers ;
branch_terminal ::=
        net_identifier
      | net_identifier [ constant_expression ]
      | net_identifier [ constant_range_expression ]
      | hierarchical_net_identifier
      | hierarchical_net_identifier [ constant_expression ]
      | hierarchical_net_identifier [ constant_range_expression ]
list_of_branch_identifiers ::=                                  // from A.2.3
        branch_identifier [ range ] { , branch_identifier [ range ] }
```

Syntax 3-9—Syntax for branch declaration

If one of the terminals of a branch is a vector net, then the other terminal shall either be a scalar net or a vector net of the same size. In the latter case, the branch is referred to as a *vector branch*. When both terminals are vectors, the scalar branches that make up the vector branch connect to the corresponding scalar nets of the vector terminals, as shown in Figure 3-1.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

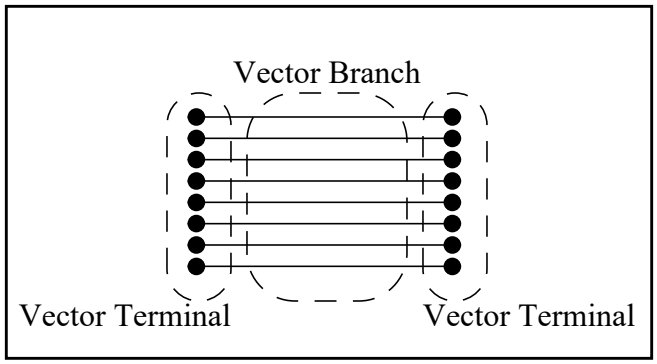


Figure 3-1: Two vector terminals

When one terminal is a vector and the other is a scalar, a singular scalar branch connects to each scalar net in the vector terminal and each terminal of the vector branch connects to the scalar terminal, as shown in Figure 3-2.

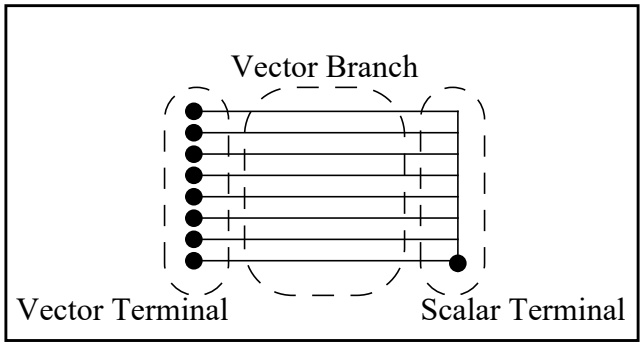


Figure 3-2: One vector and one scalar terminal

If the range of the *vector branch* is not specified then the indexing of the *vector branch* shall start at 0. For example:

```
electrical [3:5]a;
electrical [1:3]b;
branch (a,b) br1; // Branch br1 is of size 3 and can be indexed from 0 to 2
```

3.12.1 Port Branches

A port branch is a special type of branch used to access the flow into a port of a module (see 5.4.3). It is a branch between the upper and lower connections of the port. A port branch is a scalar branch if the port identifier is a scalar port. A port branch is a vector branch if the port identifier is a vector port.

Example:

```
module current_sink(p);
   electrical p;
   branch (<p>) probe_p;
   analog
      $strobe("current probed is %g", I(probe_p));
endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

3.13 Namespace

The following subsections define the namespace.

3.13.1 Nature and discipline

Natures and disciplines are defined at the same level of scope as modules. Thus, identifiers defined as natures or disciplines have a global scope, which allows nets to be declared inside any module in the same manner as an instance of a module.

3.13.2 Access functions

Each access function name, defined before a module is parsed, is automatically added to that module's namespace unless there is another identifier defined with the same name as the access function in that module's namespace. Furthermore, the access function of each base nature shall be unique.

3.13.3 Net

The scope rules for net identifiers are the same as the scope rules for any other identifier declarations, except nets can not be declared anywhere other than in the port of a module or in the module itself. A net can only be declared inside a module scope; a net can not be declared local to a block.

Access functions are uniquely defined for each net based on the discipline of the net. Each access function is used with the name of the net as its argument and a net can only be accessed through its access functions.

The hierarchical reference character (`.`) can be used to reference a net across the module boundary according to the rules specified in IEEE Std 1364 Verilog.

3.13.4 Branch

The scope rules for branch identifiers are the same as the scope rules for net identifiers. A branch can only be declared inside a module scope; a branch can not be declared local to a block.

Access functions are uniquely defined for each branch based on the discipline of the branch. The access function is used with the name of the branch as its argument and a branch can only be accessed through its access functions.

The hierarchical reference character (`.`) can be used to reference a branch across the module boundary according to the rules specified in IEEE Std 1364 Verilog.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

4. Expressions

4.1 Overview

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

An *expression* is a construct which combines *operands* with *operators* to produce a result which is a function of the values of the operands and the semantic meaning of the operator. Any legal operand, such as an integer or an indexed element from an array of reals, without a operator is also considered an expression. Wherever a value is needed in a Verilog-AMS HDL statement, an expression can be used.

Some statement constructs require an expression to be a *constant expression*. The operands of a constant expression consists of constant numbers and parameter names, but they can use any of the operators defined in Table 4-1, Table 4-14, and Table 4-15.

4.2 Operators

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

Table 4-1—Operators

{} {{}}	Concatenation, replication
unary +, unary -	Unary operators
+ - * / **	Arithmetic
%	Modulus
> >= < <=	Relational
!	Logical negation
&&	Logical and
||	Logical or
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality
~	Bitwise negation
&	Bitwise and
|	Bitwise inclusive or
^	Bitwise exclusive or
^~ or ~^	Bitwise equivalence
&	Reduction and
~&	Reduction nand
\| | Reduction or |

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 4-1—Operators *(continued)*

~|	Reduction nor
^	Reduction xor
~^ or ^~	Reduction xnor
<<	Logical left shift
>>	Logical right shift
<<<	Arithmetic left shift
>>>	Arithmetic right shift
?:	Conditional

4.2.1 Operators with real operands

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

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

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

The result of using logical or relational operators on real numbers is an integer value `0` (*false*) or `1` (*true*).

If a real expression is used for the replication factor of a concatenation, the expression will first be converted to an integer value using the rules described in 4.2.1.1, before it is used as the replication factor for the concatenation.

4.2.1.1 Real to integer conversion

Real numbers are converted to integers by rounding the real number to the nearest integer, rather than by truncating it. Implicit conversion takes 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.

Examples:

The real numbers 35.7 and 35.5 both become 36 when converted to an integer and 35.2 becomes 35.

Converting -1.5 to integer yields -2, converting 1.5 to integer yields 2.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

4.2.1.2 Integer to real conversion

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 an error (see 7.3.2).

4.2.1.3 Arithmetic conversion

For operands, a common data type for each operand is determined before the operator is applied. If either operand is real, the other operand is converted to real. Implicit conversion takes place when a integer number is used with a real number in an operand.

Examples:

```
a = 3 + 5.0;
```

The expression `3 + 5.0` is evaluated by "casting" the integer 3 to the real 3.0, and the result of the expression is 8.0.

```
b = 1 / 2;
```

The above is integer division and the result is 0.

```
c = 8.0 + (1/2);
```

(1/2) is treated as integer division, but the result is cast to a real (0.0) during the addition, and the result of the expression is 8.0.

```
d = 1 / 2.0;
```

Since the denominator is expressed as a real number (2.0) the above is treated as real division and the result is 0.5;

4.2.2 Operator precedence

The precedence order of *operators* is shown in Table 4-3.

Table 4-3—Precedence rules for operators

+ - ! ~ & ~& | ~| ^ ~^ ^~ (unary)	Highest precedence
**	
* / %	
+ - (binary)	
<< >> <<< >>>	
< <= > >=	
== != === !==	
& (bitwise)	
^ ^~ ~^ (bitwise)	
| (bitwise)	
&&	
|| (logical) or (event) , (event)	

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 4-3—Precedence rules for operators *(continued)*

?: (conditional operator)	
{} {{}}	Lowest precedence

Operators shown on the same row in Table 4-3 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 associate left to right with the exception of the conditional operator which associates right to left. Associativity refers to the order in which the operators having the same precedence are evaluated.

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 associate first.

In the following example, `B` is divided by `C` (division has higher precedence than addition) and then the result is added to `A`.

```
A + B / C
```

Parentheses can be used to change the operator precedence.

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

4.2.3 Expression evaluation order

The operators shall follow the associativity rules while evaluating an expression as described in 4.2.2. Some operators (`&&`, `||`, and `?:`) shall use *short-circuit evaluation*; in other words, some of their operand expressions shall not be evaluated as long as the expression contains no analog operators and their value is not required to determine the final value of the operation. All other operators shall not use *short-circuit evaluation* - all of their operand expressions are always evaluated. When short circuiting occurs, any side effects or runtime errors that would have occurred due to evaluation of the short-circuited operand expression shall not occur.

Example 1 - All operand expressions being evaluated:

```
integer varA, varB, varC, result;
analog function integer myFunc;
   ...
endfunction
result = varA & (varB | myFunc(varC));
```

Even if `varA` is known to be zero, the subexpression (`varB | myFunc(varC)`) will be evaluated and any side effects caused by calling `myFunc(varC)` will occur.

Example 2 - Short-circuiting being applied:

```
integer varA, varB, varC, result;
result = varA && (varB || varC);
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

If `varA` is known to be zero (`0`), the result of the expression can be determined as zero (`0`) without evaluating the sub-expression `(varB || varC)`.

Note that implementations are free to optimize by omitting evaluation of subexpressions as long as the simulation behavior (including side effects) is as if the standard rules were followed.

4.2.4 Arithmetic operators

Table 4-4 shows the binary arithmetic operators.

Table 4-4—Arithmetic operators defined

a + b	a plus b
a – b	a minus b
a * b	a multiply by b
a / b	a divide by b
a % b	a modulo b
a ** b	a to power of b

Integer division truncates any fractional part toward zero (`0`).

The unary arithmetic operators take precedence over the binary operators. Table 4-5 shows the unary operators.

Table 4-5—Unary operators defined

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

The *modulus* operator, (for example `a % b`), gives the remainder when the first operand is divided by the second, and thus is zero (0) when `b` divides a exactly. The result of a modulus operation takes the sign of the first operand.

It shall be an error to pass zero (0) as the second argument to the modulus operator.

For the case of the modulus operator where either argument is real, the operation performed is:

```
a % b = ((a/b) < 0) ? (a - ceil(a/b)*b) : (a - floor(a/b)*b);
```

Table 4-6 gives examples of modulus operations.

Table 4-6—Examples of modulus operations

Modulus expression	Result	Comments
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.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 4-6—Examples of modulus operations *(continued)*

Modulus expression	Result	Comments
11 % -3	2	The result takes the sign of the first operand.
10 % 3.75	2.5	[10 - floor(10/3.75)*3.75] yields a remainder of 2.5.

4.2.5 Relational operators

Table 4-7 lists and defines the relational operators.

Table 4-7—The relational operators defined

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

An expression using these *relational operators* yields the value zero (`0`) if the specified relation is *false* or the value one (`1`) if it is *true*.

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

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)` is evaluated, the relational expression is evaluated first and then either zero (0) or one (1) is subtracted from `foo`. When `foo - 1 < a` is evaluated, the value of `foo` operand is reduced by one (1) and then compared with a.

4.2.6 Case equality operators

The *case equality operators* share the same level of precedence as the *logical equality operators*. These operators have limited support in the **`analog`** block (see 7.3.2). Additional information on these operators can also be found in the IEEE Std 1364 Verilog.

4.2.7 Logical equality operators

The *logical equality operators* rank lower in precedence than the relational operators. Table 4-8 lists and defines the equality operators.

Table 4-8—The equality operators defined

a ==b	a equal to b
a !=b	a not equal to b

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Both equality operators have the same precedence. These operators compare the value of the operands. As with the relational operators, the result shall be zero (`0`) if comparison fails, one (`1`) if it succeeds.

4.2.8 Logical operators

The operators *logical and* (`&&`) and *logical or* (`||`) are logical connectives. The result of the evaluation of a logical comparison can be one (1) (defined as *true*) or zero (0) (defined as *false*). 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 zero (`0`) and a zero or false operand into one (`1`).

The following expression performs a *logical and* `(&&)` of three sub-expressions without needing any parentheses:

```
a < param1 && b != c && index != lastone
```

However, parentheses can be used to clearly show the precedence intended, as in the following rewrite of the above example:

```
(a < param1) && (b != c) && (index != lastone)
```

4.2.9 Bitwise operators

The *bitwise operators* perform bitwise manipulations on the operands—that is, the operator combines a bit in one operand with its corresponding bit in the other operand to calculate one bit for the result. The following logic tables (Table 4-9 — Table 4-13) show the results for each possible calculation.

Table 4-9—Bitwise binary and operator

&	**0**	**1**
0	0	0
1	0	1

Table 4-10—Bitwise binary or operator

|	**0**	**1**
0	0	1
1	1	1

Table 4-11—Bitwise binary exclusive or operator

^	**0**	**1**
0	0	1
1	1	0

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 4-12—Bitwise binary exclusive nor operator

^~ ~^	**0**	**1**
0	1	0
1	0	1

Table 4-13—Bitwise unary negation operator

~	
0	1
1	0

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

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

4.2.10 Reduction operators

The reduction operators can not be used inside the **`analog`** block and only have meaning when used in the digital context. Information on these operators can also be found in the IEEE Std 1364 Verilog.

4.2.11 Shift operators

There are two types of *shift operators*: the logical shift operators, **<<** and **>>**, and the arithmetic shift operators, **<<<** and **>>>**. The arithmetic shift operators can not be used in an **`analog`** block. Further information on these operators can be found in IEEE Std 1364 Verilog. The logical shift operators, **<<** and **>>**, perform left and right shifts of their left operand by the number of bit positions given by the right operand. Both the **<<** and **>>** shift operators fill the vacated bit positions with zeroes (0).The right operand is always treated as an unsigned number and has no effect on the signedness of the result.

Examples:

```
integer start, result;
analog begin
   start = 1;
   result = (start << 2);
end
```

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

```
integer start, result;
analog begin
   start = 3;
   result = (start >> 1);
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
end
```

In this example, the integer `result` is assigned the binary value `0001`, which is `0011` shifted to the right one position and zero-filled.

4.2.12 Conditional operator

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

```
conditional_expression ::=                                          // from A.8.3
        expression1 ? { attribute_instance } expression2 : expression3
```

Syntax 4-1—Syntax for conditional operator

The evaluation of a conditional operator begins with the evaluation of *expression1*. If *expression1* evaluates to *false* (`0`), then *expression3* is evaluated and used as the result of the conditional expression. If *expression1* evaluates to *true* (any value other than zero (`0`)), then *expression2* is evaluated and used as the result.

4.2.13 Concatenations

A concatenation is the result of the joining together of bits resulting from one or more expressions into a single value. The concatenation shall be expressed using the brace characters **{** and **}**, with commas separating the expressions within. It should not be confused with the assignment pattern **'{ }** which is used in Verilog-AMS to specify literal lists of constants and expressions for purposes such as the assignment of array initializers and coefficient arguments to the Laplace analog filters. Confusion can arise because **{ }** is used to describe lists of values for array initialization in the C language whereas it means something very different (concatenation) in the Verilog HDL and Verilog-AMS HDL languages.

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

This example concatenates two expressions:

```
{1'b1, 3'b101}
```

It is equivalent to the following example:

```
{1'b1, 1'b1, 1'b0, 1'b1}
```

Its value is `4'b1101`.

The next example concatenates three strings:

```
{ "hello", " ", "world" }
```

Its value is `"hello world"`.

An operator that can be applied only to concatenations is replication, which is expressed by a concatenation preceded by a non-negative, non-x and non-z constant expression, called a replication constant, enclosed together within brace characters, and which indicates a joining together of that many copies of the concatenation. Unlike regular concatenations, expressions containing replications shall not appear on the left-hand side of an assignment and shall not be connected to output or inout ports.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The following example replicates `w` four times:

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

The next example illustrates a replication nested within a concatenation:

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

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

```
parameter P = 32;

// The following is legal for all P from 1 to 32
assign b[31:0] = { {32-P{1'b1}}, a[P-1:0] };

// The following is illegal for P=32 because the zero
// replication appears alone within a concatenation
assign c[31:0] = { {{32-P{1'b1}}}, a[P-1:0] };

// The following is illegal for P=32
initial
   $displayb({32-P{1'b1}}, a[P-1:0]);
```

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

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

would be computed as:

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

4.2.14 Assignment patterns

The assignment pattern **'{ }**, is the way to specify lists of expressions of particular type in Verilog-AMS during assignments, particularly array assignments. It is a feature imported from the IEEE Std 1800 SystemVerilog language.

---

assignment_pattern ::= // from A.8.1
  '{ expression { **,** expression } **}**
| **'{** constant_expression **{** expression { **,** expression } **} }**

constant_assignment_pattern ::=
  '{ constant_expression { **,** constant_expression } **}**
| **'{** constant_expression **{** constant_expression { **,** constant_expression } **} }**

Syntax 4-2—Syntax for assignment pattern

In the example below, a real array is initialized using an assignment pattern

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
parameter real data1[0:4] = '{3.4, 5.6, 2.3, 4.5, 7.1};
```

In the example below, a real array variable is initialized using an assignment pattern. The example also uses a replication operator to repeat 0.0 five times so that every element of `data2` is assigned to 0.0.

```
parameter real data2[0:4] = '{ 5{0.0} };
```

The example below assigns the array `measurements` in the *analog* block using an assignment pattern composed of three variables; `a,b,c`.

```
real measurements[0:2];
real a,b,c;
analog begin
   ...
   measurements = '{a,b,c};
```

Here are the contexts in Verilog-AMS where an array assignment pattern is allowed;

- Analog operator arguments which are expected to be of type array (see 4.5.1)
- The data_source argument of the $table_model system task
- Parameter array assignment in an instantiation
- The RHS of an array variable or array parameter default assignment
- The RHS of an array variable assignment
- Array arguments in calls to user-defined functions

IEEE Std 1800 SystemVerilog has additional uses for the assignment pattern beyond array assignments. IEEE Std 1800 SystemVerilog disallows the usage of the assignment pattern in particular contexts e.g. arguments to system tasks: `$my_system_task('{4.2,5.1,6.3} )`. Verilog-AMS also adopts these restrictions. IEEE Std 1800 SystemVerilog should be consulted for a more detailed understanding of these restrictions.

4.3 Built-in mathematical functions

Verilog-AMS HDL supports both the standard and transcendental mathematical functions. Both the IEEE Std 1364 Verilog system function syntax style and the traditional Verilog-AMS HDL style are supported. Users are encouraged to adopt the IEEE Std 1364 Verilog system function style when using the mathematical functions but the traditional Verilog-AMS HDL style will continue to be supported for backwards compatibility. The following tables Table 4-14 and Table 4-15 show both syntax styles as well as the equivalent C function.

4.3.1 Standard mathematical functions

The standard mathematical functions supported by Verilog-AMS HDL are shown in Table 4-14. The operands shall be numeric (integer or real). For **`min()`**, and **`max()`**, if both operands are integer, then the result

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

is an integer, else both operands are converted to real, as is the result.For **`abs()`**, if the operand is an integer, then the result is an integer, else the result is real. All other arguments are converted to real.

Table 4-14—Standard functions

Verilog function style	Traditional Verilog-AMS function style	Equivalent C function	Description	Domain
`$ln`(*x*)	**`ln`**(*x*)	**`log`**(*x*)	Natural logarithm	*x* > 0
`$ln1p`(*x*)	**`ln1p`**(*x*)	**`log1p`**(*x*)	Natural logarithm of 1 plus x	*x* > -1
`$log10`(*x*)	**`log`**(*x*)	**`log10`**(*x*)	Decimal logarithm	*x* > 0
`$exp`(*x*)	**`exp`**(*x*)	**`exp`**(*x*)	Exponential	All *x*
`$expm1`(*x*)	**`expm1`**(*x*)	**`expm1`**(*x*)	Exponential minus 1	All *x*
`$sqrt`(*x*)	**`sqrt`**(*x*)	**`sqrt`**(*x*)	Square root	*x* >= 0
`$min`(*x,y*)	**`min`**(*x, y*)	**`fmin`**(*x,y*)	Minimum	All *x*, all *y*
`$max`(*x,y*)	**`max`**(*x, y*)	**`fmax`**(*x,y*)	Maximum	All *x*, all *y*
`$abs`(*x*)	**`abs`**(*x*)	**`fabs`**(*x*)	Absolute	All *x*
`$pow`(*x,y*)	**`pow`**(*x, y*)	**`pow`**(*x,y*)	Power (x^y)	if *x* > 0, all *y*; if *x* = 0, *y* > 0; if *x* < 0,all integer y
`$floor`(*x*)	**`floor`**(*x*)	**`floor`**(*x*)	Floor	All *x*
`$ceil`(*x*)	**`ceil`**(*x*)	**`ceil`**(*x*)	Ceiling	All *x*

The **`min()`**, **`max()`**, and **`abs()`** functions have discontinuous derivatives; it is necessary to define the behavior of the derivative of these functions at the point of the discontinuity. In this context, these functions are defined so:

`min(x,y)` is equivalent to `(x < y) ? x : y`

`max(x,y)` is equivalent to `(x > y) ? x : y`

`abs(x)` is equivalent to `(x > 0) x : -x`

The **`ln1p`**(*x*) function returns the natural logarithm of one plus x: **`ln`**(*1+x*). For small magnitude values of `x`, **`ln1p`**(*x*) can be more accurate than **`ln`**(*1+x*).

The **`expm1`**(*x*) function returns the exponential raised to the power x minus one: e^x-1. For small magnitude values of `x`, **`expm1`**(*x*) can be more accurate than **`exp`**(*x*)-*1*.

4.3.2 Transcendental functions

The trigonometric and hyperbolic functions supported by Verilog-AMS HDL are shown in Table 4-15. All operands shall be numeric (integer or real) and are converted to real if necessary. Arguments to the trigono-

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

metric functions (**sin**, **cos**, **tan**) and return values of the inverse trigonometric functions (**asin**, **acos**, **atan**, **atan2**) are in radians. Input values outside of the valid range for the operator shall report an error.

Table 4-15—Trigonometric and hyperbolic functions

Verilog function style	Traditional Verilog-AMS function style	Equivalent C function	Description	Domain
$sin(*x*)	**sin**(*x*)	**sin**(*x*)	Sine	All *x*
$cos(*x*)	**cos**(*x*)	**cos**(*x*)	Cosine	All *x*
$tan(*x*)	**tan**(*x*)	**tan**(*x*)	Tangent	*x* != *n* (π / 2), *n* is odd
$asin(*x*)	**asin**(*x*)	**asin**(*x*)	Arc-sine	-1 <= *x* <= 1
$acos(*x*)	**acos**(*x*)	**acos**(*x*)	Arc-cosine	-1 <= *x* <= 1
$atan(*x*)	**atan**(*x*)	**atan**(*x*)	Arc-tangent	All *x*
$atan2(y,*x*)	**atan2**(*y,x*)	**atan2**(*y,x*)	Arc-tangent of *y*/*x*	All *x,* all *y* ; atan2(0,0) = 0
$hypot(*x,y*)	**hypot**(*x,y*)	**hypot**(*x,y*)	$\sqrt{x^2 + y^2}$	All *x,* all *y*
$sinh(*x*)	**sinh**(*x*)	**sinh**(*x*)	Hyperbolic sine	All *x*
$cosh(*x*)	**cosh**(*x*)	**cosh**(*x*)	Hyperbolic cosine	All *x*
$tanh(*x*)	**tanh**(*x*)	**tanh**(*x*)	Hyperbolic tangent	All *x*
$asinh(*x*)	**asinh**(*x*)	**asinh**(*x*)	Arc-hyperbolic sine	All *x*
$acosh(*x*)	**acosh**(*x*)	**acosh**(*x*)	Arc-hyperbolic cosine	*x* >= 1
$atanh(*x*)	**atanh**(*x*)	**atanh**(*x*)	Arc-hyperbolic tangent	-1 < *x* < 1

4.4 Signal access functions

Access functions are used to access signals on nets, ports, and branches. There are two types of access functions, *branch access functions* and *port access function*s. The name of the access function for a signal is taken from the discipline of the net, port, or branch where the signal or port is associated and utilizes the **()** operator. A port access function also takes its name from the discipline of the port to which it is associated but utilizes the port access (**< >**) operator.

As an alternative to using the access attribute specified in the discipline, the generic **potential** and **flow** access functions are also supported (see 5.5.1).

If the signal or port access function is used in an expression, the access function returns the value of the signal. If the signal access function is being used on the left side of a branch assignment or contribution statement, it assigns a value to the signal. A port access function can not be used on the left side of a branch assignment or contribution statement.

Table 4-16 shows how access functions can be applied to branches, nets, and ports. In this table, *b1* refers to a branch, *n1* and *n2* represent either nets or ports, and *p1* represents a port. These branches, nets, and ports

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

are assumed to belong to the electrical discipline, where *V* is the name of the access function for the voltage (potential) and *I* is the name of the access function for the current (flow).

Table 4-16—Access functions examples

Example	Comments
`V(b1)`	Accesses the voltage across branch *b1*
`potential(b1)`	Alternative access of the voltage across the branch *b1*
`V(n1)`	Accesses the voltage of *n1* (a net or a port) relative to ground
`V(n1,n2)`	Accesses the voltage difference between *n1* and *n2* (nets or ports)
`V(n1,n1)`	Error
`I(b1)`	Accesses the current flowing in branch *b1*
`I(n1)`	Accesses the current flowing in the unnamed branch from *n1* to ground
`flow(n1)`	Alternative access of the current flowing in the unnamed branch from *n1* to ground
`I(n1,n2)`	Accesses the current flowing in the unnamed branch between *n1* and *n2*
`I(n1,n1)`	Error
`I(<p1>)`	Accesses the current flow into the module through port *p1*

The argument expression list for signal access functions shall be a branch identifier, or a list of one or two nets or port expressions. If two net expressions are given as arguments to a flow access function, they shall not evaluate to the same signal. The net identifiers shall be scalar or resolve to a constant net of a composite net type (array or bus) accessed by a genvar expression. If only one net expression is given as the argument to a signal access function, it is implicitly assumed that the second terminal of that unnamed branch is ground.

The operands of an expression shall be unique to define a valid branch. The access function name shall match the discipline declaration for the nets, ports, or branch given in the argument expression list. In this case, `V` and `I` are used as examples of access functions for electrical potential and flow.

For port access functions, the expression list is a single port of the module. The port identifier shall be scalar or resolve to a constant net of a bus port accessed by a genvar expression. The access function name shall match the discipline declaration for the port identifier.

4.5 Analog operators

Analog operators are functions which operate on more than just the current value of their arguments. Instead, they maintain their internal state and their output is a function of both the input and the internal state.

Analog operators are also referred to as analog filter functions. They include the time derivative, time integral, and delay operators from calculus. They also include the transition and slew filters, which are used to remove discontinuity from piecewise constant and piecewise continuous waveforms. Finally, they include more traditional filters, such as those described with Laplace and Z-transform descriptions.

One special analog operator is the **`limexp()`** function, which is a version of the **`exp()`** function with built-in limits to improve convergence.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The syntax for the analog operators is shown in Syntax 4-3.

```
analog_filter_function_call ::=                                    // from A.8.2
        ddt ( analog_expression [ , abstol_expression ] )
      | ddx ( analog_expression , branch_probe_function_call )
      | idt ( analog_expression [ , analog_expression [ , analog_expression [ , abstol_expression ] ] ] )
      | idtmod ( analog_expression [ , analog_expression [ , analog_expression [ , analog_expression
          [ , abstol_expression ] ] ] ] )
      | absdelay ( analog_expression , analog_expression [ , constant_expression ] )
      | transition ( analog_expression [ , analog_expression [ , analog_expression
          [ , analog_expression [ , constant_expression ] ] ] ] )
      | slew ( analog_expression [ , analog_expression [ , analog_expression ] ] )
      | last_crossing ( analog_expression [ , analog_expression ] )
      | limexp ( analog_expression )
      | laplace_filter_name ( analog_expression , [ analog_filter_function_arg ] ,
          [ analog_filter_function_arg ] [ , constant_expression ] )
      | zi_filter_name ( analog_expression , [ analog_filter_function_arg ] ,
          [ analog_filter_function_arg ] , constant_expression
          [ , analog_expression [ , constant_expression ] ] )
analog_filter_function_arg ::=
        parameter_identifier
      | parameter_identifier [ msb_constant_expression : lsb_constant_expression ]
      | constant_assignment_pattern_or_null
```

Syntax 4-3—Syntax for the analog operators

4.5.1 Vector or array arguments to analog operators

Certain analog operators require arrays or vectors to be passed as arguments: Laplace filters, Z-transform filters, **`noise_table()`** and **`noise_table_log()`**. An array can either be passed as an *array_identifier* (e.g. an array parameter or an array variable) or an array assignment pattern (see 4.2.14).

4.5.2 Analog operators and equations

Generally, simulators formulate the mathematical description of the system in terms of first-order differential equations and solve them numerically. There is no direct way to solve a set of nonlinear differential equations so iterative approaches are used. When using iterative approaches, some criteria (*tolerances*) is needed to determine when the algorithm knows when it is close enough to the solution and then stops the iteration. Thus, each equation, at a minimum, shall have a tolerance defined and associated with it.

Occasionally, analog operators require new equations and new unknowns be introduced by the simulator to convert a module description into a set of first-order differential equations. In this case, the simulator attempts to determine from context which tolerance to associate with the new equation and new unknown. Alternatively, these operators can be used to specify tolerances.

Specifying natures also directly enforces reusability and allows other signal attributes to be accessed by the simulator.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

4.5.3 Time derivative operator

The **ddt** operator computes the time derivative of its argument, as shown in Table 4-17.

Table 4-17—Time derivative

Operator	Comments
ddt(*expr*)	Returns $\frac{d}{dt}x(t)$, the time-derivative of *x, where x is expr*.
ddt(*expr, abstol*)	Same as above, except absolute tolerance is specified explicitly.
ddt(*expr, nature*)	Same as above, except nature is specified explicitly.

In DC analysis, **ddt()** returns zero (0). The optional parameter *abstol* is used as an absolute tolerance if needed. Whether an absolute tolerance is needed depends on the context where **ddt()** is used. See 4.5.2 for more information on the application of tolerances to equations. The absolute tolerance, *abstol* or derived from *nature*, applies to the output of the **ddt** operator and is the largest signal level that is considered negligible.

4.5.4 Time integral operator

The **idt** operator computes the time-integral of its argument, as shown in Table 4-18.

Table 4-18—Time integral

Operator	Comments
idt(*expr*)	Returns $\int_{t_0}^{t} x(\tau)d\tau + c$, where $x(\tau)$ is the value of *expr* at time τ, t_0 is the start time of the simulation, t is the current time, and c is the initial starting point as determined by the simulator and is generally the DC value (the value that makes expr equal to zero).
idt(*expr,ic*)	Returns $\int_{t_0}^{t} x(\tau)d\tau + c$, where in this case c is the value of *ic* at t_0.
idt(*expr,ic,assert*)	Returns $\int_{t_a}^{t} x(\tau)d\tau + c$, where c is the value of *ic* at t_a, which is the time when *assert* was last nonzero or t_0 if *assert* was never nonzero.
idt(*expr,ic,assert,abstol*)	Same as above, except the absolute tolerance used to control the error in the numerical integration process is specified explicitly with *abstol*.
idt(*expr,ic,assert,nature*)	Same as above, except the absolute tolerance used to control the error in the numerical integration process is taken from the specified nature.

When used in DC or IC analyses, **idt()** returns the initial condition (*ic*) if specified. If not specified, the idt operator must be contained within a negative feedback loop that forces its argument to zero. Otherwise the output of the idt operator is undefined.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

When specified with initial conditions but without assert, **idt()** returns the value of the initial condition on the initial point of a transient analysis. When specified with both initial conditions and *assert*, **idt()** returns the initial conditions during DC and IC analyses, and whenever *assert* is nonzero. Once assert becomes zero, **idt()** returns the integral of the argument starting from the last instant where assert was nonzero.

The optional parameter *abstol* or *nature* is used to derive an absolute tolerance if needed. Whether an absolute tolerance is needed depends on the context where **idt()** is used. (See 4.5.2 for more information.) The absolute tolerance applies to the input of the **idt** operator and is the largest signal level that is considered negligible.

A simple example that demonstrates the first form is a simple model for an opamp.

```
module opamp(out, pin, nin);
   output out;
   input pin, nin;
   voltage out, pin, nin;
   analog
      V(out) <+ idt(V(pin,nin));
endmodule
```

Here the opamp is simply modeled as an integrator. In this case the initial condition for the integrator is found by the simulator, generally the DC operating point is used. For the DC operating point to exist for an integrator that does not have an initial condition explicitly specified, the integrator must exist within a negative feedback loop that drives its argument to 0. Forcing the output of the integration operator to be a particular value at start of the simulation using something like

```
V(out) <+ idt(V(pin,nin), 0);
```

avoids this issue.

Using the *assert* argument, the output of the integration operator can be reset to a given value at any time. This feature is demonstrated in the following model, which uses the **idt()** operator to generate a periodic ramp waveform:

```
module ramp_generator(out);
   output out;
   voltage out;
   integer reset;
   analog begin
      reset = 0;
      @(timer(1, 1))
         reset = 1;
      V(out) <+ idt(1.0, 0, reset);
   end
endmodule
```

The output of this model is shown in Figure 4-3. Notice that in this model the reset occurs instantaneously.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

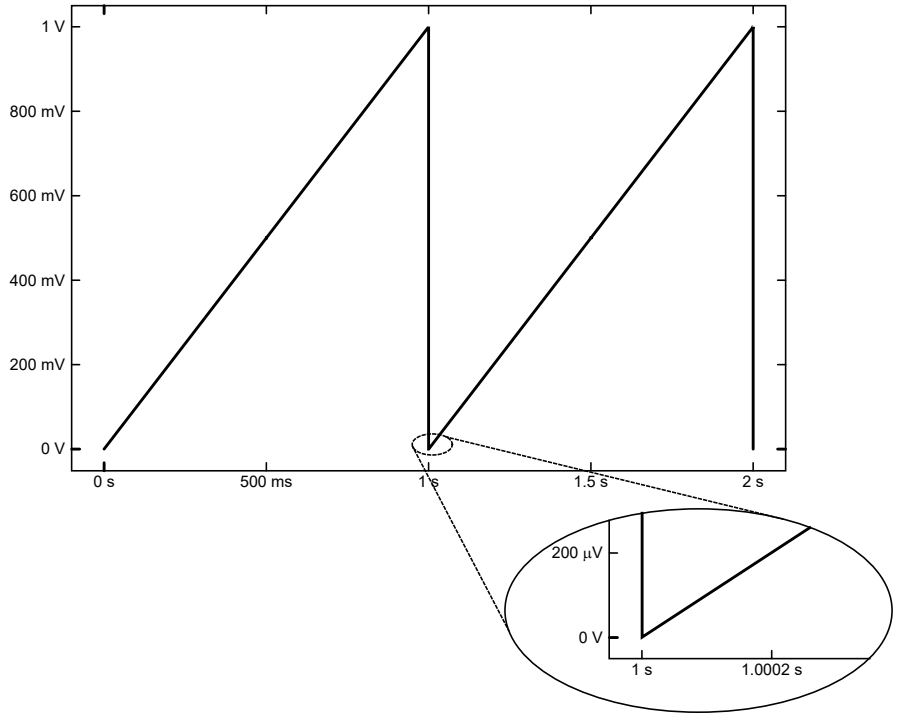


Figure 4-3: The output from the ramp generator

4.5.5 Circular integrator operator

The **`idtmod`** operator, also called the *circular integrator*, converts an expression argument into its indefinitely integrated form similar to the **`idt`** operator, as shown in Table 4-19.

Table 4-19—Circular integrator

Operator	Comments
`idtmod`(*expr*)	Returns $\int_{t_0}^{t} x(\tau)d\tau + c$, where $x(\tau)$ is the value of *expr* at time τ, t_0 is the start time of the simulation, t is the current time, and c is the initial starting point as determined by the simulator and is generally the DC value (the value that makes expr equal to zero).
`idtmod`(*expr*,*ic*)	Returns $\int_{t_0}^{t} x(\tau)d\tau + c$, where in this case c is the value of *ic* at t_0.
`idtmod`(*expr*,*ic*,*modulus*)	Returns k, where $0 \le k < modulus$ and k is $\int_{t_0}^{t} x(\tau)d\tau + c = n \times modulus + k$, n = ... –3, –2, –1, 0, 1, 2, 3 ..., and c is the value of *ic* at t_0.
`idtmod`(*expr*,*ic*,*modulus*,*offset*)	Returns k, where $offset \le k < offset + modulus$, k is $\int_{t_0}^{t} x(\tau)d\tau + c = n \times modulus + k$, and c is the value of *ic* at t_0.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 4-19—Circular integrator *(continued)*

Operator	Comments
idtmod(*expr,ic,modulus,offset,abstol*)	Same as above, except the absolute tolerance used to control the error in the numerical integration process is specified explicitly with *abstol*.
idtmod(*expr,ic,modulus,offset, nature*)	Same as above, except the absolute tolerance used to control the error in the numerical integration process is taken from the specified nature.

The initial condition is optional. If the initial condition is not specified, it defaults to zero (`0`). Regardless, the initial condition shall force the DC solution to the system.

If **`idtmod()`** is used in a system with feedback configuration which forces `expr` to zero (`0`), the initial condition can be omitted without any unexpected behavior during simulation. For example, an operational amplifier alone needs an initial condition, but the same amplifier with the right external feedback circuitry does not need a forced DC solution.

The output of the **`idtmod()`** function shall remain in the range

```
offset <= idtmod < offset+modulus
```

The `modulus` shall be an expression which evaluates to a positive value. If the modulus is not specified, then **`idtmod()`** shall behave like **`idt()`** and not limit the output of the integrator.

The default for `offset` shall be zero (`0`).

The following relationship between **`idt()`** and **`idtmod()`** shall hold at all times.

If

```
y = idt(expr, ic);
z = idtmod(expr, ic, modulus, offset);
```

then

```
y = n * modulus + z;    // n is an integer
```

where

```
offset ≤ z < modulus + offset
```

In this example, the circular integrator is useful in cases where the integral can get very large, such as a VCO. In a VCO, only the output values in the range [0,2π] are of interest, e.g.,

```
phase = idtmod(fc + gain*V(in), 0, 1, 0);
V(OUT) <+ sin(2*`M_PI*phase);
```

Here, the circular integrator returns a value in the range [0,1].

4.5.6 Derivative operator

`ddx()` provides access to symbolically-computed partial derivatives of expressions in the **`analog`** block. The analog simulator computes symbolic derivatives of expressions used in contribution statements in order to use Newton-Raphson iteration to solve the system of equations. In many cases in compact modeling, the

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

values of these derivatives are useful quantities for design, such as the trans conductance of a transistor (gm) or the capacitance of a nonlinear charge-storage element such as a varactor. The syntax for this operator is shown in Syntax 4-3.

The general form for the **ddx()** operator is:

```
ddx ( expr , unknown_quantity )
```

where:

— *expr* is the expression for which the symbolic derivative needs to be calculated.

— *unknown_quantity* is the branch probe (voltage or current probe) with respect to which the derivative of the expression needs to be computed.

The operator returns the partial derivative of its first argument with respect to the unknown indicated by the second argument, holding all other unknowns fixed and evaluated at the current operating point. The second argument shall be the potential of a scalar net or port or the flow through a branch, because these are the unknown variables in the system of equations for the analog solver. For the modified nodal analysis used in most SPICE-like simulators, these unknowns are the node voltages and certain branch currents.

If the expression does not depend explicitly on the unknown, then **ddx()** returns zero (0). Care must be taken when using implicit equations or indirect assignments, for which the simulator may create internal unknowns; derivatives with respect to these internal unknowns cannot be accessed with **ddx()**.

Unlike the **ddt()** operator, no tolerance is required because the partial derivative is computed symbolically and evaluated at the current operating point.

This first example uses **ddx()** to obtain the conductance of the diode. The variable gdio is declared as an output variable (see 3.2.1) so that its value is available for inspection by the designer.

```
module diode(a,c);
   inout a, c;
   electrical a, c;
   parameter real IS = 1.0e-14;
   real idio;
   (*desc="small-signal conductance"*)
   real gdio;
   analog begin
      idio = IS * (limexp(V(a,c)/$vt) - 1);
      gdio = ddx(idio, V(a));
      I(a,c) <+ idio;
   end
endmodule
```

The next example adds a series resistance to the diode using an implicit equation. Note that gdio does not represent the total conductance because the flow access I(a,c) requires introduction of another unknown in the system of equations. The conductance of the diode is properly reported as geff, which includes the effects of RS and the nonlinear equation.

```
module diode(a,c);
   inout a, c;
   electrical a, c;
   parameter real IS = 1.0e-14;
   parameter real RS = 0.0;
   real idio, gdio;
   (*desc="effective conductance"*)
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   real geff;
   analog begin
      idio = IS * (limexp((V(a,c)-RS*I(a,c))/$vt) - 1);
      gdio = ddx(idio, V(a));
      geff = gdio / (RS * gdio + 1.0);
      I(a,c) <+ idio;
   end
endmodule
```

The final example implements a voltage-controlled dependent current source and is used to illustrate the computations of partial derivatives.

```
module vccs(pout,nout,pin,nin);
   inout pout, nout, pin, nin;
   electrical pout, nout, pin, nin;
   parameter real k = 1.0;
   real vin, one, minusone, zero;
   analog begin
      vin = V(pin,nin);
      one = ddx(vin, V(pin));
      minusone = ddx(vin, V(nin));
      zero = ddx(vin, V(pout));
      I(pout,nout) <+ k * vin;
   end
endmodule
```

The names of the variables indicate the values of the partial derivatives: +1, -1, or 0. A SPICE-like simulator would use these values (scaled by the parameter `k`) in the Newton-Raphson solution method.

4.5.7 Absolute delay operator

`absdelay()` implements the absolute transport delay for continuous waveforms (use the **`transition()`** operator to delay discrete-valued waveforms). The general form is

```
absdelay ( input , td [ , maxdelay ] )
```

input is delayed by the amount *td*. In all cases *td* shall be a positive number. If the optional *maxdelay* is specified, then *td* can vary. If *td* becomes greater than *maxdelay*, *maxdelay* will be used as a substitute for *td*. If *maxdelay* is not specified, the value of *td* when the **`absdelay()`** is first evaluated shall be used and any future changes to *td* shall be ignored.

In DC and operating point analyses, **`absdelay()`** returns the value of its *input*. In AC and other small-signal analyses, the **`absdelay()`** operator phase-shifts the input expression to the output of the delay operator based on the following formula.

$$Output(\omega) = Input(\omega) \cdot e^{-j\omega \mathrm{td}}$$

`td` is evaluated as a constant at a particular time for any small signal analysis. In time-domain analyses, **`absdelay()`** introduces a transport delay equal to the instantaneous value of *td* based on the following formula.

$$Output(t) = Input(max(t - \mathrm{td}, 0))$$

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The transport delay *td* can be either constant (typical case) or vary as a function of time (when *maxdelay* is defined). When calculating the output at time *t*, the **`absdelay()`** operator will use linear interpolation as needed to determine the input around time:

$$max(t - \text{td}, 0)$$

A time-dependent transport delay is shown in Figure 4-4, with a ramp input to the **`absdelay`** operator for both positive and negative changes in the transport delay *td* and a maxdelay of 5.

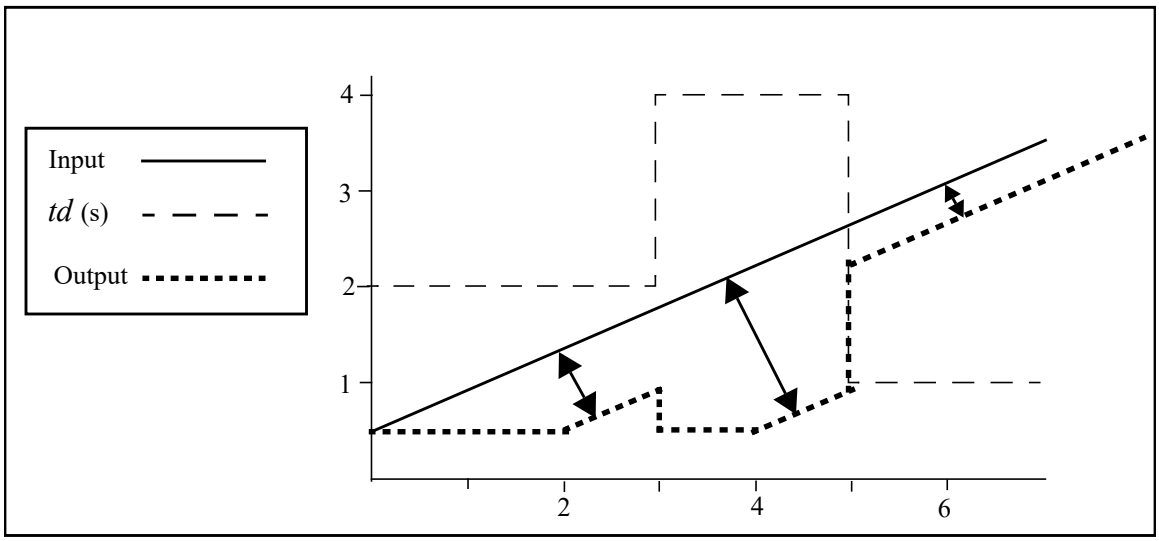


Figure 4-4: Transport delay example

From time 0 until `2s`, the output remains at `input(0)`. With a delay of `2s`, the output then starts tracking `input(t - 2)`. At `3s`, the transport delay changes from `2s` to `4s`, switching the output back to `input(0)`, since `input(max(t-td,0))` returns 0. The output remains at this level until `4s` when it once again starts tracking the input from `t = 0`. At `5s` the transport delay goes to `1s` and the output correspondingly jumps from its current value to the value defined by `input(t - 1)`.

4.5.8 Transition filter

`transition()` smooths out piecewise constant waveforms. The transition filter is used to imitate transitions and delays on digital signals (for non-piecewise constant signals, see 4.5.9). This function provides controlled transitions between discrete signal levels by setting the rise time and fall time of signal transitions, as shown in Figure 4-5.

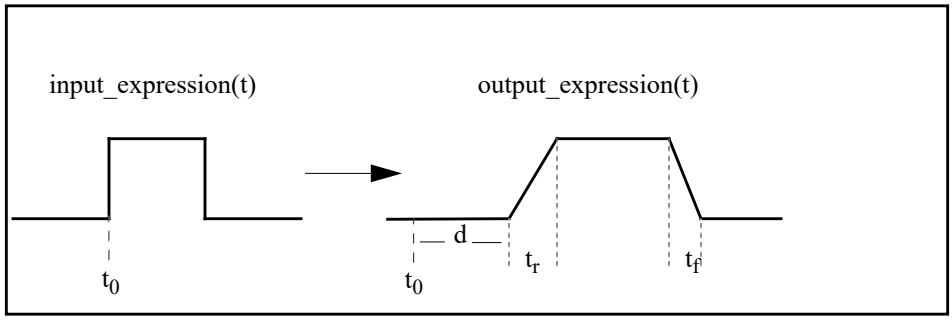


Figure 4-5: Transition filter example

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

`transition()` stretches instantaneous changes in signals over a finite amount of time and can delay the transitions, as shown in Figure 4-6.

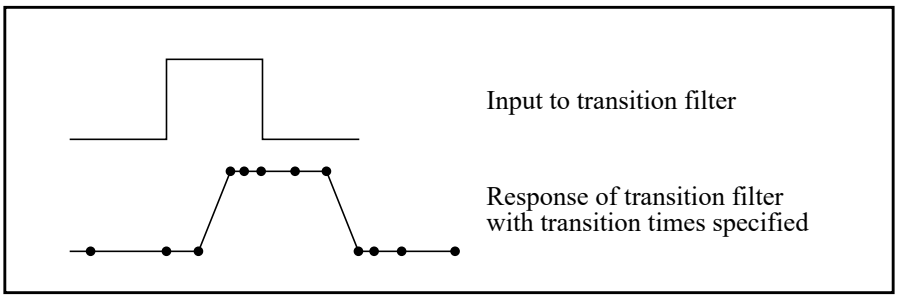


Figure 4-6: Shifting the transition filter

The general form of the **`transition()`** filter is

```
transition ( expr [ , td [ , rise_time [ , fall_time [ , time_tol ] ] ] ] )
```

The input expression is expected to evaluate over time to a piecewise constant waveform. When applied, **`transition()`** forces all positive transitions of *expr* to occur over *rise_time* and all negative transitions to occur in *fall_time* (after an initial delay of *td*). Thus, *td* models transport delay and *rise_time* and *fall_time* model inertial delay.

`transition()` returns a real number which describes a piecewise linear function over time. The transition function causes the simulator to place time points at both corners of a transition. If *time_tol* is not specified, the transition function causes the simulator to assure each transition is adequately resolved.

td, rise_time, fall_time, and *time_tol* are optional, but if specified shall be non-negative. If *td* is not specified, it is taken to be zero (`0.0`). If only a positive *rise_time* value is specified, the simulator uses it for both rise and fall times. If neither *rise_time* nor *fall_time* are specified or are equal to zero (`0.0`), the rise and fall time default to the value defined by **`` `default_transition ``**.If a time_tol value of zero (0.0) is specified, the simulator shall apply a suitable value.

If **`` `default_transition ``** is not specified the default behavior approximates the ideal behavior of a zero-duration transition. Forcing a zero-duration transition is undesirable because it could cause convergence problems. Instead, a negligible, but non-zero, transition time is used. The small non-zero transition time allows the simulator to shrink the timestep small enough so a smooth transition occurs and any convergence problems are avoided. The simulator does not force a time point at the trailing corner of a transition to avoid causing the simulator to take very small time steps, which would result in poor performance.

In DC analysis, **`transition()`** passes the value of the *expr* directly to its output. The **`transition`** filter is designed to smooth out piecewise constant waveforms. When applied to waveforms which vary smoothly, the simulation results are generally unsatisfactory. In addition, applying the transition function to a continuously varying waveform can cause the simulator to run slowly. Use **`transition()`** for discrete signals and **`slew()`** (see 4.5.9) for continuous signals.

A transition is created when the input expression changes, and at this point it uses the value of *td*, *rise_time*, *fall_time* and *time_tol* to determine the new pending transition operator. If the effects are immediate, *td=0*, then current transitions and scheduled ones are canceled, and the new one is created. If *td* is before a previously scheduled transition, then the previously scheduled transition(s) are canceled and a new one is created. If *td* is after previously scheduled transition then it adds the new transition to the pending transition(s) allowing an arbitrary number of pending transitions. Consider a digital clock that is required to be driven out onto an analog port.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
always #5 clk = ~clk;
analog V(aclk) <+ transition(clk,0,1p);
```

If the delay on the transition is greater than ½ period, then multiple pending transitions are stored on the transition operator.

```
always #5 clk = ~clk;
analog V(aclk) <+ transition(clk,5.1n,1p);
```

A transition is considered active during the period of time (rise or fall) that we are transitioning the output from one value to another.

An active transition shall be interrupted if the input to the **`transition()`** changes value while in this active transitioning region. An interrupted transition is not considered a new transition, but rather a readjustment of the original transition. To determine the time that the readjusted transition will reach the new destination, the slope shall be calculated using either the original transition's origin or destination as the new origin based on the following criteria:

— If the original transition was rising and the new destination value is below the value at the interruption, then the original transition's destination shall be used to compute the new origin. Referring to Figure 4-7, consider an original transition that rises from (t1,v1) to (t2,v2) with a rise time of tr1=t2-t1, which is interrupted at the point (ti,vi) with a new destination value (v3), where v3 < vi. Then the original transition's destination (v2) shall be used along with the rescheduled transition's actual fall time (tf3), when calculating the slope: (v3-v2)/tf3. This slope will be applied from the point of interruption (ti,vi), and the readjusted transition's expected end time (t3) is then calculated using this slope, now shifted left, along with the time and value level at the point of the interruption: t3 = ti + (v3-vi)/slope. The new origin for the transition is now (t4,v4), which will be used if the transition is interrupted again.

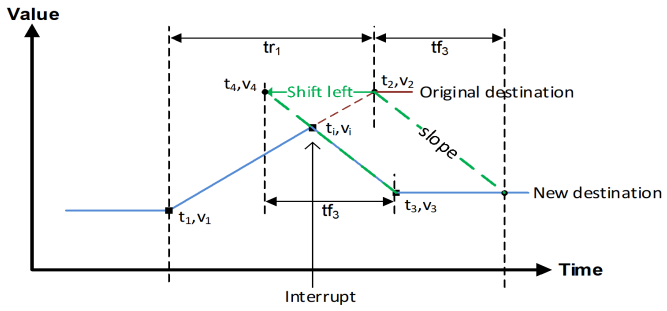


Figure 4-7: Interrupted rising transition (falling)

— If the original transition was rising and the new destination value is above the value at the interruption, then the original transition's origin shall be used to compute the new origin. Referring to Figure 4-8, consider an original transition that rises from (t1,v1) to (t2,v2) with a rise time of tr1=t2-t1, which is interrupted at the point (ti,vi) with a new destination value (v3), where v3 > vi, and with a new transition time tr3. Then the original transition's origin (v1) shall be used along with the new rise time (tr3), to calculate the slope: (v3-v1)/tr3.
This slope will be applied from the point of interruption (ti,vi), and the readjusted transition's expected end time (t3) is then calculated using this slope, along with the time and value level at the point of the interruption: t3 = ti + (v3-vi)/slope. The new origin for the transition is now (t4,v4),

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

which will be used if the transition is interrupted again. In Figure 4-8, the transition is shifted left and t4 < t1 because the new rise time is longer (tr3 > tr1). In Figure 4-9 the new rise time is shorter (tr3 < tr1), so the transition is shifted right.

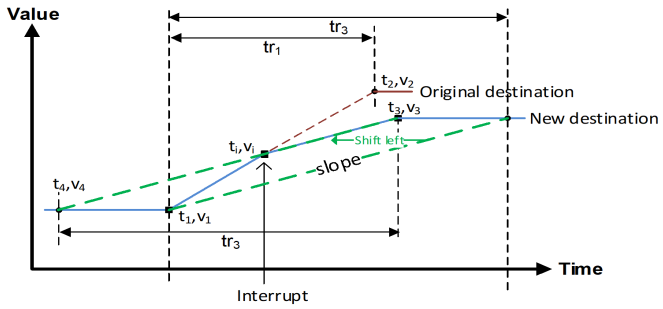


Figure 4-8: Interrupted rising transition (rising tr3>tr1)

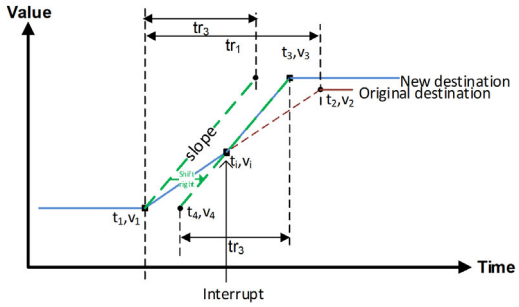


Figure 4-9: Interrupted rising transition (rising tr3<tr1)

— If the original transition was falling and the new destination value is below the value at the interruption, then the original transition's origin shall be used to compute the new origin. Referring to Figure 4, consider an original transition that falls from (t1,v1) to (t2,v2) with a fall time of tf1=t2-t1, which is interrupted at the point (ti,vi) with a new destination value (v3), where v3 < vi, and with a new transition time tf3. Then the original transition's origin (v1) shall be used along with the new fall time (tf3), to calculate the slope: (v3-v1)/tf3.
This slope will be applied from the point of interruption (ti,vi), and the readjusted transition's expected end time (t3) is then calculated using this slope, along with the time and value level at the point of the interruption: t3 = ti + (v3-vi)/slope. The new origin for the transition is now (t4,v4), which will be used if the transition is interrupted again. In Figure 4-10, the transition is shifted left and t4 < t1 because the new fall time is longer (tf3 > tf1). In Figure 4-11, the new fall time is shorter (tf3 < tf1), so the transition is shifted right.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

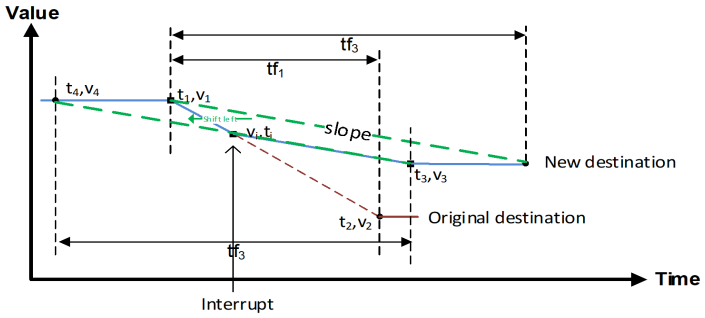


Figure 4-10: Interrupted falling transition (falling tf3>tf1)

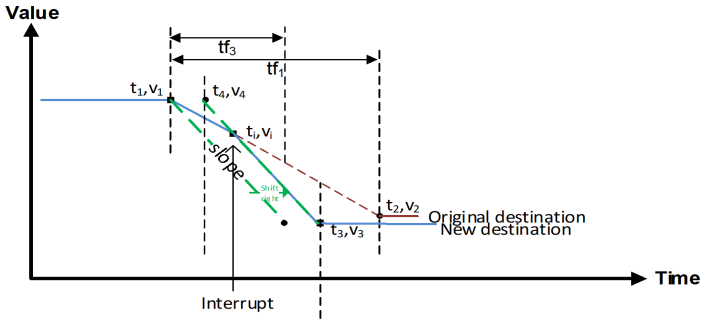


Figure 4-11: Interrupted falling transition (falling tf3<tf1)

— If the original transition was falling and the new destination value is above the value at the interruption, then the original transition's destination shall be used to compute the new origin. Referring to Figure 4-12, consider an original transition that falls from (t1,v1) to (t2,v2) with a fall time of tf1=t2-t1, which is interrupted at the point (ti,vi) with a new destination value (v3), where v3 > vi. Then the original transition's destination (v2) shall be used along with the rescheduled transition's actual rise time (tr3), when calculating the slope: (v3-v2)/tr3. This slope will be applied from the point of interruption (ti,vi), and the readjusted transition's expected end time (t3) is then calculated using this slope, now shifted left, along with the time and value level at the point of the interruption: t3 = ti + (v3-vi)/slope. The new origin for the transition is now (t4,v4), which will be used if the transition is interrupted again.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

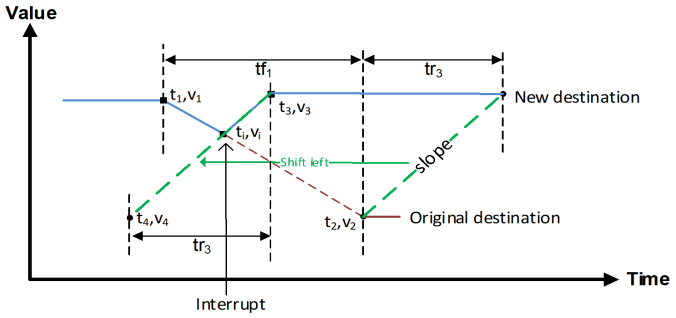


Figure 4-12: Interrupted falling transition (rising)

Because the transition function can not be linearized in general, it is not possible to accurately represent a transition function in AC analysis. The AC transfer function is approximately modeled as having unity transmission for all frequencies in all situations. Because the transition function is intended to handle discrete-valued signals, the small signals present in AC analysis rarely reach transition functions. As a result, the approximation used is generally sufficient.

Example 1 — QAM modulator

In this example, the transition function is used to control the rate of change of the modulation signal in a QAM modulator.

```
module qam16(in, out);
    input [0:3] in;
    output out;
    voltage [0:3] in;
    voltage out;

    parameter real freq = 1.0 from (0:inf);
    parameter real ampl = 1.0;
    parameter real thresh = 2.5;
    parameter real tdelay = 0 from [0:inf);
    localparam real ttransit = 1/freq;

    real x, y, phi;
    integer row, col;

    analog begin
        row = 2 * (V(in[3]) > thresh) + (V(in[2]) > thresh);
        col = 2 * (V(in[1]) > thresh) + (V(in[0]) > thresh);

        x = transition(row - 1.5, tdelay, ttransit);
        y = transition(col - 1.5, tdelay, ttransit);

        phi = `M_TWO_PI * freq * $abstime;
        V(out) <+ ampl * (x * cos(phi) + y * sin(phi));
    end
endmodule
```

Example 2 — A/D converter

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

In this example, an analog behavioral N-bit analog to digital converter, demonstrates the ability of the transition function to handle vectors.

```
module adc(in, clk, out);
   parameter bits = 8, fullscale = 1.0, dly = 0, ttime = 10n;
   input in, clk;
   output [0:bits-1] out;
   electrical in, clk;
   electrical [0:bits-1] out;
   real sample, thresh;
   integer result[0:bits-1];
   genvar i;

   analog begin
      @(cross(V(clk)-2.5, +1)) begin
         sample = V(in);
         thresh = fullscale/2.0;
         for (i = bits - 1; i >= 0; i = i - 1) begin
            if (sample > thresh) begin
               result[i] = 1.0;
               sample = sample - thresh;
            end
            else begin
               result[i] = 0.0;
            end
            sample = 2.0*sample;
         end
      end
      for (i = 0; i < bits; i = i + 1) begin
         V(out[i]) <+ transition(result[i], dly, ttime);
      end
   end
endmodule
```

4.5.9 Slew filter

The **slew** analog operator bounds the rate of change (slope) of the waveform. A typical use for **slew()** is generating continuous signals from piecewise continuous signals. (For discrete-valued signals, see 4.5.8.) The general form is

```
slew ( expr [ , max_pos_slew_rate [ , max_neg_slew_rate ] ] )
```

When applied, **slew()** forces all transitions of *expr* faster than *max_pos_slew_rate* to change at *max_pos_slew_rate* rate for positive transitions and limits negative transitions to *max_neg_slew_rate* rate as shown in Figure 4-13.

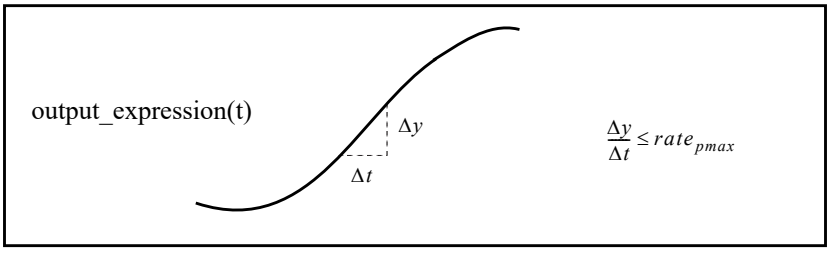


Figure 4-13: Slew filter transition

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The two rate values are optional. *max_pos_slew_rate* shall be greater than zero (0) and *max_neg_slew_rate* shall be less than zero (0). If the *max_neg_slew_rate* is not specified, it defaults to the opposite of the *max_pos_slew_rate*. If no rates are specified, **slew()** passes the signal through unchanged. If the rate of change of *expr* is less than the specified maximum slew rates, **slew()** returns the value of *expr*.

In DC analysis, **slew()** simply passes the value of the destination to its output. In small-signal analyses, the **slew()** function has a transfer function from the first argument to the output of 1.0 when not slewing (e.g. for a small-signal analysis following a dc operating point) and 0.0 when slewing.

4.5.10 last_crossing function

The **last_crossing()** function returns a real value representing the simulation time when a signal expression last crossed zero (0). The general form is

```
last_crossing ( expr [ , direction ] )
```

The optional *direction* indicator shall evaluate to an integer expression +1, -1, or 0. If it is set to 0, the **last_crossing()** will return the most recent time the input expression had either a rise or falling edge transition. If direction is +1 (-1), the **last_crossing()** will return the last time the input expression had a rising (falling) edge transition.

The **last_crossing()** function does not control the timestep to get accurate results; it uses linear interpolation to estimate the time of the last crossing. However, it can be used with the **cross()** or **above()** function for improved accuracy.

The following example measures the period of its input signal using the **cross()** and **last_crossing()** functions.

```
module period(in);
   input in;
   voltage in;
   integer crossings;
   real latest, previous;

   analog begin
      @(initial_step) begin
         crossings = 0;
         previous = 0;
      end

      @(cross(V(in), +1)) begin
         crossings = crossings + 1;
         previous = latest;
      end
      latest = last_crossing(V(in), +1);

      @(final_step) begin
         if (crossings < 2)
            $strobe("Could not measure period.");
         else
            $strobe("period = %g, crossings = %d",
               latest-previous, crossings);
      end
   end
endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Before the expression crosses zero (0) for the first time, the **last_crossing()** function returns a negative value.

4.5.11 Laplace transform filters

The Laplace transform filters implement lumped linear continuous-time filters. Each filter takes an optional parameter ε, which is a real number or a nature used for deriving an absolute tolerance (if needed). Whether an absolute tolerance is needed depends on the context where the filter is used. The zeros argument may be represented as a null argument. The null argument is characterized by two adjacent commas (,,) in the argument list.

For arguments that require a vector, the vector may be represented as either a literal vector or a reference to a vector parameter.

4.5.11.1 laplace_zp

laplace_zp() implements the zero-pole form of the Laplace transform filter. The general form is:

```
laplace_zp ( expr , ζ , ρ [ , ε ] )
```

where ζ (zeta) is a vector of M pairs of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero and the second is the imaginary part. Similarly, ρ (rho) is the vector of N real pairs, one for each pole. The poles are given in the same manner as the zeros. The transfer function is

$$H(s) = \frac{\prod_{k=0}^{M-1}\left(1 - \frac{s}{\zeta_k^r + j\zeta_k^i}\right)}{\prod_{k=0}^{N-1}\left(1 - \frac{s}{\rho_k^r + j\rho_k^i}\right)}$$

where ζ_k^r and ζ_k^i are the real and imaginary parts of the k^{th} zero (0), while ρ_k^r and ρ_k^i are the real and imaginary parts of the k^{th} pole. If a root (a pole or zero) is real, the imaginary part shall be specified as zero (0). If a root is complex, its conjugate shall also be present. If a root is zero, then the term associated with it is implemented as s, rather than $(1 - s/r)$ (where r is the root).

4.5.11.2 laplace_zd

laplace_zd() implements the zero-denominator form of the Laplace transform filter. The general form is:

```
laplace_zd ( expr , ζ , d [ , ε ] )
```

where ζ (zeta) is a vector of M pairs of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero and the second is the imaginary part. Similarly, d is the vector of N real numbers containing the coefficients of the denominator. The transfer function is

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

$$H(s) = \frac{\prod_{k=0}^{M-1}\left(1 - \frac{s}{\zeta_k^r + j\zeta_k^i}\right)}{\sum_{k=0}^{N-1} d_k s^k}$$

where ζ_k^r and ζ_k^i are the real and imaginary parts of the k^{th} zero, while d_k is the coefficient of the k^{th} power of s in the denominator. If a zero is real, the imaginary part shall be specified as zero (0). If a zero is complex, its conjugate shall also be present. If a zero is zero (0), then the term associated with it is implemented as s, rather than $(1 - s/\zeta)$.

4.5.11.3 laplace_np

laplace_np() implements the numerator-pole form of the Laplace transform filter. The general form is

```
laplace_np ( expr , n , ρ [ , ε ] )
```

where n is a vector of M real numbers containing the coefficients of the numerator. Similarly, ρ (rho) is a vector of N pairs of real numbers. Each pair represents a pole, the first number in the pair is the real part of the pole and the second is the imaginary part. The transfer function is

$$H(s) = \frac{\sum_{k=0}^{M-1} n_k s^k}{\prod_{k=0}^{N-1}\left(1 - \frac{s}{\rho_k^r + j\rho_k^i}\right)}$$

where n_k is the coefficient of the k^{th} power of s in the numerator, while ρ_k^r and ρ_k^i are the real and imaginary parts of the k^{th} pole. If a pole is real, the imaginary part shall be specified as zero (0). If a pole is complex, its conjugate shall also be present. If a pole is zero (0), then the term associated with it is implemented as s, rather than $(1 - s/\rho)$.

4.5.11.4 laplace_nd

laplace_nd() implements the numerator-denominator form of the Laplace transform filter.

The general form is:

```
laplace_nd ( expr , n , d [ , ε ] )
```

where n is an vector of M real numbers containing the coefficients of the numerator and d is a vector of N real numbers containing the coefficients of the denominator. The transfer function is:

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

$$H(s) = \frac{\sum_{k=0}^{M-1} n_k s^k}{\sum_{k=0}^{N-1} d_k s^k}$$

where n_k is the coefficient of the k^{th} power of s in the numerator and d_k is the coefficient of the k^{th} power of s in the denominator.

4.5.11.5 Examples

```
V(out) <+ laplace_zp(V(in), '{-1,0}, '{-1,-1,-1,1});
```

implements

$$H(s) = \frac{1+s}{\left(1+\frac{s}{1+j}\right)\left(1+\frac{s}{1-j}\right)}$$

and

```
V(out) <+ laplace_nd(V(in), '{0,1}, '{-1,0,1});
```

implements .

$$H(s) = \frac{s}{s^2-1}$$

This example

```
V(out) <+ laplace_zp(white_noise(k), , '{1,0,1,0,-1,0,-1,0});
```

implements a band-limited white noise source as .

$$\overline{v_{out}^2} = \frac{k}{|s^2-1|^2}$$

4.5.12 Z-transform filters

The *Z-transform* filters implement linear discrete-time filters. Each filter supports a parameter *T* which specifies the sampling period of the filter. A filter with unity transfer function acts like a simple sample-and-hold which samples every *T* seconds and exhibits no delay. The zeros argument may be represented as a null argument. The null argument is characterized by two adjacent commas (`,,`) in the argument list.

All Z-transform filters share three common arguments: *T*, τ, and t_0. *T* specifies the period of the filter, is mandatory, and shall be positive. τ specifies the transition time, is optional, and shall be nonnegative.

If the transition time is specified and is non-zero, the timestep is controlled to accurately resolve both the leading and trailing corner of the transition. If it is not specified, the transition time is taken to be one (`1`) unit

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

of time (as defined by the **`default_transition** compiler directive) and the timestep is not controlled to resolve the trailing corner of the transition. If the transition time is specified as zero (0), then the output is abruptly discontinuous. A *Z*-filter with zero (0) transition time shall not be directly assigned to a branch.

Finally t_0 specifies the time of the first transition, and is also optional. If not given, the first transition occurs at t=0.

For arguments that require a vector, the vector may be represented as either a literal vector or a reference to a vector parameter.

4.5.12.1 zi_zp

zi_zp() implements the zero-pole form of the *Z*-transform filter. The general form is:

zi_zp (expr **,** ζ **,** ρ **,** *T* [**,** τ [**,** t_0]] **)**

where ζ (zeta) is a vector of *M* pairs of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero (0) and the second is the imaginary part. Similarly, ρ (rho) is the vector of *N* real pairs, one for each pole. The poles are given in the same manner as the zeros. The transfer function is

$$H(z) = \frac{\prod_{k=0}^{M-1} 1 - z^{-1}(\zeta_k^r + j\zeta_k^i)}{\prod_{k=0}^{N-1} 1 - z^{-1}(\rho_k^r + j\rho_k^i)}$$

where ζ_k^r and ζ_k^i are the real and imaginary parts of the k^{th} zero, while ρ_k^r and ρ_k^i are the real and imaginary parts of the k^{th} pole. If a root (a pole or zero) is real, the imaginary part shall be specified as zero. If a root is complex, its conjugate shall also be present. If a root is zero (0), then the term associated with it is implemented as *z,* rather than $(1 - z/r)$ (where *r* is the root).

4.5.12.2 zi_zd

zi_zd() implements the zero-denominator form of the *Z*-transform filter. The form is:

zi_zd (expr **,** ζ **,** *d* **,** *T* [**,** τ [**,** t_0]] **)**

where ζ (zeta) is a vector of *M* pairs of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero and the second is the imaginary part. Similarly, *d* is the vector of *N* real numbers containing the coefficients of the denominator. The transfer function is

$$H(z) = \frac{\prod_{k=0}^{M-1} 1 - z^{-1}(\zeta_k^r + j\zeta_k^i)}{\sum_{k=0}^{N-1} d_k z^{-k}}$$

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

where ζ_k^r and ζ_k^i are the real and imaginary parts of the k^{th} zero, while d_k is the coefficient of the k^{th} power of s in the denominator. If a zero is real, the imaginary part shall be specified as zero (0). If a zero is complex, its conjugate shall also be present. If a zero is zero (0), then the term associated with it is implemented as z, rather than $(1 - z/\zeta)$.

4.5.12.3 zi_np

zi_np() implements the numerator-pole form of the Z-transform filter. The general form is:

zi_np (expr **,** n **,** ρ **,** T [**,** τ [**,** t_0]] **)**

where n is a vector of M real numbers containing the coefficients of the numerator. Similarly, ρ (rho) is a vector of N pairs of real numbers. Each pair represents a pole, the first number in the pair is the real part of the pole and the second is the imaginary part. The transfer function is

$$H(z) = \frac{\sum_{k=0}^{M-1} n_k z^{-k}}{\prod_{k=0}^{N-1} 1 - z^{-1}(\rho_k^r + j\rho_k^i)}$$

where n_k is the coefficient of the k^{th} power of s in the numerator, while ρ_k^r and ρ_k^i are the real and imaginary parts of the k^{th} pole. If a pole is real, the imaginary part shall be specified as zero (0). If a pole is complex, its conjugate shall also be present. If a pole is zero (0), then the term associated with it is implemented as z, rather than $(1 - z/\rho)$.

4.5.12.4 zi_nd

zi_nd() implements the numerator-denominator form of the Z-transform filter. The general form is:

zi_nd (expr **,** n **,** d **,** T [**,** τ [**,** t_0]] **)**

where n is an vector of M real numbers containing the coefficients of the numerator and d is a vector of N real numbers containing the coefficients of the denominator. The transfer function is

$$H(z) = \frac{\sum_{k=0}^{M-1} n_k z^{-k}}{\sum_{k=0}^{N-1} d_k z^{-k}}$$

where n_k is the coefficient of the k^{th} power of s in the numerator and d_k is the coefficient of the k^{th} power of s in the denominator.

4.5.13 Limited exponential

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The **limexp()** function is a special-purpose operator whose purpose is to improve convergence of the analog solver (generally, a Newton-Raphson linear solver) when faced with the strongly nonlinear behavior of the exponential function. The operator has internal state containing information about the argument on previous iterations. It returns a real value which is the exponential of its single real argument; however, it internally limits the change of its output from iteration to iteration in order to improve convergence. It is therefore not useful for any exponential whose argument does not change from iteration to iteration. On any iteration where the change in the output of the **limexp()** function is bounded, the simulator is prevented from terminating the iteration. Thus, the simulator can only converge when the output of **limexp()** equals the exponential of the input.

The general form is:

```
limexp ( expr )
```

The apparent behavior of **limexp()** is not distinguishable from **exp()**, except using **limexp()** to model semiconductor junctions generally results in dramatically improved convergence. There are different ways of implementing limiting algorithms for the exponential[1] [2].

Other nonlinearities besides the exponential may be in behavioral models. The **$limit()** system function described in 9.17.3 provides a method to indicate these nonlinearities to the simulator to improve convergence.

4.5.14 Constant versus dynamic arguments

Some of the arguments to the analog operators described in this section, the events described in Clause 5, and the **$limit()** function in 9.17.3 expect dynamic expressions and others expect constant expressions. The dynamic expressions can be functions of circuit quantities and can change during an analysis. The constant expressions remain static throughout an analysis.

Table 4-20 summarizes the arguments of the analog operators defined in this section.

Table 4-20—Analog operator arguments

Operator	Constant expression arguments	Dynamic expression arguments
absdelay	maxdelay	expr, td
ddt	abstol	expr
ddx	wrt_what	expr
idt	abstol	expr, ic, assert
idtmod	abstol	expr, ic,modulus, offset
laplace_zp	zeros, poles, abstol	expr
laplace_zd	zeros, denominator, abstol	expr
laplace_np	numerator, poles, abstol	expr
laplace_nd	numerator, denominator, abstol	expr
last_crossing		expr, dir

[1]Laurence W. Nagel, "SPICE2: A computer program to simulate semiconductor circuits," Memorandum No. ERL-M520, University of California, Berkeley, California, May 1975.

[2]W. J. McCalla, *Fundamentals of Computer-Aided Circuit Simulation*. Kluwer Academic Publishers, 1988.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 4-20—Analog operator arguments *(continued)*

Operator	Constant expression arguments	Dynamic expression arguments
limexp		expr
slew		expr, max_pos_slew_rate, max_neg_slew_rate
transition		expr, td, rise_time, fall_time, time_tol
zi_zp	zeros, poles, T, t0	expr, t
zi_zd	zeros, denominator, T, t0	expr, t
zi_np	numerator, poles, T, t0	expr, t
zi_nd	numerator, denominator, T, t0	expr, t

If a dynamic expression is passed as an argument which expects a constant expression, the value of the dynamic expression at the start of the analysis defaults to the constant value of the argument. Any further change in value of that expression is ignored during the iterative analysis.

4.5.15 Restrictions on analog operators

Analog operators are subject to several important restrictions because they maintain their internal state. It is important to ensure that all analog operators are evaluated every iteration of a simulation to ensure that the internal state is maintained. The analog operator **ddx()** is the only exception to this rule as it does not require an internal state to be maintained. All analog operators are considered to have no state history prior to time t == 0.

— Analog operators shall not be used inside conditional (**if**, **case**, or **?:**) statements unless the conditional expression controlling the statement consists of terms which can not change their value during the course of a simulation.

— Analog operators shall not be used inside event triggered statements.

— Analog operators are not allowed in the **repeat**, **while** and non-genvar **for** looping statements.

— Analog operators can only be used inside an **analog** block; they can not be used inside an **initial** or **always** block, or inside a user-defined function.

— It is illegal to specify a null argument in the argument list of an analog operator, except as specified elsewhere in this document.

These restrictions help prevent usage which could cause the internal state to be corrupted or become out-of-date, which results in anomalous behavior.

4.6 Analysis dependent functions

This section describes the **analysis()** function, which is used to determine what type of analysis is being performed, and the small-signal source functions. The small-signal source functions only affect the behavior of a module during small-signal analyses. The small-signal analyses provided by SPICE include the AC and noise analyses, but others are possible. When not active, the small-signal source functions return zero (0).

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

4.6.1 Analysis

The **`analysis()`** function takes one or more string arguments and returns one (`1`) if any argument matches the current analysis type. Otherwise it returns zero (`0`). The general form is:

```
analysis ( analysis_list )
```

There is no fixed set of analysis types. Each simulator can support its own set. However, simulators shall use the names listed in Table 4-21 to represent analyses which are similar to those provided by SPICE.

Table 4-21—Analysis types

Name	Analysis description
`"ac"`	.AC analysis
`"dc"`	.OP or .DC analysis (single point or dc sweep analysis)
`"noise"`	.NOISE analysis
`"tran"`	.TRAN analysis
`"ic"`	The initial-condition analysis which precedes a transient analysis.
`"static"`	Any equilibrium point calculation, including a DC analysis as well as those that precede another analysis, such as the DC analysis which precedes an AC or noise analysis, or the IC analysis which precedes a transient analysis.
`"nodeset"`	The phase during an equilibrium point calculation where nodesets are applied.

Any unsupported type names are assumed to not be a match.

Table 4-22 describes the implementation of the analysis function. Each column shows the return value of the function. A status of one (`1`) represents *True* and zero (`0`) represents *False.*

Table 4-22—Analysis function implementation

Analysis	Argument	DC	Sweep[a] d1 d2 dN	TRAN OP Tran	AC OP AC	NOISE OP AC
First part of `"static"` when nodesets are applied	`"nodeset"`	1	1 0 0	1 0	1 0	1 0
Initial DC state	`"static"`	1	1 1 1	1 0	1 0	1 0
Initial condition	`"ic"`	0	0 0 0	1 0	0 0	0 0
DC	`"dc"`	1	1 1 1	0 0	0 0	0 0
Transient	`"tran"`	0	0 0 0	1 1	0 0	0 0
Small-signal	`"ac"`	0	0 0 0	0 0	1 1	0 0
Noise	`"noise"`	0	0 0 0	0 0	0 0	1 1

[a]Sweep refers to a dc analysis in which a parameter is swept through multiple values. d1, d2 and dN above refer to dc points within the same sweep analysis.

Using the **`analysis()`** function, it is possible to have a module behave differently depending on which analysis is being run.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

For example, to implement nodesets or initial conditions using the **analysis()** function and switch branches, use the following.

```
if (analysis("ic"))
   V(cap) <+ initial_value;
else
   I(cap) <+ ddt(C*V(cap));
```

4.6.2 DC analysis

Verilog-AMS supports a single-point dc analysis and also a multipoint dc sweep analysis in which multiple dc points are computed over a sweep of parameter values. An operating point analysis is done for each dc point in the sweep. A single-point dc analysis is the same as an operating point analysis. The **analysis**("dc") and **analysis**("static") function calls shall return true for a single-point dc analysis and also for every dc point in a sweep analysis. The **analysis**("nodeset") function call shall return true only during the phase of an operating point analysis in which nodeset values are applied; that phase may occur in a single-point dc analysis or the first point of a multipoint dc sweep analysis, but does not occur for subsequent points of a dc sweep.

During a dc sweep analysis, the values of variables at the conclusion of the operating point analysis for one dc point shall be used as the starting values for those variables for the next dc point. However, variable values shall not be carried over between two independent dc sweep analyses (from the last dc point of one analysis to the first dc point of the next analysis). Variables shall be re-initialized to zero (or x, for integers whose values are assigned in a digital context) at the start of each new analysis.

4.6.3 AC stimulus

A small-signal analysis computes the steady-state response of a system which has been linearized about its operating point and is driven by a small sinusoid. The sinusoidal stimulus is provided using the **ac_stim()** function. The general form is:

```
ac_stim ( [ analysis_name [ , mag [ , phase ] ] ] )
```

The AC stimulus function returns zero (0) during large-signal analyses (such as DC and transient) as well as on all small-signal analyses using names which do not match *analysis_name.* The name of a small-signal analysis is implementation dependent, although the expected name (of the equivalent of a SPICE AC analysis) is “ac”, which is the default value of *analysis_name*. When the name of the small-signal analysis matches *analysis_name*, the source becomes active and models a source with magnitude *mag* and phase *phase*. The default magnitude is one (1) and the default phase is zero (0). *phase* is given in radians.

4.6.4 Noise

Several functions are provided to support noise modeling during small-signal analyses. To model large-signal noise during transient analyses, use the **$random()** or **$arandom()** system tasks. The noise functions are often referred to as noise sources. There are four noise functions, **white_noise()** models white noise processes, **flicker_noise()** models *1/f* or flicker noise processes, **noise_table()** interpolates a vector to model a process where the spectral density of the noise varies as a piecewise linear function of frequency, and lastly, **noise_table_log()** interpolates a vector to model a process where the spectral density of the noise varies as a piecewise linear function of the base-10 logarithm of frequency. The noise functions are only active in small-signal noise analyses and return zero (0) otherwise.

The syntax for noise functions is shown in Syntax 4-4.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
analog_small_signal_function_call ::=                                    // from A.8.2
    ...
  | white_noise ( analog_expression [ , string ] )
  | flicker_noise ( analog_expression , analog_expression [ , string ] )
  | noise_table ( noise_table_input_arg [ , string ] )
  | noise_table_log ( noise_table_input_arg [ , string ] )
```

Syntax 4-4—Syntax for the noise functions

4.6.4.1 white_noise

White noise processes are those whose current value is completely uncorrelated with any previous or future values. This implies their spectral density does not depend on frequency. They are modeled using:

```
white_noise ( pwr [ , name ] )
```

which generates white noise with a power of *pwr*.

For example, the thermal noise of a resistor could be modeled using:

```
I(a,b) <+ V(a,b)/R +
   white_noise(4 * `P_K * $temperature/R, "thermal");
```

The optional *name* argument acts as a label for the noise source used when the simulator outputs the individual contribution of each noise source to the total output noise. The contributions of noise sources with the same *name* from the same instance of a module are combined in the noise contribution summary.

4.6.4.2 flicker_noise

The **`flicker_noise()`** function models flicker noise. The general form is:

```
flicker_noise ( pwr , exp [ , name ] )
```

which generates pink noise with a power of *pwr* at 1Hz which varies in proportion to $1/f^{exp}$.

The optional *name* argument acts as a label for the noise source used when the simulator outputs the individual contribution of each noise source to the total output noise. The contributions of noise sources with the same *name* from the same instance of a module are combined in the noise contribution summary.

4.6.4.3 noise_table

The **`noise_table()`** function interpolates a set of values to model a process where the spectral density of the noise varies as a piecewise linear function of frequency. The general form is:

```
noise_table ( input [ , name ] )
```

The argument *input* can either be a vector or a string indicating a filename.

When the *input* is a vector it contains pairs of real numbers: the first number in each pair is the frequency in Hertz and the second is the power. The vector can either be specified as an array parameter or an array assignment pattern.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

When the *input* is a file name, the indicated file will contain the frequency / power pairs. The file name argument shall be constant and will be either a string literal or a string parameter. Each frequency / power pair shall be separated by a newline and the numbers in the pair shall be separated by one or more spaces or tabs. To increase the readability of the data file, comments may be inserted before or after any frequency / power pair. Comments begin with '#' and end with a newline. The input file shall be in text format only and the numbers shall be real or integer.

The following shows an example of the input file:

```
# noise_table_input.tbl
# Example of input file format for noise_table
#
#     freq        pwr
      1.0e0    1.657580e-23
      1.0e1    3.315160e-23
      1.0e2    6.636320e-23
      1.0e3    1.326064e-22
      1.0e4    2.652128e-22
      1.0e5    5.304256e-22
      1.0e6    1.060851e-21

# End of example input file.
```

Although the user is encouraged to specify each noise pair in order of ascending frequency, the simulator shall internally sort the pairs into ascending frequency if required. Each frequency value must be unique. **`noise_table()`** performs piecewise linear interpolation to compute the power spectral density generated by the function at each frequency between the lowest and highest frequency in the set of values. For frequencies lower than the lowest frequency in the value set, **`noise_table()`** returns the power specified for the lowest frequency, and for frequencies higher than the highest frequency, **`noise_table()`** returns the power specified for the highest frequency.

The optional *name* argument acts as a label for the noise source used when the simulator outputs the individual contribution of each noise source to the total output noise. The contributions of noise sources with the same *name* from the same instance of a module are combined in the noise contribution summary.

4.6.4.4 noise_table_log

The **`noise_table_log()`** function interpolates a set of values to model a process where the spectral density of the noise varies as a piecewise linear function of the base-10 logarithm of frequency. The general form is:

```
noise_table_log ( input [ , name ] )
```

The argument *input* can either be a vector or a string indicating a filename; in either case, the meaning and restrictions on the input are the same as for **`noise_table()`**. The difference is that **`noise_table_log()`** interpolates logarithmically. For a frequency *f* not specified in the input data, the noise power shall be computed using the two pairs (*f1*,*p1*) and (*f2*,*p2*) in the input (whether an array or file), where *f1* is the largest frequency value in the input data less than *f* and *f2* is the smallest frequency larger than *f* (that is, $f1 < f < f2$); the noise power *P* is computed as:

```
P = pow(10, log(p1) + (log(p2)-log(p1)) * (log(f)-log(f1)) / (log(f2)-
log(f1)) )
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

As with **noise_table()**, for frequencies lower than the lowest frequency in the value set, **noise_table_log()** returns the power specified for the lowest frequency, and for frequencies higher than the highest frequency, **noise_table_log()** returns the power specified for the highest frequency.

The optional *name* argument acts as a label for the noise source used when the simulator outputs the individual contribution of each noise source to the total output noise. The contributions of noise sources with the same *name* from the same instance of a module are combined in the noise contribution summary.

The difference between **noise_table()** and **noise_table_log()** is illustrated in Figure 4-14.

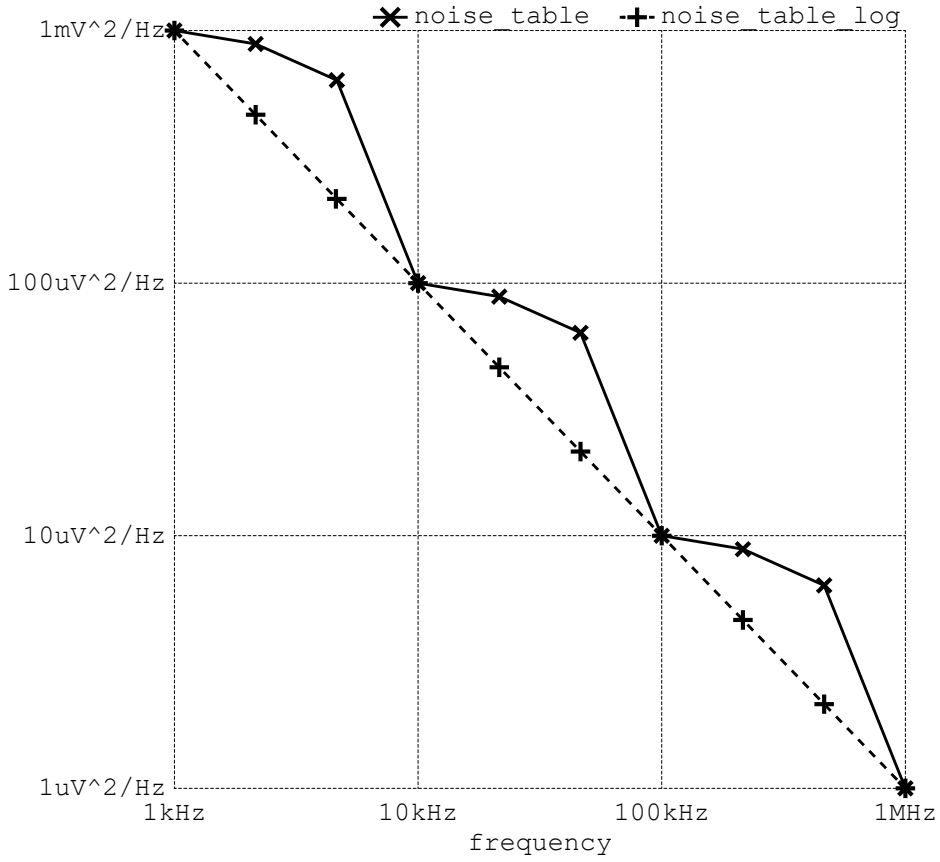


Figure 4-14: Comparison of noise_table and noise_table_log

The **noise_table_log()** function produces a straight line on a log-log plot from just two points:

```
V(out) <+ noise_table_log('{1,1, 1e6,1e-6});
```

whereas the linear interpolation of noise_table produces a series of curves between the interpolating points, depending on the number of points specified in the function call and the number of points per decade in the small-signal analysis. Here, one point per decade is specified:

```
V(out) <+ noise_table('{1,1, 1e1,1e-1, 1e2,1e-2, 1e3,1e-3, 1e4,1e-4,
                  1e5,1e-5, 1e6,1e-6});
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

4.6.4.5 Noise model for diode

The noise of a junction diode could be modeled as shown in the following example.

```
I(a,c) <+ is*(exp(V(a,c) / (n * $vt)) - 1)
   + white_noise(2*`P_Q*I(<a>))
   + flicker_noise(kf*pow(abs(I(<a>)), af), ef);
```

4.6.4.6 Correlated noise

Each noise function generates noise which is uncorrelated with the noise generated by other functions. *Perfectly correlated noise* is generated by using the output of one noise function for more than one noise source. *Partially correlated noise* is generated by combining the output of shared and unshared noise functions.

Example 1 — Two noise voltages are perfectly correlated.

```
n = white_noise(pwr);
V(a,b) <+ c1*n;
V(c,d) <+ c2*n;
```

Example 2 - Partially correlated noise sources can also be modeled.

```
n1 = white_noise(1-corr);
n2 = white_noise(1-corr);
n12 = white_noise(corr);
V(a,b) <+ Kv*(n1 + n12);
I(b,c) <+ Ki*(n2 + n12);
```

4.7 User-defined functions

A user-defined function can be used to return a value (for an expression). All functions are defined within modules. Each function can be an analog user-defined function or a digital function (as defined in IEEE Std 1364 Verilog).

4.7.1 Defining an analog user-defined function

The syntax for defining an analog user-defined function is shown in Syntax 4-5.

---

analog_function_declaration ::= *// from A.2.6*
    analog function [analog_function_type] analog_function_identifier **;**
        analog_function_item_declaration { analog_function_item_declaration }
        analog_function_statement
    endfunction

analog_function_type ::= **integer** | **real** | **string**

analog_function_item_declaration ::=
    analog_block_item_declaration
  | input_declaration **;**
  | output_declaration **;**
  | inout_declaration **;**

analog_block_item_declaration ::= *// from A.2.8*
    { attribute_instance } parameter_declaration **;**
  | { attribute_instance } integer_declaration
  | { attribute_instance } real_declaration

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
| { attribute_instance } string_declaration
```

Syntax 4-5—Syntax for an analog user-defined function declaration

An analog user-defined function declaration shall begin with the keywords **`analog function`**, optionally followed by the type of the return value from the function, then the name of the function and a semicolon, and ending with the keyword **`endfunction`**.

The *analog_function_type* specifies the return value of the function; its use is optional. *type* can be **`real`**, **`integer`**, or **`string`**; if unspecified, the default is **`real`**.

An analog user-defined function:

- can use any statements available for conditional execution (see 5.2);
- shall not use access functions;
- shall not use analog filter functions;
- shall not use contribution statements or event control statements;
- shall have at least one formal argument declared;
- all formal arguments shall have an associated block item declaration specifying the data type of the argument;
- all formal arguments shall have an associated direction specification that shall be either **`input`**, **`output`**, or **`inout`**;
- shall not use named blocks;
- shall only reference locally-defined variables, variables passed as arguments, locally-defined parameters and module-level parameters; and
- if a locally-defined parameter with the specified name does not exist, then the module-level parameter of the specified name will be used.

Example 1 — Determine max value:

This example defines an analog user-defined function called `maxValue`, which returns the potential of whichever signal is larger.

```
analog function real maxValue;
   input n1, n2;
   real n1, n2;
   begin
      // code to compare potential of two signals
      maxValue = (n1 > n2) ? n1 : n2;
   end
endfunction
```

Example 2 — Area and perimeter of a rectangle

This example defines an analog user-defined function called `geomcalc`, which returns both the area and perimeter of a rectangle.

```
analog function real geomcalc;
   input l, w;
   output area, perim;
   real l, w, area, perim;
   begin
      area = l * w;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
      perim = 2 * ( l + w );
   end
endfunction
```

Example 3 — Initialization of a vector variable

The analog user-defined function called `arrayadd` adds the contents of a second array to the first.

```
analog function real arrayadd;
   inout [0:1]a;
   input [0:1]b;
   real a[0:1], b[0:1];
   integer i;
   begin
      for(i = 0; i < 2; i = i + 1) begin
         a[i] = a[i] + b[i];
      end
   end
endfunction
```

4.7.2 Returning a value from an analog user-defined function

There are four ways to return a value from an analog user-defined function: using the implicit analog user-defined function identifier variable, using a **`return`** statement, using an output argument, or using an inout argument.

4.7.2.1 Analog user-defined function identifier variable

The analog user-defined function definition implicitly declares a variable, internal to the analog user-defined function, with the same name as the *analog_function_identifier*. This variable inherits the same type as the type specified in the analog user defined function declaration. This internal variable is initialized to zero (0) if the inherited type is numerical, or the empty string (“”) if a string, and can be used within the body of the analog user-defined function. The last value assigned to this variable will be the return value of the analog user-defined function. If this internal variable is not assigned during the execution of the analog user-defined function, then the analog user-defined function will return the variables default initial value. An analog user-defined function shall always return a scalar value, either numeric or string based on the type specified for the analog user-defined function.

The following line (from the first example in 4.7.1) illustrates this concept:

```
maxValue = (n1 > n2) ? n1 : n2;
```

4.7.2.2 Analog function return statement

The **`return`** statement shall override any value assigned to the function name. When the **`return`** statement is used, the function shall specify an expression with the **`return`** of the correct type for the function.

```
return (n1 > n2) ? n1 : n2;
```

4.7.2.3 Output arguments

An **`output`** argument allows the user to return more than one value. The argument passed to an **`output`** argument must be an analog variable reference. If the **`output`** argument is defined as an array then the argument passed into the function must be an analog variable or an array assignment pattern of analog variables of equivalent size. All **`output`** arguments of an analog user-defined function are initialized, zero (0)

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

if numeric, empty string (“”) if a string, which in turn means that the argument passed to it is reset to zero (0) or the empty string (“”) accordingly. During the execution of the function, these variables can be read and assigned in the flow. At the end of the execution of the analog user-defined function, the last value assigned to the **output** argument is then assigned to the corresponding analog variable reference that was passed into the function.

The following lines (from the second example in 4.7.1) illustrate this concept:

```
area = l * w;
perim = 2 * ( l + w );
```

4.7.2.4 Inout arguments

inout arguments allow the user to pass in a value to the function and return a different value from it using the same argument. The argument passed to an **inout** argument must be an analog variable reference. If the **inout** argument is defined as an array then the argument passed into the function must be an analog variable or an array assignment pattern of analog variables of equivalent size. The **inout** arguments of an analog user-defined function do not get initialized like those defined as **output**. During the execution of the function, these variables can be read and assigned in the flow. At the end of the execution of the analog user-defined function, the last value assigned to the **inout** argument is then assigned to the corresponding analog variable reference that was passed into the function. If a value was not assigned to the **inout** argument during the execution of the analog user-defined function, then the corresponding analog variable reference is left untouched.

The following lines (from the third example in 4.7.1) illustrate the use of an **inout** argument.

```
for(i = 0; i < 2; i = i + 1) begin
   a[i] = a[i] + b[i];
end
```

Note: **inout** arguments are not “pass by reference”, but more closely related to “copy in” and “copy out”. Care should be taken to avoid passing the same analog variable reference to different **inout** and **output** arguments of the same analog user-defined function as the results are undefined.

4.7.3 Calling an analog user-defined function

An analog user-defined function call is an operand within an expression. Syntax 4-6 shows the analog user-defined function call.

---

analog_function_call ::= *// from A.8.2*
    analog_function_identifier { attribute_instance } (analog_expression { , analog_expression })

Syntax 4-6—Syntax for function call

The order of evaluation of the arguments to an analog user-defined function call is undefined. The argument expressions are assigned to the declared inputs, outputs, and inouts in the order of their declaration.

An analog user-defined function:

— shall not call itself directly or indirectly, i.e., recursive functions are not permitted; and

— shall only be called within the analog context, either from an **analog** block or from within another analog user-defined function;

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The following example uses the `maxValue` function defined in 4.7.1.

```
V(out) <+ maxValue(val1, val2);
```

The following example uses the `geomcalc` function defined in 4.7.1.

```
dummy = geomcalc(l-dl, w-dw, ar, per);
```

Note that the first two arguments are expressions, and match up with the inputs `l` and `w` for the function; the second two arguments must be real identifiers because they match up with the function outputs.

The following example incorrectly uses the `geomcalc` function defined in 4.7.1.

```
dummy = geomcalc(l-dl, w-dw, ar, V(a));
```

Here the last two arguments to the user-defined function `geomcalc` are declared as **`output`** arguments, but the fourth argument is passed the potential probe *V(a)*. Only analog variable references can be passed to **`output`** and **`inout`** arguments of an analog user-defined function so this example will result in a compilation error.

The following example uses the `arrayadd` example defined in 4.7.1, to add values from one array to another.

```
x[0] = 5; x[1] = 10;
y = 3; z = 6;
dummy = arrayadd(x,'{y,z});
```

Here the first and second arguments are both expecting vectors. A vector variable is passed for the first argument and an array assignment pattern of two scalar analog variables has been used for the second argument. Since the first argument is an **`inout`** argument, the result of calling the `arrayinit` function will update the vector variable `x` with values `x[0] = 8 and x[1] = 16`.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

5. Analog behavior

5.1 Overview

The description of an analog behavior consists of setting up contributions for various signals under certain procedural or timing control. This section describes an analog procedural block, analog signals, contribution statements, procedural control statements, and analog timing control functions.

5.2 Analog procedural block

Discrete time behavioral definitions within IEEE Std 1364 Verilog are encapsulated within the **`initial`** and **`always`** procedural blocks. Every **`initial`** and **`always`** block starts a separate concurrent activity flow. For continuous time simulation, the behavioral description is encapsulated within the analog procedural block. The syntax for **`analog`** block is shown in Syntax 5-1.

```
analog_construct ::=                                          // from A.6.2
        analog analog_statement
      | analog initial analog_function_statement
analog_statement ::=                                          // from A.6.4
        { attribute_instance } analog_loop_generate_statement
      | { attribute_instance } analog_loop_statement
      | { attribute_instance } analog_case_statement
      | { attribute_instance } analog_conditional_statement
      | { attribute_instance } analog_procedural_assignment
      | { attribute_instance } analog_seq_block
      | { attribute_instance } analog_system_task_enable
      | { attribute_instance } contribution_statement
      | { attribute_instance } indirect_contribution_statement
      | { attribute_instance } analog_event_control_statement
analog_statement_or_null ::=
        analog_statement
      | { attribute_instance } ;
```

Syntax 5-1—Syntax for analog procedural block

The analog procedural block defines the behavior as a procedural sequence of statements. The conditional and looping constructs are available for defining behaviors within the analog procedural block. Because the description is a continuous-time behavioral description, no blocking event control statements (such as blocking delays, blocking events, or waits) are supported.

All the statements within the **`analog`** block shall be executed sequentially at a given point of time, however the effects on the analog variables, nets, and branches contained in various modules in a design are considered concurrently. Analog blocks shall be executed at every point in a simulation. Multiple analog blocks can also be used within a module declaration. Refer 6.2 for more details on multiple analog blocks.

5.2.1 Analog initial block

An *analog initial block* is a special analog (procedural) block, beginning with the keywords **`analog initial`**, for simulation initialization purposes.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Like a regular **analog** block, an **analog initial** block is also comprised of a procedural sequence of statements. If there are multiple **analog initial** blocks, they are executed as if concatenated. However, statements in **analog initial** blocks are restricted for initialization purposes. So an **analog initial** block shall not contain the following statements:

— statements with access functions or analog operators;

— contribution statements;

— event control statements.

This is similar to the restrictions on the statements in analog functions.

This is because an **analog initial** block is executed before a matrix solution is available so statements in an **analog initial** block are restricted to initialization purposes prior to the availability of a solution of both the digital and the analog modules.

Additionally, digital values cannot be accessed from the **analog initial** block as they have not yet been assigned when the **analog initial** block is executed.

The **analog initial** block is executed once for each analysis, and can be executed for each sub-task of parameter sweep analysis (such as dc sweep). The initialization sequence of analog and digital blocks/statements is described in 8.2 and 8.4.1. If a parameter or variable that is referenced from an **analog initial** block is changed during a sub-task of a parameter sweep analysis, then the **analog initial** block shall be re-executed so that the new value is taken into account.

5.3 Block statements

The *block statements*, also referred to as *sequential blocks*, are a means of grouping procedural statements. The statements within the block shall be executed in sequence, one after another in the given order and the control shall pass out of the block after the last statement is executed. The block statements are delimited by the keywords **begin** and **end**.

5.3.1 Sequential blocks

The syntax for sequential blocks is shown in Syntax 5-2.

```
analog_seq_block ::=                                              // from A.6.3
        begin [ : analog_block_identifier { analog_block_item_declaration } ]
            { analog_statement } end
analog_block_item_declaration ::=                                 // from A.2.8
          { attribute_instance } parameter_declaration ;
        | { attribute_instance } integer_declaration
        | { attribute_instance } real_declaration
```

Syntax 5-2—Syntax for the sequential blocks

5.3.2 Block names

A sequential block can be named by adding a **:***block_identifier* after the keyword **begin**. The naming of a block allows local variables to be declared for that block. The block names give a means of uniquely identifying all variables at any simulation time.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

All named block variables are static—that is, an unique location exists for all variables and leaving or entering the block do not affect the values stored in them. All identifiers declared within a named sequential block can be accessed outside the scope in which they are declared. Named block variables cannot be assigned outside the scope of the block in which they are declared.

Parameters declared within a named block have local scope and cannot be assigned outside the scope. Named and ordered parameter overrides at module instantiation can only affect parameters declared at module scope.

```
module example;
   parameter integer p1 = 1;
   real moduleVar;

   analog begin
      begin: myscope
         parameter real p2 = p1;
         real localVar = 1.5 * p2;
      end
      moduleVar = myscope.localVar;
   end
endmodule

module top;
   example #(.p1(4)) inst1();             // allowed
   example #(.myscope.p2(4)) inst2();     // error
endmodule
```

5.4 Analog signals

Analog signals are distinguished from digital signals in that an *analog signal* has a discipline with a continuous domain. Disciplines, nets, nodes, and branches are described in and ports are described in Clause 6.

This section describes analog branch assignments, signal access mechanisms, and operators in Verilog-AMS HDL.

5.4.1 Access functions

Flows and potentials on nets, ports, and branches are accessed using *access functions*. The name of the access function is taken from the discipline of the net, port, or branch associated with the signal.

Example 1 — Consider a named electrical branch `b` where *electrical* is a discipline with *V* as the access function for the potential and *I* as the access function for the flow. The potential (voltage) is accessed via `V(b)` and the flow (current) is accessed via `I(b)`.

— There can be any number of named branches between any two signals.

— Unnamed branches are accessed in a similar manner, except the access functions are applied to net names or port names rather than branch names.

Example 2 — If *n1* and *n2* are electrical nets or ports, then `V(n1, n2)` creates an unnamed branch from *n1* to *n2* (if it does not already exist) and then accesses the branch potential (or the potential difference between *n1* to *n2),* and `V(n1)` does the same from *n1* to the global reference node (*ground*).

— In other words, accessing the potential from a net or port to a net or port defines an unnamed branch. Accessing the potential on a single net or port defines an unnamed branch from that net or port to the global reference node (*ground*). There can only be one unnamed branch between any two nets or between a net and implicit ground (in addition to any number of named branches).

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

— An analogous access method is used for flows.

Example 3 — `I(n1, n2)` creates an unnamed branch from *n1* to *n2* (if it does not already exist) and then accesses the branch flow, and `I(n1)` does the same from *n1* to the global reference node (*ground*).

— Thus, accessing the flow from a net or port to a net or port defines an unnamed branch. Accessing the potential on a single net or port defines an unnamed branch from that net or port to the global reference node (*ground*).

— It is also possible to access the flow passing through a port into a module. The name of the access function is derived from the flow nature of the discipline of the port. In this case, `(<>)` is used to delimit the port name rather than `()`.

Example 4 — `I(<p1>)` is used to access the current flow into the module through the electrical port `p1`. This capability is discussed further in 5.4.3.

5.4.2 Probes and sources

An analog component can be represented using a network of probes and controlled sources. The Verilog-AMS HDL uses the concept of *probes* and *sources* as a means of unambiguously representing a network. The mapping between these representations are defined in following subsections.

5.4.2.1 Probes

If no value is specified for either the potential or the flow, the branch is a *probe*. If the flow of the branch appears in an expression anywhere in the module, the branch is a *flow probe*, otherwise the branch is a *potential probe*. Using both the potential and the flow of a probe branch is illegal. The models for probe branches are shown in Figure 5-1.

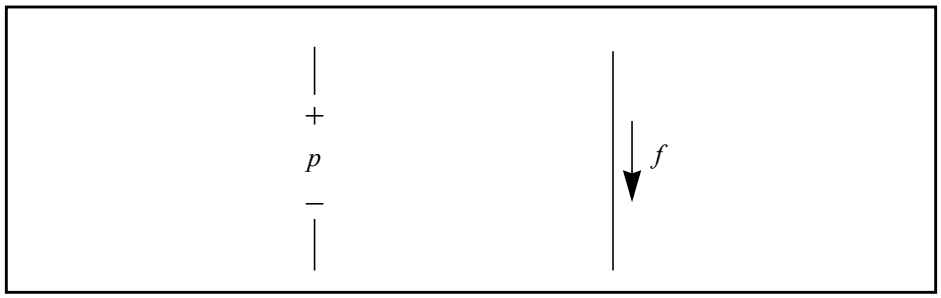


Figure 5-1: Equivalent circuit models for probe branches

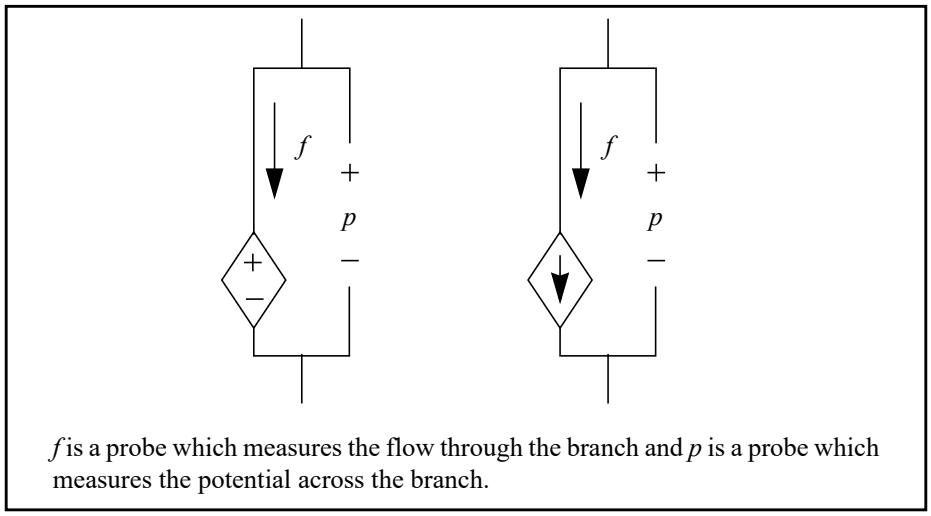
The branch potential of a flow probe is zero (`0`). The branch flow of a potential probe is zero (`0`).

5.4.2.2 Sources

A branch, either named or unnamed, is a *source branch* if either the potential or the flow of that branch is assigned a value by a contribution statement (see 5.6) anywhere in the module. It is a *potential source* if the branch potential is specified and is a *flow source* if the branch flow is specified. A branch cannot simultaneously be both a potential and a flow source, although it can switch between them (a *switch branch*).

Both the potential and the flow of a source branch are accessible in expressions anywhere in the module. The models for potential and flow sources are shown in Figure 5-2.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.



f is a probe which measures the flow through the branch and *p* is a probe which measures the potential across the branch.

Figure 5-2: Equivalent circuit models for source branches

5.4.3 Accessing flow through a port

The port access function accesses the flow into a port of a module. The name of the access function is derived from the flow nature of the discipline of the port. However `(<>)` is used to delimit the port name, e.g., `I(<a>)` accesses the current through module port *a*.

Example 1 — Consider the junction `diode` below, where the total diode current is monitored and a message is issued if it exceeds a given value.

```
module diode (a, c);
   inout a, c;
   electrical a, c;
   branch (a, c) i_diode, junc_cap;
   parameter real is = 1e-14, tf = 0, cjo = 0, imax = 1, phi = 0.7;
   analog begin
      I(i_diode) <+ is*(limexp(V(i_diode)/$vt) - 1);
      I(junc_cap) <+
         ddt(tf*I(i_diode) - 2*cjo*sqrt(phi*(phi*V(junc_cap))));
      if (I(<a>) > imax)
         $strobe( "Warning: diode is melting!" );
   end
endmodule
```

The expression `V(<a>)` is invalid for ports and nets, where `V` is a potential access function. The port access function shall not be used on the left side of a contribution operator `<+`.

Example 2 — An ideal relay (a controlled switch) can be implemented as:

```
module relay (p, n, ps, ns);
   inout p, n, ps, ns;
   electrical p, n, ps, ns;
   parameter vth=0.5;
   integer closed;
   analog begin
      closed = (V(ps,ns) >vth ? 1 : 0);
      if (closed)
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
            V(p,n) <+ 0;
         else
            I(p,n) <+ 0;
      end
   endmodule
```

A discontinuity of order zero (0) is assumed to occur when the branch switches and so it is not necessary to use the **$discontinuity** function with switch branches.

5.4.4 Unassigned sources

If a value is not assigned to a branch, and it is not a probe branch, the branch flow is set to zero (0).

Examples:

```
if (closed)
   V(p,n) <+ 0;
```

is equivalent to

```
if (closed)
   V(p,n) <+ 0;
else
   I(p,n) <+ 0;
```

5.5 Accessing net and branch signals and attributes

The methods for accessing signal and attributes of nets and branches are described in this section.

5.5.1 Accessing net and branch signals

Signals on nets and branches can be accessed only by either the access functions of the discipline associated with them or by the generic potential or flow access functions. The name of the net or the branch shall be specified as the argument to the access function. The syntax for analog signal access is shown in Syntax 5-3.

---

nature_access_function ::= *// from A.8.2*
    nature_attribute_identifier
  | **potential**
  | **flow**

branch_probe_function_call ::=
    nature_access_function **(** branch_reference **)**
  | nature_access_function **(** analog_net_reference [**,** analog_net_reference] **)**

port_probe_function_call ::= nature_access_function **(** **<** analog_port_reference **>** **)**

branch_reference ::= *// from A.8.9*
    hierarchical_branch_identifier
  | hierarchical_branch_identifier **[** constant_expression **]**
  | hierarchical_unnamed_branch_reference

hierarchical_unnamed_branch_reference ::=
    hierarchical_inst_identifier**.branch** **(** branch_terminal [**,** branch_terminal] **)**
  | hierarchical_inst_identifier**.branch** **(** **<** port_identifier **>** **)**
  | hierarchical_inst_identifier**.branch** **(** **<** hierarchical_port_identifier **>** **)**

analog_net_reference ::=

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    port_identifier
  | port_identifier [ constant_expression ]
  | net_identifier
  | net_identifier [ constant_expression ]
  | hierarchical_port_identifier
  | hierarchical_port_identifier [ constant_expression ]
  | hierarchical_net_identifier
  | hierarchical_net_identifier [ constant_expression ]
analog_port_reference ::=
    port_identifier
  | port_identifier [ constant_expression ]
  | hierarchical_port_identifier
  | hierarchical_port_identifier [ constant_expression ]
```

Syntax 5-3—Syntax for analog signal access

Branch or port probe function calls shall only reference nets and ports that have been declared to belong to a continuous discipline; references to branches require that the branch terminals belong to a continuous discipline. The nature attribute identifier for a branch probe function call must be the access function name for the potential or flow nature defined for the discipline associated with the nets or branches. For a port probe function call, the nature attribute identifier must be the access function name for the flow nature associated with the port, and the port reference many not use hierarchical specifications, i.e., it must be a declared port of the module in which the port access function is used.

The examples below use the electrical discipline defined in 3.6.2.1 and its associated natures and their access functions defined in 3.6.1.

```
module transamp(out, in);
   inout out, in;
   electrical out, in;
   parameter real gm = 1;
   analog
      I(out) <+ gm*V(in);
endmodule

module resistor(p, n);
   inout p, n;
   electrical p, n;
   branch (p,n) res;
   parameter real R = 50;
   analog
      V(res) <+ R*I(res);
endmodule
```

The **potential** and **flow** access functions can also be used to access the potential or flow of a named or unnamed branch. The example below demonstrates the potential access functions being used. Note V cannot be used as an access function because there is a parameter called V declared in the module.

```
module measure1(p);
   output p;
   electrical p;
   parameter real V = 1.1;
   analog begin
      $strobe("%M: voltage ratio is %g", potential(p) / V);
   end
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
endmodule
```

When the potential and flow access functions are used on an unnamed branch composed of two nets – the discipline of both nets must be the same.

5.5.2 Signal access for vector branches

Verilog-AMS HDL allows ports, nets, and branches to be arranged as vectors, however, the access functions can only be applied to scalars or individual elements of a vector. The scalar element of a vector is selected with an index, e.g., `V(in[1])` accesses the voltage `in[1]`.

The index must be a constant expression, though it may include genvar variables. Genvar variables can only be assigned to as the iteration index of for loops; they allow signal access within looping constructs.

The following examples illustrate applications of access functions to elements of a an analog signal vector or bus. In the N-bit `DAC` example, the analog vector `in` is accessed within an analog for-loop containing the genvar variable `i`. In the following fixed-width `DAC8` example, literal values are used to access elements of the bus directly.

```
//
// N-bit DAC example.
//

module dac(out, in, clk);
   parameter integer width = 8 from [2:24];
   parameter real fullscale = 1.0, vth = 2.5, td = 1n, tt = 1n;
   output out;
   input [0:width-1] in;
   input clk;
   electrical out;
   electrical [0:width-1] in;
   electrical clk;

   real aout;
   genvar i;

   analog begin
      @(cross(V(clk) - vth, +1)) begin
         aout = 0;
         for (i = width - 1; i >= 0; i = i - 1) begin
            if (V(in[i]) > vth) begin
               aout = aout + fullscale/pow(2, width - i);
            end
         end
      end
      V(out) <+ transition(aout, td, tt);
   end
endmodule

//
// 8-bit fixed-width DAC example.
//

module dac8(out, in, clk);
   parameter real fullscale = 1.0, vth = 2.5, td = 1n, tt = 1n;
   output out;
   input [0:7] in;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    input clk;
    electrical out;
    electrical [0:7] in;
    electrical clk;

    real aout;

    analog begin
       @(cross(V(clk) - 2.5, +1)) begin
          aout = 0;
          aout = aout + ((V(in[7]) > vth) ? fullscale/2.0 : 0.0);
          aout = aout + ((V(in[6]) > vth) ? fullscale/4.0 : 0.0);
          aout = aout + ((V(in[5]) > vth) ? fullscale/8.0 : 0.0);
          aout = aout + ((V(in[4]) > vth) ? fullscale/16.0 : 0.0);
          aout = aout + ((V(in[3]) > vth) ? fullscale/32.0 : 0.0);
          aout = aout + ((V(in[2]) > vth) ? fullscale/64.0 : 0.0);
          aout = aout + ((V(in[1]) > vth) ? fullscale/128.0 : 0.0);
          aout = aout + ((V(in[0]) > vth) ? fullscale/256.0 : 0.0);
       end

       V(out) <+ transition(aout, td, tt);
    end

endmodule
```

5.5.3 Accessing attributes

Attributes are attached to the nature of a potential or flow. Therefore, the attributes for a net or a branch can be accessed by using the hierarchical referencing operator (**.**) to the potential or flow for the net or branch.

Example:

```
module twocap(a, b, n1, n2);
   inout a, b, n1, n2;
   electrical a, b, n1, n2;
   branch (n1, n2) cap;
   parameter real c= 1p;
   analog begin
      I(a,b) <+ c*ddt(V(a,b), a.potential.abstol);
      I(cap) <+ c*ddt(V(cap), n1.potential.abstol);
   end
endmodule
```

The syntax for referencing access attributes is shown in Syntax 5-4. This syntax shall not be used for the **access**, **ddt_nature**, or **idt_nature** attributes of a nature, nor any other attribute whose value is not a constant expression.

---

nature_attribute_reference ::= *// from A.8.9*
    net_identifier . potential_or_flow . nature_attribute_identifier

potential_or_flow ::= **potential** | **flow** *// from A.1.7*

Syntax 5-4—Syntax for referencing attributes of a net

The **abstol** attribute of a nature may also be accessed simply by using the nature's identifier as the appropriate argument to the **ddt()**, **idt()**, or **idtmod()** operators described in 4.5.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

5.5.4 Creating unnamed branches using hierarchical net references

An access function in a module can have one or more hierarchical net references to nets in other module instances. In these cases, a new unnamed branch is created in the module containing the access function call.

Example:

```
module signal_monitor;
   parameter refv = 2.3;
   electrical a;
   analog begin
      V(a) <+ refv;

      // Creates an unnamed branch in module signal_monitor between
      // nets top.drv.a and implicit ground.
      $strobe("voltage at top.drv.a = %g volts", V(top.drv.a));

      // Creates an unnamed branch in module signal_monitor between
      // nets top.drv.a and top.drv.b
      $strobe("voltage diff in top.drv = %g volts", V(top.drv.a, top.drv.b));

      // Creates an unnamed branch in module signal_monitor between
      // local net a and top.drv.a
      $strobe("voltage diff from ref in top.drv = %g volts", V(a,top.drv.a));

      // References the unnamed branch created in the first $strobe()
      // statement
      if(V(top.drv.a) > 10.0) $strobe("voltage limit exceeded at top.drv.a");

   end
endmodule
```

Note that even if the instance *top.drv* already has an unnamed branch between nodes *a* and `ground` and *a* and *b*, the new unnamed branches are created in the module *signal_monitor*.

5.5.5 Accessing nets and branch signals hierarchically

A module is allowed to access the potential and flow of a branch in another module instance using an access function providing that value is available in the other instance. If it is not available, then an error shall be reported. Reasons why it would be unavailable are:

— The branch does not exist in the other instance
— The access function is not the valid access function for that named branch

An example of a hierarchical access of the potential of a named branch is:

```
module top;
   A a1();
   B b1();
endmodule

module A;
   electrical n,p;
   branch (n,p) b;
   analog V(b) <+ 1.34;
endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
module B;
   analog $strobe("voltage == %g", V(top.a1.b));
endmodule
```

To access an existing unnamed branch in another module instance, the *hierarchical_unnamed_branch_reference* syntax is used.

Example:

```
analog begin
   // strobes the voltage of the unnamed branch between
   // nets a and b in top.drv.
   $strobe("Voltage == %g", V(top.drv.branch(a,b)));

   // strobes the current flowing through the unnamed port
   // branch for the port p in top.drv
   $strobe("Current == %g", I(top.drv.branch(<p>)));
end
```

5.6 Contribution statements

The branch contribution statement is used in the **analog** block to describe continuous-time behavior between a module's analog nets and ports. Contribution statements may be described in direct or indirect form.

5.6.1 Direct branch contribution statements

The direct contribution statement uses the *branch contribution operator* **<+** to describe the mathematical relationship between one or more analog nets within the module. The mapping is done with contribution statements using the form shown in Syntax 5-5:

```
contribution_statement ::= branch_lvalue <+ analog_expression ;          // from A.6.10
branch_lvalue ::= branch_probe_function_call                             // from A.8.5
branch_probe_function_call ::=                                           // from A.8.2
        nature_access_function ( branch_reference )
      | nature_access_function ( analog_net_reference [ , analog_net_reference ] )
```

Syntax 5-5—Syntax for branch contribution

In general, a branch contribution statement consists of two parts, a left-hand side and a right-hand side, separated by a branch contribution operator. The right-hand side can be *analog_expression* can be any combination of linear, nonlinear, or differential expressions of module signals, constants, and parameters which evaluates to or can be promoted to a real value. The left-hand side specifies the source branch signal where the right-hand side shall be assigned. It shall consist of a signal access function applied to a branch.

If the branch contribution statement is conditionally executed, the expression shall not include an analog filter function, as described in 4.5, unless the conditional expression is a constant expression.

Electrical behavior can be described using:

```
V(n1, n2) <+ expression;
```

or

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
I(n1, n2) <+ expression;
```

where `(n1,n2)` represents an unnamed source branch and `V(n1,n2)` refers to the potential on the branch, while `I(n1,n2)` refers to the flow through the branch. The ‘V’ and ‘I’ functions (access attributes of the nature) are obtained from the discipline’s potential and flow bindings of the electrical net (refer 3.6 for further details on disciplines and natures).

Implementations may issue a warning if a contribution is made to an analog port declared with an input direction. There are no restrictions on the probing of an analog port declared with an output direction.

There shall be no contributions to an implicit net; contributions shall be done only on analog nets declared with a continuous discipline.

For example, the following modules model a resistor and a capacitor.

```
module resistor(p, n);
   inout p, n;
   electrical p, n;
   branch (p,n) path;    // named branch
   parameter real r = 0;

   analog
      V(path) <+ r*I(path);
endmodule

module capacitor(p, n);
   inout p, n;
   electrical p, n;
   parameter real c = 0;

   analog
      I(p,n) <+ c*ddt(V(p, n)); // unnamed branch p,n
endmodule
```

The **potential** and **flow** access functions can also be used to contribute to the potential or flow of a named or unnamed branch. The example below demonstrates the **potential** access functions being used to contribute to a branch and the **flow** and **potential** access functions being used to probe branches. Note V and I cannot be used as access functions because there are parameters called V and I declared in the module.

```
module measure2(p);
   output p;
   electrical p;
   parameter real V = 1.1;
   parameter real I = 1u;
   parameter real R = 10k;
   analog begin
      potential(p) <+ flow(p) * R; // create a resistor
      $strobe("voltage ratio at port 'p' is %g", potential(p) / V);
      $strobe("current ratio through port 'p' is %g", flow(<p>) / I);
   end
endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

5.6.1.1 Relations

Branch contribution statements implicitly define source branch relations. The branch is directed from the first net of the access function to the second net. If the second net is not specified, the global reference node (*ground*) is used as the reference net.

A branch relation is a path of the flow between two nets in a module. Each net has two quantities associated with it—the potential of the net and the flow out of the net. In electrical circuits, the potential of a net is its voltage, whereas the flow out of the net is its current. Similarly, each branch has two quantities associated with it—the potential across the branch and the flow through the branch.

For example, the following module models a simple single-ended amplifier.

```
module amp(out, in);
   input in;
   output out;
   electrical out, in;
   parameter real Gain = 1;

   analog
      V(out) <+ Gain*V(in);
endmodule
```

5.6.1.2 Evaluation

A statement is evaluated as follows for source branch contributions:

1) The simulator evaluates the right-hand side.
2) The simulator adds the value of the right-hand side to any previously retained value of the branch for later assignment to the branch. If there are no previously retained values, the value of the right-hand side itself is retained.
3) At the end of the simulation cycle, the simulator assigns the retained value to the source branch.

Parasitics are added to the amplifier shown in 5.6.1.1 by simply adding additional contribution statements to model the input admittance and output impedance.

Examples:

```
module amp(out, in);
   inout out, in;
   electrical out, in;
   parameter real Gain = 1, Rin = 1, Cin = 1, Rout = 1, Lout = 1;

   analog begin
      // gain of amplifier
      V(out) <+ Gain*V(in);

      // model input admittance
      I(in) <+ V(in)/Rin;
      I(in) <+ Cin*ddt(V(in));

      // model output impedance
      V(out) <+ Rout*I(out);
      V(out) <+ Lout*ddt(I(out));
   end
endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

5.6.1.3 Value retention

When solving an **analog** block during an iteration, multiple contributions to the same potential branch or same flow branch will be additive. However, contributing a flow to a branch which already has a value retained for the potential results in the potential being discarded and the branch being converted to a flow source. Conversely, contributing a potential to a branch which already has a value retained for the flow results in the flow being discarded and the branch being converted into a potential source.

Unlike variables, the contributed value for a branch is only valid for the current iteration. If a branch is not contributed to, directly or indirectly, for any particular iteration, and it is not a branch probe, it shall be treated as a flow source with a value of 0.

Example 1:

```
if (closed)
   V(p,n) <+ 0;
```

is equivalent to

```
if (closed)
   V(p,n) <+ 0;
else
   I(p,n) <+ 0;
```

Example 2:

The value retention rules specify that the example below will result in an assignment of 7.0 to the potential source for the unnamed branch between ports `p` and `n`.

```
module value_ret(p, n);
   inout p, n;
   electrical p, n;
   analog begin
      V(p,n) <+ 1.0; // no previously-retained value, 1 is retained
      I(p,n) <+ 2.0; // potential discarded; flow of 2 retained
      V(p,n) <+ 3.0; // flow discarded; potential of 3 retained
      V(p,n) <+ 4.0; // 4 added to previously-retained 3
   end
endmodule
```

Example 3:

The following module defines a current-controlled current source. Because the branch flow I(ps,ns) appears in an expression on the right-hand side, 5.4.2.1 states that this unnamed branch is a probe and its potential is zero (0).

```
module cccs (p, n, ps, ns);
   inout p, n, ps, ns;
   electrical p, n, ps, ns;
   parameter real A = 1.0;
   analog begin
      I(p,n) <+ A * I(ps,ns);
   end
endmodule
```

The value retention rules are used to model switches, as described in 5.6.5.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

5.6.2 Examples

The following examples demonstrate how to formulate models and the correspondence between the behavioral description and the equivalent probe/source model.

5.6.2.1 The four controlled sources

The following example is used with each of the four behavioral statements listed below. Each statement creates a unique controlled source when inserted into this example.

```
module control_source (p, n, ps, ns);
   inout p, n, ps, ns;
   electrical p, n, ps, ns;
   parameter A=1;
   branch (ps,ns) in;
   branch (p,n) out;

    analog begin
   // add behavioral statement here
    end
endmodule
```

The model for a voltage controlled voltage source is

```
V(out) <+ A * V(in);
```

The model for a voltage controlled current source is

```
I(out) <+ A * V(in);
```

The model for a current controlled voltage source is

```
V(out) <+ A * I(in);
```

The model for a current controlled current source is

```
I(out) <+ A * I(in);
```

5.6.3 Resistor and conductor

Figure 5-3 shows the model for a linear conductor.

```
module my_conductor(p,n);
    inout p, n;
    electrical p,n;
    parameter real G=1;
    branch (p,n) cond;

    analog begin
        I(cond) <+ G * V(cond);
    end
endmodule
```

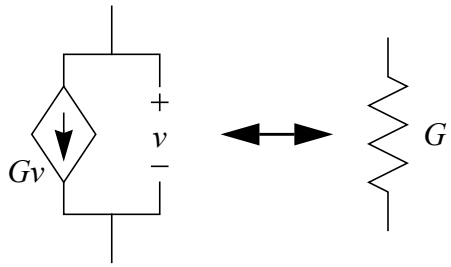


Figure 5-3: Linear conductor model

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The assignment to `I(cond)` makes `cond` a current source branch and `V(cond)` simply accesses the potential probe built into the current source branch.

Figure 5-4 shows the model for a linear resistor.

```
module my_resistor(p,n);
    inout p,n;
    electrical p,n;
    parameter real R=1;
    branch (p,n) res;

    analog begin
        V(res) <+ R * I(res);
    end
endmodule
```

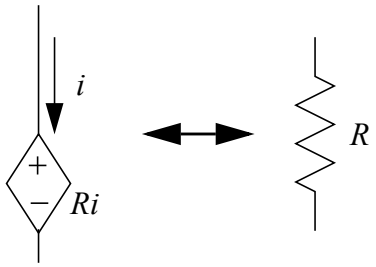


Figure 5-4: Linear resistor model

The assignment to `V(res)` makes `res` a potential source branch and `I(res)` simply accesses the optional flow probe built into the potential source branch.

5.6.4 RLC circuits

A series RLC circuit is formulated by summing the voltage across its three components,

$$v(t) = Ri(t) + L\frac{d}{dt}i(t) + \frac{1}{C}\int_{-\infty}^{t} i(\tau)d\tau$$

which can be defined as

```
V(p, n) <+ R*I(p, n) + L*ddt(I(p, n)) + idt(I(p, n))/C;
```

A parallel RLC circuit is formulated by summing the currents through its three components,

$$i(t) = \frac{v(t)}{R} + C\frac{d}{dt}v(t) + \frac{1}{L}\int_{-\infty}^{t} v(\tau)d\tau$$

which can be defined as

```
I(p, n) <+ V(p, n)/R + C*ddt(V(p, n)) + idt(V(p, n))/L;
```

5.6.5 Switch branches

Contribution to a branch may be switched between potential and a flow during a simulation. This type of branch is useful when modeling ideal switches and mechanical stops. As a result, contribution statements are allowed within conditional statements but are not allowed within event control statements. Note that the contribution statements shall not use *analog operators* when the condition can change during the course of a simulation.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

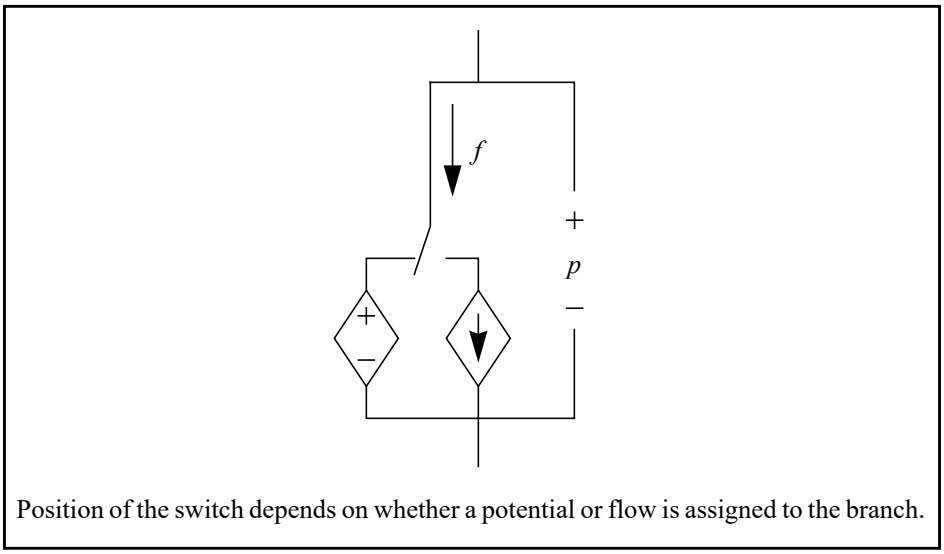


Figure 5-5: Circuit model for a switched source branch

For example, an ideal relay (a controlled switch) can be implemented using a switch branch as follows:

```
module relay (p, n, cp, cn);
   inout p, n, cp, cn;
   electrical p, n, cp, cn;
   branch (p,n) out;
   branch (cp,cn) ctrl;
   parameter real thresh = 0;

   analog begin
      @(cross(V(ctrl) - thresh, 0))
         ; // acts only to resolve threshold crossings

      if (V(ctrl) > thresh)
         V(out) <+ 0;
      else
         I(out) <+ 0; // optional due to value retention
   end
endmodule
```

A discontinuity of order zero (0) is assumed to occur when the branch switches and so it is not necessary to use the **$discontinuity** function with switch branches. Usage of contribution statements inside event control statements is disallowed as these statements may not be executed at every time point.

5.6.6 Implicit Contributions

An important feature of contribution statements is that the value of the target may be expressed in terms of itself. This is referred to as an implicit or fixed-point formulation.

Example:

```
I(diode) <+ is*(limexp((V(diode) - r*I(diode))/$vt) - 1);
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Notice that `I(diode)` is found on both sides of the contribution operator. The underlying implementation of the simulator will find the value of `I(diode)` that equals the sum of the contributions made to it, even if the contributions are a function of `I(diode)` itself.

5.6.7 Indirect branch contribution statements

Direct contribution statements are not the only way that values can be assigned to analog signals. Indirect branch contributions provide an alternative approach that is useful in cases where direct contributions do not behave as needed. One such case is the ideal opamp (or nullor). In this model, the output is driven to the voltage that results in the input voltage being zero. The constitutive equation is

```
vin = 0,      (1)
```

which can be formulated with a contribution statement as

```
V(out) <+ V(out) + V(in);
```

This statement defines the output of the opamp to be a controlled voltage source by assigning to `V(out)` and defines the input to be high impedance by only probing the input voltage. That the desired behavior is achieved can be seen by subtracting `V(out)` from both sides of the contribution operator, resulting in (1). However, this approach does not result in the right tolerances being applied to the equation if out and in have different disciplines. In this situation the tolerances for the equations would come from `V(out)` because it is the target of the contribution, but the final equation does not contain `V(out)`. It would be better if the tolerances for the equation were taken from `V(in)`.

The indirect branch assignment should be used in this situation.

```
V(out): V(in) == 0;
```

which reads “drive `V(out)` so that `V(in) == 0`”. This indicates out is driven with a voltage source and the source voltage needs to be adjusted so that the given equation is satisfied. Any branches referenced in the equation are only probed and not driven. In particular, `V(in)` acts as a voltage probe.

The left-hand side of the equality operator must either be an access function, or **ddt**, **idt** or **idtmod** applied to an access function. The tolerance for the equation is taken from the argument on the left side of the equality operator, in this case `V(in)` as desired. Syntax 5-6 shows the syntax for an indirect assignment statement.

```
indirect_contribution_statement ::=                                   // from A.6.10
        branch_lvalue : indirect_expression == analog_expression ;
indirect_expression ::=                                               // from A.8.3
        branch_probe_function_call
      | port_probe_function_call
      | ddt ( branch_probe_function_call [ , abstol_expression ] )
      | ddt ( port_probe_function_call [ , abstol_expression ] )
      | idt ( branch_probe_function_call [ , analog_expression
            [ , analog_expression [ , abstol_expression ] ] ] )
      | idt ( port_probe_function_call [ , analog_expression [ , analog_expression
            [ , abstol_expression ] ] ] )
      | idtmod ( branch_probe_function_call [ , analog_expression [ , analog_expression
            [ , analog_expression [ , abstol_expression ] ] ] ] )
      | idtmod ( port_probe_function_call [ , analog_expression [ , analog_expression
            [ , analog_expression [ , abstol_expression ] ] ] ] )
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
branch_lvalue ::= branch_probe_function_call                          // from A.8.5
branch_probe_function_call ::=                                        // from A.8.2
      nature_attribute_identifier ( branch_reference )
    | nature_attribute_identifier ( analog_net_reference [ , analog_net_reference ] )
```

Syntax 5-6—Syntax for indirect branch assignment

Indirect branch contributions shall not be used in conditional or looping statements, unless the conditional expression is a constant expression. The constant expression shall not include the **`analysis()`** function with an argument that can result in different return values during a single analysis, such as the `"ic"` or `"nodeset"` arguments.

For example, a complete description of an ideal opamp is:

```
module opamp(out, pin, nin);
   inout out, pin, nin;
   electrical out, pin, nin;
   analog
      V(out):V(pin,nin) == 0;
endmodule
```

5.6.7.1 Multiple indirect contributions

For multiple indirect contribution statements, the targets frequently can be paired with any equation.

For example, the following ordinary differential equation,

$$\frac{dx}{dt} = f(x, y, z)$$

$$\frac{dy}{dt} = g(x, y, z)$$

$$\frac{dz}{dt} = h(x, y, z)$$

can be written as

```
V(x): ddt(V(x)) == f(V(x), V(y), V(z));
V(y): ddt(V(y)) == g(V(x), V(y), V(z));
V(z): ddt(V(z)) == h(V(x), V(y), V(z));
```

or

```
V(y): ddt(V(x)) == f(V(x), V(y), V(z));
V(z): ddt(V(y)) == g(V(x), V(y), V(z));
V(x): ddt(V(z)) == h(V(x), V(y), V(z));
```

or

```
V(z): ddt(V(x)) == f(V(x), V(y), V(z));
V(x): ddt(V(y)) == g(V(x), V(y), V(z));
V(y): ddt(V(z)) == h(V(x), V(y), V(z));
```

without affecting the results.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

5.6.7.2 Indirect and direct contribution

Indirect contribution statements is incompatible with direct contribution statements across the same pair of analog nets (or any of its parallel branches). Once a value is indirectly assigned to a branch, it cannot be contributed to using the branch contribution operator <+.

5.6.8 Contributing hierarchically

5.6.8.1 Contributions to branches between hierarchical nets

Direct contribution statements can contribute to a branch between combinations of local and hierarchical nets.

In these cases, a new unnamed branch is created in the module containing the direct contribution statements.

Example:

```
module source_driver();
   electrical m;
   parameter real vref = 0.0;
   analog begin
      V(m) <+ vref;

      // creates an unnamed voltage source branch of 1.8
      // volts between the net top.drv.x and implicit ground.
      V(top.drv.x) <+ 1.8;

      // creates an unnamed voltage source branch of 1.2
      // volts between nets top.drv.x and top.drv.y
      V(top.drv.x, top.drv.y) <+ 1.2;

      // creates an unnamed voltage source branch of 0.9
      // volts between the local net m and top.drv.y
      V(m, top.drv.y) <+ 0.9;

   end
endmodule
```

The simulator shall check if the contribution produces a solvable set of equations, e.g. no voltage source loops created.

5.6.8.2 Hierarchical direct contributions to branches

Hierarchical direct contributions to named and unnamed branches is allowed provided that the branch is suitable for such contributions. Reasons that a hierarchical branch contribution would not be allowed are:

— Invalid access function used for the contribution to the particular branch
— The hierarchical contribution changes the branch into a switch branch

The simulator shall check if a hierarchical contribution produces a solvable set of equations, e.g. no voltage source loops created.

Example:

```
module source_driver();
   analog begin
      // contributes 1.8 volts to the named branch br_v in top.drv
      V(top.drv.br_v) <+ 1.8;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
        // contributes 1.2 volts to the unnamed branch between
        // nets x and y in top.drv
        V(top.drv.branch(x,y)) <+ 1.2;

        // contributes 1mA to the named branch br_i in top.drv
        I(top.drv.br_i) <+ 1m;

    end
endmodule
```

Hierarchical contributions are not allowed to branches that have been indirectly contributed to (see 5.6.7)

5.7 Analog procedural assignments

Analog procedural assignments are used for modifying analog **integer**, **real**, and **string** variables including array variables. The syntax for procedural assignments shown in Syntax 5-7.

```
analog_procedural_assignment ::= analog_variable_assignment ;                    // from A.6.2
analog_variable_assignment ::=
        scalar_analog_variable_assignment
      | array_analog_variable_assignment
scalar_analog_variable_assignment ::= scalar_analog_variable_lvalue = analog_expression
analog_variable_lvalue ::=                                                       // from A.8.5
        variable_identifier
      | variable_identifier [ analog_expression ] { [ analog_expression ] }
array_analog_variable_assignment ::= array_analog_variable_lvalue = array_analog_variable_rvalue ;
array_analog_variable_rvalue ::=
        array_variable_identifier
      | array_ variable_identifier [ analog_expression ] { [ analog_expression ] }
      | assignment_pattern
```

Syntax 5-7—Syntax for procedural assignments

For scalar variable assignments the following requirements hold;

- The left-hand side of a procedural assignment shall be scalar, either an **integer**, **real**, or **string** identifier or an element of an **integer**, **real** or **string** array.
- The right-hand side expression can be any arbitrary expression constituted from legal operands and operators as described in Clause 4 that evaluates to a scalar.
- A *scalar_analog_variable_assignment* is defined as a variable assignment whose right-hand side *expression* is an *analog_expression* involving analog operators.
- The following semantic restrictions are applicable to the *analog_expression* in the *scalar_analog_-variable_assignment* syntax:
  - Concatenation expressions cannot be used as part of the *analog_expression* (assigning to list of values in the analog context is not allowed).
  - Analog filter functions cannot be used as part of the *analog_expression* syntax if the statement is conditionally executed during simulation.
  - Hierarchical assignment of a variable from another scope/module is not allowed

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Verilog-AMS supports both packed arrays and unpacked arrays of data. The term packed array is used to refer to the dimensions declared before the data identifier name. The term unpacked array is used to refer to the dimensions declared after the data identifier name.

Examples:

```
wire [7:0] c1; // packed array of scalar wire types
real u [7:0]; // unpacked array of real types
```

The requirements of unpacked array variable assignments are a subset of the requirements of IEEE Std 1800 SystemVerilog.

- The array on the LHS of the assignment shall be an array variable, a slice of an array variable or an array parameter (when the default value of the parameter is assigned).
- The arrays on the LHS and the RHS of the assignment must be unpacked.
- Array assignments shall only be done with arrays that are compatible. An array, or a slice of such an array, shall be assignment compatible with any other such array or slice if all the following conditions are satisfied:
  - The element types of source and target shall be equivalent.
  - Every dimension of the source array shall have the same number of elements as the target array. Example:

```
int A[10:1]; // fixed-size array of 10 elements
int B[0:9]; // fixed-size array of 10 elements
int C[24:1]; // fixed-size array of 24 elements
A = B; // ok. Compatible type and same size
A = C; // type check error: different sizes
```

5.8 Analog conditional statements

There are two types of conditional statement allowed in analog behavior:

- if-else-if statements
- case statements

5.8.1 if-else-if statement

The *if-else statement* is used to determine whether a statement is executed or not. The syntax of an *analog conditional statement* is shown in Syntax 5-8. If any of the conditionally-executed statements (*analog_statement_or_null*) contains an *analog operator*, the conditional expression (*analog_expression*) shall be a *analysis_or_constant_expression*. (See the discussion in 4.5.15 regarding restrictions on the usage of analog operators.)

```
analog_conditional_statement ::=                                    // from A.6.6
    if ( analog_expression ) analog_statement_or_null
    { else if ( analog_expression ) analog_statement_or_null }
    [ else analog_statement_or_null ]
```

Syntax 5-8—Syntax of conditional statement

If the expression evaluates to *True* (that is, has a non-zero value), the *analog statements specified as part of the true conditional* shall be executed. If it evaluates to *False* (has a zero value (0)), the *analog statements*

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

specified as part of the true conditional shall not be executed. If analog statements are specified as part of the false condition using **else** and expression is *False*, these statements shall be executed.

Since the numeric value of the **if** expression is tested for being zero (0), certain shortcuts are possible (see 4.2).

5.8.2 Examples

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()** which lacks an **else**.

In the example below, the **else** goes with the inner **if()**, as shown by indentation.

```
if (index > 0)
   if (i > j)
      result = i;
   else  // else applies to preceding if
      result = j;
```

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

```
if (index > 0) begin
   if (i > j)
      result = i;
end
else result = j;
```

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

5.8.3 Case statement

The *case statement* is a multi-way decision statement which tests if an expression matches one of a number of other expressions, and if so, branches accordingly. The case statement has the syntax shown in Syntax 5-9.

---

analog_case_statement ::= *// from A.6.7*
  case (analog_expression) analog_case_item { analog_case_item } **endcase**
| **casex** (analog_expression) analog_case_item { analog_case_item } **endcase**
| **casez** (analog_expression) analog_case_item { analog_case_item } **endcase**

analog_case_item ::=
  analog_expression { , analog_expression } : analog_statement_or_null
| **default** [:] analog_statement_or_null

Syntax 5-9—Syntax for case statement

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

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

The *analog_expression* and the *analog_case_item* expression can be computed at runtime; neither expression is required to be a constant expression.

The *analog_case_item* expressions are evaluated and compared in the exact order in which they are given. During this linear search, if one of the *analog_case_item* expressions matches the *analog_expression* given in parentheses, then the statement associated with that *analog_case_item* is executed. If all comparisons fail, and the default item is given, then the default item statement is executed; otherwise none of the *analog_case_item* statements are executed.

The **casex** and the **casez** versions of the *case* statement are described in 7.3.2 and IEEE Std 1364 Verilog.

5.8.4 Restrictions on conditional statements

Since analog filter functions have to be evaluated at every time point these are restricted to be used inside conditional statements (if-else-if and case) unless the conditional expression is a *constant expression.* Also, for the use of analog filter functions, the conditional statements cannot be conditionally executed (nested conditional statements). Contributions statements are allowed as part of the conditional analog statements (refer 5.6.5 for details on switch branches).

Event control statements (e.g.: **timer**, **cross**) cannot be used inside conditional statements unless the conditional expression is a constant expression.

5.9 Looping statements

There are several types of looping statements: **repeat**, **while**, and **for**. These statements provide a means of controlling the execution of a statement zero (0), one (1), or more times.

The **for** looping statements can be used to describe analog behaviors using analog operators.

The following restrictions are applied to looping statements (**repeat**, **while** and **for**) except for *analog_for* statements, refer 5.9.3

- Analog filter functions are not allowed
- Event control statements are not allowed
- Contribution statements are not allowed

5.9.1 Repeat and while statements

repeat() executes a statement a fixed number of times. Evaluation of the expression decides how many times a statement is executed.

while() executes a statement until an expression becomes *False*. If the expression starts out *False*, the statement is not executed at all.

The *repeat* and *while* expressions shall be evaluated once before the execution of any statement in order to determine the number of times, if any, the statements are executed. The syntax for **repeat()** and **while()** statements is shown in Syntax 5-10.

---

analog_loop_statement ::= // *from A.6.8*
    repeat (analog_expression) analog_statement

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
| while ( analog_expression ) analog_statement
...
```

Syntax 5-10—Syntax for repeat and while statements

5.9.2 For statements

The **for()** statement is a looping construct which controls execution of its associated statement(s) using an index variable. In the case of *analog_for* statement the control mechanism shall consist of *genvar_initialization* and *genvar_expression*s to adhere to the restrictions associated with the use of analog operators. Syntax 5-11 shows the syntax for the looping statements that can be used in analog behavior.

```
analog_loop_statement ::=                                    // from A.6.8
    ...
    | for ( analog_variable_assignment ; analog_expression ; analog_variable_assignment )
        analog_statement
```

Syntax 5-11—Syntax for the for statements

The **for()** statement controls execution of its associated statement(s) by a three-step process:

1) it executes an assignment normally used to initialize an integer which controls the number of loops executed.
2) it evaluates an expression—if the result is zero (0), the *for-loop* exits; otherwise, the *for-loop* executes its associated statement(s) and then performs Step 3.
3) it executes an assignment normally used to modify the value of the loop-control variable and repeats Step 2.

5.9.3 Analog For Statements

The *analog_for* statements are syntactically equivalent to **for()** statements except the associated analog statement can contain analog operators. The *analog_loop_generate_statement* puts the additional restriction upon the procedural assignment and conditional expressions of the *for-loop* to be statically evaluatable. Verilog-AMS HDL provides genvar-derived expressions for this purpose. Syntax 5-12 shows the syntax for the *analog_for* statement.

```
analog_loop_generate_statement ::=                           // from A.4.2
    for ( genvar_initialization ; genvar_expression ; genvar_iteration )
        analog_statement
```

Syntax 5-12—Syntax for the analog_for statements

Examples:

```
module genvarexp(out, dt);
   parameter integer width = 1;
   output out;
   input [1:width] dt;
   electrical out;
   electrical [1:width] dt;
   genvar k;
   real tmp;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    analog begin
       tmp = 0.0;
       for (k = 1; k <= width; k = k + 1) begin
          tmp = tmp + V(dt[k]);
          V(out) <+ ddt(V(dt[k]));
       end
    end
endmodule
```

See the discussion in 4.5.15 regarding other restrictions on the usage of analog operators.

5.10 Analog event control statements

The analog behavior of a component can be controlled using *events*. *events* have the following characteristics:

- events have no time duration
- events can be triggered and detected in different parts of the behavioral model
- events do not block the execution of an **analog** block
- events can be detected using the @ operator
- events do not hold any data
- there can be both digital and analog events

There are three types of analog events, *global events* (5.10.2), *monitored events* (5.10.3), and *named events* (5.10.4). Null arguments are not allowed in analog events. Analog event detection consist of an event expression followed by a procedural statement, as shown in Syntax 5-13.

```
analog_event_control_statement ::= analog_event_control analog_event_statement        // from A.6.5
analog_event_control ::=
      @ hierarchical_event_identifier
    | @ ( analog_event_expression )
analog_event_expression ::=
      expression
    | posedge expression
    | negedge expression
    | hierarchical_event_identifier
    | initial_step [ ( " analysis_identifier "{ , " analysis_identifier " } ) ]
    | final_step [ ( " analysis_identifier " { , " analysis_identifier " } ) ]
    | analog_event_functions
    | analog_event_expression or analog_event_expression
    | analog_event_expression , analog_event_expression
analog_event_functions ::=
      cross ( analog_expression [ , analog_expression_or_null
          [ , analog_expression_or_null [ , analog_expression_or_null [ , analog_expression ] ] ] ] )
    | above ( analog_expression [ , analog_expression_or_null
          [ , analog_expression_or_null [ , analog_expression ] ] ] )
    | timer ( analog_expression [ , analog_expression_or_null
          [ , analog_expression_or_null [ , analog_expression ] ] ] )
    | absdelta ( analog_expression , analog_expression
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
        [ , analog_expression_or_null [ , analog_expression_or_null [ , analog_expression ] ] ] )
analog_event_statement ::=                                                    // from A.6.4
        { attribute_instance } analog_loop_statement
      | { attribute_instance } analog_case_statement
      | { attribute_instance } analog_conditional_statement
      | { attribute_instance } analog_procedural_assignment
      | { attribute_instance } analog_event_seq_block
      | { attribute_instance } analog_system_task_enable
      | { attribute_instance } disable_statement
      | { attribute_instance } event_trigger
      | { attribute_instance } ;
```

Syntax 5-13—Syntax for event detection in analog context

The procedural statements following the event expression is executed whenever the event described by the expression changes. The analog event detection is non-blocking, meaning the execution of the procedural statement is skipped unless the analog event has occurred. The event expression consists of one or more signal names, global events, or monitored events separated by the **or** operator.

The following restrictions applies to the statements that can be specified within an event control block:

- Analog filter functions cannot be used as part of the event control statement. This statement cannot maintain its internal state since it is only executed intermittently when the corresponding event is triggered
- Contribution statements cannot be used inside an event control block because it can generate discontinuity in analog signals
- Nested event control statements are not allowed

The parentheses around the event expression are required.

Analog events can also be detected within digital blocks. Syntax 5-14 shows the usage of analog event control statements inside digital to monitor analog values in the digital context. The usage of *initial_step* and *final_step* analog events are not allowed in the digital context. Refer 7.3.4 for further details on detecting continuous events in a discrete context.

```
event_expression ::=                                                          // from A.6.5
        expression
      | posedge expression
      | negedge expression
      | hierarchical_event_identifier
      | event_expression or event_expression
      | event_expression , event_expression
      | analog_event_functions
      | driver_update expression
      | analog_variable_lvalue
```

Syntax 5-14—Syntax for analog event detection in digital context

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

5.10.1 Event OR operator

The “OR-ing” of events indicates the occurrence of any one of the events specified shall trigger the execution of the procedural statement following the event. The keyword **or** is used as an event OR operator. A comma (,) can be used interchangeably with the keyword **or** to OR event expressions.

Examples:

```
analog begin
   @(initial_step or cross(V(smpl)-2.5,+1)) begin
      vout = (V(in) > 2.5);
   end
   V(out) <+ vout;
end
```

Here, **initial_step** is a global event and **cross()** returns a monitored event. The variable `vout` is set to zero (`0`) or one (`1`) whenever either event occurs.

5.10.2 Global events

Global events are generated by a simulator at various stages of simulation. The user model cannot generate these events. These events are detected by using the name of the global event in an event expression with the `@` operator.

Global events are pre-defined in Verilog-AMS HDL. These events cannot be redefined in a model.

The pre-defined global events are shown in Syntax 5-15.

---

analog_event_expression ::= *// from A.6.5*

```
...
| initial_step [ ( " analysis_identifier "{ , " analysis_identifier " } ) ]
| final_step [ ( " analysis_identifier " { , " analysis_identifier " } ) ]
...
```

Syntax 5-15—Global events

initial_step and **final_step** generate global events on the first and the last point in an analysis respectively. **final_step** will also generate a global event upon the termination of the simulation due to a **$finish()** simulation control task (see 9.7.1). They are useful when performing actions which should only occur at the beginning or the end of an analysis. Both global events can take an optional argument, consisting of an analysis list for the active global event.

Examples:

```
@(initial_step("ac", "dc")) // active for dc and ac only
@(initial_step("tran"))       // active for transient only
```

Table 5-1 describes the return value of **initial_step** and **final_step** for standard analysis types. Each column shows the return-on-event status. One (`1`) represents *Yes* and zero (`0`) represents *No*. A Ver-

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

ilog-AMS HDL simulator can use any or all of these typical analysis types. Additional analysis names can also be used as necessary for specific implementations. (See 4.6.1 for further details.)

Table 5-1—Return value of initial_step and final_step

Analysis[a]	DCOP OP	Sweep[b] d1 d2 dN	TRAN OP p1 pN	AC OP p1 pN	NOISE OP p1 pN
initial_step	1	1 0 0	1 0 0	1 0 0	1 0 0
initial_step("ac")	0	0 0 0	0 0 0	1 0 0	0 0 0
initial_step("noise")	0	0 0 0	0 0 0	0 0 0	1 0 0
initial_step("tran")	0	0 0 0	1 0 0	0 0 0	0 0 0
initial_step("dc")	1	1 0 0	0 0 0	0 0 0	0 0 0
initial_step(*unknown*)	0	0 0 0	0 0 0	0 0 0	0 0 0
final_step	1	0 0 1	0 0 1	0 0 1	0 0 1
final_step("ac")	0	0 0 0	0 0 0	0 0 1	0 0 0
final_step("noise")	0	0 0 0	0 0 0	0 0 0	0 0 1
final_step("tran")	0	0 0 1	0 0 1	0 0 0	0 0 0
final_step("dc")	1	0 0 1	0 0 0	0 0 0	0 0 0
final_step(*unknown*)	0	0 0 0	0 0 0	0 0 0	0 0 0

[a]pX designates frequency/time analysis point X, X = 1 to N; OP designates the Operating Point.
[b]Sweep refers to a dc analysis in which a parameter is swept through multiple values and an operating point analysis is performed for each value. d1 refers to the first point in the sweep; d2 through dN are subsequent points.

The following example measures the bit-error rate of a signal and prints the result at the end of the simulation.

```
module bitErrorRate (in, ref);
   input in, ref;
   electrical in, ref;
   parameter real period=1, thresh=0.5;
   integer bits, errors;

   analog begin
      @(initial_step) begin
         bits = 0;
         errors = 0;
      end

      @(timer(0, period)) begin
         if ((V(in) > thresh) != (V(ref) > thresh))
            errors = errors + 1;
         bits = bits + 1;
      end

      @(final_step)
          $strobe("bit error rate = %f%%", 100.0 * errors / bits );
   end
endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

initial_step and **final_step** take a list of quoted strings as optional arguments. The strings are compared to the name of the analysis being run. If any string matches the name of the current analysis name, the simulator generates an event on the first point and the last point of that particular analysis, respectively.

If no analysis list is specified, the **initial_step** global event is active during the solution of the first point (or initial DC analysis) of every analysis. The **final_step** global event, without an analysis list, is only active during the solution of the last point of every analyses.

5.10.3 Monitored events

Monitored events are detected using event functions with the **@** operator. The triggering of a monitored event is implicit due to change in signals, simulation time, or other runtime conditions.

5.10.3.1 cross function

The **cross()** function is used for generating a monitored analog event to detect threshold crossings in analog signals when the expression crosses zero (0) in the specified direction. In addition, **cross()** controls the timestep to accurately resolve the crossing.

```
analog_event_functions ::=                                          // from A.6.5
    cross ( analog_expression [ , analog_expression_or_null
        [ , analog_expression_or_null [ , analog_expression_or_null [ , analog_expression ] ] ] ] )
  ...
```

*Syntax 5-16—The **cross** analog event function*

The expressions in this syntax have the following meanings:

```
cross ( expr [ , dir [ , time_tol [ , expr_tol [ , enable ] ] ] ] )
```

where *expr* is required, and *dir*, *time_tol*, *expr_tol*, and *enable* are optional. The *expr*, *dir*, and *enable* arguments are specified as *analog_expression*.The tolerances (*time_tol* and *expr_tol*) are specified as *analog_expression* and shall be non-negative.If a value of zero (0.0) is specified, the simulator shall apply a suitable value. The *dir* and *enable* arguments, if specified, shall evaluate to integers. If the tolerances are not specified, then the tool (e.g., the simulator) sets them. If either or both tolerances are defined, then the direction shall also be defined.

If the direction indicator is set to 0 or is not specified, the event and timestep control occur on both positive and negative crossings of the signal. If *dir* is +1, the event and timestep control only occur on rising edge transitions of the signal. If *dir* is –1, the event and timestep control only occur on falling edge transitions of the signal. For any other values of *dir*, the **cross()** function does not generate an event and does not act to control the timestep.

expr_tol and *time_tol* are absolute tolerances and are defined as shown in Figure 5-6. They represent the maximum allowable error between the true crossing point and when the event triggers. The event shall occur

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

after the threshold crossing, and while the signal remains in the box defined by actual crossing and *expr_tol* and *time_tol*.

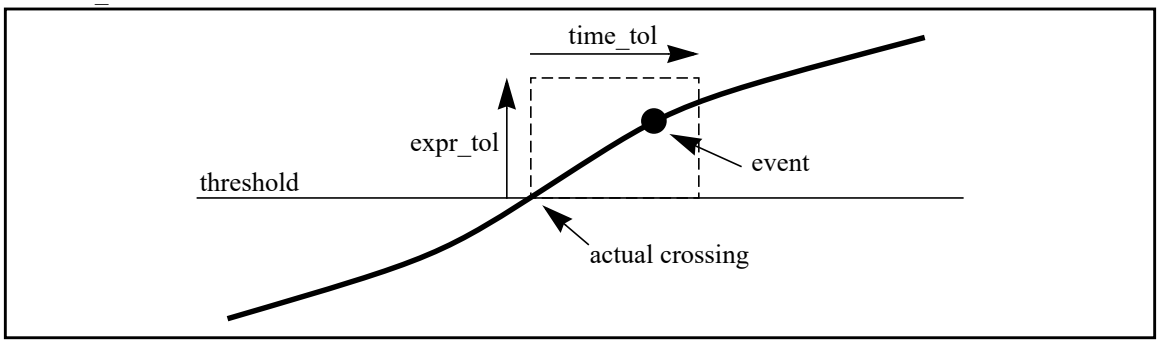


Figure 5-6: Timing of event relative to threshold crossing.

If *expr_tol* is specified, *time_tol* shall also be specified and both tolerances shall be satisfied at the crossing.

Specifying a small *time_tol* ensures a slowly-varying expression triggers an event within a reasonable time from the actual crossing point, and specifying a small *expr_tol* ensures a rapidly-varying expression triggers an event within a reasonable value of the actual crossing point. However, setting either of these tolerances to unrealistically small values can adversely affect the simulator's performance.

Although changes to either *time_tol* or *expr_tol* during the simulation are permitted, care should be taken to ensure that they do not vary from iteration to iteration. Use of constant tolerances, or changing the tolerances via an event statement, is desirable.

The following description of a sample-and-hold illustrates how the **`cross()`** function can be used.

```
module sh (in, out, smpl);
   parameter real thresh = 0.0;
   parameter integer dir = +1 from [-1:+1] exclude 0;
   output out;
   input in, smpl;
   electrical in, out, smpl;
   real state;

   analog begin
      @(cross(V(smpl) - thresh, dir))
         state = V(in);
      V(out) <+ transition(state, 0, 10n);
   end
endmodule
```

If *enable* is specified and nonzero, then **`cross()`** functions as just described. If *enable* argument is specified and it is zero, then **`cross()`** is inactive, meaning that it does not generate an event at threshold crossings and does not act to control the timestep. Thus, there are two ways to disable the cross function, either by specifying *enable* as 0, or giving a value other than –1, 0, or 1 to *dir*. In the following example, the first way is used to allow the sample and hold to be disabled. Notice that in this example, the tolerances are not specified, and so take their default values.

```
module sh (in, out, smpl, en);
   parameter real thresh = 0.0;
   parameter integer dir = +1 from [-1:+1] exclude 0;
   output out;
   input in, smpl, en;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    electrical in, out, smpl;
    real state;

    analog begin
       @(cross(V(smpl) - thresh, dir, , , en === 1'b1))
          state = V(in);
       V(out) <+ transition(state, 0, 10n);
    end
endmodule
```

The **`cross()`** function maintains its internal state and has the same restrictions as analog operators. In particular, it shall not be used inside an **`if`**, **`case`**, **`casex`**, or **`casez`** statement unless the conditional expression is a genvar expression. In addition, **`cross()`** is not allowed in the **`repeat`** and **`while`** iteration statements. It is allowed in the *analog_for* statements.

5.10.3.2 above function

The **`above()`** function is almost identical to the **`cross()`** function, except that it also triggers during initialization or dc analysis. It generates a monitored analog event to detect threshold crossings in analog signals when the expression crosses zero (0) from below. As with the **`cross()`** function, **`above()`** controls the timestep to accurately resolve the crossing during transient analysis.

---

analog_event_functions ::= *// from A.6.5*

```
    ...
    | above ( analog_expression [ , analog_expression_or_null
        [ , analog_expression_or_null [ ,  analog_expression ] ] ] )
    ...
```

---

*Syntax 5-17—The **`above`** analog event function*

The expressions in this syntax have the following meanings:

`above` **`(`** *expr* [**`,`** *time_tol* [**`,`** *expr_tol* [**`,`** *enable*]]] **`)`**

where *expr* is required. The tolerances (*time_tol* and *expr_tol*) are optional, but if specified shall be non-negative.If a value of zero (0.0) is specified, the simulator shall apply a suitable value. The *enable* argument, if specified, shall evaluate to an integer, all other arguments are real expressions. If the tolerances are not specified, then the tool (e.g., the simulator) sets them.

The **`above()`** function can generate an event during initialization. If the expression is positive at the conclusion of the initial condition analysis that precedes a transient analysis, the **`above()`** function shall generate an event. In contrast, the **`cross()`** function can only generate an event after the simulation time has advanced from zero. The **`cross()`** function will not generate events for non-transient analyses, such as ac, dc, or noise analyses of SPICE (see 4.6.1), but the **`above()`** function can. During a dc sweep, the **`above()`** function shall also generate an event when the expression crosses zero from below; however, the step size of the dc sweep is not controlled to accurately resolve the crossing.

The following example uses the **`above()`** function in place of the **`cross()`** function in the description of a simplified version of the sample-and-hold module introduced in the previous section. If the voltage on the `smpl` port is above 2.5V initially (at time=0), then use of the **`above()`** function ensures that the input is sampled and passed to the output when solving for the initial state of the circuit. If the voltage on the `smpl` port never crosses 2.5V in the positive direction, then the **`cross()`** function of the previous example would never trigger, even if the voltage on the `smpl` port is always above 2.5V.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
module sh (in, out, smpl);
   output out;
   input in, smpl;
   electrical in, out, smpl;
   real state;

   analog begin
      @(above(V(smpl) - 2.5))
         state = V(in);
      V(out) <+ transition(state, 0, 10n);
   end
endmodule
```

If *enable* is specified and nonzero, then **above()** functions as just described. If *enable* argument is specified and it is zero, then **above()** is inactive, meaning that it does not generate an event at threshold crossings and does not act to control the timestep.

The **above()** function maintains its internal state and has the same restrictions on its use as the **cross()** function.

5.10.3.3 timer function

The **timer()** function is used to generate analog events to detect specific points in time.

---

analog_event_functions ::= // *from A.6.5*

```
    ...
  | timer ( analog_expression [ , analog_expression_or_null
      [ , analog_expression_or_null [ ,  analog_expression ] ] ] )
```

---

*Syntax 5-18—The **timer** analog event function*

The expressions in this syntax have the following meanings:

```
timer ( start_time [ , period [ , time_tol [ , enable ] ] ] )
```

where *start_time* is required; *period*, *time_tol* and *enable* are optional arguments. The *start_time* and *period* arguments are *analog_expressions*. The tolerance (*time_tol*) is an *analog_expression* and shall be non-negative.If a value of zero (0.0) is specified, the simulator shall apply a suitable value. The *enable* argument, if specified, shall evaluate to an integer.

The **timer()** function schedules an event which occurs at an absolute time (*start_time*). The analog simulator places a time point within *time_tol* of an event. At that time point, the event evaluates to *True*.

If *time_tol* is not specified, the default time point is at, or just beyond, the time of the event.Although changes to *time_tol* during the simulation are permitted, care should be taken to ensure that it does not vary from iteration to iteration. Use of a constant *time_tol*, or changing it via an event statement, is desirable.

If the *period* is specified as greater than zero (0), the timer function schedules subsequent events at multiples of *period*. If the *period* expression evaluates to a value less than or equal to 0.0, the timer shall trigger only once at the specified *start_time* (if the *start_time* is in the future with respect to the current simulation time).

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

If the *start_time* or *period* expressions change value during the evaluation of the **analog** block, the next event will be scheduled based on the latest value of the *start_time* and *period*.

If *enable* is specified and nonzero, then **timer()** functions as just described. If *enable* argument is specified and it is zero, then **timer()** is inactive, meaning that it does not generate events as long as *enable* is zero. However, it will start generating events once *enable* returns to being nonzero as if it had never been disabled.

A pseudo-random bit stream generator is an example how the timer function can be used.

```
module bitStream (out);
   output out;
   electrical out;
   parameter period = 1.0;
   integer x;

   analog begin
      @(timer(0, period))
         x = $random + 0.5;
      V(out) <+ transition( x, 0.0, period/100.0 );
   end
endmodule
```

5.10.3.4 absdelta function

The **absdelta()** event function enables efficient and accurate sampling of analog signals for use in digital behavioral code. The **absdelta()** event function is particularly useful for the conversion of analog-owned variables to real-typed digital-owned variables just as the **above()** event function is particularly useful for the conversion of analog-owned variables to logic-typed digital-owned variables.

According to criteria you set, the simulator can generate an **absdelta** event when an analog expression changes more than a specified amount, a capability that is typically used to discretize analog signals. Use the **absdelta()** function to specify when the simulator generates an **absdelta** event. This function is only allowed in an **initial** or **always** block of a Verilog-AMS module.

---

analog_event_functions ::= *// from A.6.5*
  absdelta (analog_expression **,** analog_expression
    [**,** analog_expression_or_null [**,** analog_expression_or_null [**,** analog_expression]]] **)**
  ...

---

*Syntax 5-19—The **absdelta** analog event function*

The expressions in this syntax have the following meanings:

absdelta (*expr* **,** *delta* [**,** *time_tol* [**,** *expr_tol* [**,** *enable*]]] **)**

where *expr* and *delta* are required; *time_tol*, *expr_tol* and *enable* are optional arguments. The mandatory *expr* and *delta* arguments are specified as an *analog_expression* and *delta* shall be non-negative. Both the tolerances (*time_tol* and *expr_tol*) are specified as an *analog_expression* and shall be non-negative.If a value of zero (0.0) is specified, the simulator shall apply a suitable value. The *enable* argument is specified as an *analog_expression* and shall evaluate to an integer.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

A specified *time_tol* that is smaller than the time precision is ignored and the time precision is used instead. The *expr_tol* argument specifies the largest difference in *expr* that you consider negligible. If the tolerances are not specified, then the tool (e.g., the simulator) sets them.

The **absdelta()** function does not force timesteps in the analog solver - it just observes the expr argument and generates events at the appropriate times to meet the requirements above. To avoid forcing analog timesteps just to determine an event time, **absdelta()** may interpolate the time at which an event occurred if necessary.

The **absdelta()** function generates events for the following times and conditions.

- During initialization or dc sweep analysis.
- When the enable argument changes from zero to non-zero.
- When the expr value changes in absolute value by more than delta, relative to the previous **absdelta()** event (but not when the current time is within *time_tol* of the previous **absdelta()** event). The simulator is allowed to schedule this event at any time between the time corresponding to the interpolated absolute change of (*delta* - *expr_tol*) and the time corresponding to the interpolated absolute change of (*delta* + *expr_tol*) for performance or other reasons.
- When expr changes direction (but not when the amount of the change is less than *expr_tol*).

If *delta* is set to zero, an event is generated every timestep the expression value changes with zero tolerances. *expr_tol* and *time_tol*, if specified, will have no effect and be ignored. No events are generated at times calculated by interpolation. Generating events on every value change may severely impact simulation performance and so setting *delta* to zero should only be done with great care.

expr_tol and *time_tol* can be changed during the simulation. For example, one might change *expr_tol* when *delta* is changed so that the error is proportionate. *time_tol* can be changed to help filter fast changing signals to only enforce minimum time-placed events. The values of *delta*, *expr_tol* and *time_tol* do not directly trigger an **absdelta()** event. Instead, they are used by the function to determine when an event should be generated due to a change in *expr*; this could be during any interpolation of *expr* to satisfy the tolerance requirements.

If *enable* is specified and nonzero, then **absdelta()** functions as just described. If *enable* argument is specified and it is zero, then the **absdelta()** function is inactive, meaning that it does not generate events.

The following example describes an event-driven electrical to **wreal** conversion module where the **absdelta()** function is used to determine when the electrical input signal is converted to a **wreal** output signal.

```
`include "disciplines.vams"
`timescale 1ns / 100ps
module electrical_sampler (e_in, r_out);
   input e_in;
   output r_out;
   electrical e_in;
   wreal r_out;
   parameter real vdelta=0.1 from (0:inf); // voltage delta
   parameter real ttol=1n from (0:1m]; // time tolerance
   parameter real vtol=0.01 from (0:inf); // voltage tolerance
   real sampled;

   assign r_out = sampled;
   always @(absdelta(V(e_in), vdelta, ttol, vtol))
      sampled = V(e_in);
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
endmodule
```

5.10.4 Named events

An identifier declared as an event data type is called a *named event*. A named event is triggered explicitly and is used in an event expression to control the execution of procedural statements in the same manner as event control described in 5.10. Named events can be triggered from always and initial blocks, or from an analog event statement. This allows control over the enabling of multiple actions in other procedures.

An event name shall be declared explicitly before it is used. Syntax 5-20 gives the syntax for declaring events.

---

event_declaration ::= **event** list_of_event_identifiers **;** // *from* A.2.1.3
list_of_event_identifiers ::= event_identifier { dimension } { **,** event_identifier { dimension } // *from* A.2.3
dimension ::= **[** dimension_constant_expression **:** dimension_constant_expression **]** // *from* A.2.5

Syntax 5-20—Syntax for event declaration

A declared event is made to occur by the activation of an event triggering statement with the syntax given in Syntax 5-21. An event is not made to occur by changing the index of an event array in an event control expression.

---

event_trigger ::= // *from* A.6.5
-> hierarchical_event_identifier { **[** expression **]** } **;**

Syntax 5-21—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 signaling the occurrence of an explicit event periodically while the circuit waits for the event to occur.

The following example show how an event (*ana_event*) can be generated from an analog event statement, not only to trigger an event-controlled statement in the **analog** block, but also in an **always** statement.

```
event ana_event;
event dig_event;

initial #10 -> dig_event;
always @(ana_event) $display("Event: ana_event detected in digital");
analog begin
   @(timer(1n)) -> ana_event;
   @(ana_event) $display("Event: ana_event detected in analog");
   @(dig_event) $display("Event: dig_event detected in analog");
end
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

5.10.5 Digital events in analog behavior

To model mixed signal functionality, analog behavior can be made sensitive to digital events, including **posedge** events, **negedge** events, state change events, and named events. In the example above it shows how a digital event (***dig_event***) can be detected within the *analog* context.

5.11 Jump statements

```
jump_statement ::=                                          // from A.6.5
      return [ expression ] ;
    | break ;
    | continue ;
```

Syntax 5-22—Syntax for jump statements

Verilog-AMS HDL provides C-like jump statements **break**, **continue**, and **return**.

```
break               // break out of loop as in C
continue            // skip to end of loop, as in C
return expression   // exit from analog user-defined function
```

The **continue** and **break** statements can only be used in a loop. The **continue** statement jumps to the end of the loop and executes the loop control if present. The **break** statement jumps out of the loop.

The **continue** and **break** statements cannot be used inside an analog for loop. Refer 5.9.3

The **return** statement can only be used from within an analog user-defined function.

In a function returning a value, the **return** statement shall have an expression of the correct type.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

6. Hierarchical structures

6.1 Overview

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

Verilog-AMS HDL provides a mechanism to customize the behavior of embedded modules using parameters. The embedded module parameter default value can be modified through a higher-level module's parameter override or a hierarchy independent `defparam` statement.

To describe a hierarchy of modules, the user provides textual definitions of 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.

6.2 Modules

A module definition shall be enclosed between the keywords **`module`** and **`endmodule`**, as shown in Syntax 6-1. 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 of 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 6.2.2). The identifiers in this list shall be declared in input, output, or inout declaration 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.

A module definition may have multiple **`analog`** blocks. The behavior of multiple **`analog`** blocks shall be defined by assuming that the multiple **`analog`** blocks internally combine into a single **`analog`** block in the order that the **`analog`** blocks appear in the module description. In other words, they are concatenated in the order they appear in the module. Concurrent evaluation of the multiple **`analog`** blocks is implementation dependent as long as the behavior in that case is similar to what would happen if they had been concatenated.

A module can have a description attribute, which shall be used by the simulator when generating help messages for the module.

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

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


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
module_keyword ::= module | macromodule | connectmodule
module_parameter_port_list ::= # ( parameter_declaration { , parameter_declaration } )      // from A.1.3
list_of_ports ::= ( port { , port } )
list_of_port_declarations ::=
        ( port_declaration { , port_declaration } )
port ::=
        [ port_expression ]
      | . port_identifier ( [ port_expression ] )
port_expression ::=
        port_reference
      | { port_reference { , port_reference } }
port_reference ::=
        port_identifier [ [ constant_range_expression ] ]
port_declaration ::=
        {attribute_instance} inout_declaration
      | {attribute_instance} input_declaration
      | {attribute_instance} output_declaration
module_item ::=                                                                              // from A.1.4
        port_declaration ;
      | non_port_module_item
module_or_generate_item ::=
        { attribute_instance } module_or_generate_item_declaration
      | { attribute_instance } local_parameter_declaration ;
      | { attribute_instance } parameter_override
      | { attribute_instance } continuous_assign
      | { attribute_instance } gate_instantiation
      | { attribute_instance } udp_instantiation
      | { attribute_instance } module_instantiation
      | { attribute_instance } initial_construct
      | { attribute_instance } always_construct
      | { attribute_instance } loop_generate_construct
      | { attribute_instance } conditional_generate_construct
      | { attribute_instance } analog_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
      | branch_declaration
      | analog_function_declaration
non_port_module_item ::=
        module_or_generate_item
      | generate_region
      | specify_block
      | { attribute_instance } parameter_declaration ;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    | { attribute_instance } specparam_declaration
    | aliasparam_declaration
parameter_override ::= defparam list_of_defparam_assignments ;
```

Syntax 6-1—Syntax for module

6.2.1 Top-level modules and $root

Top-level modules are modules that are included in the source text, but do not appear in any module instantiation statement, as described in 6.2.2. This applies even if the module instantiation appears in a generate block that is not itself instantiated (see Syntax 6.6).

Verilog-AMS incorporates hierarchical identifier prefix **$root** from IEEE Std 1800 SystemVerilog. The name **$root** is used to unambiguously refer to a top-level instance or to an instance path starting from the root of the instantiation tree. **$root** is the root of the instantiation tree.

For example:

```
$root.A.B // item B within top instance A
$root.A.B.C // item C within instance B within top instance A
```

$root allows explicit access to the top of the instantiation tree. This is useful to disambiguate a local path (which takes precedence) from the rooted path. If **$root** is not specified, a hierarchical path is ambiguous.

For example, A.B.C can mean the local A.B.C or the top-level A.B.C (assuming there is an instance A that contains an instance B at both the top level and in the current module). The ambiguity is resolved by giving priority to the local scope and thereby preventing access to the top-level path. **$root** allows explicit access to the top level in those cases in which the name of the top-level module is insufficient to uniquely identify the path.

6.2.2 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 does 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.

Syntax 6-2 gives the syntax for specifying instantiations of modules.

```
module_instantiation ::=                                          // from A.4.1
        module_or_paramset_identifier [ parameter_value_assignment ]
            module_instance { , module_instance } ;
parameter_value_assignment ::= # ( list_of_parameter_assignments )
list_of_parameter_assignments ::=
        ordered_parameter_assignment { , ordered_parameter_assignment }
      | named_parameter_assignment { , named_parameter_assignment }
ordered_parameter_assignment ::= expression
named_parameter_assignment ::=
        . parameter_identifier ( [ mintypmax_expression ] )
      | . system_parameter_identifier ( [ constant_expression ] )
module_instance ::= name_of_module_instance ( [ list_of_port_connections ] )
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
name_of_module_instance ::= module_instance_identifier [ range ]
list_of_port_connections ::=
        ordered_port_connection { , ordered_port_connection }
      | named_port_connection { , named_port_connection }
ordered_port_connection ::= { attribute_instance } [ expression ]
named_port_connection ::= { attribute_instance } . port_identifier ( [ expression ] )
```

Syntax 6-2—Syntax for module instantiation

The instantiations of modules can contain a range specification. This allows an array of instances to be created.

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

The list of module connections shall be provided only for modules defined with ports. The parentheses, however, are always required. When a list of 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 6.5 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 if it is a digital 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 ()`]

The example below illustrates a comparator and an integrator (lower-level modules) which are instantiated in sigma-delta A/D converter module (the higher-level module).

```
module comparator(cout, inp, inm);
   output cout;
   input inp, inm;
   electrical cout, inp, inm;
   parameter real td = 1n, tr = 1n, tf = 1n;
   real vcout;
   analog begin
      @(cross(V(inp) - V(inm), 0))
         vcout = ((V(inp) > V(inm)) ? 1 : 0);
      V(cout) <+ transition(vcout, td, tr, tf);
   end
endmodule

module integrator(out, in);
   output out;
   input in;
   electrical in, out;
   parameter real gain = 1.0;
   parameter real ic = 0.0;
   analog begin
      V(out) <+ gain*idt(V(in), ic);
   end
endmodule

module sigmadelta(out, aref, in);
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    output out;
    input aref, in;
    electrical out, aref, in;
    electrical gnd; ground gnd;

    comparator C1(.cout(aa0), .inp(in), .inm(aa2));
    integrator #(1.0) I1(.out(aa1), .in(aa0));
    comparator C2(out, aa1, gnd);
    d2a #(.width(1)) D1(aa2, aref, out);   // a D/A converter
endmodule
```

The comparator instance `C1` and the integrator instance `I1` in Figure 6-1 use named port connections, whereas the comparator instance `C2` and the `d2a` (not described here) instance `D1` use ordered port connections. Note the integrator instance `I1` overrides gain parameter positionally, whereas the `d2a` instance `D1` overrides width parameter by named association.

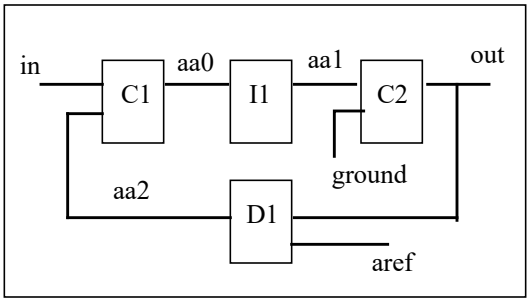


Figure 6-1: Comparator and integrator modules

6.3 Overriding module parameter values

When one module instantiates another module, it can alter the values of any parameters declared within the instantiated module, as well as the values of various system parameters that are implicitly declared for all modules. There are three ways to alter parameter values: the *defparam statement*, which allows assignment to parameters using their hierarchical names, the *module instance parameter value assignment*, which allows values to be assigned inline during module instantiation, and the *paramset*, which is described in 6.4. If a defparam assignment conflicts with a module instance parameter, the parameter in the module shall take the value specified by the defparam. If a defparam assignment conflicts with a paramset instance parameter, the paramset selection will occur with the parameter value specified by the defparam.

The module instance parameter value assignment comes in two forms, by ordered list or by name. The first form is *module instance parameter value assignment by order*, which allows values to be assigned in-line during module instantiation in the order of their declaration. The second form is *module instance parameter value assignment by name*, which allows values to be assigned in-line during module instantiation by explicitly associating parameter names with the overriding values.

6.3.1 Defparam statement

Using the defparam statement, parameter values can be changed in any module instance throughout the design using the hierarchical name of the parameter. See 6.7 for details about hierarchical names.

However, a defparam statement in a hierarchy in or under a generate block instance (see 6.6) or an array of instances (see 6.2.2) shall not change a parameter value outside that hierarchy. Additionally, a defparam statement is not allowed in a hierarchy in or under a paramset instance (see 6.4).

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Each instantiation of a generate block is considered to be a separate hierarchy scope. Therefore, this rule implies that a defparam statement in a generate block may not target a parameter in another instantiation of the same generate block, even when the other instantiation is created by the same loop generate construct.

For example, the following code is not allowed:

```
genvar i;

generate
   for (i = 0; i < 8; i = i + 1) begin : somename
      flop my_flop(in[i], in1[i], out1[i]);
      defparam somename[i+1].my_flop.xyz = i ;
   end
endgenerate
```

Similarly, a defparam statement in one instance of an array of instances may not target a parameter in another instance of the array.

The expression on the right-hand side of a defparam assignments shall be a constant expression involving only constant numbers and references to parameters. The referenced parameters (on the right-hand side of a 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. Its syntax is shown in Syntax 6-3.

---

parameter_override ::= **defparam** list_of_defparam_assignments **;** *// from A.1.4*

list_of_defparam_assignments ::= defparam_assignment { **,** defparam_assignment } *// from A.2.3*

defparam_assignment ::= hierarchical_parameter_identifier **=** constant_mintypmax_expression *// from A.2.4*

Syntax 6-3—Syntax for defparam

Examples:

```
module tgate ();
   electrical io1,io2,control,control_bar;
   mosn m1 (io1, io2, control);
   mosp m2 (io1, io2, control_bar);
endmodule

module mosp (drain,gate,source);
   inout drain, gate, source;
   electrical drain, gate, source;
   parameter gate_length = 0.3e-6,
             gate_width  = 4.0e-6;

   spice_pmos #(.l(gate_length),.w(gate_width)) p (drain, gate, source);
endmodule

module mosn (drain,gate,source);
   inout drain, gate, source;
   electrical drain, gate, source;
   parameter gate_length = 0.3e-6,
             gate_width  = 4.0e-6;
   spice_nmos #(.l(gate_length),.w(gate_width)) n (drain, gate, source);
endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
module annotate ();
   defparam
      tgate.m1.gate_width = 5e-6,
      tgate.m2.gate_width = 10e-6;
endmodule
```

6.3.2 Module instance parameter value assignment by order

The order of the assignments in module instance parameter value assignment shall follow the order of declaration of the parameters within the module. Local parameter declarations are not considered when assigning by order; parameter alias declarations are also skipped. It is not necessary to assign values to all of the parameters within a module when using this method. However, the left-most parameter assignment(s) can not be skipped. Therefore, to assign values to a subset of the parameters declared within a module, the declarations of the parameters which make up this subset shall precede the declarations of the remaining (optional) parameters. An alternative is to assign values to all of the parameters, but use the default value (the same value assigned in the declaration of the parameter within the module definition) for those parameters which do not need new values.

Consider the following example, where the parameters within module instance `weakp` are changed during instantiation.

```
module m ();
   electrical clk;
   electrical out_a, in_a;
   electrical out_b, in_b;

   // create an instance and set parameters
   mosp #(2e-6,1e-6) weakp (out_a, in_a, clk);

   // create an instance leaving default values
   mosp plainp (out_b, in_b, clk);
endmodule
```

6.3.3 Module instance parameter value assignment by name

Parameter assignment by name consists of explicitly linking the parameter name and its value. The name of the parameter shall be the name specified in the instantiated module. It is not necessary to assign values to all the parameters within a module when using this method. Only those parameters which are assigned new values need to be specified.

The parameter expression is optional so 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.

In the following example of instantiating a voltage-controlled oscillator, the parameters are specified on a named-association basis much as they are for ports.

```
module n (lo_out, rf_in);
   output lo_out;
   input rf_in;
   electrical lo_out, rf_in;

   //create an instance and set parameters
   vco #(.centerFreq(5000), .convGain(1000)) vco1(lo_out, rf_in);
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
endmodule
```

Here, the name of the instantiated `vco` module is `vco1`. The `centerFreq` parameter is passed a value of 5000 and the `convGain` parameter is passed a value of 1000. The positional assignment mechanism for ports assigns `lo_out` as the first port and `rf_in` as the second port of `vco1`.

6.3.4 Parameter dependence

A parameter (for example, `gate_cap`) can be defined with an expression containing another parameter (for example, `gate_width` or `gate_length`). Since `gate_cap` depends on the value of `gate_width` and `gate_length`, a modification of `gate_width` or `gate_length` changes the value of `gate_cap`.

In the following parameter declaration, an update of `gate_width`, whether by a defparam statement or in an instantiation statement for the module which defined these parameters, automatically updates `gate_cap`.

```
parameter
    gate_width  = 0.3e-6,
    gate_length = 4.0e-6,
    gate_cap    = gate_length * gate_width * `COX;
```

6.3.5 Detecting parameter overrides

In some cases, it is important to be able to determine whether a parameter value was obtained from the default value in its declaration statement or if that value was overridden. In such a case, the **`$param_given()`** function described in 9.19 can be used.

6.3.6 Hierarchical system parameters

In addition to the parameters explicitly declared in a module's header, there are six system parameters that are implicitly declared for every module: **`$mfactor`**, **`$xposition`**, **`$yposition`**, **`$angle`**, **`$hflip`**, and **`$vflip`**. The values of these parameters may be accessed in a module or paramset using these names, as described in 9.18. The value of these parameters may be overridden using the defparam statement, module instance parameter value assignment by name, or a paramset; in all three methods, the system parameter identifier is prefixed by a period (**`.`**), just as for explicitly-declared parameters.

If an instance of a module has a non-unity value of **`$mfactor`**, then the following rules are applied automatically by the simulator:

- All contributions to a branch flow quantity in the **`analog`** block shall be multiplied by **`$mfactor`**
- The value returned by any branch flow probe in the **`analog`** block, including those used in indirect assignments, shall be divided by **`$mfactor`**
- Contributions to a branch flow quantity using the noise functions of 4.6.4 (**`white_noise`**, **`flicker_noise`**, and **`noise_table`**) shall have the noise power multiplied by **`$mfactor`**
- Contributions to a branch potential quantity using the noise functions of 4.6.4 shall have the noise power divided by **`$mfactor`**
- The module's value of **`$mfactor`** is also propagated to any module instantiated by the original module, according to the rules found in 9.18.

Application of these rules guarantees that the behavior of the module in the design is identical to the behavior of a quantity **`$mfactor`** of identical modules with the same connections; however, the simulator only has to evaluate the module once.

Verilog-AMS does not provide a method to disable the automatic **`$mfactor`** scaling. The simulator shall issue a warning if it detects a misuse of the **`$mfactor`** in a manner that would result in double-scaling.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The two resistor modules below show ways that the **$mfactor** might be used in a module. The first example, badres, misuses the **$mfactor** such that the contributed current would be multiplied by **$mfactor** twice, once by the explicit multiplication and once by the automatic scaling rule. The simulator will generate an error for this module.

```
module badres(a, b);
   inout a, b;
   electrical a, b;
   parameter real r = 1.0 from (0:inf);
   analog begin
      I(a,b) <+ V(a,b) / r * $mfactor; // ERROR
   end
endmodule
```

In this second example, parares, **$mfactor** is used only in the conditional expression and does not scale the output. No error will be generated for this module. In cases where the effective resistance r/**$mfactor** would be too small, the resistance is simply shorted out, and the simulator may collapse the node to reduce the size of the system of equations.

```
module parares(a, b);
   inout a, b;
   electrical a, b;
   parameter real r = 1.0 from (0:inf);
   analog begin
      if (r / $mfactor < 1.0e-3)
         V(a,b) <+ 0.0;
      else
         I(a,b) <+ V(a,b) / r;
   end
endmodule
```

The values of the five geometrical system parameters, **$xposition**, **$yposition**, **$angle**, **$hflip**, and **$vflip**, do not have any automatic effect on the simulation. The paramset or module may use these values to compute geometric layout-dependent effects, as shown in the following example.

In the next example, it is assumed that a top-level module named processinfo contains values for the properties of polysilicon resistors in the manufacturing process, including the nominal value processinfo.rho and the gradients processinfo.drho_dx and processinfo.drho_dy, in the x and y direction respectively.

```
module polyres(a, b);
   inout a, b;
   electrical a, b;
   parameter real length = 1u from (0:inf);
   parameter real width = 1u from (0:inf);
   real rho, reff;
   analog begin
      rho = processinfo.rho
               + $xposition * processinfo.drho_dx
               + $yposition * processinfo.drho_dy;
      reff = rho * length / width;
      I(a,b) <+ V(a,b) / reff;
   end
endmodule
```

The resistor just defined could be instantiated in the following manner so as to cancel out the process gradients:

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
module matchedres(a, b);
   inout a, b;
   electrical a, b;
   parameter real length = 1u from (0:inf);
   parameter real width = 1u from (0:inf);
   polyres #(.width(width/4.0), .length(length),
             .$xposition(-1u),  .$yposition(-1u))  R1 (a, b);
   polyres #(.width(width/4.0), .length(length),
             .$xposition(+1u),  .$yposition(-1u))  R2 (a, b);
   polyres #(.width(width/4.0), .length(length),
             .$xposition(-1u),  .$yposition(+1u))  R3 (a, b);
   polyres #(.width(width/4.0), .length(length),
             .$xposition(+1u),  .$yposition(+1u))  R4 (a, b);
endmodule
```

Unfortunately, if the module `matchedres` is itself instantiated off-center, then the process gradients will not be canceled.

6.4 Paramsets

A *paramset definition* is enclosed between the keywords **paramset** and **endparamset**, as shown in Syntax 6-4. The first identifier following the keyword **paramset** is the name of the paramset being defined. The second identifier will usually be the name of a module with which the paramset is associated. The second identifier may instead be the name of a second paramset. A chain of paramsets may be defined in this way, but the last paramset in the chain shall reference a module.

---

paramset_declaration ::= *// from A.1.9*
  { attribute_instance } **paramset** paramset_identifier module_or_paramset_identifier ;
    paramset_item_declaration { paramset_item_declaration }
    paramset_statement { paramset_statement }
  endparamset

paramset_item_declaration ::=
  { attribute_instance } parameter_declaration ;
| { attribute_instance } local_parameter_declaration ;
| aliasparam_declaration
| { attribute_instance } integer_declaration
| { attribute_instance } real_declaration

paramset_statement ::=
  .module_parameter_identifier = paramset_constant_expression ;
| .module_output_variable_identifier = paramset_constant_expression ;
| .system_parameter_identifier = paramset_constant_expression ;
| analog_function_statement

paramset_constant_expression ::=
  constant_primary
| hierarchical_parameter_identifier
| unary_operator { attribute_instance } constant_primary
| paramset_constant_expression binary_operator { attribute_instance } paramset_constant_expression
| paramset_constant_expression ? { attribute_instance } paramset_constant_expression : { attribute_instance } paramset_constant_expression

---

Syntax 6-4—Syntax for paramset

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The paramset itself contains no behavioral code; all of the behavior is determined by the associated module. The restrictions on statements in the paramset are described in 6.4.1.

The paramset provides a convenient way to collect parameter values for a particular module, such that an instance need only provide overrides for a smaller number of parameters. A simulator can use this information to optimize data storage for the instances: multiple instances may share a paramset, and the simulator can share storage of parameters of the underlying module. The shared storage of paramsets makes them similar to the SPICE model card. Also like the SPICE model card, paramsets may be overloaded, as described in 6.4.2.

The only restriction on the associated module for a paramset is that it does not contain a defparam statement in or under its hierarchy, see 6.3.1.

A paramset can have a description attribute, which shall be used by the simulator when generating help messages for the paramset.

The following example shows how one might convert a SPICE model card into a Verilog-AMS paramset. Suppose one has the following lines in a SPICE netlist:

```
m1 d1 g 0 0 nch l=1u w=10u
m2 d2 g 0 0 nch l=1u w=5u
.model nch nmos (level=3 kp=5e-5 tox=3e-8 u0=650 nsub=1.3e17
                 + vmax=0 tpg=1 nfs=0.8e12)
```

These lines could be written in Verilog-AMS as follows, assuming that `nmos3` is a behavioral module that contains the same equations as the SPICE primitive.

```
nch #(.l(1u), .w(10u)) m1 (.d(d1), .g(g), .s(0), .b(0));
nch #(.l(1u), .w(5u))  m2 (.d(d2), .g(g), .s(0), .b(0));

paramset nch nmos3; // default paramset
   parameter real l=1u from [0.25u:inf);
   parameter real w=1u from [0.2u:inf);
   .l=l; .w=w; .ad=w*0.5u; .as=w*0.5u;
   .kp=5e-5; .tox=3e-8; .u0=650; .nsub=1.3e17;
   .vmax=0; .tpg=1; .nfs=0.8e12;
endparamset
```

Note that the paramset has only two parameters, `l` and `w`; an instance of the paramset that attempts to override any of the other parameters of the underlying module `nmos3` would generate an error. Analog simulators are expected to optimize the storage of paramset values in a manner similar to the way SPICE optimizes model parameter storage.

6.4.1 Paramset statements

The restrictions on statements or assignments allowed in a paramset are similar to the restrictions for analog functions. Specifically, a paramset:

- can use any statements available for conditional execution (see 5.2);
- shall not use access functions;
- shall not use contribution statements or event control statements; and
- shall not use named blocks.

The special syntax

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
.module_parameter_identifier = paramset_constant_expression ;
```

is used to assign values to the parameters of the associated module. The expression on the right-hand side can be composed of numbers, parameters and hierarchical out-of-module references to local parameters of a different module. Hierarchical out-of-module references to non-local parameters of a different module is disallowed. The expression may also use the **$arandom** function from 9.13.1 and the **$rdist_** functions from 9.13.2, so long as the arguments to these functions are constant.

Paramset statements may assign values to variables declared in the paramset; the values need not be constant expressions. However, these variables shall not be used to assign values to the module's parameters. Paramset variables may be used to provide output variables for instances that use the paramset; see 6.4.3.

The following example shows how to use the **$rdist_normal** function of 9.13.2 to model two kinds of statistical variation.

```
module semicoCMOS ();
   localparam real tox = 3e-8;
   localparam real dtox_g = $rdist_normal(1,0,1n,"global");
   localparam real dtox_mm = $rdist_normal(2,0,5n,"instance");
endmodule

paramset nch nmos3; // mismatch paramset
   parameter real l=1u from [0.25u:inf);
   parameter real w=1u from [0.2u:inf);
   parameter integer mm=0 from (0:1];
   .l=l; .w=w; .ad=w*0.5u; .as=w*0.5u;
   .kp=5e-5; .u0=650; .nsub=1.3e17;
   .vmax=0; .tpg=1; .nfs=0.8e12;
   .tox = semicoCMOS.tox + semicoCMOS.dtox_g + semicoCMOS.dtox_mm;
endparamset

module top ();
   electrical d1, d2, g, vdd, gnd;
   ground gnd;
   nch #(.l(1u), .w(5u), .mm(1)) m1(.d(d1), .g(g), .s(gnd), .b(gnd));
   nch #(.l(1u), .w(5u), .mm(1)) m2(.d(d2), .g(g), .s(gnd), .b(gnd));
   resistor #(.r(1k)) R1 (vdd, d1);
   resistor #(.r(1k)) R2 (vdd, d2);
   vsine #(.dc(2.5)) Vdd (vdd, gnd);
   vsine #(.dc(0), .ampl(1.0), .offset(1.5), .freq(1k)) Vg (g, gnd);
endmodule
```

Because the local parameter `dtox_mm` is obtained from **$rdist_normal** with the string `"instance"`, the instances `m1` and `m2` will get different values of `tox`. Though the local variation has a smaller standard deviation than the global variation, only the local variation will affect the differential voltage between nodes `d1` and `d2`.

6.4.2 Paramset overloading

Paramset identifiers need not be unique: multiple paramsets can be declared using the same *paramset_identifier*, and they may refer to different modules. During elaboration, the simulator shall choose an appropriate paramset from the set that shares a given name for every instance that references that name.

When choosing an appropriate paramset, the following rules shall be enforced:

— All parameters overridden on the instance shall be parameters of the paramset

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

— The parameters of the paramset, with overrides and defaults, shall be all within the allowed ranges specified in the paramset parameter declaration.
— The local parameters of the paramset, computed from parameters, shall be within the allowed ranges specified in the paramset.
— The underlying module shall have a port declared for each port connected in the instance line.

The rules above may not be sufficient for the simulator to pick a unique paramset, in which case the following rules shall be applied in order until a unique paramset has been selected:

— The paramset with the fewest number of un-overridden parameters shall be selected.
— The paramset with the greatest number of local parameters with specified ranges shall be selected.
— The paramset with the fewest ports not connected in the instance line shall be selected.

It shall be an error if there are still more than one applicable paramset for an instance after application of these rules.

If a paramset assigns a value to a module parameter and this value is outside the range specified for that module parameter, it shall be an error. The simulator shall consider only the ranges of the paramset's own parameters when choosing a paramset.

The following example illustrates some of the rules for paramset selection. Consider a design that includes the two paramsets defined previously (in the examples of 6.4 and 6.4.1) as well as the following paramsets:

```
paramset nch nmos3; // short-channel paramset
   parameter real l=0.25u from [0.25u:1u);
   parameter real w=1u from [0.2u:inf);
   parameter real ad=0.5*w from (0:inf);
   parameter real as=0.5*w from (0:inf);
   .l=l; .w=w; .ad=ad; .as=as;
   .kp=5e-5; .tox=3e-8; .u0=650; .nsub=1.3e17;
   .vmax=0; .tpg=1; .nfs=0.8e12;
endparamset

paramset nch nmos3; // long-channel paramset
   parameter real l=1u from [1u:inf);
   parameter real w=1u from [0.2u:inf);
   parameter real ad=0.4*w from (0:inf);
   parameter real as=0.4*w from (0:inf);
   .l=l; .w=w; .ad=ad; .as=as;
   .kp=5e-5; .tox=3e-8; .u0=640; .nsub=1.3e17;
   .vmax=0; .tpg=1; .nfs=0.7e12;
endparamset
```

The following instances might exist in the design:

```
nch #(.l(1u), .w(5u), .mm(1)) m1(.d(d1), .g(g), .s(0), .b(0));
nch #(.l(1u), .w(5u), .mm(1)) m2(.d(d2), .g(g), .s(0), .b(0));
nch #(.l(1u), .w(10u)) m3 (.d(g), .g(g), .s(0), .b(0));
nch #(.l(3u), .w(5u), .ad(1.2p), .as(1.3p))
         m4 (.d(d1), .g(g2), .s(d2), .b(0));
```

The instances `m1` and `m2` will use the mismatch paramset from 6.4.1, because it is the only one for which `mm` is a parameter. The instance `m4` will use the long-channel paramset defined in this example, because while the short-channel paramset also has `ad` and `as` as parameters, the length of `m4` is only allowed by the range for `l` in the long-channel paramset. The instance `m3` will use the default paramset defined in 6.4; it cannot use the mismatch paramset because the default value of `mm` for that paramset is not allowed by the range, and

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

it discriminates against the long-channel paramset because that paramset would have two un-overridden parameters.

6.4.3 Paramset output variables

As with modules, integer or real variables in the paramset that are declared with descriptions are considered output variables; see 3.2.1. A few special rules apply to paramset output variables and output variables of modules referenced by a paramset:

— If a paramset output variable has the same name as an output variable of the module, the value of the paramset output variable is the value reported for any instance that uses the paramset.

— If a paramset variable without a description has the same name as an output variable of the module, the module output variable of that name shall not be available for instances that use the paramset.

— A paramset output variable's value may be computed from values of any output parameters of the module by using the special syntax

.module_output_variable_identifier

The following example declares an output variable `ft` for instances of the paramset `smnpn`.The module is assumed to have output variables named `gm`, `cpi`, and `cmu`. If the module `npn` had an output variable named `ft`, the paramset's output variable would replace it.

```
paramset smnpn npn; // small npn paramset
   (*desc="cut-off frequency"*) real ft;
   .is=2.0e-17; .bf=120.0; .br=10; rb=145; .rc=75; .re=12;
   .cje=2.0e-14; .vje=0.9; .mje=0.4;
   .cjc=3.0e-14; .vjc=0.6; .mjc=0.3; .xcjc=0.2;
   ft = .gm/(`M_TWO_PI*(.cpi + .cmu));
endparamset
```

6.5 Ports

Ports provide a means of interconnecting instances of modules. For example, if a module `A` instantiates module `B`, the ports of module `B` are associated with either the ports or the internal nets of module `A`.

6.5.1 Port definition

The syntax for a port association is shown in Syntax 6-5.

```
port ::=                                                    // from A.1.3
      [ port_expression ]
    | . port_identifier ( [ port_expression ] )
port_expression ::=
      port_reference
    | { port_reference { , port_reference } }
```

Syntax 6-5—Syntax for port

The port expression in the port definition can be one of the following:

— a simple net identifier

— a scalar member of a vector net or port declared within the module

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

— a sub-range of a vector net or port declared within the module
— a vector net formed as a result of the concatenation operator

The port expression is optional because ports can be defined which do not connect to anything internal to the module.

6.5.2 Port declarations

The type and direction of each port listed in the module definition's list of ports are declared in the body of the module.

6.5.2.1 Port type

The type of a port is declared by giving its discipline, as shown in Syntax 6-6. If the type of a port is not declared, the port can only be used in a structural description. (It can be passed to instances of modules, but cannot be accessed in a behavioral description.)

```
net_declaration ::=                                                    // from A.2.1.3
        ...
      | discipline_identifier [ range ] list_of_net_identifiers ;
      | discipline_identifier [ range ] list_of_net_decl_assignments ;
        ...
range ::= [ msb_constant_expression : lsb_constant_expression ]        // from A.2.5
list_of_net_decl_assignments ::= net_decl_assignment { , net_decl_assignment }   // from A.2.3
list_of_net_identifiers ::= ams_net_identifier { , ams_net_identifier }
net_decl_assignment ::= ams_net_identifier = expression                // from A.2.4
```

Syntax 6-6—Syntax for port type declarations

6.5.2.2 Port direction

Each port listed in the list of ports for the module definition shall be declared in the body of the module as an **`input`**, **`output`**, or **`inout`** (bidirectional). This is in addition to any other declaration for a particular port—for example, a *net_discipline*, **`reg`**, or **`wire`**. The syntax for port declarations is shown in Syntax 6-7.

```
inout_declaration ::=                                                  // from A.2.1.2
        inout [ discipline_identifier ] [ net_type | wreal ] [ signed ] [ range ] list_of_port_identifiers
input_declaration ::=
        input [ discipline_identifier ] [ net_type | wreal ] [ signed ] [ range ] list_of_port_identifiers
output_declaration ::=
        output [ discipline_identifier ] [ net_type | wreal ] [ signed ] [ range ] list_of_port_identifiers
      | output [ discipline_identifier ] reg [ signed ] [ range ] list_of_variable_port_identifiers
      | output output_variable_type  list_of_variable_port_identifiers
```

Syntax 6-7—Syntax for port direction declarations

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

A port can be declared in both a *port type* declaration and a *port direction* declaration. If a port is declared as a vector, the range specification between the two declarations of a port shall be identical. For example:

```
input [0:3] in;
electrical [0:3] in; // valid, MSB and LSB in both the port type and port
                     // direction declaration evaluate to the same value

input [0:3] in;
electrical [0:4-1] in;  // valid, MSB and LSB in both the port type and port
                        // direction declaration evaluate to the same value

input [3:0] in;
electrical [0:3] in; // error, MSB and LSB in the port type declaration does
                     // not evaluate to the same value as the port direction
                     // declaration.
```

Implementations can limit the maximum number of ports in a module definition, but this shall be a minimum of 256 ports per implementation.

6.5.3 Real valued ports

Verilog-AMS HDL supports ports which are declared to be real-valued and have a discrete-time discipline. This is done using the net type **`wreal`** (defined in 3.7). There can be a maximum of one driver of a real-valued net.

Examples:

```
module top();
   wreal stim;
   reg clk;
   wire [1:8] out;

   testbench tb1 (stim, clk);
   a2d dut (out, stim, clk);

   initial clk = 0;
   always #1 clk = ~clk;
endmodule

module testbench(wout, clk);
   output wout;
   input clk;
   real out;
   wire clk;
   wreal wout;

   assign wout = out;

   always @(posedge clk) begin
      out = out + $abstime;
   end
endmodule

module a2d(dout, in, clk);
   output [1:8] dout;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   input in, clk;
   wreal in;
   wire clk;
   reg [1:8] dout;
   real residue;
   integer i;

   always @(negedge clk) begin
      residue = in;
      for (i = 8; i >= 1; i = i - 1) begin
         if (residue > 0.5) begin
            dout[i] = 1'b1;
            residue = residue - 0.5;
         end
         else begin
            dout[i] = 1'b0;
         end
         residue = residue*2;
      end
   end
endmodule
```

6.5.4 Connecting module ports by ordered list

One way to connect the ports listed in a module instantiation with the ports defined by the instantiated module is via an ordered list—that is, the ports listed for the module instance shall be in the same order as the ports listed in the module definition.

Examples:

```
module adc4 (out, rem, in);
   output [3:0] out; output rem;
   input in;
   electrical [3:0] out;
   electrical in, rem, rem_chain;

   adc2 hi2 (out[3:2], rem_chain, in);
   adc2 lo2 (out[1:0], rem, rem_chain);
endmodule

module adc2 (out, remainder, in);
   output [1:0] out; output remainder;
   input in;
   electrical [1:0] out;
   electrical in, remainder, r;

   adc hi1 (out[1], r, in);
   adc lo1 (out[0], remainder, r);
endmodule

module adc (out, remainder, in);
   output out, remainder;
   input in;
   electrical out, in, remainder;
   integer d;

   analog begin
      d = (V(in) > 0.5);
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
      V(out) <+ transition(d);
      V(remainder) <+ 2.0 * V(in);
      if (d)
         V(remainder) <+ -1.0;
   end
endmodule
```

6.5.5 Connecting module ports by name

The second way to connect module ports consists of explicitly linking the two names for each side of the connection — specify the name used in the module definition, followed by the name used in the instantiating module. This compound name is then placed in the list of module connections.

The following rules apply:

- The name of port shall be the name specified in the module definition.
- The name of port cannot be a bit select or a part select.
- The port expression shall be the name used by the instantiating module and can be one of the following:
  - a simple net identifier
  - a scalar member of a vector net or port declared within the module
  - a sub-range of a vector net or port declared within the module
  - a vector net formed as a result of the concatenation operator
- The port expression is optional so 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 can not be mixed; connections to the ports of a particular module instance shall be all by order or all by name.

Examples:

```
module adc4 (out, rem, in);
   input in;
   output [3:0] out;    output rem;
   electrical [3:0] out;
   electrical in, rem, rem_chain;

   adc2 hi (.in(in), .out(out[3:2]), .remainder(rem_chain));
   adc2 lo (.in(rem_chain), .out(out[1:0]), .remainder(rem));
endmodule

module adc2 (out, in, remainder);
   output [1:0] out;    output remainder;
   input in;
   electrical [1:0] out;
   electrical in, remainder, r;

   // adc is same as defined in 6.5.4
   adc hi1 (out[1], r, in);
   adc lo1 (out[0], remainder, r);
endmodule
```

Since these connections were made by port name, the order in which the connections appear is irrelevant.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

6.5.6 Detecting port connections

When a module is instantiated, all of its ports need not be connected. For example, a clock module may provide outputs `clk` and `clkbar`, but a design may only need `clk`. In some cases, it may be important to know whether a particular port is connected. For example, if the **`transition()`** filter of 4.5.8 is used on the outputs, it might speed up the simulation if the filter is only used when the port is connected. The **`$port_connected()`** function described in 9.19 can be used to determine whether a port is connected.

6.5.7 Port connection rules

All digital ports connected to a net shall be of compatible disciplines, as shall all analog ports connected to a net. Ports of both analog and digital discipline may be connected to a net provided the appropriate connect statements exist (see 7.7).

6.5.7.1 Matching size rule

A scalar port can be connected to a scalar net and a vector port can be connected to a vector net or concatenated net expression of the matching width. In other words, the sizes of the ports and net need to match.

6.5.7.2 Resolving discipline of undeclared interconnect signal

Verilog-AMS HDL supports undeclared interconnects between module instances when describing hierarchical structures. That is, a signal appearing in the connection list of a module instantiation need not appear in any port declaration or discipline declaration (see 7.4).

6.5.8 Inheriting port natures

A net of continuous discipline shall have a potential nature and may have a flow nature. Because of hierarchical connections, an analog node may be associated with a number of analog nets, and thus, a number of continuous disciplines. The node shall be treated as having a **`potential abstol`** with a value equal to the smallest **`abstol`** of all the potential natures of all the disciplines with which it is associated. The node shall be treated as having a **`flow abstol`** with a value equal to the smallest **`abstol`** of all the flow natures, if any, of all the disciplines with which it is associated.

6.6 Generate constructs

Generate constructs are used to either conditionally or multiply instantiate generate blocks into a model. A generate block is a collection of one or more module items. A generate block may not contain port declarations, parameter declarations, specify blocks, or specparam declarations. All other module items, including other generate constructs, are allowed in a generate block. Generate constructs provide the ability for parameter values to affect the structure of the model. They also allow for modules with repetitive structure to be described more concisely, and they make recursive module instantiation possible.

There are two kinds of generate constructs: *loops* and *conditionals*. Loop generate constructs allow a single generate block to be instantiated into a model multiple times. Conditional generate constructs, which include if-generate and case-generate constructs, instantiate at most one generate block from a set of alternative generate blocks. The term *generate scheme* refers to the method for determining which or how many generate blocks are instantiated. It includes the conditional expressions, case alternatives, and loop control statements that appear in a generate construct.

Generate schemes are evaluated during elaboration of the model. Elaboration occurs after parsing the HDL and before simulation; and it involves expanding module instantiations, computing parameter values, resolving hierarchical names (see 6.7), establishing net connectivity and in general preparing the model for simula-

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

tion. Although generate schemes use syntax that is similar to behavioral statements, it is important to recognize that they do not execute at simulation time. They are evaluated at elaboration time, and the result is determined before simulation begins. Therefore, all expressions in generate schemes shall be constant expressions, deterministic at elaboration time. For more details on elaboration, see 6.9.

The elaboration of a generate construct results in zero or more instances of a generate block. An instance of a generate block is similar in some ways to an instance of a module. It creates a new level of hierarchy. It brings the objects, behavioral constructs, and module instances within the block into existence. These constructs act the same as they would if they were in a module brought into existence with a module instantiation, except that object declarations from the enclosing scope can be referenced directly (see 6.8). Names in instantiated named generate blocks can be referenced hierarchically as described in 6.7.

The keywords **`generate`** and **`endgenerate`** may be used in a module to define a *generate region*. A generate region is a textual span in the module description where generate constructs may appear. Use of generate regions is optional. There is no semantic difference in the module when a generate region is used. A parser may choose to recognize the generate region to produce different error messages for misused generate construct keywords. Generate regions do not nest, and they may only occur directly within a module. If the generate keyword is used, it shall be matched by an endgenerate keyword.

The syntax for generate constructs is given in Syntax 6-8.

```
module_or_generate_item ::=                                   // from A.1.4
        { attribute_instance } module_or_generate_item_declaration
      | { attribute_instance } local_parameter_declaration ;
      | { attribute_instance } parameter_override
      | { attribute_instance } continuous_assign
      | { attribute_instance } gate_instantiation
      | { attribute_instance } udp_instantiation
      | { attribute_instance } module_instantiation
      | { attribute_instance } initial_construct
      | { attribute_instance } always_construct
      | { attribute_instance } loop_generate_construct
      | { attribute_instance } conditional_generate_construct
      | { attribute_instance } analog_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
      | branch_declaration
      | analog_function_declaration
generate_region ::=                                           // from A.4.2
        generate { module_or_generate_item } endgenerate
genvar_declaration ::=
        genvar list_of_genvar_identifiers ;
list_of_genvar_identifiers ::=
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
        genvar_identifier { , genvar_identifier }
analog_loop_generate_statement ::=
        for ( genvar_initialization ; genvar_expression ; genvar_iteration )
            analog_statement
loop_generate_construct ::=
        for ( genvar_initialization ; genvar_expression ; genvar_iteration )
            generate_block
genvar_initialization ::=
        genvar_identifier = constant_expression
genvar_expression ::=
        genvar_primary
      | unary_operator { attribute_instance } genvar_primary
      | genvar_expression binary_operator { attribute_instance } genvar_expression
      | genvar_expression ? { attribute_instance } genvar_expression : genvar_expression
genvar_iteration ::=
      genvar_identifier = genvar_expression
genvar_primary ::=
        constant_primary
      | genvar_identifier
conditional_generate_construct ::=
        if_generate_construct
      | case_generate_construct
if_generate_construct ::=
        if ( constant_expression ) generate_block_or_null
        [ else generate_block_or_null ]
case_generate_construct ::=
        case ( constant_expression ) case_generate_item { case_generate_item } endcase
case_generate_item ::=
        constant_expression { , constant_expression } : generate_block_or_null
      | default [ : ] generate_block_or_null
generate_block ::=
        module_or_generate_item
      | begin [ : generate_block_identifier ] { module_or_generate_item } end
generate_block_or_null ::=
        generate_block
      | ;
```

Syntax 6-8—Syntax for generate constructs

6.6.1 Loop generate constructs

A loop generate construct permits a generate block to be instantiated multiple times using syntax that is similar to a for loop statement. The loop index variable shall be declared in a genvar declaration prior to its use in a loop generate scheme.

The genvar is used as an integer during elaboration to evaluate the generate loop and create instances of the generate block, but it does not exist at simulation time. A genvar shall not be referenced anywhere other than in a loop generate scheme.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Both the initialization and iteration assignments in the loop generate scheme shall assign to the same genvar. The initialization assignment shall not reference the loop index variable on the right-hand side.

Within the generate block of a loop generate construct, there is an implicit localparam declaration. This is an integer parameter that has the same name and type as the loop index variable, and its value within each instance of the generate block is the value of the index variable at the time the instance was elaborated. This parameter can be used anywhere within the generate block that a normal parameter with an integer value can be used. It can be referenced with a hierarchical name.

Because this implicit localparam has the same name as the genvar, any reference to this name inside the loop generate block will be a reference to the localparam, not to the genvar. As a consequence, it is not possible to have two nested loop generate constructs that use the same genvar.

Generate blocks in loop generate constructs can be named or unnamed, and they can consist of only one item, which need not be surrounded by begin/end keywords. Even if the begin/end keywords are absent, it is still a generate block, which, like all generate blocks, comprises a separate scope and a new level of hierarchy when it is instantiated.

If the generate block is named, it is a declaration of an array of generate block instances. The index values in this array are the values assumed by the genvar during elaboration. This can be a sparse array because the genvar values do not have to form a contiguous range of integers. The array is considered to be declared even if the loop generate scheme resulted in no instances of the generate block. If the generate block is not named, the declarations within it cannot be referenced using hierarchical names other than from within the hierarchy instantiated by the generate block itself.

It shall be an error if the name of a generate block instance array conflicts with any other declaration, including any other generate block instance array. It shall be an error if the loop generate scheme does not terminate. It shall be an error if a genvar value is repeated during the evaluation of the loop generate scheme. It shall be an error if any bit of the genvar is set to x or z during the evaluation of the loop generate scheme.

For example, this module implements a continuously running (unclocked) analog-to-digital converter.

```
module adc (in, out);
   parameter bits=8, fullscale=1.0, dly=0.0, ttime=10n;
   input in;
   output [0:bits-1] out;
   electrical in;
   electrical [0:bits-1] out;

   real sample, thresh;
   genvar i;

   analog begin
      thresh = fullscale/2.0;
      sample = V(in);
   end

   generate
      for (i=bits-1; i>=0; i=i-1)
         analog begin
            V(out[i]) <+ transition(sample > thresh, dly, ttime);
            if (sample > thresh) sample = sample - thresh;
            sample = 2.0*sample;
         end
   endgenerate
endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The model in the next two examples are parametrized modules that use a loop to generate SPICE primitive instances. The second of these examples makes a net declaration inside of the generate loop to generate the nodes needed to connect the analog primitives for each iteration of the loop.

This module implements an interconnect line constructed from RC sections.

```
module rcline (n1, n2);
   inout n1, n2;
   electrical n1, n2, gnd;
   ground gnd;
   parameter integer N = 10 from (0:inf);
   electrical [0:N] n;
   parameter Cap = 1p, Res = 1k;
   localparam Csec = Cap/N, Rsec = Res/N;

   genvar i;

   // "generate" and "endgenerate" keywords are not required.
   for (i=0; i <N; i=i+1) begin
      resistor #(.r(Rsec)) R(n[i], n[i+1]);
      capacitor #(.c(Csec)) C(n[i+1], gnd);
   end

   analog begin
      V(n1, n[0]) <+ 0.0;
      V(n2, n[N]) <+ 0.0;
   end
endmodule
```

This module also implements an interconnect line constructed from RC sections, but the sections are now symmetric. Additionally, the capacitor is now implemented by an **analog** block.

```
module rcline2 (n1, n2);
   inout n1, n2;
   electrical n1, n2, gnd;
   ground gnd;
   parameter integer N = 10 from (0:inf);
   electrical [0:N] n;
   parameter Cap = 1p, Res = 1k;
   localparam Csec = Cap/N, Rsec = Res/(2*N);

   genvar i;

   for (i=0; i <N; i=i+1) begin : section
      electrical n_int;

      resistor #(.r(Rsec)) R1(n[i], n_int);
      resistor #(.r(Rsec)) R2(n_int, n[i+1]);
      analog
         I(n_int, gnd) <+ Csec * ddt(V(n_int));
   end

   analog begin
      V(n1, n[0]) <+ 0.0;
      V(n2, n[N]) <+ 0.0;
   end
endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

In the above example the block inside the generate loop is a named `block`. For each block instance created by the generate loop, the generate block identifier for the loop is indexed by adding the "[genvar value]" to the end of the generate block identifier. These names can be used in hierarchical path names (see 6.7).

6.6.2 Conditional generate constructs

The conditional generate constructs, *if-generate* and *case-generate*, select at most one generate block from a set of alternative generate blocks based on constant expressions evaluated during elaboration. The selected generate block, if any, is instantiated into the model.

Generate blocks in conditional generate constructs can be named or unnamed, and they may consist of only one item, which need not be surrounded by **`begin-end`** keywords. Even if the **`begin-end`** keywords are absent, it is still a generate block, which, like all generate blocks, comprises a separate scope and a new level of hierarchy when it is instantiated.

Because at most one of the alternative generate blocks is instantiated, it is permissible for there to be more than one block with the same name within a single conditional generate construct. It is not permissible for any of the named generate blocks to have the same name as generate blocks in any other conditional or loop generate construct in the same scope, even if the blocks with the same name are not selected for instantiation. It is not permissible for any of the named generate blocks to have the same name as any other declaration in the same scope, even if that block is not selected for instantiation.

If the generate block selected for instantiation is named, then this name declares a generate block instance and is the name for the scope it creates. Normal rules for hierarchical naming apply. If the generate block selected for instantiation is not named, it still creates a scope; but the declarations within it cannot be referenced using hierarchical names other than from within the hierarchy instantiated by the generate block itself.

If a generate block in a conditional generate construct consists of only one item that is itself a conditional generate construct and if that item is not surrounded by begin/end keywords, then this generate block is not treated as a separate scope. The generate construct within this block is said to be directly nested. The generate blocks of the directly nested construct are treated as if they belong to the outer construct. Therefore, they can have the same name as the generate blocks of the outer construct, and they cannot have the same name as any declaration in the scope enclosing the outer construct (including other generate blocks in other generate constructs in that scope). This allows complex conditional generate schemes to be expressed without creating unnecessary levels of generate block hierarchy.

The most common use of this would be to create an if-else-if generate scheme with any number of else-if clauses, all of which can have generate blocks with the same name because only one will be selected for instantiation. It is permissible to combine if-generate and case-generate constructs in the same complex generate scheme. Direct nesting applies only to conditional generate constructs nested in conditional generate constructs. It does not apply in any way to loop generate constructs.

The following module implements a non-linear resistor that internally uses the SPICE resistor primitive if the non-linear coefficients are not given or a short if the resistance value is 0.

```
module nlres (inout electrical a, inout electrical b);
   parameter real res = 1k from (0:inf);
   parameter real coeff1 = 0.0;

   generate
      if ($param_given(coeff1) && coeff1 != 0.0)
         analog
            V(a, b) <+ res * (1.0 + coeff1 * I(a, b)) * I(a, b);
      else if (res == 0.0)
         analog
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
            V(a, b) <+ 0.0;
      else
         resistor #(.r(res)) R1(a, b);
   endgenerate
endmodule
```

For compact modeling of semiconductor devices where the delay time of signals through the device needs to be taken into account (non-quasi-static models) introduction of extra nodes and branches can be controlled through a module parameter.

```
module nmosfet (d, g, s, b);
   inout electrical d, g, s, b;
   parameter integer nqsMod = 0 from [0:1];

   // "generate" and "endgenerate" keywords are not required.
   if (nqsMod) begin : nqs
      electrical GP;
      electrical BP;
      electrical BI;
      electrical BS;
      electrical BD;
      ...
   end
endmodule
```

Conditional generate constructs make it possible for a module to contain an instantiation of itself. The same can be said of loop generate constructs, but it is more easily done with conditional generates. With proper use of parameters, the resulting recursion can be made to terminate, resulting in a legitimate model hierarchy. Because of the rules for determining top-level modules, a module containing an instantiation of itself will not be a top-level module.

The following example is a continuously running (unclocked) pipeline analog-to-digital converter that instantiates a lower resolution version of itself as part of its structure.

```
module pipeline_adc (in, out);
   parameter bits=8, fullscale=1.0;
   inout in;
   inout [0:bits-1] out;
   electrical in;
   electrical [0:bits-1] out;

   comparator #(.ref(fullscale/2)) cmp (in, out[bits-1]);

   generate
      if (bits > 1) begin
         electrical n1, n2;
         subtractor #(.level(fullscale)) sub (in, out[bits-1], n1);
         amp2x amp (n1, n2);
         pipeline_adc #(.bits(bits-1)) section (n2, out[0:bits-2]);
      end
   endgenerate
endmodule
```

Some of the functionality of conditional generate constructs can also be achieved using paramset overloading, see 6.4.2. For instance, selection of a particular module based on the value or presence of a parameter can also be handled by constructing appropriate paramsets.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

6.6.2.1 Dynamic parameters

A special case exists for dc sweep simulations: a series of operating point analyses where one or more parameters of the circuit change value between each analysis, see also 4.6.1 on the Analysis function. Digital simulations do not normally allow a parameter to vary during simulation; in analog simulation it is quite common to sweep a parameter during simulation to get information on how the parameter values influence the circuit behavior and hence the simulation results.

In connection with the conditional generate construct, an implementation may choose to limit the possible parameters to sweep to those that do not influence the structure of the circuit.

6.6.3 External names for unnamed generate blocks

Although an unnamed generate block has no name that can be used in a hierarchical name, it needs to have a name by which external interfaces can refer to it. A name will be assigned for this purpose to each unnamed generate block as described in the next paragraph.

Each generate construct in a given scope is assigned a number. The number will be 1 for the construct that appears textually first in that scope and will increase by 1 for each subsequent generate construct in that scope. All unnamed generate blocks will be given the name "genblk<n>" where <n> is the number assigned to its enclosing generate construct. If such a name would conflict with an explicitly declared name, then leading zeroes are added in front of the number until the name does not conflict.

Each generate construct is assigned its number as described in the previous paragraph even if it does not contain any unnamed generate bocks.

Example:

```
module top ();
   parameter genblk2 = 0;
   genvar i;

   // The following generate block is implicitly named genblk1
   if (genblk2) electrical a; // top.genblk1.a
   else         electrical b; // top.genblk1.b

   // The following generate block is implicitly named genblk02
   // as genblk2 is already a declared identifier
   if (genblk2) electrical a; // top.genblk02.a
   else         electrical b; // top.genblk02.b

   // The following generate block would have been named genblk3
   // but is explicitly named g1
   for (i = 0; i < 1; i = i + 1) begin : g1 // block name
      // The following generate block is implicitly named genblk1
      // as the first nested scope inside of g1
      if (1) electrical a; // top.g1[0].genblk1.a
   end

   // The following generate block is implicitly named genblk4 since
   // it belongs to the fourth generate construct in scope "top".
   // The previous generate block would have been named genblk3
   // if it had not been explicitly named g1
   for (i = 0; i < 1; i = i + 1)
      // The following generate block is implicitly named genblk1
      // as the first nested generate block in genblk4
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
      if (1) electrical a; // top.genblk4[0].genblk1.a

   // The following generate block is implicitly named genblk5
   if (1) electrical a; // top.genblk5.a

endmodule
```

6.7 Hierarchical names

Every identifier in Verilog-AMS HDL has a unique *hierarchical path name*. The hierarchy of modules and the definition of items such as named blocks within the modules define these names. The hierarchy of names can be viewed as a tree structure, where each module instance or a named begin-end block defines a new hierarchical level, or as a scope (of a particular branch of the tree).

At the top of the name hierarchy are the names of modules where no instances have been created. This is the *root* of the hierarchy. Inside any module, each module instance and named begin-end block define a new branch of the hierarchy. Named blocks within named blocks also create new branches.

Each node in the hierarchical name tree is treated as a separate scope with respect to identifiers. A particular identifier can be declared only once in any scope.

Any named object can be referenced uniquely in its full form by concatenating the names of the module instance or named blocks that contain it. The period character (**.**) is used to separate names in the hierarchy. The complete path name to any object starts at a top-level module. This path name can be used from any level in the description. The first name in a path name can also be the top of a hierarchy which starts at the level where the path is being used.

The syntax for hierarchical path names is given in Syntax 6-9.

---

hierarchical_identifier ::= [**$root .**] { identifier [**[** constant_expression **]**] **.** } identifier

---

Syntax 6-9—Syntax for hierarchical path name

Hierarchical names consist of instance names separated by periods, where an instance name can be an array element. The instance name **$root** refers to the top of the instantiated design and is used to unambiguously gain access to the top of the design.

```
$root.mymodule.u1 // absolute name
u1.struct1.field1 // u1 must be visible locally or above, including globally
adder1[5].sum
```

Examples:

```
module samplehold (in, cntrl, out);
   input in, cntrl;
   output out;
   electrical in, cntrl, out;
   electrical store, sample;
   parameter real vthresh = 0.0;
   parameter real cap = 10e-9;
   amp op1 (in, sample, sample);
   amp op2 (store, out, out);

   analog begin
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
      I(store) <+ cap * ddt(V(store));
      if (V(cntrl) > vthresh)
         V(store, sample) <+ 0;
      else
         I(store, sample) <+ 0;
   end
endmodule

module amp(inp, inm, out);
   input inp, inm;
   output out;
   electrical inp, inm, out;
   parameter real gain=1e5;

   analog begin
      V(out) <+ gain*V(inp,inm);
   end
endmodule
```

Figure 6-2 illustrates the hierarchy implicit in the example code.

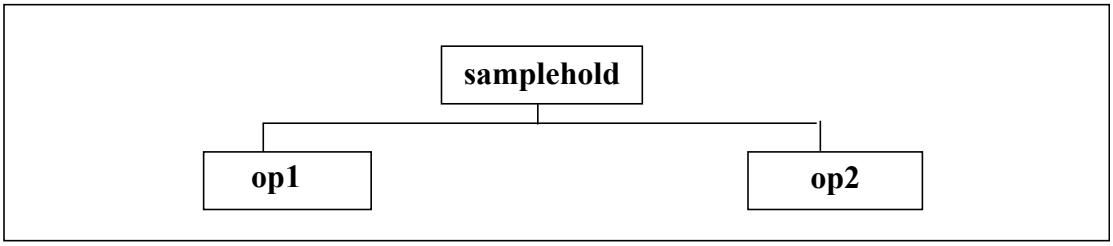


Figure 6-2: Hierarchy in a model

Figure 6-3 is a list of the hierarchical forms of the names of all the objects defined in the example code.

```
samplehold     in, cntrl, out, sample, store, vthresh, cap
op1            op1.inp, op1.inm, op1.out, op1.gain
op2            op2.inp, op2.inm, op2.out, op2.gain
```

Figure 6-3: Hierarchical path names in a design

6.7.1 Usage of hierarchical references

The following usage rules and semantic restrictions shall be applied to analog identifiers referred hierarchically using an *out-of-module reference* (OOMR) in a mixed signal module:

- Potential and flow access for named and unnamed branches (including port branches) can be done hierarchically.
- Hierarchical reference of an implicit net is allowed when the referenced net is first coerced to a specific discipline.
- Access of parameters can be done hierarchically. However, parameter declaration statements shall not make out-of-module references (e.g., for setting default values).
- Analog user-defined functions can be accessed hierarchically.
- It shall be an error to access analog variables hierarchically.
- Potential and flow contributions to named and unnamed branches can be done hierarchically.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

— It shall be an error to assign to an analog variable using hierarchical notation.

Hierarchical references to analog branches and nets can be done in both analog as well as digital blocks.

Verilog-AMS HDL follows the rules for hierarchical upward referencing as described in 12.6 of IEEE Std 1364 Verilog with the addition that the *scope_name* shall be restricted to a *hierarchical_inst_identifier*.

6.8 Scope rules

The following elements define a new scope in Verilog-AMS HDL:

— modules
— tasks
— named blocks
— functions
— generate blocks
— analog functions

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 which have the same name, or to name a task the same as a variable within the same module, or to give an instance the same name as the name of the net connected to its output. For generate blocks, this rule applies regardless of whether the generate block is instantiated. An exception to this is made for generate blocks in a conditional generate construct. See 6.6.3 for a discussion of naming conditional generate blocks.

If an identifier is referenced directly (without a hierarchical path) within a named block, or generate block it shall be declared either within the named block, or generate block locally or within a module, or within a named block, or generate block that is higher in the same branch of the name tree containing the named block, or generate block. If it is declared locally, 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 named block, or generate block, it continues to search higher level modules until found.

Because of the upward searching process, path names which are not strictly on a downward path can be used.

6.9 Elaboration

Elaboration is the process that occurs between parsing and simulation. It binds modules to module instances, builds the model hierarchy, computes parameter values, selects paramsets, resolves hierarchical names, establishes net connectivity, resolves disciplines and inserts connect modules, and prepares all of this for simulation. With the addition of generate constructs, the order in which these tasks occur becomes significant.

6.9.1 Concatenation of analog blocks

A module definition may have multiple **`analog`** blocks. The simulator shall internally combine the multiple **`analog`** blocks into a single **`analog`** block in the order that the **`analog`** blocks appear in the module description. In other words, the **`analog`** blocks shall execute in the order that they are specified in the module.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Concatenation of the **analog** blocks occurs after all generate constructs have been evaluated, i.e. after the loop generate constructs have been unrolled, and after the conditional generate constructs have been selected. If an **analog** block appears in a loop generate statement, then the order in which the loop is unrolled during elaboration determines the order in which the **analog** blocks are concatenated to the eventual single **analog** block after elaboration.

6.9.2 Elaboration and paramsets

If a generate construct contains an instantiation of an overloaded paramset, then the paramset selection is performed after the generate construct has been evaluated. The evaluation of the generate construct may influence the values and connections of the paramset instance, and hence the selection of matching paramset and module.

6.9.3 Elaboration and connectmodules

Automatic insertion of connect modules is a post-elaboration operation, as first the disciplines of the various nets needs to be resolved. This is described in detail in 7.8.

Discipline resolution can only occur after elaboration of the generate constructs once the connections of all nets has been resolved. It should also occur after the paramset selection as the choice for a particular module instantiation may affect the disciplines of the connected nets.

6.9.4 Order of elaboration

Because of generate constructs and paramsets, the model hierarchy can depend on parameter values. Because defparam statements can alter parameter values from almost anywhere in the hierarchy, the result of elaboration can be ambiguous when generate constructs are involved. The final model hierarchy can depend on the order in which defparams and generate constructs are evaluated.

The use of paramsets cannot introduce ambiguity as no defparam inside the hierarchy below a paramset instantiation is allowed, see 6.3.1 and 6.4.

The following algorithm defines an order that produces the correct hierarchy:

1) A list of starting points is initialized with the list of top-level modules.
2) The hierarchy below each starting point is expanded as much as possible without elaborating generate constructs. All parameters encountered during this expansion are given their final values by applying initial values, parameter overrides, defparam statements, and paramset selections.
3) In other words, any defparam statement whose target can be resolved within the hierarchy elaborated so far must have its target resolved and its value applied. defparam statements whose target cannot be resolved are deferred until the next iteration of this step. Because no defparam inside the hierarchy below a generate construct is allowed to refer to a parameter outside the generate construct, it is possible for parameters to get their final values before going to step 3).
4) Each generate construct encountered in step 2) is revisited, and the generate scheme is evaluated. The resulting generate block instantiations make up the new list of starting points. If the new list of starting points is not empty, go to step 2).

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

7. Mixed signal

7.1 Overview

With the mixed use of digital and analog simulators, a common terminology is needed. This clause provides the core terminology used in this LRM and highlights the behavior of the mixed-signal capabilities of Verilog-AMS HDL.

Verilog-AMS HDL provides the ability to accurately model analog, digital, and mixed-signal blocks. *Mixed-signal blocks* provide the ability to access data and be controlled by events from the other domain. In addition to providing mixed-signal interaction directly through behavioral descriptions, Verilog-AMS HDL also provides a mechanism for the mixed-signal interaction between modules.

Verilog-AMS HDL is a hierarchical language which enables top-down design of mixed-signal systems. Connect modules are used in the language to resolve the mixed-signal interaction between modules. These modules can be manually inserted (by the user) or automatically inserted (by the simulator) based on rules provided by the user.

Connect rules and the discipline of the mixed signals can be used to control auto-insertion throughout the hierarchy. Prior to insertion, all net segments of a mixed signal shall first be assigned a discipline. This is commonly needed for interconnect, which often does not have a discipline declared for it. Once a discipline has been assigned (usually through use of a discipline resolution algorithm), *connect modules* shall be inserted based on the specified connect rules. *Connect rules* control which connect modules are used and where are they inserted.

Connect modules are a special form of a mixed-signal module which allow accurate modeling of the interfaces between analog and digital blocks. They help ensure the drivers and receivers of a connect module are correctly handled so the simulation results are not impacted.

This clause also details a feature which allows analog to accurately model the effects the digital receivers for mixed signals containing both drivers and receivers. In addition, special functions provide access to driver values so a more accurate connect module can be created.

The following subclauses define these capabilities in more detail.

7.2 Fundamentals

The most important feature of Verilog-AMS HDL is that it combines the capabilities of both analog and digital modeling into a single language. This subclause describes how the continuous (analog) and discrete (digital) domains interact together, as well as the mixed-signal-specific features of the language.

7.2.1 Domains

The domain of a value refers to characteristics of the computational method used to calculate it. In Verilog-AMS HDL, a variable is calculated either in the *continuous* (analog) *domain* or the *discrete* (digital) *domain* every time. The potentials and flows described in natures are calculated in the continuous domain, while register contents and the states of gate primitives are calculated in the discrete domain. The values of real and integer variables can be calculated in either the continuous or discrete domain depending on how their values are assigned.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Values calculated in the discrete domain change value instantaneously and only at integer multiples of a minimum resolvable time. For this reason, the derivative with respect to time of a digital value is always zero (`0`). Values calculated in the continuous domain, on the other hand, are continuously varying.

7.2.2 Contexts

The domain of a variable is determined based upon the context in which it is assigned. Assignment statements which appear within an **`analog`** block or **`analog initial`** block are considered to be in the *continuous* (analog) context. Assignment statements within an **`initial`** or **`always`** block, or within a *continuous_assign* statement, are said to be within the *discrete* (digital) context. Module-level variable declaration assignments are considered to be context free and do not associate the variable with a particular domain. It shall be an error to assign to a given variable in both contexts. The domain of an unassigned variable is undefined and left up to implementations to determine. The implementation may issue a warning if an unassigned variable is referenced.

7.2.3 Nets, nodes, ports, and signals

In Verilog-AMS HDL, hierarchical structures are created when higher-level modules create instances of lower level modules and communicate with them through input, output, and bidirectional ports. A *port* represents the physical connection between an expression in the instantiating or parent module and an expression in the instantiated or child module. The expressions involved are referred to as *nets*, although they can include registers, variables, and nets of both continuous and discrete disciplines. A port of an instantiated module has two nets, the upper connection (`vpiHiConn`) which is a net in the instantiating module and the lower connection (`vpiLoConn`) which is a net in the instantiated module, as shown in Figure 7-1. The `vpiLoConn` and `vpiHiConn` connections to a port are frequently referred to as the *formal* and *actual connection*s respectively.

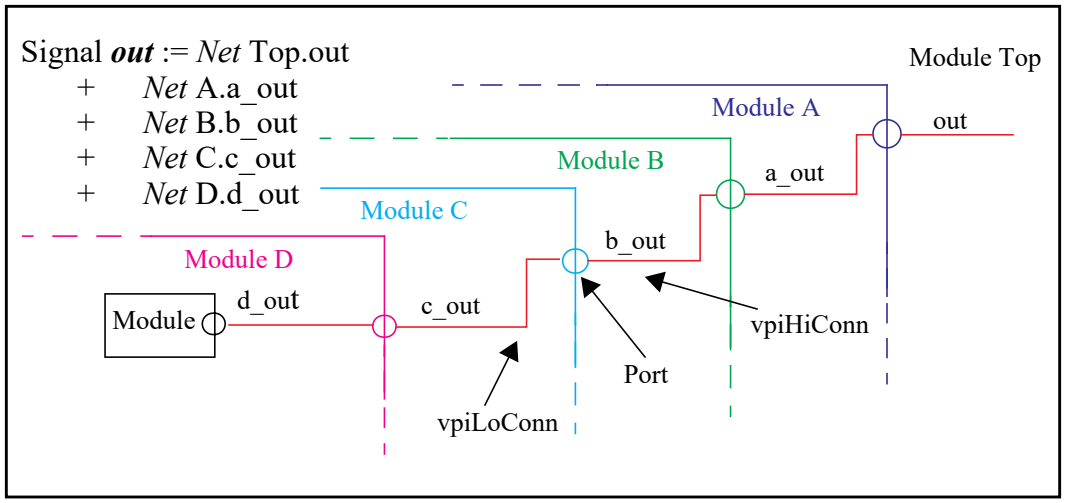


Figure 7-1: Signal “out” hierarchy of net segments

A net can be declared with either a discrete or analog *discipline* or no *discipline* (neutral interconnect). Within the Verilog-AMS language, only digital blocks and primitives can drive a discrete net (*drivers*), and only **`analog`** blocks can contribute to an analog net (*contributions*). A *signal* is a hierarchical collection of nets which, because of port connections, are contiguous. If all the nets that make up a signal are in the discrete domain, the signal is a *digital signal*. If all the nets that make up a signal are in the continuous domain, the signal is an *analog signal*. A signal that consists of nets from both domains is called a *mixed signal*.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Similarly, a port whose connections are both analog is an *analog port*, a port whose connections are both digital is a *digital port*, and a port whose connections are analog and digital is a *mixed port*.

Since it is physically one wire in the design, Kirchhoff's current law applies to the whole signal, and it forms one node in analog simulation (see 3.6). Drivers in the digital domain are converted to contributions in the analog domain using auto-inserted digital-to-analog connection modules (D2As), and the signal value is calculated in the analog domain. Instead of determining the final digital receiver value of the signal by resolving all the digital drivers, the resolved analog signal is converted back to a digital value. A digital behavioral block that reads the value of a signal is a *receiver*, but since Verilog-AMS has no syntax that identifies multiple receivers within a module as distinct, the associated net can be viewed as a single receiver for the purposes of analog to digital conversion. Drivers are created by declaring a reg, instantiating a digital primitive or using a continuous assign statement. Since it is only possible to insert connect modules at port boundaries, when multiple continuous assign statements exist in a module, they are handled by a single connect module.

The drivers and receivers of a mixed signal are associated with their locally-declared net; the discipline of that net is used to determined which connection modules to use. The discipline of the whole signal is found by discipline resolution, as described in 7.4, and is used to determine the attributes of the node in simulation.

7.2.4 Mixed-signal and net disciplines

One job of the discipline of a continuous net is to specify the tolerance (**`abstol`**) for the potential of the associated node. A mixed signal can have a number of compatible continuous nets, with different continuous disciplines and different abstols. In this case, the `abstol` of the associated node shall be the smallest of the *abstols* specified in the disciplines associated with all the continuous nets of the signal.

If an undeclared net segment has multiple compatible disciplines connected to it, a connect statement shall specify which discipline to use during discipline resolution.

7.3 Behavioral interaction

Verilog-AMS HDL supports several types of block statements for describing behavior, such as **`analog`** blocks, **`initial`** blocks, and **`always`** blocks. Typically, non-analog behavior is described in **`initial`** and **`always`** blocks, assignment statements, or **`assign`** declarations. There can be any number of **`initial`**, **`always`** and **`analog`** blocks in a particular Verilog-AMS HDL module.

Nets and variables in the continuous domain are termed *continuous nets* and *continuous variables* respectively. Likewise nets, regs and variables in the discrete domain are termed *discrete nets*, *discrete regs*, and *discrete variables*. In Verilog-AMS HDL, the nets and variables of one domain can be referenced in the other's context. This is the means for passing information between two different domains (continuous and discrete). Read operations of nets and variables in both domains are allowed from both contexts. Write operations of nets and variables are only allowed from the context of their domain.

Verilog-AMS HDL provides ways to:

- access discrete primaries (e.g., nets, regs, or variables) from a continuous context
- access continuous primaries (e.g., flows, potentials, or variables) from a discrete context
- detect discrete events in a continuous context
- detect continuous events in a discrete context

The specific time when an event from one domain is detected in the other domain is subject to the synchronization algorithm described in 7.3.6 and Clause 8. This algorithm also determines when changes in nets and variables of one domain are accessible in the other domain.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

7.3.1 Accessing discrete nets and variables from a continuous context

Discrete nets and variables can be accessed from a continuous context. However, because the data types which are supported in continuous contexts are more restricted than those supported in discrete contexts, certain discrete types can not be accessed in a continuous context.

Table 7-1 lists how the various discrete net/variable types can be accessed from a continuous context.

Table 7-1—Discrete net/variable access from continuous context

Discrete net/reg/ variable type	Examples	Equivalent continuous variable type	Access to this discrete net/reg/variable type from a continuous context
real	**real** r; **real** rm[0:8];	**real**	Discrete reals are accessed in the continuous context as real numbers.
integer	**integer** i; **integer** im[0:4];	**integer**	Discrete integers are accessed in continuous context as integer numbers.
bit	**reg** r1; **wire** w1; **reg** [0:9] r[0:7]; **reg** r[0:66]; **reg** [0:34] rb;	**integer**	Discrete bit and bit groupings (buses and part selects) are accessed in the continuous context as integer numbers. The sign bit (bit 31) of the integer is always set to zero (0). The lowest bit of the bit grouping is mapped to the zeroth bit of the integer. The next bit of the bus is mapped to the first bit of the integer and so on. If the bus width is less than 31 bits, the higher bits of the integer are set to zero (0). Access of discrete bit groupings with greater than 31 bits is illegal.

The syntax for a Verilog-AMS HDL primary is defined in Syntax 7-1.

```
primary ::=                                                    // from A.8.4
        number
      | hierarchical_identifier [ { [ expression ] } [ range_expression ] ]
      | concatenation
      | multiple_concatenation
      | function_call
      | system_function_call
      | ( mintypmax_expression )
      | string_literal
      | branch_probe_function_call
      | port_probe_function_call
```

Syntax 7-1—Syntax for primary

The following example accesses the discrete primary `in` from a continuous context.

```
module onebit_dac (in, out);
   input in;
   inout out;
   wire in;
   electrical out;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   real x;

   analog begin
      if (in == 0)
         x = 0.0;
      else
         x = 3.0;
      V(out) <+ x;
   end
endmodule
```

7.3.2 Accessing X and Z bits of a discrete net in a continuous context

Discrete nets can contain bits which are set to `x` (*unknown*) or `z` (*high impedance*). Verilog-AMS HDL supports accessing of 4-state logic values within the analog context. The `x` and `z` states must be translated to equivalent analog real or integer values before being used within the analog context. The language supports the following specific features, which provide a mechanism to perform this conversion.

— the case equality operator (`===`)
— the case inequality operator (`!==`)
— the **`case`**, **`casex`**, and **`casez`** statements
— binary, octal and hexadecimal numeric constants which can contain `x` and `z` as digits.

The case equality and case inequality operators have the same precedence as the equality operator.

Example:

```
module a2d(dnet, anet);
   input dnet;
   output anet;
   wire dnet;
   ddiscrete dnet;
   electrical anet;
   real avar;

   analog begin
      if (dnet === 1'b1)
         avar = 5;
      else if (dnet === 1'bx)
         avar = avar; // hold value
      else if (dnet === 1'b0)
         avar = 0;
      else if (dnet === 1'bz)
         avar = 2.5; // high impedance - float value

      V(anet) <+ avar;
   end
endmodule
```

A **`case`** statement could also have been used as an alternative to the above *if-else-if* statement to perform the 4-state logic value comparisons.

Example:

```
      case (dnet)
         1'b1: avar = 5;
         1'bx: avar = avar; // hold value
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
      1'b0: avar = 0;
      1'bz: avar = 2.5; // high impedance - float value
   endcase
```

Accessing digital net and digital binary constant operands are supported within analog context expressions. It is an error if these operands return `x` or `z` bit values when solved. It will be an error if the value of the digital variable being accessed in the analog context goes either to `x` or `z`.

Example:

```
module converter(dnet, anet);
   output dnet;
   inout anet;
   reg dnet;
   electrical anet;
   integer var1;
   real var2;

   initial begin
      dnet = 1'b1;
      #50 dnet = 1'bz;
      $finish;
   end

   analog begin
      var1 = 1'bx;         // error
      var2 = 1'bz;         // error
      var1 = 1 + dnet;     // error after #50

      if (dnet === 1'bx)   // error
         $display("Error to access x bit in continuous context");

      V(anet) <+ 1'bz;     // error
      V(anet) <+ dnet;     // error after #50
   end
endmodule
```

The syntax for the features that support `x` and `z` comparisons in a continuous context is defined in 2.6 and 5.8.3. Support for `x` and `z` is limited in the **`analog`** blocks as defined above.

NOTE—Consult section 5.1.8 in IEEE Std 1364 Verilog for a description of the semantics of these operators.

7.3.2.1 Special floating point values

Floating point arithmetic can produce special values representing plus and minus infinity and Not-a-Number (NaN) to represent a bad value. While use of these special numbers in digital expressions is not an error, it is illegal to assign these values to a branch through contribution in the analog context.

7.3.3 Accessing continuous nets and variables from a discrete context

All continuous nets can be probed from a discrete context using access functions. All probes which are legal in a continuous context of a module are also legal in the discrete context of a module.

The following example accesses the continuous net `V(in)` from the discrete context is.

```
module sampler (in, clk, out);
   inout in;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    input clk;
    output out;
    electrical in;
    wire clk;
    reg out;

    always @(posedge clk)
       out = V(in);

endmodule
```

Continuous variables can be accessed for reading from any discrete context in the same module where these variables are declared. Because the discrete domain can fully represent all continuous types, a continuous variable is fully visible when it is read in a discrete context. If the current time in the continuous and discrete kernels differ, interpolation is used to determine the value to be used in the discrete context for the continuous variable unless the value of the continuous variable was last assigned in an analog event statement. In this case, the value used in the digital context is exactly the same as the last value assigned to the continuous variable.

7.3.4 Detecting discrete events in a continuous context

Discrete events can be detected in a Verilog-AMS HDL continuous context. The arguments to discrete events in continuous contexts are in the discrete context. A discrete event in a continuous context is non-blocking like the other event types allowed in continuous contexts. The syntax for events in a continuous context is shown in Syntax 7-2.

```
analog_event_control_statement ::= analog_event_control analog_event_statement        // from A.6.5
analog_event_control ::=
      @ hierarchical_event_identifier
    | @ ( analog_event_expression )
analog_event_expression ::=
      expression
    | posedge expression
    | negedge expression
    | hierarchical_event_identifier
    | initial_step [ ( " analysis_identifier "{ , " analysis_identifier " } ) ]
    | final_step [ ( " analysis_identifier " { , " analysis_identifier " } ) ]
    | analog_event_functions
    | analog_event_expression or analog_event_expression
analog_event_functions ::=
      cross ( analog_expression [ , analog_expression_or_null
          [ , analog_expression_or_null [ , analog_expression_or_null [ , analog_expression ] ] ] ] )
    | above ( analog_expression [ , analog_expression_or_null
          [ , analog_expression_or_null [ , analog_expression ] ] ] )
    | timer ( analog_expression [ , analog_expression_or_null
          [ , analogt_expression_or_null [ , analog_expression ] ] ] )
    |absdelta ( analog_expression , analog_expression
          [ , analog_expression_or_null [ , analog_expression_or_null [ , analog_expression ] ] ] )
analog_event_statement ::=
      { attribute_instance } analog_loop_statement
    | { attribute_instance } analog_case_statement
    | { attribute_instance } analog_conditional_statement
    | { attribute_instance } analog_procedural_assignment
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
| { attribute_instance } analog_event_seq_block
| { attribute_instance } analog_system_task_enable
| { attribute_instance } disable_statement
| { attribute_instance } event_trigger
| { attribute_instance } ;
```

Syntax 7-2—Syntax for event control statement

The following example shows a discrete event being detected in an **analog** block.

```
module sampler3 (in, clk1, clk2, out);
   input in, clk1, clk2;
   output out;
   wire clk1;
   electrical in, clk2, out;
   real vout;

   analog begin
      @(posedge clk1 or cross(V(clk2), 1))
         vout = V(in);
      V(out) <+ vout;
   end
endmodule
```

7.3.5 Detecting continuous events in a discrete context

In Verilog-AMS HDL, monitored continuous events can be detected in a discrete context. The arguments to these events are in the continuous context. A continuous event in a discrete context is blocking like other discrete events. The syntax for analog events in a discrete context is shown in Syntax 7-3.

```
event_expression ::=                                          // from A.6.5
      expression
    | posedge expression
    | negedge expression
    | hierarchical_event_identifier
    | event_expression or event_expression
    | event_expression , event_expression
    | analog_event_functions
    | driver_update expression
    | analog_variable_lvalue
```

Syntax 7-3—Syntax for analog event detection in digital context

The following example detects a continuous event in an always block.

```
module sampler2 (in, clk, out);
   input in, clk;
   output out;
   wire in;
   reg out;
   electrical clk;

   always @(cross(V(clk) - 2.5, 1))
      out = in;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
endmodule
```

7.3.6 Concurrency

Verilog-AMS HDL provides synchronization between the continuous and discrete domains. Simulation in the discrete domain proceeds in integer multiples of the digital tick. This is the smallest value of the second argument of the **`timescale** directive (see 19.8 in IEEE Std 1364 Verilog).

Simulation in the continuous domain appears to proceed continuously. Thus, there is no time granularity below which continuous values can be guaranteed to be constant.

The rest of this subclause describes synchronization semantics for each of the four types of mixed-signal behavioral interaction. Any synchronization method can be employed, provided the semantics preserved. A typical synchronization algorithm is described in 8.2.

7.3.6.1 Analog event appearing in a digital event control

In this case, an analog event, such as **cross** or **timer**, appears in an `@()` statement in the digital context.

Example:

```
always begin
   @(cross(V(x) - 5.5,1))
      n = 1;
end
```

Besides using analog event functions, one can also use analog variables that are only assigned values in analog event statements in a digital event control statement. An event occurs whenever a value is assigned to the variable, regardless of whether the variable changes value or not. This might be done when one wants to sample a value in the continuous time domain to avoid jitter being created by the discrete nature of time in the digital context, but wish to process the sample in the digital context.

Example:

```
analog @(timer(0,100n))
   smpl = V(in);

always @(smpl) begin
   ...
```

When it is determined the event has occurred in the analog domain, the statements under the event control shall be scheduled in the digital domain at the nearest digital time tick to the time of the analog event. This event shall not be schedule in the digital domain earlier than the last or current digital event (see 8.3.3), however it may appear to be in a delta cycle belonging to a tick started at an earlier or later time.

Zero-delay scheduling is not rounded, so in the case where the digital event causes another event on the digital to analog boundary with zero delay, it will be handled at the current analog time.

7.3.6.2 Digital event appearing in an analog event control

Example:

```
analog begin
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    @(posedge n)
       r = 3.14;
end
```

In this case, a digital event, such as **`posedge`** or **`negedge`**, appears in an `@()` statement in the analog context.

When it is determined the event has occurred in the digital domain, the statements under the event control shall be executed in the analog domain at the time corresponding to a real promotion of the digital time (e.g., `27ns` to `27.0e-9`).

7.3.6.3 Analog primary appearing in a digital expression

In this case, an analog primary (variable, potential, or flow) whose value is calculated in the continuous domain appears in a expression which is in the digital context; thus the analog primary is evaluated in the digital domain.

The expression shall be evaluated using the analog value calculated for the time corresponding to a real promotion of the digital time at which the expression is evaluated.

If the current time in the continuous and discrete kernels differ, interpolation is used to determine the value to be used in the discrete context for the continuous variable unless the value of the continuous variable was last assigned in an analog event statement. In this case, the value used in the digital context is exactly the same as the last value assigned to the continuous variable.

7.3.6.4 Analog variables appearing in continuous assigns

Analog variables that are only assigned values within analog event statements can be used in the expressions that drive continuous assigns, both when the target of the continuous assign is a **`wreal`** or a traditional Verilog wire type (**`wire`**, **`trireg`**, **`wor`**, **`wand`**, etc.).

7.3.6.5 Digital primary appearing in an analog expression

In this case, a digital primary (**`reg`**, **`wire`**, **`integer`**, etc.) whose value is calculated in the discrete domain appears in an expression which is in the analog context; thus the analog primary is evaluated in the continuous domain.

The expression shall be evaluated using the digital value calculated for the greatest digital time tick which is less than or equal to the analog time when the expression is evaluated.

7.3.7 Function calls

Digital functions cannot be called from within the analog context. Analog functions cannot be called from within the digital context.

7.4 Discipline resolution

In general a mixed signal is a contiguous collection of nets, some with discrete discipline(s) and some with continuous discipline(s). A continuous signal is a contiguous collection of nets where all the nets are in the continuous domain (see 1.3). Additionally, some of the nets can have undeclared discipline(s). Discipline resolution assigns disciplines and domains to those nets whose discipline is undeclared. This is done to (1) control auto-insertion of connect modules, according to the rules embodied in *connect statements* and (2) to ensure that the nets of all mixed signals and continuous signals have a known discipline and domain.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The assignments are based on: discipline declarations, **`default_discipline** directives (see 3.8), and the hierarchical connectivity of the design. Once all net segments of every mixed signal has been resolved, insertion of connect modules shall be performed.

7.4.1 Compatible discipline resolution

One factor which influences the resolved discipline of a net whose discipline is undeclared is the disciplines of nets to which it is connected via ports; i.e., if multiple compatible disciplines are connected to the same net via multiple ports only one discipline can be assigned to that net. This is controlled by the **resolveto** form of the connect statement; the syntax of this form is described in 7.7.2.

If disciplines at the lower connections of ports (where the undeclared net is an upper connection) are among the disciplines in discipline_list, the result_discipline is the discipline which is assigned to the undeclared net. If all the nets are of the same discipline, no rule is needed; that discipline becomes the resolved discipline of the net.

In the example shown in Figure 7-2, `NetA` and `NetB` are undeclared interconnects. `NetB` has `cmos3` and `cmos4` at the lower connection ports, while it is an upper connection.

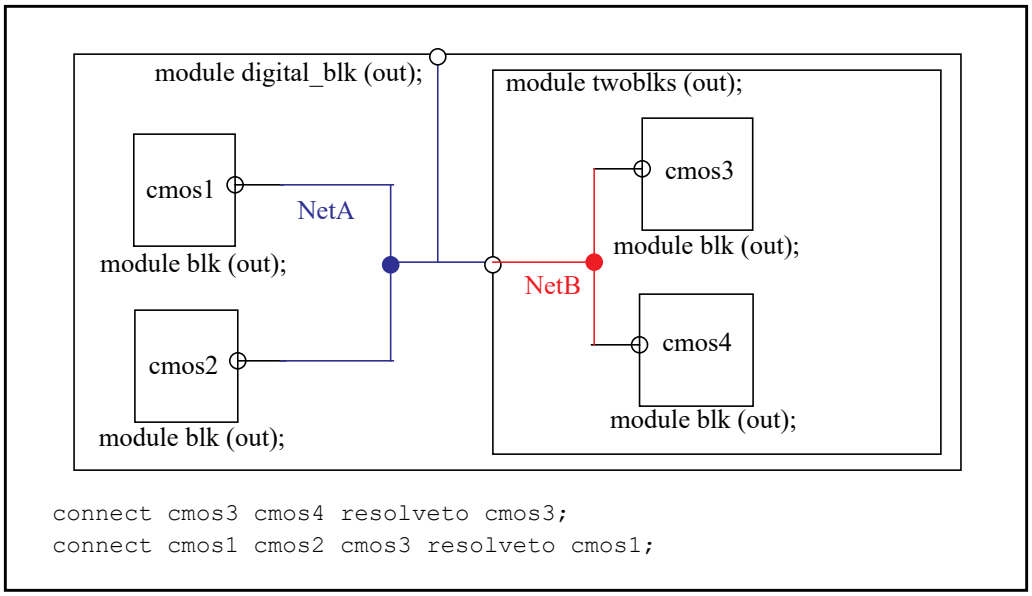


Figure 7-2: Compatible discipline resolution

The first connect statement resolves `NetB` to be assigned the discipline `cmos3`.

`NetA` has `cmos1`, `cmos2` and the resulting `cmos3` from module `twoblks` at the lower connection ports; based on the second connect statement, it resolves to be assigned the discipline cmos1.

7.4.2 Connection of discrete-time disciplines

Ports of discrete-time disciplines (ports where digital signals appear at both upper (`vpiHiConn`) and lower (`vpiLoConn`) connections) shall obey the rules imposed by IEEE Std 1364 Verilog on such connections.

In addition, the real-value nets shall obey the rules imposed by 3.7.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

7.4.3 Connection of continuous-time disciplines

Ports of continuous-time disciplines (ports where analog signals appear at both upper (`vpiHiConn`) and lower (`vpiLoConn`) connections) shall obey the rules imposed in 3.11. It shall be an error to connect incompatible continuous disciplines together.

7.4.4 Resolution of mixed signals

Once discipline declarations have been applied, if any mixed-signal or continuous-signal nets don't have a discipline and domain assigned additional resolution is needed. This section provides an additional method for discipline resolution of remaining undeclared nets.

There are two modes for this method of resolution, *basic* (the default) and *detail*, which determine how known disciplines are used to resolve these undeclared nets. For the entire design, undeclared nets shall be resolved at each level of the hierarchy where continuous (analog) has precedence over discrete (digital). The selection of these discipline resolution modes shall be vendor-specific.

More than one conflicting discipline declaration from the same context (in or out of context) for the same hierarchical segment of a signal is an error. In this case, *conflicting* simply means an attempt to declare more than one discipline regardless of whether the disciplines are compatible or not.

Sample algorithms for the complete discipline resolution process are listed in Annex F.

7.4.4.1 Basic discipline resolution algorithm

In this mode (the default), both continuous and discrete disciplines propagate up the hierarchy to meet one another. At each level of the hierarchy where continuous and discrete meet for an undeclared net that net segment is declared continuous. This typically results in connect modules being inserted higher up the design hierarchy.

In the example shown in Figure 7-3, `NetA`, `NetB`, `NetC`, and `NetD` are undeclared interconnects.

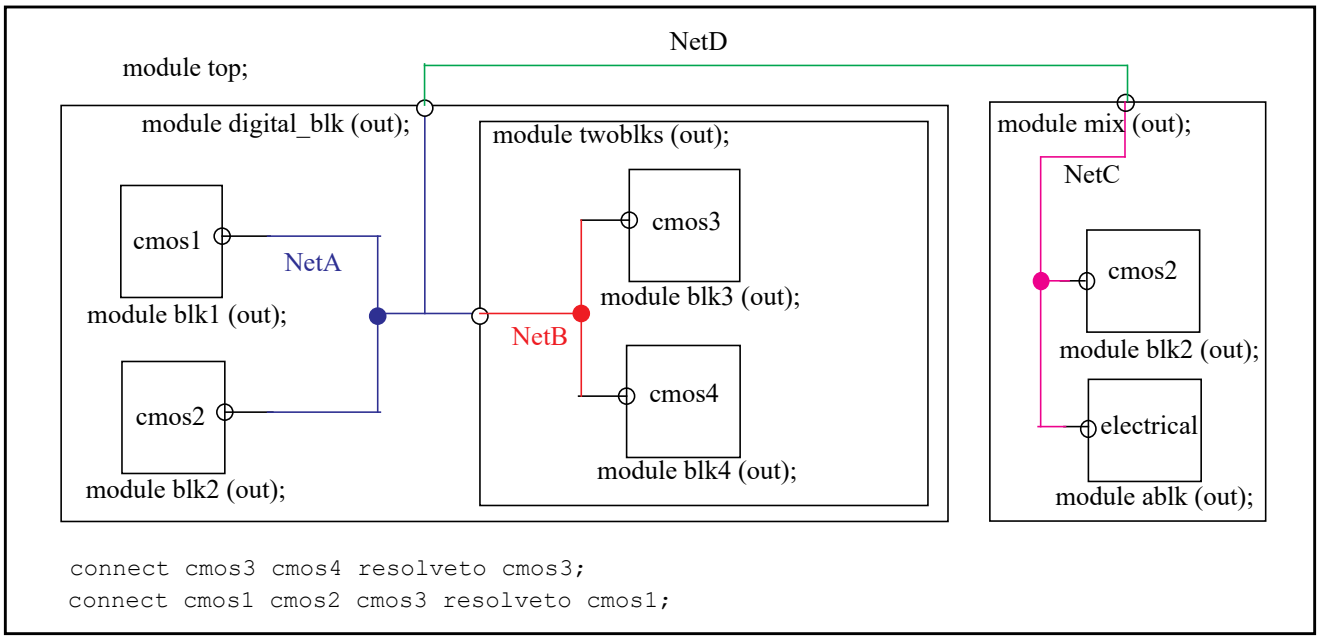


Figure 7-3: Discipline resolution mode: basic

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Using the basic mode of discipline resolution and the specified **`resolveto`** connect statements for this example results in the following:

— `NetB` resolves to `cmos3` based on the first **`resolveto`** connect statement.
— `NetA` resolves to `cmos1` based on the second **`resolveto`** connect statement.
— `NetC` resolves to electrical based on continuous (electrical) winning over discrete (`cmos2`).
— `NetD` resolves to electrical based on continuous (electrical) winning over discrete (`cmos1`).

7.4.4.2 Detail discipline resolution algorithm

In this mode continuous disciplines propagate up and then back down to meet discrete disciplines. Discrete disciplines do not propagate up the hierarchy. This can result in more connect modules being inserted lower down into discrete sections of the design hierarchy for added accuracy.

In the example shown in Figure 7-4, `NetA`, `NetB`, `NetC`, and `NetD` are undeclared interconnects.

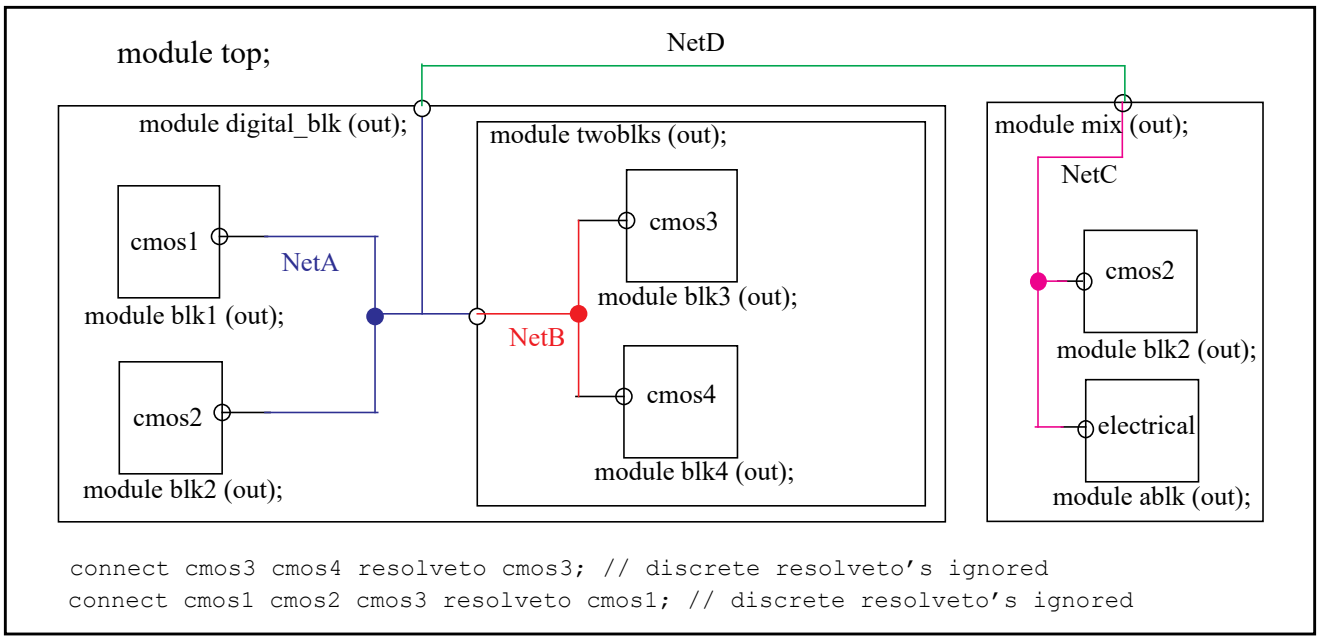


Figure 7-4: Discipline resolution mode: detail

Using the detail mode of discipline resolution for this example results in the following:

— *Continuous up*: `NetC` resolves to electrical based on continuous (electrical) winning over discrete (`cmos2`).
— *Continuous up*: `NetD` resolves to electrical based on continuous (electrical) winning over undeclared.
— *Continuous down*: `NetA` resolves to electrical based on continuous (electrical) winning over undeclared.
— *Continuous down*: `NetB` resolves to electrical based on continuous (electrical) winning over undeclared.

The specified **`resolveto`** connect statements are ignored in this mode unless coercion (see 7.8.1) is used.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

7.4.4.3 Coercing discipline resolution

Connect module insertion can be affected by *coercion* i.e., declaring disciplines for the interconnect in the hierarchy. If an interconnect is assigned a discipline, that discipline shall be used unless the **`resolveto`** connect statement overrides the discipline.

The example in Figure 7-5 shows several effects of coercion on auto-insertion.

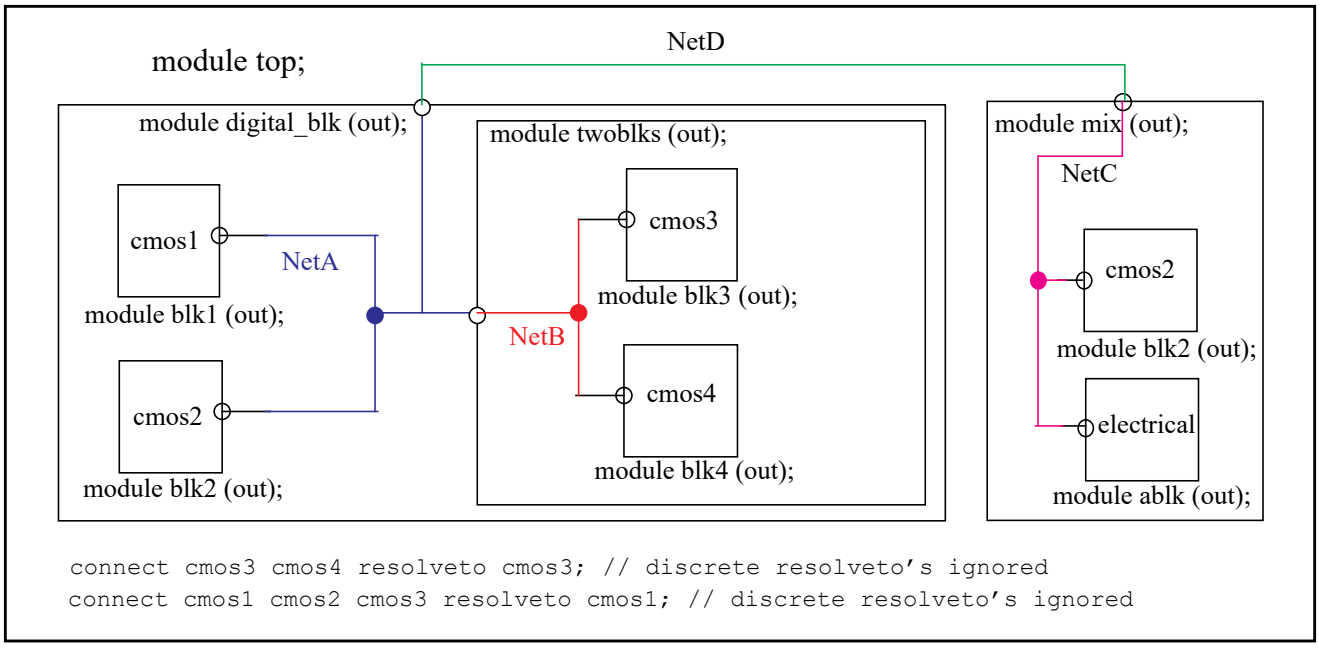


Figure 7-5: Coercion effects on auto- insertion

Case1: `NetB` is declared as `cmos3` (the others are undeclared)

```
cmos3 top.digital_blk.twoblks.NetB
```

discipline resolution basic: Same as without *coercion*.

discipline resolution detail: `NetB` stays `cmos3`; `NetA`, `NetC`, and `NetD` become electrical.

Case2: `NetA` is declared as `cmos1` (the others are undeclared)

discipline resolution basic: `NetA` stays `cmos1`, `NetB` is assigned `cmos3`, and `NetC` and `NetD` become electrical.

discipline resolution detail: Same as *basic* mode.

Case3: `NetC` is declared as `cmos2` (the others are undeclared)

discipline resolution basic: `NetC` stays `cmos2`, `NetB` is assigned `cmos3`, `NetA` is assigned `cmos1`, and `NetD` is assigned `cmos1`.

discipline resolution detail: Same as *basic* mode.

7.4.5 Discipline resolution of continuous signals

The discipline of nets, without a declared discipline, of a continuous signal shall also be determined by using the discipline resolution algorithms listed in Annex F. Both algorithms will give the same result as there are no discrete disciplines to propagate upwards.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

7.5 Connect modules

Connect modules are automatically inserted to connect the continuous and discrete disciplines (mixed nets) of the design hierarchy together. The continuous and discrete disciplines of the ports of the connect modules and their directions are used to determine the circumstances in which the module can be automatically inserted.

The connect module is a special form of a module; its definition is shown in Syntax 7-4.

---

module_declaration ::= *// from A.1.2*
{ attribute_instance } module_keyword module_identifier [module_parameter_port_list]
list_of_ports ; { module_item }
endmodule
| { attribute_instance } module_keyword module_identifier [module_parameter_port_list]
[list_of_port_declarations] ; { non_port_module_item }
endmodule

module_keyword ::= **module** | **macromodule** | **connectmodule**

Syntax 7-4—Syntax for connect modules

7.6 Connect module descriptions

The disciplines of mixed nets are determined prior to the connect module insertion phase of elaboration. Connect module declarations with matching port discipline declarations and directions are instantiated to connect the continuous and discrete domains of the mixed net.

The port disciplines define the default type of disciplines which shall be bridged by the connect module. The directional qualifiers of the discrete port determine the default scenarios where the module can be instantiated. The following combinations of directional qualifiers are supported for the continuous and discrete disciplines of a connect module:

Table 7-2—Connect module directional qualifier combinations

continuous	discrete
input	output
output	input
inout	inout

Example 1:

```
connectmodule d2a (in, out);
   input in;
   output out;
   ddiscrete in;
   electrical out;
   // insert connect module behavioral here
endmodule
```

can bridge a mixed input port whose upper connection is compatible with discipline `ddiscrete` and whose lower connection is compatible with `electrical`, or a mixed output port whose upper con-

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

nection is compatible with discipline `electrical` and whose lower connection is compatible with `ddiscrete`.

Example 2:

```
connectmodule a2d (out, in);
   output out;
   input in;
   ddiscrete out;
   electrical in;
   // insert connect module behavioral here
endmodule
```

can bridge a mixed `output` port whose upper connection is compatible with discipline `ddiscrete` and whose lower connection is compatible with `electrical`, or a mixed `input` port whose upper connection is compatible with discipline `electrical` and whose lower connection is compatible with `ddiscrete`.

Example 3:

```
connectmodule bidir (out, in);
   inout out;
   inout in;
   ddiscrete out;
   electrical in;
   // insert connect module behavioral here
endmodule
```

can bridge any mixed port whose one connection is compatible with discipline `ddiscrete` and whose connection is compatible with `electrical`.

7.7 Connect specification statements

Any number of connect modules can be defined. The designer can choose and specialize those in the design via the connect specification statements. The connect specification statements allow the designer to define:

- specification of which connect module is used, including parameterization, for bridging given discrete and continuous disciplines
- overrides for the connect module default disciplines and port directions
- resolution of incompatible disciplines

The syntax for connect specifications is shown in Syntax 7-5.

```
connectrules_declaration ::=                                    // from A.1.8
      connectrules connectrules_identifier ;
         { connectrules_item }
      endconnectrules
connectrules_item ::=
      connect_insertion
    | connect_resolution
```

Syntax 7-5—Syntax for connect specification statements

The two forms of the connect specification statements and their syntaxes are detailed in the following subclauses.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

7.7.1 Connect module auto-insertion statement

The connect module insertion statement declares which connect modules are automatically inserted when mixed nets of the appropriate types are encountered, as shown in Syntax 7-6.

This specifies the connect module *connect_module_identifier* is used to determine the mixed nets of the type used in the declaration of the connect module.

There can be multiple connect module declarations of a given (**discrete** — **continuous**) discipline pair and the connect module specification statement specifies which is to be used in the auto-insertion process. In addition, parameters of the connect module declaration can be specified via the *connect_attributes*.

```
connect_insertion ::= connect connectmodule_identifier [ connect_mode ]          // from A.1.8
        [ parameter_value_assignment ] [ connect_port_overrides ] ;
connect_mode ::= merged | split
connect_port_overrides ::=
      discipline_identifier , discipline_identifier
    | input discipline_identifier , output discipline_identifier
    | output discipline_identifier , input discipline_identifier
    | inout discipline_identifier , inout discipline_identifier
```

Syntax 7-6—Syntax for connect configuration statements

Connect modules can be reused for different, but compatible disciplines by specifying different discipline combinations in which the connect module can be used. The form is

connect *connect_module*_identifier connect_attributes *discipline*_identifier **,** *discipline*_identifier **;**

where the specified disciplines shall be compatible for both the continuous and discrete disciplines of the given connect module.

It is also possible to override the port directions of the connect module, which allows a module to be used both as a unidirectional and bidirectional connect module. This override also aids library based designs by allowing the user to specify the connect rules, rather than having to search the entire library. The form is

connect *connect_module*_identifier connect_attributes direction *discipline*_identifier **,**
direction *discipline*_identifier **;**

where the specified disciplines shall be compatible for both the continuous and discrete disciplines of the given connect module and the specified directions are used to define the type of connect module.

7.7.2 Discipline resolution connect statement

The discipline resolution connect statement specifies a single discipline to use during the discipline resolution process when multiple nets with compatible disciplines are part of the same mixed net, as shown in Syntax 7-7.

```
connect_resolution ::= connect discipline_identifier { , discipline_identifier } resolveto   // from A.1.8
        discipline_identifier_or_exclude ;
discipline_identifier_or_exclude ::=
      discipline_identifier
    | exclude
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Syntax 7-7—Syntax for connect configuration resolveto statements

where the discipline identifiers before the *resolveto* keyword are the list of compatible disciplines and the discipline identifier after is the discpline to be used. If the keyword *exclude* follows *resolveto* rather than a discipline identifier, then the otherwise compatible disciplines are deemed to be incompatible and an error is indicated if they are found on the same net.

Example:

```
connect logic18 logic32 resolveto exclude ;
connect electrical18 electrical32 resolveto exclude ;
```

In the first case, two discrete disciplines, and the second case two continuous disciplines, are declared to be incompatible. In both cases, the discipline ending in 18 is associated with 1.8V logic and the discipline ending in 32 is associated with 3.2V logic. These connect statements prevent ports associated with one supply voltage to be connected to nets associated with the other.

7.7.2.1 Connect rule resolution mechanism

When there is an exact match for the set of disciplines specified as part of the discipline_list, the resolved discipline would be as per the rule specified in the exact match. When more than one specified rule applies to a given scenario a warning message shall be issued by the simulator and the first match would be used.

When there is no exact fit, then the resolved discipline would be based on the subset of the rules specified. If there is more than one subset matching a set of disciplines, the simulator shall give a warning message and apply the first subset rule that satisfies the current scenario.

The resolved discipline need not be one of the disciplines specified in the discipline list.

The **`connect`**...**`resolveto`** shall not be used as a mechanism to set the disciplines of simulator primitives but used only for discipline resolution.

Example 1:

```
connect x,y,a resolveto a;
connect x,y resolveto x;
```

For the above set of connect rule specifications:

- disciplines `x,y` would resolve to discipline `x`.
- disciplines `x,y,a` would resolve to discipline `a`.
- disciplines `y,a` would resolve to discipline `a`.

Example 2:

```
connect x,y,a resolveto y;
connect x,y,a resolveto a;
connect x,y,b resolveto b;
```

For the above set of connect rule specifications:

- disciplines `x,y` would resolve to discipline `y` with a warning.
- disciplines `x,y,a` would resolve to discipline `y` with a warning.
- disciplines `y,b` would resolve to `b`.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

7.7.3 Parameter passing attribute

An attribute method can be used with the connect statement to specify parameter values to pass into the Verilog-AMS HDL connect module and override the default values. Any parameters declared in the connect module can be specified.

Example:

```
connect a2d_035u #(.tt(3.5n), .vcc(3.3));
```

Here each parameter is listed with the new value to be used for that parameter.

7.7.4 connect_mode

This can be used to specify additional segregation of connect modules at each level of the hierarchy. Setting *connect_mode* to **split** or **merged** defines whether all ports of a common discrete discipline and port direction share an connect module or have individual connect modules.

Example:

```
connect a2d_035u split #(.tt(3.5n), .vcc(3.3));
```

Here each digital port has a separate connect module.

7.8 Automatic insertion of connect modules

Automatic insertion of connect modules is performed when signals and ports with continuous time domain and discrete time domain disciplines are connected together. The connect module defines the conversion between these different disciplines.

An instance of the connect module shall be inserted across any mixed port that matches the rule specified by a **connect** statement. Rules for matching connect statements with ports take into account the port direction (see 7.8.1) and the disciplines of the signals connected to the port.

Each **connect** statement designates a module to be a connect module. When two disciplines are specified in a connect statement, one shall be discrete and the other continuous.

Example:

```
module dig_inv(in, out);
   input in;
   output out;
   reg out;
   ddiscrete in, out;
   always begin
      out = #10 ~in;
   end
endmodule

module analog_inv(in, out);
   input in;
   output out;
   electrical in, out;
   parameter real vth = 2.5;
   real outval;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   analog begin
      if (V(in) > vth)
         outval = 0;
      else
         outval = 5 ;
      V(out) <+ transition(outval);
   end
endmodule

module ring;
   dig_inv d1 (n1, n2);
   dig_inv d2 (n2, n3);
   analog_inv a3 (n3, n1);
endmodule

connectmodule elect_to_logic(el,cm);
   input el;
   output cm;
   reg cm;
   electrical el;
   ddiscrete cm;
   always
      @(cross(V(el) - 2.5, 1))
         cm = 1;
   always
      @(cross(V(el) - 2.5, -1))
         cm = 0;
endmodule

connectmodule logic_to_elect(cm,el);
   input cm;
   output el;
   ddiscrete cm;
   electrical el;
   analog
      V(el) <+ transition((cm == 1) ? 5.0 : 0.0);
endmodule

connectrules mixedsignal;
   connect elect_to_logic;
   connect logic_to_elect;
endconnectrules
```

Here two modules, `elect_to_logic` and `logic_to_elect`, are specified as the connect modules to be automatically inserted whenever a signal and a module port of disciplines `electrical` and `ddiscrete` are connected.

Module `elect_to_logic` converts signals on port `out` of instance `a3` to port `in` of instance `d1`. Module `logic_to_elect` converts the signal on port `out` of instance `d2` to port `in` of instance `a3`.

7.8.1 Connect module selection

The selection of a connect module for automatic insertion depends upon the disciplines of nets connected together at ports. It is, therefore, a post elaboration operation since the signal connected to a port is only known when the module in which the port is declared has been instantiated.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Auto-insertion of connect modules is done hierarchically. The connect modules are inserted based on the net disciplines and ports at each level of the hierarchy. The *connect_mode* **`split`** and **`merged`** are applied at each level of the hierarchy. This insertion supports the ability to coerce the placement of connect modules by declaring the disciplines of interconnect.

Figure 7-6 shows an example of auto-insertion with coercion.

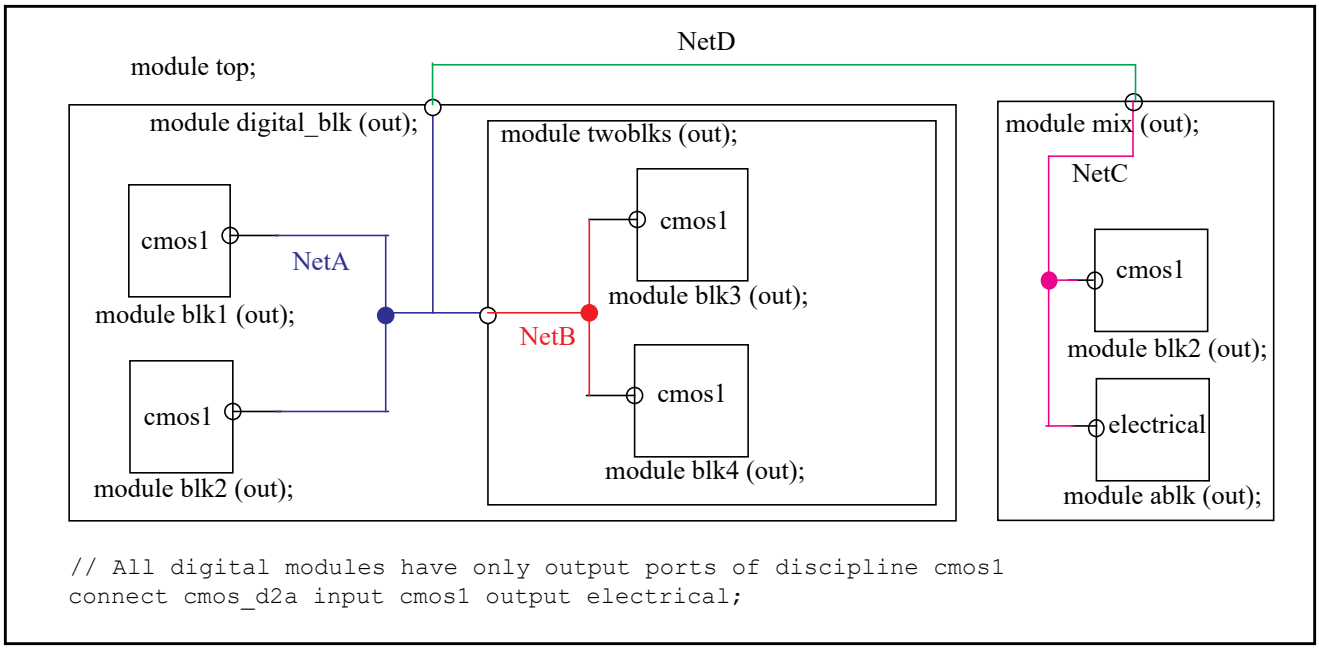


Figure 7-6: Auto-insertion with coercion

Case1: All interconnects are undeclared

- *discipline resolution basic*:
  - *merged*: `d2a` at `top.mix.blk2` and `d2a` at `top.digital_blk` (two connect modules).
  - *split*: Same as merged.
- *discipline resolution detail*:
  - *merged*: `d2a` at `top.mix.blk2`, `d2a` at `top.digital_blk.(blk1-blk2)`, and `d2a` at `top.digital_blk.twoblks` (three connect modules).
  - *split*: `d2a` at each of the five `cmos1` blocks.

Case2: If `NetB` is declared as `cmos1` and the remaining interconnect is undeclared

- *discipline resolution basic*:
  - *merged*: `d2a` at `top.mix.blk2` and `d2a` at `top.digital_blk` (two connect modules).
  - *split*: Same as merged.
- *discipline resolution detail*:
  - *merged*: `d2a` at `top.mix.blk2`, `d2a` at `top.digital_blk.(blk1-blk2)`, and `d2a` at `top.digital_blk.twoblks` (three connect modules).
  - *split*: `d2a` at `top.mix.blk2`, `d2a` at `top.digital_blk.blk1`, `d2a` at `top.digital_blk.blk2`, and `d2a` at `top.digital_blk.twoblks` (four connect modules).

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

7.8.2 Signal segmentation

Once a connect module has been selected it can not be inserted until it can be determined whether there should be one connect module per port or one connect module for all the ports on the net of a signal which match a given **`connect`** statement. Inserting multiple copies of the same connect module on one signal (i.e., between the signal and the multiple ports) has the effect of creating distinct segments of the signal with the same discipline at that level of the hierarchy.

This segmentation of the signal which connects ports is only performed in the case of digital ports (i.e., ports with discrete-time domain or digital discipline). For analog (or continuous-time domain) disciplines, it is not desirable to segment the signal between the ports; i.e, there shall never be more than one analog node representing a signal. However, it can be desirable for the simulator's internal representation of the signal to consist of various separate digital segments, each with its own connect module.

Figure 7-7 shows how to model the loading effect of each individual digital port on the analog node.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

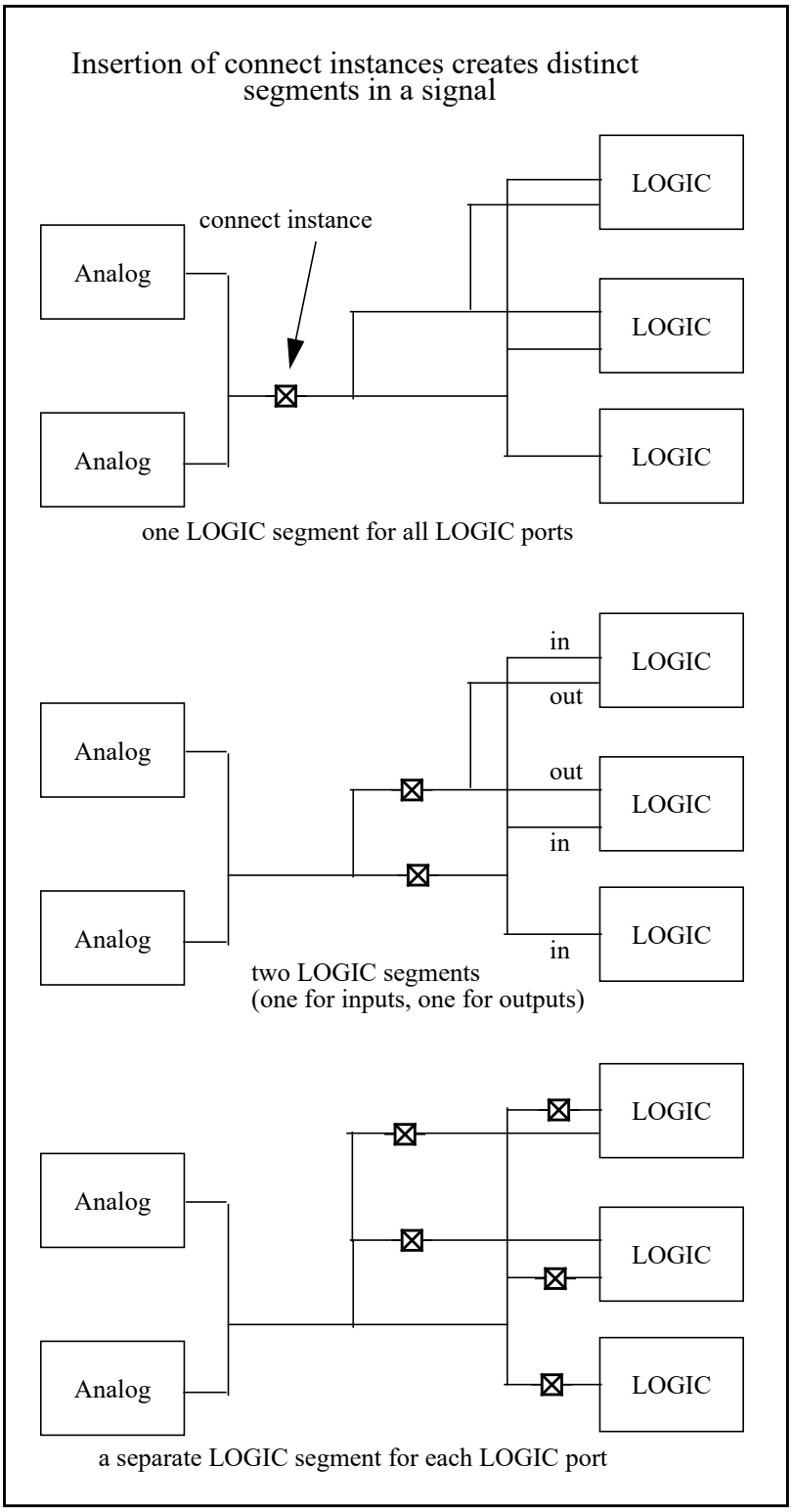


Figure 7-7: Signal segmentation by connect modules

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

7.8.3 connect_mode parameter

This parameter can be used in the `connect` statement to direct the segmentation of the signal at each level of the hierarchy, which can occur while inserting a connect module. It can be one of two predefined values, **`split`** or **`merged`.** The default is **`merged`**.

The *connect_mode* indicates how input, output, or inout ports of the given discipline shall be combined for the purpose of inserting connect modules. It is applied when there is more than one port of discrete discipline on a net of a signal where the **`connect`** statement applies.

7.8.3.1 merged

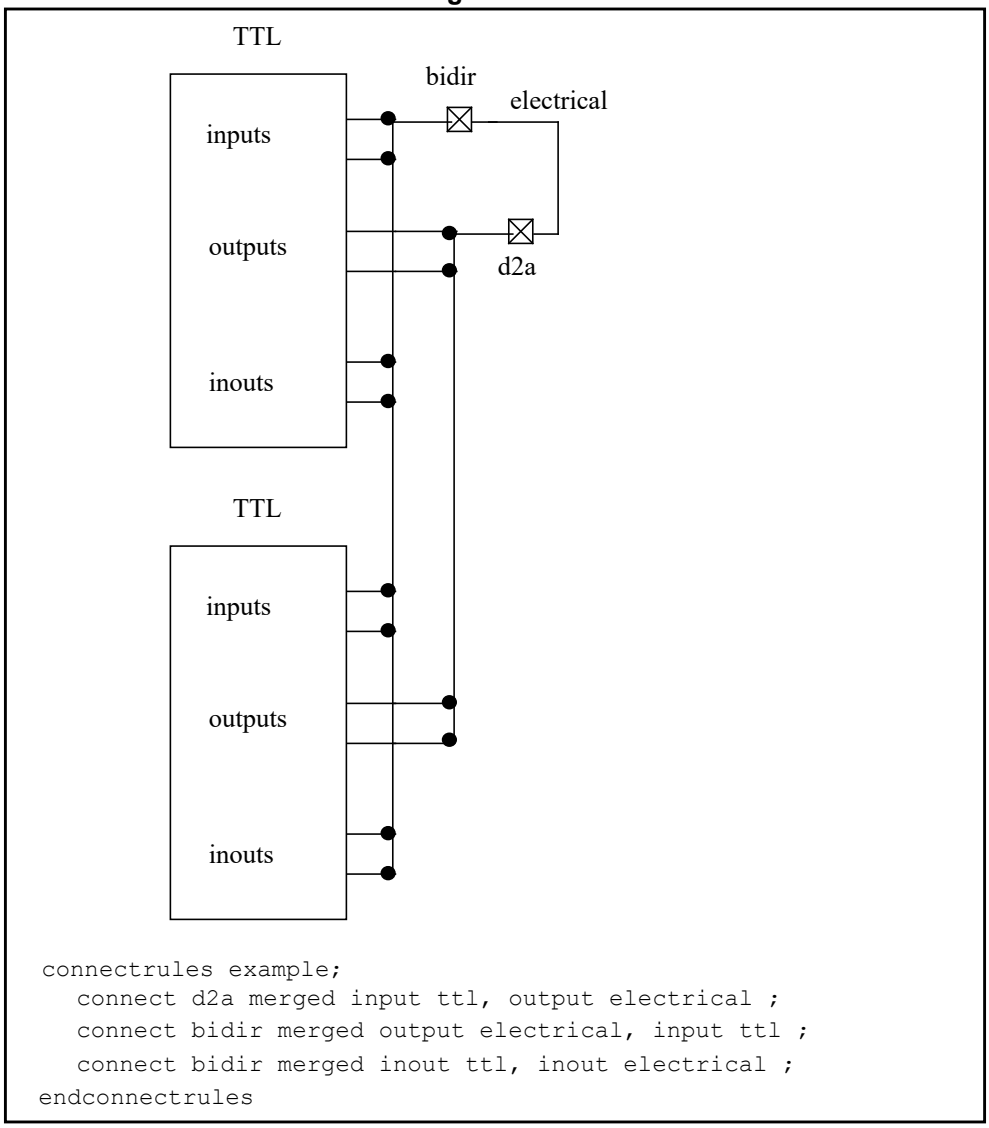
This instructs the simulator to try to group all ports (whether they are input, output, or inout) and to use just one connector module, provided the module is the same.

Figure 7-9 illustrates the effect of the **`merged`** attribute.

Connection of the `electrical` signal to the `ttl` inout ports and `ttl` input ports results in a single connector module, `bidir`, being inserted between the ports and the `electrical` signal. The `ttl` output ports are merged, but with a different connect module; i.e., there is one connector module inserted between the electrical signal and all of the `ttl` output ports.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Figure 7-8:




```
connectrules example;
   connect d2a merged input ttl, output electrical ;
   connect bidir merged output electrical, input ttl ;
   connect bidir merged inout ttl, inout electrical ;
endconnectrules
```

Figure 7-9: Connector insertion using merged

7.8.3.2 split

If more than one input port is connected at a net of a signal, using **`split`** forces there to be one connect module for each port which converts between the net discipline and the port discipline. In this way, the net connecting to the ports is segmented by the insertion of one connect module for each port.

Example 1:

```
connect elect_to_logic split;
```

This **`connect`** statement specifies the module `elect_to_logic` shall be split across the discrete module ports:

— if an input port has `ddiscrete` discipline and the signal connecting to the port has `electrical` discipline, or

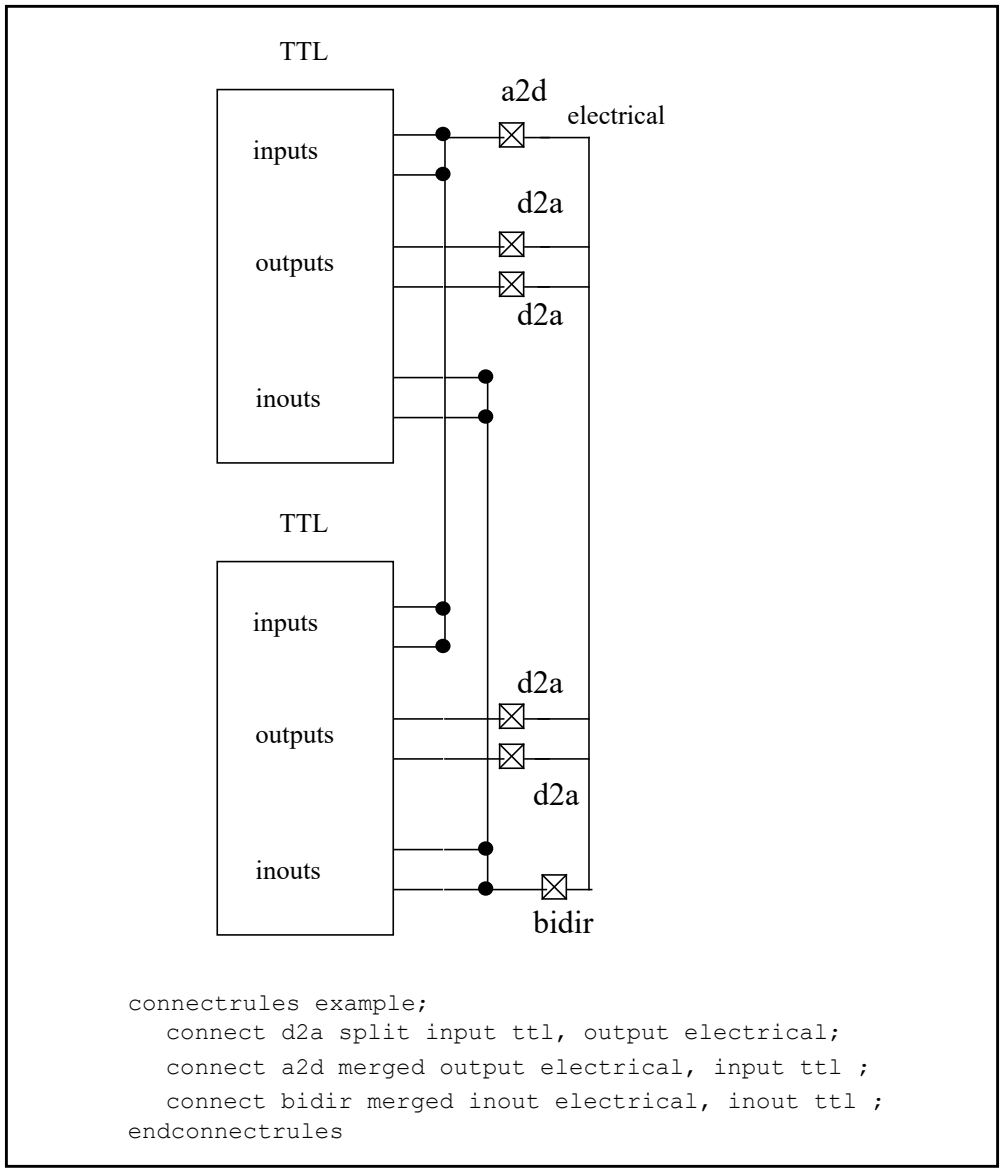
Copyright © 2024 Accellera Systems Initiative. All rights reserved.

— if an output port has `electrical` discipline and the signal connecting to the port has `ddiscrete` discipline.

Example 2:

In Figure 7-10, the connections of an `electrical` signal to `ttl` output ports results in a distinct instance of the `d2a` connect module being inserted for each output port. This is mandated by the **`split`** parameter.

Connection of the `electrical` signal to `ttl` input ports results in a single instance of the `a2d` connect module being inserted between the `electrical` signal and all the `ttl` input ports. This is mandated by merged parameter. This behavior is also seen for the `ttl` inout ports where the **`merged`** parameter is used.




```
connectrules example;
   connect d2a split input ttl, output electrical;
   connect a2d merged output electrical, input ttl ;
   connect bidir merged inout electrical, inout ttl ;
endconnectrules
```

Figure 7-10: Connect module insertion with signal segmentation

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Example 3

```
connect cmosA2d split #(.r(30k) input electrical, output cmos02u;
```

performs three functions:

1) Connects an instance of `cmosA2d` module between a signal with `electrical` discipline and the input port with `cmos02u` discipline, or an output port with `electrical` discipline and the signal with `cmos02u` discipline;
2) Sets the value of the parameter `r` to `30k`; and
3) Uses one module instance for each input port.

If there are many output ports where this rule applies, by definition there is no segmentation of the signal between these ports, since the ports have discipline `electrical` (an analog discipline).

Example 4

```
connect cmosA2d merged #(.r(15k) input electrical, output cmos04u;
```

does three things:

1) Connects an instance of `cmosA2d` module between a signal with `electrical` discipline and an input port with `cmos04u` discipline, or an output port with `electrical` discipline and a signal with `cmos4u` discipline;
2) Sets the value of the parameter `r` to `15k`; and
3) Uses one module instance regardless of the number of ports.

7.8.4 Rules for driver-receiver segregation and connect module selection and insertion

Driver-receiver segregation and connect module insertion is a post elaboration operation. It depends on a complete hierarchical examination of each signal in the design, i.e., an examination of the signal in all the contexts through which it passes. If the complete hierarchy of a signal is digital, i.e., the signal has a digital discipline in all contexts through which is passes, it is a *digital signal* rather than a mixed signal. Similarly, if the complete hierarchy of a signal is analog, it is an *analog signal* rather than a mixed signal. Rules for driver-receiver segregation and connect module insertion apply only to *mixed signals*, i.e., signals which have an analog discipline in one or more of the contexts through which they pass and a digital discipline in one or more of the contexts. In this case, *context* refers to the appearance of a signal in a particular module instance.

For a particular signal, a module instance has a digital context if the signal has a digital discipline in that module or an analog context if the signal has an analog discipline. The appearance of a signal in a particular context is referred to as a *segment* of the signal. In general, a *signal* in a fully elaborated design consists of various segments, some of which can be analog and some of which can be digital.

A *port* represents a connection between two net segments of a signal. The context of one of the net segments is an instantiated module and the context of the other is the module which instantiates it. The segment in the instantiated module is called the *lower* or *formal connection* and the segment in the instantiating module is the *upper* or *actual connection*. A connection element is selected for each port where one connection is analog and the other digital.

The following rules govern driver-receiver segregation and connect module selection. These rules apply only to mixed signals.

1) A mixed signal is represented in the analog domain by a single node, regardless of how its analog contexts are distributed hierarchically.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

2) Digital drivers of mixed signals are segregated from receivers so the digital drivers contribute to the analog state of the signal and the analog state determines the value seen by the receivers.
3) A connection shall be selected for a port only if one of the connections to the port is digital and the other is analog. In this case, the port shall match one (and only one) connect statement. The module named in the connect statement is the one which shall be selected for the port.

Once connect modules have been selected, they are inserted according to the `connect_mode` parameter in the pertinent connect statements. These rules apply to connect module insertion:

1) The connect mode of a port for which a connect module has been selected shall be determined by the value of the `connect_mode` parameter of the `connect` statement which was used to select the connect module.
2) The connect module for a port shall be instantiated in the context of the ports upper connection.
3) All ports connecting to the same signal (upper connection), sharing the same connect module, and having `merged` parameter shall share a single instance of the selected connect module.
4) All other ports shall have an instance of the selected connect module, i.e., one connect module instance per port.

7.8.5 Instance names for auto-inserted instances

Parameters of auto-inserted connect instances can be set on an instance-by-instance basis with the use of the **`defparam`** statement. This requires predictable instance names for the auto-inserted modules.

The following naming scheme is employed to unambiguously distinguish the connector modules for the case of auto-inserted instances.

1) **`merged`**

   In the merged case, one or more ports have a given discipline at their bottom connection, call it `BottomDiscipline`, and a common signal, call it `SigName`, of another discipline at their top connection. A single connect module, call it `ModuleName`, is placed between the top signal and the bottom signals. In this case, the instance name of the connect module is derived from the signal name, module name, and the bottom discipline:

   ```
   SigName__ModuleName__BottomDiscipline
   ```

2) **`split`**

   In the split case, one or more ports have a given discipline at their bottom connection and a common signal of another discipline, call it `TopDiscipline`, at their top connection. One module instance is instantiated for each such port. In this case, the instance name of the connect module is

   ```
   SigName__InstName__PortName
   ```

   where `InstName` and `PortName` are the local instance name of the port and its instance respectively.

NOTE—The __ between the elements of these generated instance names is a double underscore.

7.8.5.1 Port names for Verilog built-in primitives

In the cases of instances of modules and instances of UDPs, port names are well defined. In these cases the port name is the name of the signal at the lower connection of the port. In the case of built-in digital primitives, however, IEEE Std 1364 Verilog does not define port names. In order to support the unique naming of auto inserted connect modules and the ability to override the parameters of those connect modules, built-in digital primitives ports will be provided with predictable names. These names are only for the purpose of

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

naming the connect modules and do not define actual port names. These port names may not be used to instantiate or to do access of these primitives.

The following naming conventions shall be used when generating connect module instance names that are connected to built-in digital primitives.

1) For N-input gates (**and**, **nand**, **nor**, **or**, **xnor**, **xor**) the output will be named `out`, and the inputs reading from left to right will be in1, `in2`, `in3`, and so forth.
2) For N-output gates (**buf**, **not**) The input will be named `in`, and the outputs reading from left to right will be named `out1`, `out2`, `out3`, and so forth.
3) For 3 port MOS switches (**nmos**, **pmos**, **rnmos**, **rpmos**) the ports reading from left to right will be named `source`, `drain`, `gate`.
4) For 4 port MOS switches (**cmos**, **rcmos**) the ports reading from left to right will be named `source`, `drain`, `ngate`, `pgate`.
5) For bidirectional pass switches (**tran**, **tranif1**, **tranif0**, **rtran**, **rtranif1**, rtranif) the ports reading from left to right will be named `source`, `drain`, `gate`.
6) For single port primitives (**pullup**, **pulldown**) the port will be named `out`.

7.8.6 Supply sensitive connect module examples

The connect modules described so far in Clause 7 use a constant parameter value to set the supply and threshold voltage levels used in the behavioral blocks of each module. When we need to consider the time dependent effect of supplies on the switching behavior of connect modules then using elaboration time constants is not sufficient. The following example demonstrates how a string parameter can be used to hierarchically access a branch quantity (see 9.20) so that the connect modules are now dependent upon a supply voltage defined elsewhere in the design,

```
module dig_inv(in, out);
   input in;
   output out;
   reg out;
   ddiscrete in, out;
   always begin
      out = #10 ~in;
   end
endmodule

module analog_inv(in, out, vdd);
   input in;
   output out;
   electrical in, out;
   electrical vdd;
   real outval;
   analog begin
      if (V(in) > V(vdd)/2 )
         outval = 0;
      else
         outval = V(vdd) ;

      V(out) <+ transition(outval);
   end
endmodule


module global_supply;
   electrical vdd;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   analog V(vdd) <+ 5.0;
endmodule

module ring;
   dig_inv d1 (n1, n2);
   dig_inv d2 (n2, n3);
   analog_inv a3 (n3, n1, $root.global_supply.vdd);
endmodule

connectmodule elect_to_logic(el, cm);
   input el;
   output cm;
   reg cm;
   electrical el;
   ddiscrete cm;
   always
      @(cross(V(el) - V($root.global_supply.vdd)/2.0, 1))
         cm = 1;

   always
      @(cross(V(el) - V($root.global_supply.vdd)/2.0, -1))
         cm = 0;
endmodule

connectmodule logic_to_elect(cm, el);
   input cm;
   output el;
   ddiscrete cm;
   electrical el;
   analog
      V(el) <+ V($root.global_supply.vdd) * transition((cm == 1) ? 1 : 0);
endmodule

connectrules mixedsignal;
   connect elect_to_logic;
   connect logic_to_elect;
endconnectrules
```

The additional top level module `global_supply`, now defines a supply voltage `vdd`. The connect modules access this supply via a hierarchical reference,

```
$root.global_supply.vdd
```

They are now sensitive to changes in the supply as the simulation proceeds. In this example the name of the supply is hard coded into the module. Using the analog node alias system functions (see 9.20), a more generic connect module may be written where the supply name is provided via a string parameter. This parameter may be set via connect rules.

```
module dig_inv(in, out);
   input in;
   output out;
   reg out;
   ddiscrete in, out;
   always out = #10 ~in;
endmodule

module analog_inv(in, out, vdd);
   input in;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   output out;
   electrical in, out;
   electrical vdd;
   real outval;
   analog begin
      if (V(in) > V(vdd)/2.0)
         outval = 0;
      else
         outval = V(vdd);

      V(out) <+ transition(outval);
   end
endmodule

module global_supply;
   electrical vdd;
   analog V(vdd) <+ 5.0;
endmodule

module ring;
   dig_inv d1 (n1, n2);
   dig_inv d2 (n2, n3);
   analog_inv a3 (n3, n1, $root.global_supply.vdd);
endmodule

connectmodule elect_to_logic(el, cm);
   input el;
   output cm;
   reg cm;
   electrical el;
   ddiscrete cm;
   electrical vdd;
   parameter string vddname = "(not_given)"; // Set via the CR
   analog initial begin
      if($analog_node_alias(vdd, vddname) == 0)
         $error("Unable to resolve power supply: %s", vddname);
   end

   always @(cross(V(el) - V(vdd)/2.0, 1))
      cm = 1;

   always @(cross(V(el) - V(vdd)/2.0, -1))
      cm = 0;

endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
connectmodule logic_to_elect(cm, el);
   input cm;
   output el;
   ddiscrete cm;
   electrical el;
   electrical vdd;
   parameter string vddname = "(not_given)"; // Set via the CR
   analog initial begin
      if($analog_node_alias(vdd, vddname) == 0)
         $error("Unable to resolve power supply: %s", vddname);
   end

   analog V(el) <+ V(vdd) * transition((cm == 1) ? 1 : 0);
endmodule

connectrules mixedsignal;
   connect elect_to_logic #(.vddname("$root.global_supply.vdd"));
   connect logic_to_elect #(.vddname("$root.global_supply.vdd"));
endconnectrules
```

When there are multiple supplies in the design distinct disciplines must be specified for each digital net that is associated with a given supply. This may be done by explicitly specifying the discipline of the digital nets or by using remote disciplines. Supply sensitivity in multi supply designs is managed in the same way as above but with the specific supply hierarchical reference provided on each connect rule as the following example shows.

```
`include "disciplines.vams"
`timescale 1ns/1ns

discipline ddiscrete_1v2
   domain discrete;
enddiscipline

discipline ddiscrete_1v8
   domain discrete;
enddiscipline

module global_supply;
   electrical vdd_1v2;
   electrical vdd_1v8;
   analog begin
      V(vdd_1v2) <+ 1.2;
      V(vdd_1v8) <+ 1.8;
   end
endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
module analog_inv(in, out, vdd);
   input in;
   output out;
   electrical in, out;
   electrical vdd;
   real outval;
   analog begin
      if (V(in) > V(vdd)/2.0)
         outval = 0;
      else
         outval = V(vdd);

      V(out) <+ transition(outval);
   end
endmodule

// 1.2v supply level
module dig_inv_1v2(in, out);
   input in;
   output out;
   ddiscrete_1v2 in, out;
   reg out;
   always out = #10 ~in;
endmodule

module ring_1v2;
   dig_inv_1v2 d1(n1, n2);
   dig_inv_1v2 d2(n2, n3);
   analog_inv a3 (n3, n1, $root.global_supply.vdd_1v2);
endmodule

// 1.8v supply level
module dig_inv_1v8(in, out);
   input in;
   output out;
   ddiscrete_1v8 in, out;
   reg out;
   always out = #10 ~in;
endmodule

module ring_1v8;
   dig_inv_1v8 d1(n1, n2);
   dig_inv_1v8 d2(n2, n3);
   analog_inv a3 (n3, n1, $root.global_supply.vdd_1v8);
endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
connectmodule elect_to_logic(el, cm);
   input el;
   output cm;
   reg cm;
   electrical el;
   ddiscrete cm;
   electrical vdd;
   parameter string vddname = "(not_given)"; // Set via the connect rule
   analog initial begin
      if($analog_node_alias(vdd, vddname) == 0)
         $error("Unable to resolve power supply: %s", vddname);
   end

   always @(cross(V(el) - V(vdd)/2.0, 1))
      cm = 1;

   always @(cross(V(el) - V(vdd)/2.0, -1))
      cm = 0;

endmodule

connectmodule logic_to_elect(cm, el);
   input cm;
   output el;
   ddiscrete cm;
   electrical el;
   electrical vdd;
   parameter string vddname = "(not_given)"; // Set via the connect rule
   analog initial begin
      if($analog_node_alias(vdd, vddname) == 0)
         $error("Unable to resolve power supply: %s", vddname);
   end

   analog V(el) <+ V(vdd) * transition((cm == 1) ? 1 : 0);
endmodule

connectrules mixedsignal;
   connect elect_to_logic #(.vddname("$root.global_supply.vdd_1v2"))
            input electrical, output ddiscrete_1v2;

   connect logic_to_elect #(.vddname("$root.global_supply.vdd_1v2"))
            input ddiscrete_1v2, output electrical;

   connect elect_to_logic #(.vddname("$root.global_supply.vdd_1v8"))
            input electrical, output ddiscrete_1v8;

   connect logic_to_elect #(.vddname("$root.global_supply.vdd_1v8"))
            input ddiscrete_1v8, output electrical;
endconnectrules
```

In the above examples the supply hierarchical reference is specified as an absolute name via the **$root** prefix (see 6.2.1). If **$root** is not supplied then the hierarchical reference search proceeds in the usual way as defined in 6.7. The hierarchical reference search will start from the connect module insertion point. This should be taken into consideration when naming the supplies as changes to discipline resolution controls will affect the connect module location and therefore may change how the supply hierarchical reference resolves.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

7.9 Driver-receiver segregation

If the hierarchical segments of a signal are all digital or all analog, the signal is not a mixed signal and the internal representation of the signal does not differ from that of a purely digital or an analog signal.

If the signal has both analog and digital segments in its hierarchy, it is a mixed signal. In this case, the appropriate conversion elements are inserted, either manually or automatically, based on the following rules.

— All the analog segments of a mixed signal are representations of a single analog node.
— Each of the non-contiguous digital segments of a signal shall be represented internally as a separate digital signal, with its own state.
— Each non-contiguous digital segment shall be segregated into the collection of drivers of the segment and the collection of receivers of the segment.

In the digital domain, signals can have drivers and receivers. A driver makes a contribution to the state of the signal. A receiver accesses, or reads, the state of the signal. In a pure digital net, i.e., one without an analog segment, the simulation kernel resolves the values of the drivers of a signal and it propagates the new value to the receivers by means of an event when there is a change in state.

In the case of a mixed net, i.e., one with digital segments and an analog segment, it can be useful to propagate the change to the analog simulation kernel, which can then detect a threshold crossing, and then propagate the change in state back to the digital kernel. This, among other things, allows the simulation to account for rise and fall times caused by analog parasitics.

Within digital segments of a mixed-signal net, drivers and receivers of ordinary modules shall be segregated, so transitions are not propagated directly from drivers to receivers, but propagate through the analog domain instead. In this case, the drivers and receivers of connect modules shall be oppositely segregated; i.e., the connect module drivers shall be grouped with the ordinary module receivers and the ordinary module drivers shall be grouped with the connect module receivers.

Thus, digital transitions are propagated from drivers to receivers by way of analog (through using connect module instances). Figure 7-11 shows driver-receiver segregation in modules having bidirectional and unidirectional ports, respectively.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

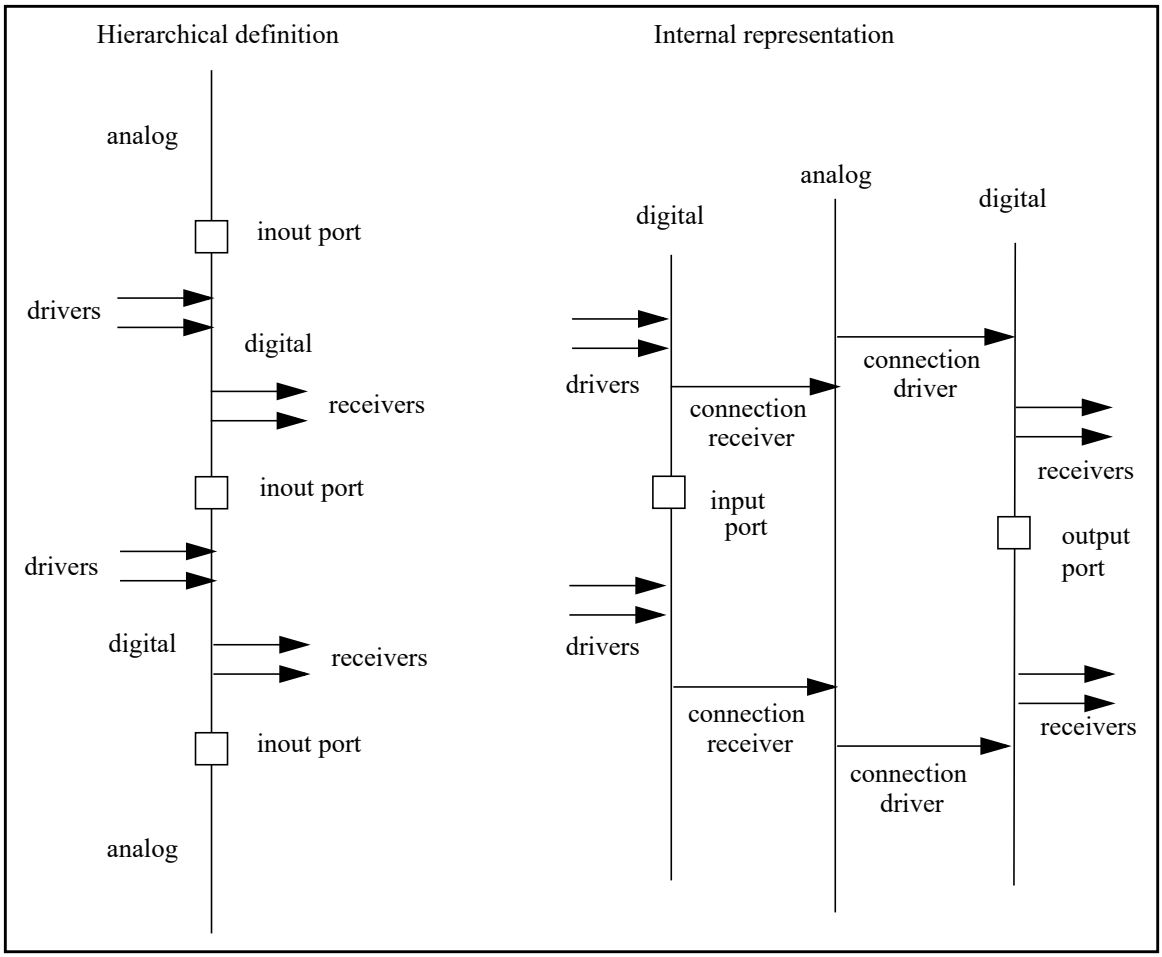


Figure 7-11: Driver-receiver segregation in modules with bidirectional ports

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

8. Scheduling semantics

8.1 Overview

This clause details the simulation cycles for analog simulation and mixed A/D simulations.

A mixed-signal simulator shall contain an analog solver that complies with the analog simulation cycle described in 8.3. This component of the mixed-signal simulator is termed the analog engine. A mixed signal simulator shall also contain a discrete event simulator that complies with the scheduling semantics described in 8.5. This component is termed the digital engine.

In a mixed-signal circuit, an *analog macro-process* is a set of continuous nodes that must be solved together because they are joined by analog blocks or analog primitives. A mixed-signal circuit can comprise one or more analog macro-process separated by digital processes.

8.2 Simulation initialization

Before simulation of the network or system can be proceed, initialization of the system must first be performed as outlined in Figure 8-1.

The system initialization is divided into three main processes, compilation, elaboration, and simulation. Compilation refers to the process where the design artifacts are incorporated into the simulator. Elaboration is the process where the system is hierarchically instantiated. During this process, as each design element is elaborated, parameter declaration assignments are evaluated and module-level generate constructs expanded. Once the system has been elaborated, the simulator will move onto the simulation process. It is during this process that module-level variable declaration assignments are evaluated followed by the execution of **`analog initial`** blocks. Once **`analog initial`** blocks are evaluated, analog net declaration assignments and simulator nodeset values are then applied.

At this point, the system is initialized and the simulation cycle will proceed as detailed in 8.3.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

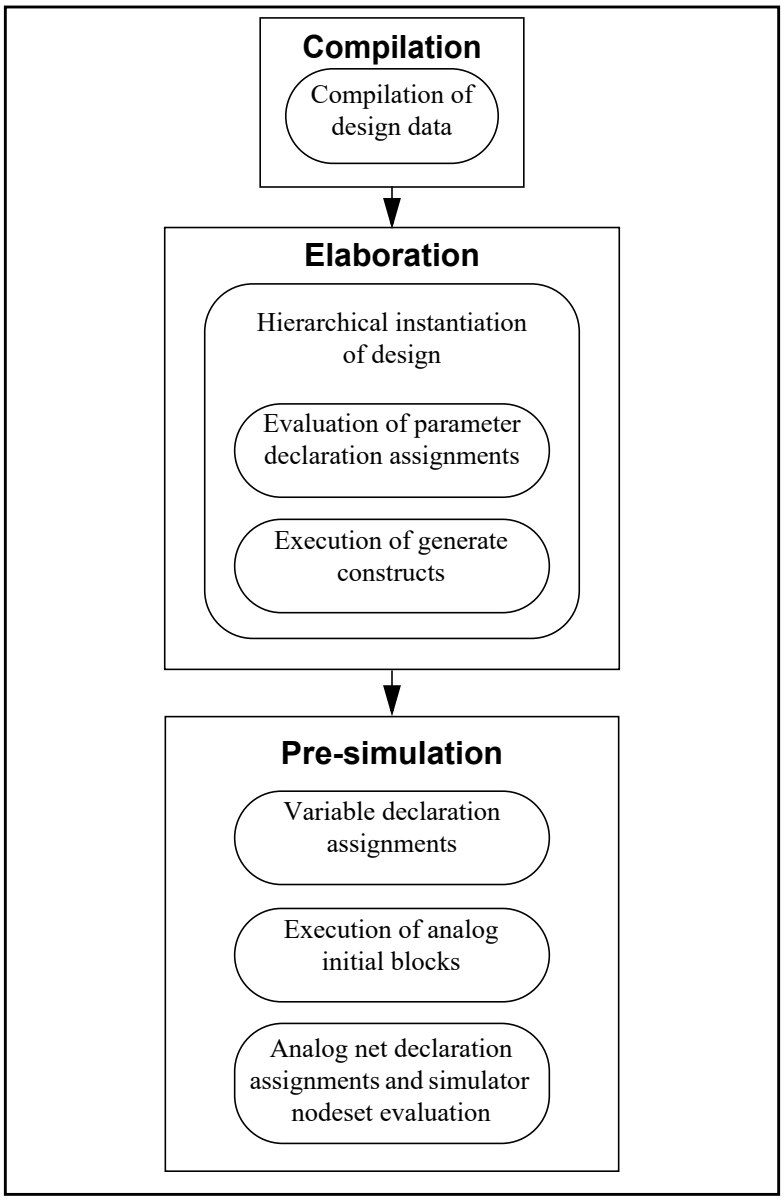


Figure 8-1: System initialization flow

It is important to note that for parametric sweep type analyses like dc sweep, the tool shall re-evaluate the Elaboration and Pre-simulation steps as outlined in Figure 8-1 for each sweep point to ensure that all parameter value changes are captured.

8.3 Analog simulation cycle

Simulation of a network, or system, starts with an analysis of each node to develop equations which define the complete set of values and flows in a network. Through transient analysis, the value and flow equations are solved incrementally with respect to time. At each time increment, equations for each signal are iteratively solved until they converge on a final solution.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

8.3.1 Nodal analysis

To describe a network, simulators combine constitutive relationships with Kirchhoff's Laws in *nodal analysis* to form a system of differential-algebraic equations of the form

$$f(v, t) = \frac{dq(v, t)}{dt} + i(v, t) = 0$$

$$v(0) = v_0$$

These equations are a restatement of Kirchhoff's Flow Law (KFL).

v is a vector containing all node values

t is time

q and i are the dynamic and static portions of the flow

$f( )$ is a vector containing the total flow out of each node

v_0 is the vector of initial conditions

This equation was formulated by treating all nodes as being conservative (even signal flow nodes). In this way, signal-flow and conservative terminals can be connected naturally. However, this results in unnecessary KFL equations for those nodes with only signal-flow terminals attached. This situation is easily recognized and those unnecessary equations are eliminated along with the associated flow unknowns, which shall be zero (`0`) by definition.

8.3.2 Transient analysis

The equation describing the network is differential and non-linear, which makes it impossible to solve directly. There are a number of different approaches to solving this problem numerically. However, all approaches discretize time and solve the nonlinear equations iteratively, as shown in Figure 8-2.

The simulator replaces the time derivative operator (*dq/dt*) with a discrete-time finite difference approximation. The simulation time interval is discretized and solved at individual time points along the interval. The simulator controls the interval between the time points to ensure the accuracy of the finite difference approximation. At each time point, a system of nonlinear algebraic equations is solved iteratively. Most circuit simulators use the Newton-Raphson (NR) method to solve this system.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

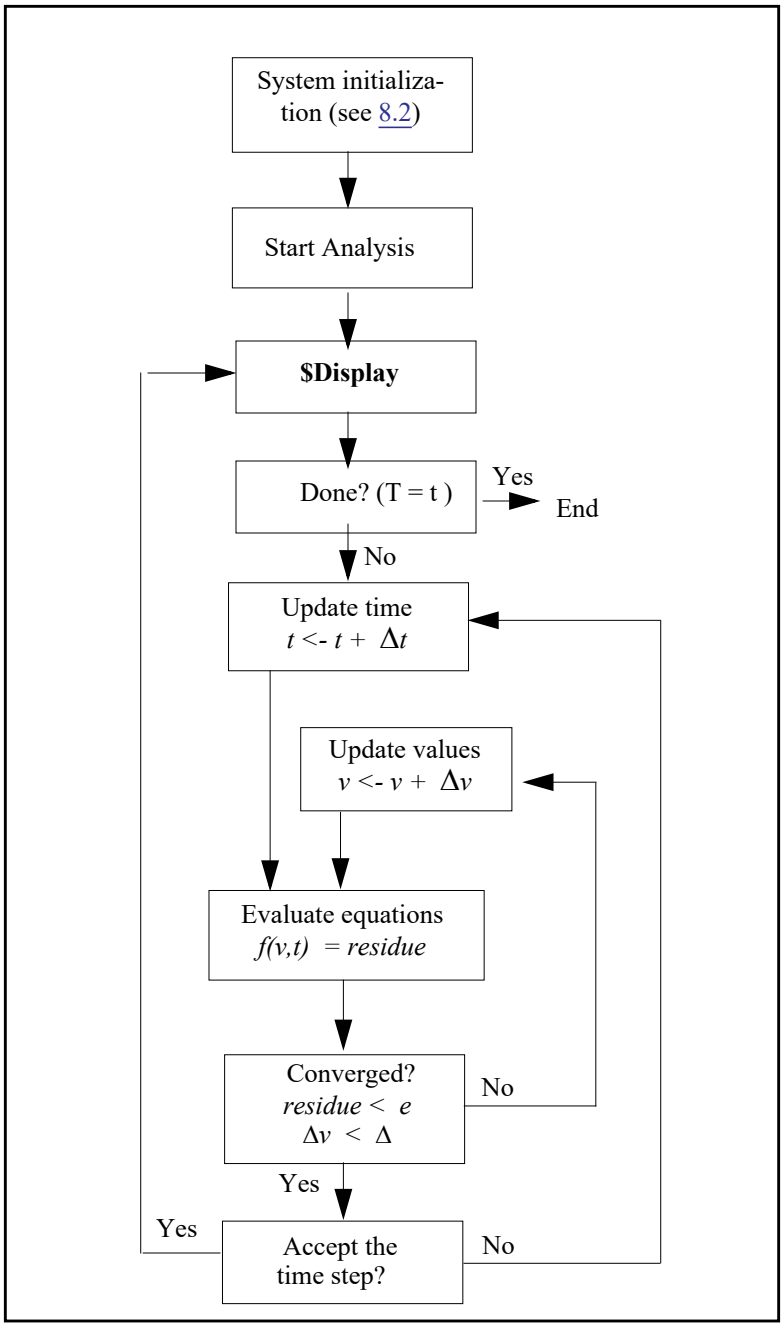


Figure 8-2: Simulation flowchart (transient analysis)

8.3.3 Convergence

In the analog kernel, the behavioral description is evaluated iteratively until the NR method converges. On the first iteration, the signal values used in expressions are approximate and do not satisfy Kirchhoff's Laws.

In fact, the initial values might not be reasonable, so models need to be written so they do something reasonable even when given unreasonable signal values.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

For example, the log or square root of a signal value is being computed, some signal values cause the arguments to these functions to become negative, even though a real-world system never exhibits negative values.

As the iteration progresses, the signal values approach the solution. Iteration continues until two convergence criteria are satisfied. The first criterion is the proposed solution on this iteration, $v^{(j)}(t)$, shall be close to the proposed solution on the previous iteration, $v^{(j-1)}(t)$, and

$$| v_n^{(j)} - v_n^{(j-1)} | < reltol\ (max(| v_n^{(j)}| , |v_n^{(j-1)}|)) + abstol$$

where *reltol* is the relative tolerance and *abstol* is the absolute tolerance.

reltol is set as a simulator option and typically has a value of `0.001`. There can be many absolute tolerances, which one is used depends on the quantity the signal represents (volts, amps, etc.). The absolute tolerance is important when v_n is converging to zero (`0`). Without *abstol*, the iteration never converges.

The second criterion ensures Kirchhoff's Flow Law is satisfied:

$$\left|\sum_n f_n(v^{(j)})\right| < reltol(max(|f^i{}_n(v^{(j)})|)) + abstol$$

where $f_n^i(v^{(j)})$ is the flow exiting node *n* from branch *i*.

Both of these criteria specify the absolute tolerance to ensure convergence is not precluded when v_n or $f_n(v)$ go to zero (`0`). The relative tolerance can be set once in an options statement to work effectively on any node in the circuit, but the absolute tolerance shall be scaled appropriately for its associated signal. The absolute tolerance shall be the largest signal value which is considered negligible on all the signals where it is associated.

The simulator uses absolute tolerance to get an idea of the scale of signals. Absolute tolerances are typically 1,000 to 1,000,000 times smaller than the largest typical value for signals of a particular quantity. For example, in a typical integrated circuit, the largest potential is about 5 volts, so the default absolute tolerance for voltage is 1μV. The largest current is about 1mA, so the default absolute tolerance for current is 1pA.

8.4 Mixed-signal simulation cycle

This section describes the semantics of the initialization, the process of mixed-signal DC analysis, and the synchronization of analog and digital in transient analysis for Verilog-AMS simulation.

8.4.1 Circuit initialization

The initialization phase of mixed-signal simulation is the process of initializing the circuit state for analysis tasks such as DC, transient, and AC. It is a one time execution of nodeset statements (3.6.3.2), then the procedural statements in **`analog initial`** block, and then the procedural statements in the Verilog initial block for time zero. These procedures can also be used for assertion of circuit/module parameters and initial state.

8.4.2 Mixed-signal DC analysis

Mixed-signal DC analysis is the process of finding the steady state of the circuit, which is the DC operating point for transient and AC analysis. The steady state of the digital circuit is defined as the final state at time 0 when all analog and digital events are executed. For mixed-signal DC analysis, the processes of the analog

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

DC analysis and the digital simulation at time 0 are executed iteratively, starting with the initialization phase (including analog and digital) defined in circuit initialization (8.4.1), until all signals at the A/D boundaries reach steady state. The signal propagation at the A/D boundaries follows the same scheduling semantics as are defined in transient analysis in the following sections.

8.4.3 Mixed-signal transient analysis

A Verilog-AMS simulation consists of a number of analog and digital processes communicating via events, shared memory and conservative nodes. Analog processes that share conservative nodes are "solved" jointly and can be viewed as a "macro" process, there may be any number "macro" processes, and it is left up to the implementation whether it solves them in a single matrix, multiple matrices or uses other techniques but it should abide by the accuracy stipulated in the disciplines and analog functions.

8.4.3.1 Concurrency

Most (current) simulators are single-threaded in execution, meaning that although the semantics of Verilog-AMS imply processes are active concurrently, the reality is that they are not. If an implementation is genuinely multi-threaded, it should not evaluate processes that directly share memory concurrently, as there are no data locking semantics in Verilog-AMS.

8.4.3.2 Analog macro process scheduling semantics

The internal evaluation of an analog macro process is described in 8.3.2. Once the analog engine has determined its behavior for a given time, it must communicate the results to other processes in the mixed signal simulation through events and shared variables. When an analog macro process is evaluated, the analog engine finds a potential "solution" at a future time (the "acceptance time"), and it stores (but does not communicate) values[1] for all the process's nodes up to that time. A "wake up" event is scheduled for the acceptance time of the process, and the process is then inactive until it is either woken up or receives an event from another process. If it is woken up by its own "wake up" event, it calculates a new solution point, acceptance time (and so forth) and deactivates. If it is woken up prior to acceptance time by an event that disturbs its current solution, it will cancel its own "wake up" event, accept at the wake-up time, recalculate its solution and schedule a new "wake up" event for the new acceptance time. The process may also wake itself up early for reevaluation by use of a timer (which can be viewed as just another process).

If the analog process identifies future analog events such as "crossings" or timer events (see 5.10.3) then it will schedule its wake-up event for the time of the first such event rather than the acceptance time. If the analog process is woken by such an analog event it will communicate any related events at that time and deactivate, rescheduling its wake-up for the next analog event or acceptance. Events to external processes generated from analog events are not communicated until the global simulation time reaches the time of the analog event.

If the time to acceptance is infinite then no wake-up event needs to be scheduled[2].

Analog processes are sensitive to changes in all variables and digital signals read by the process unless that access is only in statements 'guarded' by event expressions. For example the following code implements a simple digital to analog convertor:

```
module d2a(val,vo); // 16 bit D->A
   parameter   Vgain = 1.0/65536;
   input       val;
   wire [15:0] val;
```

[1]Or derivatives w.r.t. time used to calculate the values.

[2]The case when all derivatives are zero - the circuit is stable.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   electrical  vo;
   analog begin
      V(vo) <+ Vgain * val;
   end
endmodule
```

The output voltage `V(vo)` is reevaluated when any bit in `val` changes, which is not a problem if all the bits change simultaneously and no 'X' values occur. A practical design would require that the digital value is latched to avoid bad bit sequences, as in the following version:

```
module d2aC(clk,val,vo); // Clocked 16 bit D2A
   parameter   Vgain = 1.0/65536;
   input       clk;
   input       val;
   wire [15:0] val;
   electrical  vo;
   real        v_clkd;
   analog begin
      @(posedge clk) v_clkd = Vgain * val;
      V(vo) <+ v_clkd;
   end
endmodule
```

Since *val* is now guarded by the `@(posedge clock)` expression the **`analog`** block is not sensitive to changes in `val` and only reevaluates when `clk` changes.

Macro processes can be evaluated separately but may be evaluated together[1], in which case, the wake up event for one process will cause the re-evaluation of all or some of the processes. Users should bear this in mind when writing mixed-signal code, as it will mean that the code should be able to handle re-evaluation at any time (not just at its own event times).

8.4.3.3 A/D boundary timing

In the analog kernel, time is a floating point value. In the digital kernel time is an integer value. Hence, A2D events generally do not occur exactly at digital integer clock ticks.

For the purpose of reporting results and scheduling delayed future events, the digital kernel converts analog event times to digital times such that the error is limited to half the precision base for the module where the conversion occurs. For the examples below the timescale is 1ns/1ns, so the maximum scheduling error when swapping a digital module for its analog counterpart will be 0.5ns.

Consequently an A2D event that results in a D2A event being scheduled with zero (`0`) delay, shall have its effect propagated back to the analog kernel with zero (`0`) delay.

[1]This is implementation-dependent.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

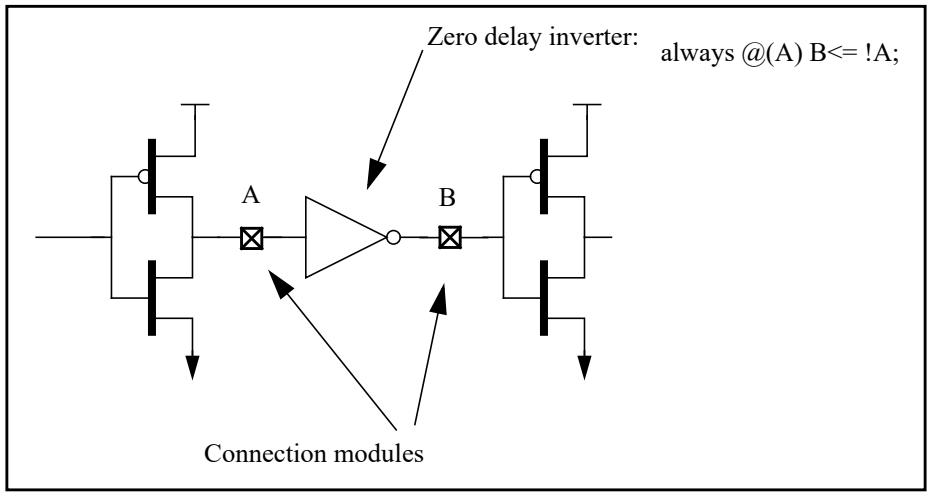


Figure 8-3: A zero delay inverter

If the circuit shown in Figure 8-3 is being simulated with a digital time resolution of `1e-9` (one (`1`) nanosecond) then all digital events shall be reported by the digital kernel as having occurred at an integer multiple of `1e-9`. The A2D and D2A modules inserted are a simple level detector and a voltage ramp generator:

```
connectmodule a2d(i,o);
   parameter vdd = 1.0;
   ddiscrete   o;
   input       i;
   output      o;
   reg         o;
   electrical  i;
   always begin   @(cross(V(i) - vdd/2,+1))o = 1; end
   always begin   @(cross(V(i) - vdd/2,-1))o = 0; end
endmodule
connectmodule d2a(i, o);
   parameter   vdd   =  1.0;
   parameter   slewrate = 2.0/1e-9; // V/s
   input       i;
   output      o;
   electrical  o;
   reg         qd_val,  // queued value
               nw_val;
   real        et;      // delay to event
   real  start_delay;   // .. to ramp start
   always @(driver_update i) begin
      nw_val = $driver_next_state(i,0); // assume one driver
      if (nw_val == qd_val) begin
         // no change (assume delay constant)
      end else begin
         et       = $driver_delay(i,0) * 1e-9; // real delay
         qd_val   = nw_val;
      end
   end
   analog begin
      @(qd_val)   start_delay = et - (vdd/2)/slewrate;
      V(o) <+ vdd * transition(qd_val,start_delay,vdd/slewrate);
   end
endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

If connector `A` detects a positive threshold crossing, the resulting falling edge at connector `B` generated by the propagation of the signal through verilog inverter model shall be reported to the analog kernel with no further advance of analog time. The digital kernel will treat these events as if they occurred at the nearest nanosecond.

Example:

If `A` detects a positive crossing as a result of a transient solution at time 5.2e-9, the digital kernel shall report a rising edge at `A` at time 5.0e-9 and falling edge at `B` at time 5.0e-9, but the analog kernel shall see the transition at `B` begin at time 5.2e-9, as shown in Figure 8-4. D2As fed with zero delay events cannot be preemptive, so the crossover on the return is delayed from the digital event; zero-delay inverters are not physically realizable devices.

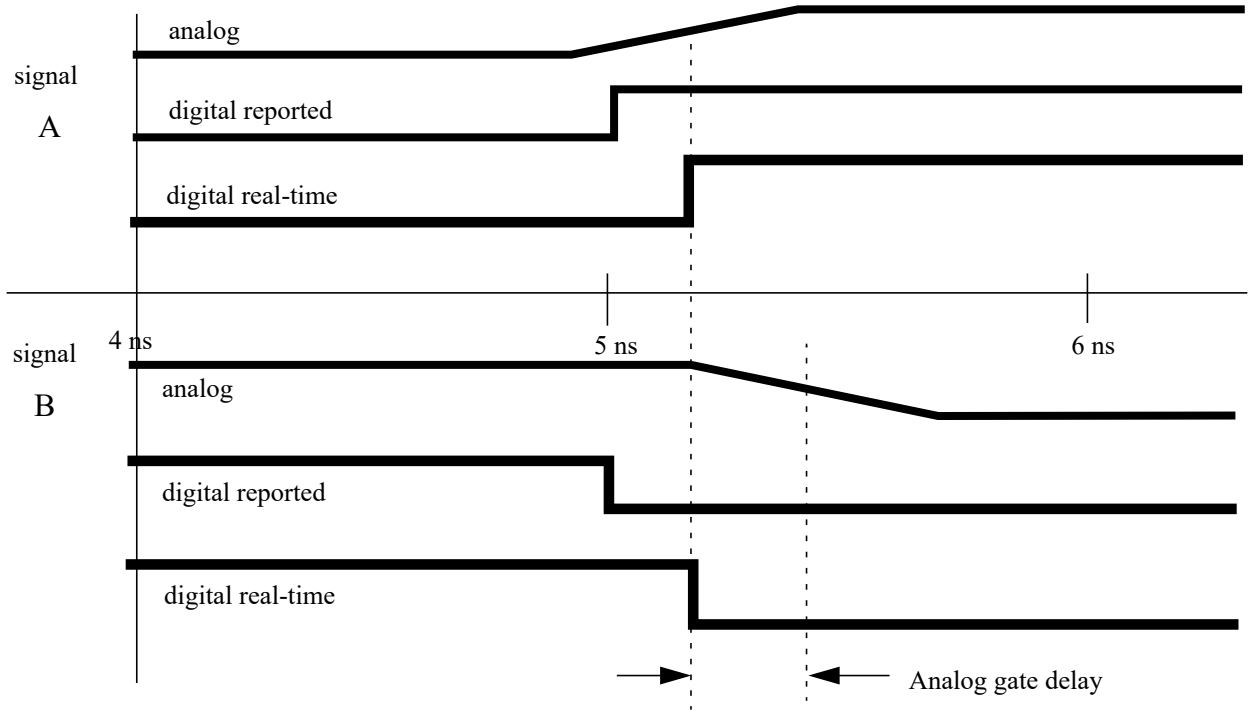


Figure 8-4: Zero delay transient solution times

If the inverter equation is changed to use a one unit delay (`always @(A) B<= #1 !A`), then the timing is as in Figure 8-5.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

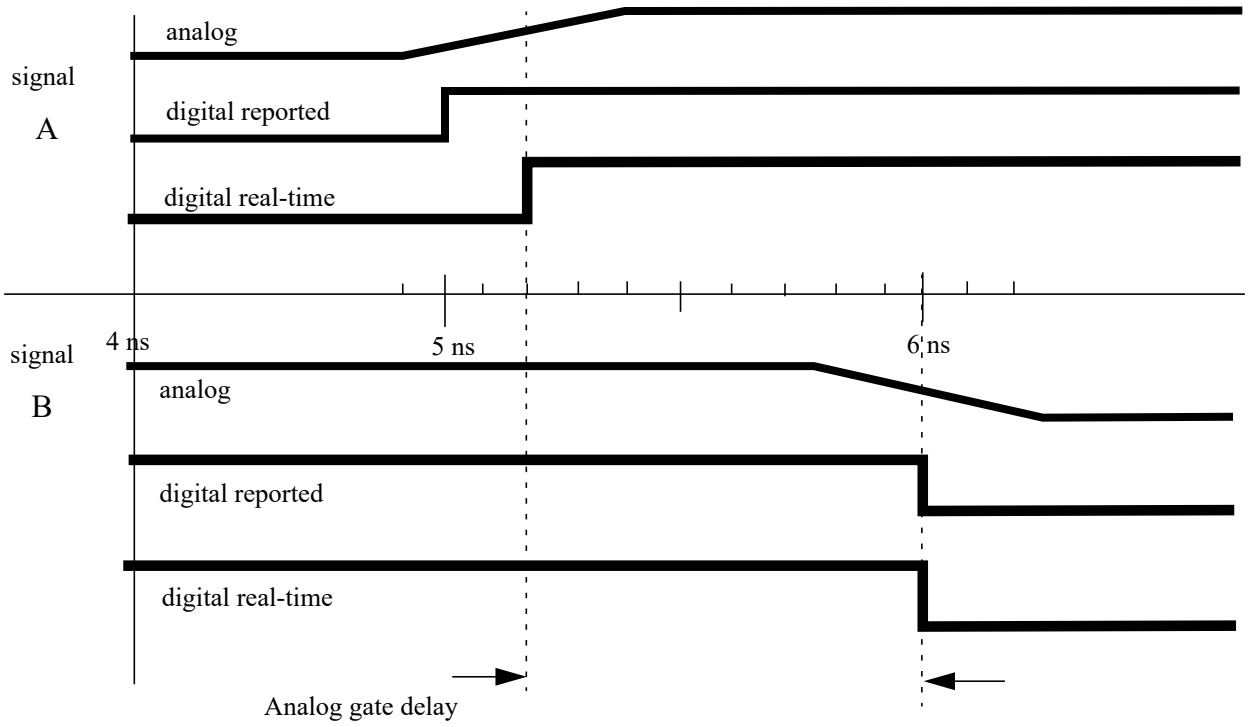


Figure 8-5: Unit delay transient solution times

8.4.4 The synchronization loop

Verilog-AMS uses a “conservative” simulation algorithm, the analog and digital processes that are managed by the simulation kernel are synchronized such that neither computes results that will invalidate signal values that have already been assigned; time never goes backwards. While the implementation of the simulator may have separate event queues for analog and digital events (see 8.4.5), it can be viewed as a single event queue logically with a common global time. Analog processes are similar to Verilog *initial* statements in that they start automatically at time zero. The event sequence for the transient simulation shown in Figure 8-5 would be as follows:

Time	Event Queue
4.9ns	Evaluate the first analog inverter
	Evaluate acceptance at 5.4ns, but schedule wake-up for 5.2 for crossing.
5.2ns	Evaluate crossing event
	The A2D logic sets the digital signal A, which triggers the evaluation of the non-blocking assign to B, which schedules the actual assignment for 6ns (rounded 1ns delay).
	D2A notices queued event and schedules wake-up for 5.75 via rampgen module.
	Schedule wake-up at 5.4ns (as previously calculated).

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

5.4ns Evaluate acceptance

Circuit evaluates stable, nothing scheduled.

5.75ns D2A/rampgen process wake-up

Start ramp in analog domain.

6.0ns Non blocking assign performed (digital event).

D2A may be sensitive, but doesn't need to do anything.

6.25ns D2A/rampgen process wake-up

Drive 0V to complete ramp. Nothing more to schedule.

Any events queued ahead of the current global event time may be canceled. For instance, if the sequence above is interrupted by a change on the primary input before digital assignment takes place as shown in Figure 8-6.

Time Event Queue

4.9ns Evaluating the first analog inverter

Evaluate acceptance at 5.4ns, but schedule wake-up

for 5.2 for crossing.

5.2ns Evaluate crossing event

The A2D logic sets the digital signal A, which triggers the evaluation of the non-blocking assign to B, which schedules the actual assignment for 6ns (rounded 1ns delay).

D2A notices queued event and changes value using transition filter.

Schedule wake-up at 5.4ns (as previously calculated).

5.3ns Analog event disturbs the solution

Accept at 5.3ns.

Cancel 5.4ns wake-up.

New acceptance is 5.45ns, but schedule wake-up for crossing at 5.4ns.

5.4ns Evaluate crossing event

The A2D logic sets the digital signal A, which triggers the evaluation of the non-blocking assign to B, which schedules the actual assignment for 6ns (rounded 1ns delay), canceling previous event.

D2A detects the driver change and *qd_val* toggles back to 1 before the 0 propagates through the transition filter, so no analog change occurs at B.

Schedule wake-up at 5.45ns (as previously calculated).

5.45ns Evaluate acceptance

Circuit evaluates stable, nothing scheduled.

6.00ns Non blocking assign performed (digital event).

Value of B doesn't change.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

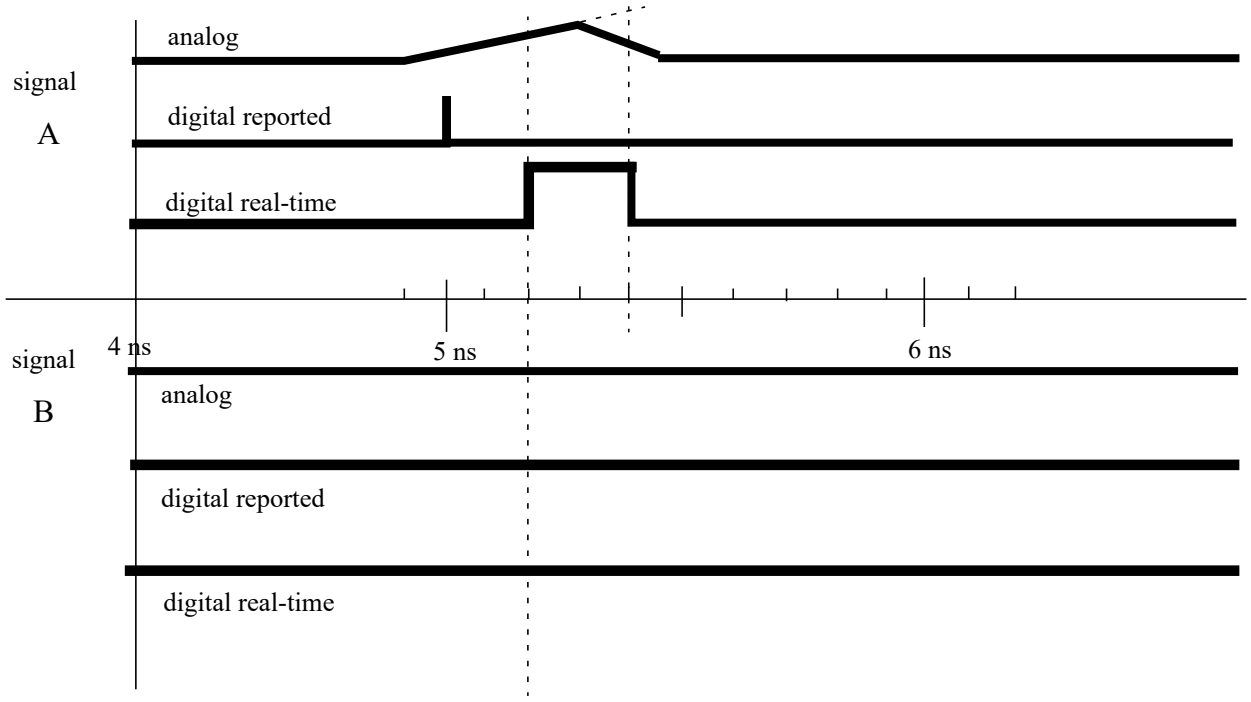


Figure 8-6: Transient solution times with glitch

If the canceling event arrived after the ramp on `B` had started but before the assignment to the digital `B`, it is possible to see the glitch propagate back into the analog domain without an event appearing on `B`.

8.4.5 Synchronization and communication algorithm

Figure 8-7 is an abstract representation of how the analog engine simulating an analog macro process communicates and synchronizes with the digital engine and vice-versa.

The synchronization algorithm can exploit characteristics of the analog and digital kernels described in the next section. The arrows represent an engine moving from one synchronization point to another, which in the case of an analog macro-process involves one or more time-steps and in the case of a digital engine, involves once or more discrete times at which events are processed.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

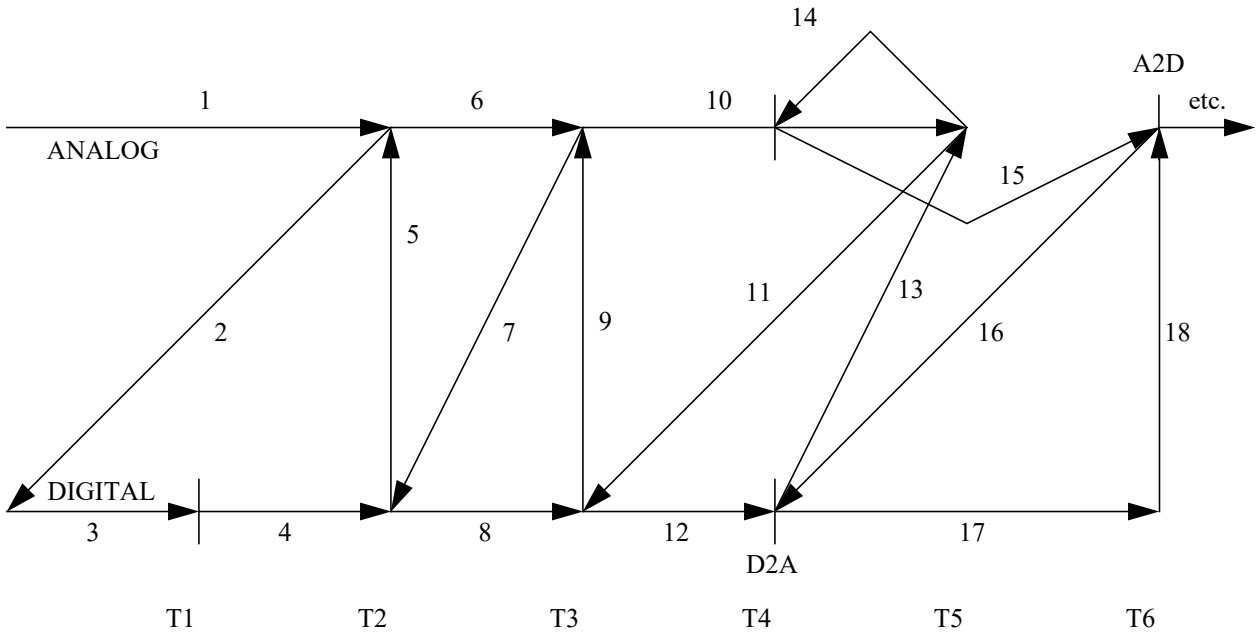


Figure 8-7: Sample run

1) The analog engine begins transient analysis and sends state information (that it is good up to T2) to the digital engine (1, 2).
2) The digital engine begins to run using its own time steps (3); however, if there is no D2A event, the analog engine is not notified and the digital engine continues to simulate until it can not advance its time without surpassing the time of the analog solution (4). Control of the simulation is then returned to the analog engine (5), which accepts at T2. This process is repeated (7, 8, 9, 10, and 11).
3) If the digital engine produces a D2A event (12), control of the simulation is returned to the Analog engine (13). The analog engine accepts at the time of the D2A event (14, which may involve recalculating from T3). The analog engine then calculates the next time step (15).
4) If the analog engine produces an A2D event, it returns control to the digital engine (16), which simulates up to the time of the A2D event, and then surrenders control (17 and 18).
5) This process continues until transient analysis is complete.

8.4.6 absdelta interpolated A2D events

The **`absdelta()`** monitored event function allows the analog solver to generate A2D events by interpolating the times between the last time step and next time step at which the **`absdelta`** expression changed by delta and schedules them in the digital event queue. The digital engine will then consume these events by simulating up to either

a) the time of the next D2A event or
b) the time of the next time step in the analog engine. At this point, it will surrender control to the analog engine.

In the case of a), unconsumed **`absdelta`** A2D events in the digital engine are rejected.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

8.4.7 Assumptions about the analog and digital algorithms

1) Advance of time in a digital algorithm
   a) The digital engine has some minimum time granularity and all digital events occur at a time which is some integer multiple of that granularity.
   b) The digital engine can always accept events for a given simulation time provided it has not yet executed events for a later time. Once it executes events for a given time, it can not accept events for an earlier time.
   c) The digital engine can always report the time of the most recently executed event and the time of the next pending event.
2) Advance of time in an analog algorithm
   a) The analog engine advances time by calculating a sequence of solutions. Each solution has an associated time which, unlike the digital time, is not constrained to a particular minimum granularity.
   b) The analog engine can not tell for certain the time when the next solution converges. Thus, it can tell the time of the most recently calculated solution, but not the time of the next solution.
   c) In general, the analog solution is a function of one or more previous solutions. Having calculated the solution for a given time, the analog engine can either accept or reject that solution; it cannot calculate a solution for a future time until it has accepted the solution for the current time.
3) Analog to digital events
   a) Certain analog events (**`above, cross`**, **`initial_step`**, and **`final_step`**) cause an analog solution of the time where they occur. Such events are associated with the solution that produced them until they are consumed by the digital engine. Until then, they can be rejected along with the solution, if it is rejected.
   b) **`absdelta`** analog to digital events can occur at times interpolated between two analog solution times (that of the last analog solution and the next analog solution). These events are associated with the next analog solution. If the next analog solution is rejected, the **`absdelta`** events associated with that solution, that have not be consumed by the digital engine, are rejected (see 8.4.6).
4) Digital to analog events shall cause an analog solution of the time where they occur.

8.5 Scheduling semantics for the digital engine

The scheduling semantics for Verilog-HDL simulation are outlined in Clause 11 of IEEE Std 1364 Verilog.

The digital engine of a Verilog-AMS mixed-signal simulator shall comply with that section except for the changes outlined in this section.

For mixed-signal simulation, the major change from Clause 11 of IEEE Std 1364 Verilog is that two new types of event must be supported by the event queue called the *explicit D2A* (digital-to-analog) *event,* and the *analog macro-process event*.

Explicit D2A events are created when a digital event occurs to which an **`analog`** block is *explicitly sensitive*. An **`analog`** block is explicitly sensitive to event expressions mentioned in an event control statement in that **`analog`** block.

Similarly, there is also the concept of the *implicit D2A event* that is created when a digital variable to which an **`analog`** block is *implicitly sensitive* changes value. An **`analog`** block is implicitly sensitive to all digital variable references that are not guarded by event control statements in that **`analog`** block.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

An analog macro-process event is also created when either type of D2A event occurs. The analog macro-process event is associated with the analog macro-process that is sensitive to the D2A event. An analog macro-process event is evaluated by calling the analog engine to solve it. Note that implicit D2A events are not added to the stratified event queue, but as they directly cause an analog macro-process event, they effectively force a digital-analog synchronization when they occur.

8.5.1 The stratified event queue

The Verilog event queue is logically segmented into *seven* different regions. Events are added to any of the seven regions but are only removed from the *active* region. Regions 1b and 3b have been added for mixed-signal simulation.

1. Events that occur at the current simulation time and can be processed in any order. These are the *active* events.

1b. Explicit D2A events that occur at the current simulation time shall be processed after all the active events are processed.

2. Events that occur at the current simulation time, but that shall be processed after all the active and explicit D2A 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, explicit D2A and inactive events are processed. These are the non blocking assign update events.

3b. Analog macro-process events shall be processed after all active, explicit D2A events, inactive events and non blocking assign update events are processed.

4. Events that shall be processed after all the active, explicit D2A, inactive, non blocking assign update events and analog macro-process 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 non blocking 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 IEEE Std 1364 Verilog.

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 of IEEE Std 1364 Verilog) creates a non blocking assign update event, scheduled for current or a later simulation time.

The **`$monitor`**, **`$strobe`** and **`$debug`** system tasks (see 17.1 of IEEE Std 1364 Verilog) 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 Section 25.58 of IEEE Std 1364 Verilog) or `vpi_register_cb(cb_readwrite)` (see 27.33 of IEEE Std 1364 Verilog) shall be treated as inactive events.

Note that A2D events must be analog event controlled statements (e.g., **`@cross`**, **`@timer`**). These are scheduled just like other event controlled statements in Verilog-HDL (e.g., **`@posedge`**).

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

8.5.2 The Verilog-AMS digital engine reference model

In all the examples that follow, T refers to the current simulation time of the digital engine, 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 explicit D2A events) {
         activate all explicit D2A events;
      }else if (there are non blocking assign update events){
         activate all non blocking assign update events;
      }else if (there are analog macro-process events) {
         activate all analog macro-process 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 if (E is a D2A event) {
      evaluate the D2A
      modify the analog values
      add A2D events to event queue, if any
   }else if (E is an analog macro-process event) {
      evaluate the analog macro-process
      modify the analog values
      add A2D events to event queue, if any
   }else {/*shall be an evaluation event */
      evaluate the process;
      add update events to the event queue;
   }
}
```

8.5.3 Scheduling implication of assignments

Assignments are translated into processes and events as follows.

8.5.3.1 Continuous assignment

A continuous assignment statement (6.1 of IEEE Std 1364 Verilog) 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.

8.5.3.2 Procedural continuous assignment

A procedural continuous assignment (which is the **`assign`** or **`force`** statement; see 9.3 of IEEE Std 1364 Verilog) 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.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

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

8.5.3.3 Blocking assignment

A blocking assignment statement (see 9.2.1 of IEEE Std 1364 Verilog) 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.

8.5.3.4 Non blocking assignment

A nonblocking assignment statement (see 9.2.2 of IEEE Std 1364 Verilog) 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.

8.5.3.5 Switch (transistor) processing

The event-driven simulation algorithm described in 11 of IEEE Std 1364 Verilog 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 IEEE Std 1364 Verilog 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 IEEE Std 1364 Verilog 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 non-conducting 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.

8.5.3.6 Processing explicit D2A events (region 1b)

An explicit D2A event is processed by evaluating the **analog** block that is sensitive to this event. This is so that the values used for the digital variables referenced inside the explicitly sensitive event control statement in the analog block are the values of those variables after region 1 has been processed, not the values of those variables just before region 3b is processed.

8.5.3.7 Processing analog macro-process events (region 3b)

An analog macro-process event is evaluated by calling the analog engine to solve the associated analog macro-process. Note that if multiple events for a particular analog macro-process are active, then a single evaluation of the analog macro-process shall consume all of these events from the queue.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The reason for processing analog macro-processes after regions 1-3 have been processed is to minimize the number of times analog macro-processes are evaluated, because such evaluations tend to be expensive.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

9. System tasks and functions

9.1 Overview

Verilog-AMS HDL is a superset of IEEE Std 1364 Verilog and hence all the system tasks in IEEE Std 1364 Verilog are supported. Verilog-AMS adds several system tasks and system functions. These are described in this clause. In addition, Verilog AMS HDL extends the behavior of several Verilog systems tasks and functions including allowing some of them to be used in an analog context.

The system task and functions support by Verilog-AMS HDL are categorized in 9.2. A subclause is devoted to each category from 9.4 until the end of this clause.

The behavior of a system task or function which is allowed in an analog context will be described in the context of the analog simulation cycle (9.3) if required in the relevant section for that system task or function.

9.2 Categories of system tasks and functions

This subclause describes system tasks and functions that are considered part of the Verilog-AMS HDL. It also states whether a particular system task or function is supported in the digital context and if it is supported in the analog context. The system tasks and functions are divided into various categories. Each category has a table describing the support level in Verilog-AMS HDL for the system task or functions in that category.

Table 9-1—Display system tasks

Task name	Supported in digital context	Supported in analog context
$display	Yes	Yes
$displayb, $displayh, $displayo	Yes	No
$strobe	Yes	Yes
$strobeb, $strobeh, $strobeo	Yes	No
$write	Yes	Yes
$writeb, $writeh, $writeo	Yes	No
$monitor	Yes	Yes
$monitorb, $monitorh, $monitoro	Yes	No
$monitoron, $monitoroff	Yes	No
$debug	No	Yes

Table 9-2—File input-output system tasks and functions

Task/function name(s)	Supported in digital context	Supported in analog context
$fclose, $fopen	Yes	Yes
$fdisplay	Yes	Yes
$fdisplayb, $fdisplayh, $fdisplayo	Yes	No

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 9-2—File input-output system tasks and functions *(continued)*

Task/function name(s)	Supported in digital context	Supported in analog context
$fwrite	Yes	Yes
$fwriteb, $fwriteh, $fwriteo	Yes	No
$fstrobe	Yes	Yes
$fstrobeb, $fstrobeh, $fstrobeo	Yes	No
$fmonitor	Yes	Yes
$fmonitorb, $fmonitorh, $fmonitoro	Yes	No
$fgetc, $ungetc	Yes	No
$fgets	Yes	Yes
$fscanf	Yes	Yes
$swrite, $sformat, $sscanf	Yes	Yes
$swriteb, $swriteh, $swriteo	Yes	No
$fread	Yes	No
$rewind, $fseek, $ftell	Yes	Yes
$fflush	Yes	Yes
$ferror	Yes	Yes
$feof	Yes	Yes
$readmemb, $readmemh	Yes	No
$sdf_annotate	Yes	No
$fdebug	No	Yes

Table 9-3—Timescale system tasks

Task/function name(s)	Supported in digital context	Supported in analog context
$printtimescale	Yes	No
$timeformat	Yes	No

Table 9-4—Simulation control system tasks

Task name	Supported in digital context	Supported in analog context
$finish	Yes	Yes
$stop	Yes	Yes
$fatal	No	Yes

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 9-4—Simulation control system tasks *(continued)*

Task name	Supported in digital context	Supported in analog context
$warning	No	Yes
$error	No	Yes
$info	No	Yes

Table 9-5—PLA modeling system tasks

Task name	Supported in digital context	Supported in analog context
$async$and$array	Yes	No
$async$nand$array	Yes	No
$async$or$array	Yes	No
$async$nor$array	Yes	No
$sync$and$array	Yes	No
$sync$nand$array	Yes	No
$sync$or$array	Yes	No
$sync$nor$array	Yes	No
$async$and$plane	Yes	No
$async$nand$plane	Yes	No
$async$or$plane	Yes	No
$async$nor$plane	Yes	No
$sync$and$plane	Yes	No
$sync$nand$plane	Yes	No
$sync$or$plane	Yes	No
$sync$nor$plane	Yes	No

Table 9-6—Stochastic analysis system tasks

Task name	Supported in digital context	Supported in analog context
$q_initialize	Yes	No
$q_remove	Yes	No
$q_exam	Yes	No
$q_add	Yes	No
$q_full	Yes	No

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 9-7—Simulation time system functions

Function name	Supported in digital context	Supported in analog context
$realtime	Yes	No
$time	Yes	No
$stime	Yes	No
$abstime	Yes	Yes

Table 9-8—Conversion system functions

Function name	Supported in digital context	Supported in analog context
$bitstoreal	Yes	Yes
$itor	Yes	Yes
$signed	Yes	No
$realtobits	Yes	Yes
$rtoi	Yes	Yes
$unsigned	Yes	No

Table 9-9—Command line input system functions

Function name	Supported in digital context	Supported in analog context
$test$plusargs	Yes	Yes
$value$plusargs	Yes	Yes

Table 9-10—Probabilistic distribution system functions

Function name	Supported in digital context	Supported in analog context
$dist_chi_square	Yes	Yes
$dist_exponential	Yes	Yes
$dist_poisson	Yes	Yes
$dist_uniform	Yes	Yes
$dist_erlang	Yes	Yes
$dist_normal	Yes	Yes

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 9-10—Probabilistic distribution system functions *(continued)*

Function name	Supported in digital context	Supported in analog context
$dist_t	Yes	Yes
$random	Yes	Yes
$arandom	Yes	Yes
$rdist_chi_square	Yes	Yes
$rdist_exponential	Yes	Yes
$rdist_poisson	Yes	Yes
$rdist_uniform	Yes	Yes
$rdist_erlang	Yes	Yes
$rdist_normal	Yes	Yes
$rdist_t	Yes	Yes

Table 9-11—Math system functions

Function name	Supported in digital context	Supported in analog context
$clog2	Yes	Yes
$ln	Yes	Yes
$ln1p	Yes	Yes
$log10	Yes	Yes
$exp	Yes	Yes
$expm1	Yes	Yes
$sqrt	Yes	Yes
$pow	Yes	Yes
$floor	Yes	Yes
$ceil	Yes	Yes
$sin	Yes	Yes
$cos	Yes	Yes
$tan	Yes	Yes
$asin	Yes	Yes
$acos	Yes	Yes
$atan	Yes	Yes
$atan2	Yes	Yes
$hypot	Yes	Yes
$sinh	Yes	Yes

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 9-11—Math system functions *(continued)*

Function name	Supported in digital context	Supported in analog context
$cosh	Yes	Yes
$tanh	Yes	Yes
$asinh	Yes	Yes
$acosh	Yes	Yes
$atanh	Yes	Yes
$min	Yes	Yes
$max	Yes	Yes
$abs	Yes	Yes

Table 9-12—Analog kernel parameter system functions

Function Name	Supported in digital context	Supported in analog context
$temperature	Yes	Yes
$vt	Yes	Yes
$simparam	Yes	Yes
$simparam$str	Yes	Yes

Table 9-13—Dynamic simulation probe system function

Function Name	Supported in digital context	Supported in analog context
$simprobe	No	Yes

Table 9-14—Analog kernel control system tasks and functions

Task/function name	Supported in digital context	Supported in analog context
$discontinuity	No	Yes
$limit	No	Yes
$bound_step	No	Yes

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 9-15—Hierarchical parameter system functions

Function name	Supported in digital context	Supported in analog context
$mfactor	Yes	Yes
$xposition	Yes	Yes
$yposition	Yes	Yes
$angle	Yes	Yes
$hflip	Yes	Yes
$vflip	Yes	Yes

Table 9-16—Explicit binding detection system functions

Function name	Supported in digital context	Supported in analog context
$param_given	No	Yes
$port_connected	No	Yes

Table 9-17—Analog node alias system function

Function name	Supported in digital context	Supported in analog context
$analog_node_alias	No	Yes
$analog_port_alias	No	Yes

Table 9-18—Table based interpolation and lookup system function

Function name	Supported in digital context	Supported in analog context
$table_model	Yes	Yes

Table 9-19—Connectmodule driver and receiver access system functions and operator

Function/operator name	Supported in digital context of connectmodule	Supported in analog context of connectmodule
$driver_count	Yes	No
$driver_state	Yes	No

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 9-19—Connectmodule driver and receiver access system functions and operator

Function/operator name	Supported in digital context of connectmodule	Supported in analog context of connectmodule
$driver_strength	Yes	No
@(driver_update)	Yes	No
$receiver_count	Yes	Yes

Table 9-20—Supplementary connectmodule driver access system functions

Task/function name(s)	Supported in digital context of connectmodule	Supported in analog context of connectmodule
$driver_delay	Yes	No
$driver_next_state	Yes	No
$driver_next_strength	Yes	No
$driver_type	Yes	No

9.3 System tasks/functions executing in the context of the Analog Simulation Cycle

From 8.2, the analog simulation cycle has some different characteristics than the digital simulation cycle in Verilog-AMS. These differences requires some additional description for certain system tasks or functions that are supported in IEEE Std 1364 Verilog and have been extended to work in the analog context by Verilog-AMS.

A key difference is that the analog engine iteratively evaluates the **`analog`** blocks in an analog macro process until that process is converged 8.4. The behavior of a particular system task or function during the iterative evaluation process will be stated in the relevant section for that system task or function, if required. The goal of the defined behavior of a system task or function in the analog context is that a call to a such system task or function in an **`analog`** block during an iteration that is rejected should cause no side-effects on the next iteration.

Another difference is that the analog engine supports additional analyses beyond a single transient analysis. A single transient analysis is the only analysis that IEEE Std 1364 Verilog supports. Verilog-AMS extends this to allows multiple analyses, including multiple transient analyses, to be run within a single simulation process. Because of this extension, the behavior of a particular system task or function during different analysis types and between different analyses will be stated in the relevant section for that system task or function, if required.

9.4 Display system tasks

9.4.1 Behavior of the display tasks in the analog context

Verilog-AMS extends the display tasks so that they can be used in the analog context.

The syntax for these functions are shown in Syntax 9-1.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
display_tasks_in_analog_block ::=
      $strobe ( list_of_arguments ) ;
    | $display ( list_of_arguments ) ;
    | $monitor ( list_of_arguments ) ;
    | $write ( list_of_arguments ) ;
    | $debug ( list_of_arguments ) ;
```

Syntax 9-1—Syntax for the display_tasks_in_analog_block

The following rules apply to these functions.

- **`$strobe`** provides the ability to display simulation data when the simulator has converged on a solution for all nodes.
- **`$strobe`** displays its arguments in the same order they appear in the argument list. Each argument can be a quoted string, an expression which 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 specify the display format for a subsequent expression.
- Escape sequences are inserted into a string in three ways:
  - The special character `\` indicates the character to follow is a literal or non-printable character (see Table 9-21).
  - The special character `%` indicates the next character shall be interpreted as a format specification which establishes the display format for a subsequent expression argument (see Table 9-22). For each `%` character which 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 9-21).
- Any *null* argument produces a single space character in the display. (A *null argument* is characterized by two adjacent commas (`,,`) in the argument list.)
- When **`$strobe`** is invoked without arguments, it simply prints a newline character.

The **`$display`** task provides the same capabilities as **`$strobe`**. The **`$write`** task provides the same capabilities as **`$strobe`**, but with no newline. The **`$debug`** task provides the capability to display simulation data while the analog simulator is solving the equations; it displays its arguments for each iteration of the analog solver.

The **`$monitor`** task provides the ability to monitor and display the values of any variables or expressions specified as arguments to the task. The arguments for this task are specified in exactly the same manner as for the **`$strobe`** system task.

When a **`$monitor`** task is invoked with one or more arguments, the simulator sets up a mechanism whereby for each accepted step, if the variable or an expression in the argument list changes value compared with the last accepted step —with the exception of the **`$abstime`** or **`$realtime`** system functions—the entire argument list is displayed at the end of the time step as if reported by the **`$strobe`** task. If two or more arguments change value at the same time, only one display is produced that shows the new values.

9.4.2 Escape sequences for special characters

The escape sequences shown in Table 9-21, when included in a string argument, print special characters.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 9-21— Escape sequences for printing special characters

`\n`	The newline character
`\t`	The tab character
`\\`	The \ character
`\"`	The " character
`\ddd`	A character specified by 1 to 3 octal digits
`%%`	The % character

9.4.3 Format specifications

Table 9-22 shows the escape sequences used for format specifications. 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. For each `%` character (except `%m`, `%%` and `%l`) that appears in a string, a corresponding expression argument shall be supplied after the string.

Table 9-22— Escape sequences for format specifications

`%h` or `%H`	Display in hexadecimal format
`%d` or `%D`	Display in decimal format
`%o` or `%O`	Display in octal format
`%b` or `%B`	Display in binary format
`%c` or `%C`	Display in ASCII character format
`%l` or `%L`	Display library binding information
`%m` or `%M`	Display hierarchical name
`%s` or `%S`	Display as a string

The formatting specification `%l` (or `%L`) is defined for displaying the library information of the specific module. This information shall be displayed as "*library.cell*" corresponding to the library name from which the current module instance was extracted and the cell name of the current module instance. See Clause 13 of IEEE Std 1364 Verilog for information on libraries and configuring designs.

Any expression argument which has no corresponding format specification is displayed using the default decimal format in **`$strobe`**.

The format specifications in Table 9-23 are used for real numbers and have the full formatting capabilities available in the C language. For example, the format specification `%10.3g` sets a minimum field width of 10 with three (3) fractional digits.

Table 9-23— Format specifications for real numbers

`%e` or `%E`	Display 'real' in an exponential format
`%f` or `%F`	Display 'real' in a decimal format

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 9-23— Format specifications for real numbers

`%g` or `%G`	Display 'real' in exponential or decimal format, whichever format results in the shorter printed output
`%r` or `%R`	Display 'real' in engineering notation, using the scale factors defined in 2.6.2

9.4.4 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 which invokes the system task containing the format specifier. This is useful when there are many instances of the module which call the system task. One obvious application is timing check messages in a flip-flop or latch module; the `%m` format specifier pinpoints the module instance responsible for generating the timing check message.

9.4.5 String format

The `%s` format specifier is used to print ASCII codes as characters. For each `%s` specification which appears in a string, a corresponding argument shall follow the string in the argument list. The associated argument is interpreted as a sequence of 8-bit hexadecimal ASCII codes, with each 8 bits representing a single character. If the argument is a variable, its value shall be right-justified so the right-most 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 (`0`) are never printed.

9.4.6 Behavior of the display tasks in the analog block during iterative solving

All the display tasks, except **`$debug`**, shall not display output unless an iteration has been accepted.

9.4.7 Extensions to the display tasks in the digital context

For **`$strobe`**, **`$display`**, **`$write`** and **`$monitor`**

— the `%r` (or `%R`) format specifier may be used on real expressions in the digital context

9.5 File input-output system tasks and functions

Verilog-AMS HDL extends many of the file operation tasks so that they can be used in the analog context. This section describes the File I/O tasks that can be used in the analog context.

The system tasks and functions for file-based operations are divided into the following categories:

— Functions and tasks that open and close files
— Tasks that output values into files
— Tasks that output values into variables
— Tasks and functions that read values from files and load into variables

9.5.1 Opening and closing files

The syntax for **`$fopen`** and **`$fclose`** system tasks is shown in Syntax 9-2.

---

file_open_function ::=
    mcd = **`$fopen`** (filename) ;

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    | fd = $fopen ( filename , type ) ;
file_close_task ::=
      $fclose ( multi_channel_descriptor ) ;
    | $fclose ( fd ) ;
```

Syntax 9-2—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 an expression that is a string literal, **string** data type, or an integral data type containing a character string that names the file to be opened.

type is a string expression containing a character string of one of the forms in Table 9-24 that indicates how the file should be opened. If *type* is omitted, the file is opened for writing, and a multichannel descriptor mcd is returned. If *type* is supplied, the file is opened as specified by the value of *type*, and a file descriptor *fd* is returned.

The multichannel descriptor *mcd* is a 32-bit integer in which a single bit is set indicating which file is opened. The least significant bit (bit 0) of an *mcd* always refers to the standard output. Output is directed to two or more files opened with multichannel descriptors by bitwise OR-ing together their multichannel descriptors and writing to the resultant value.

The most significant bit (bit 31) of a multichannel descriptor is reserved and shall always be cleared, limiting an implementation to at most 31 files opened for output via multichannel descriptors.

The file descriptor *fd* is a 32-bit value. The most significant bit (bit 31) of a fd is reserved and shall always be set; this allows implementations of the file input and output functions to determine how the file was opened. The remaining bits hold a small number indicating what file is opened. Three file descriptors are pre-opened; they are `STDIN`, `STDOUT`, and `STDERR`, which have the values `32'h8000_0000`, `32'h8000_0001`, and `32'h8000_0002`, respectively. `STDIN` is pre-opened for reading, and `STDOUT` and `STDERR` are pre-opened for append.

Unlike multichannel descriptors, file descriptors cannot be combined via bitwise OR in order to direct output to multiple files. Instead, files are opened via file descriptor for input, output, and both input and output, as well as for append operations, based on the value of *type*, according to Table 9-24.

Table 9-24—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

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

If a file cannot be opened (either the file does not exist and the *type* specified is `"r"`, `"rb"`, `"r+"`, `"r+b"`, or `"rb+"`, or the permissions do not allow the file to be opened at that path), a zero is returned for the mcd or fd. Applications can call **`$ferror`** to determine the cause of the most recent error (see 9.5.7).

The `"b"` in the above types exists to distinguish binary files from text files. Many systems (such as Unix) make no distinction between binary and text files, and on these systems the `"b"` is ignored. However, some systems (such as machines running Windows NT) perform data mappings on certain binary values written to and read from files that are opened for text access.

The **`$fclose`** system task closes the file specified by *fd* or closes the file(s) specified by the multichannel descriptor mcd. No further output to or input from any file descriptor(s) closed by **`$fclose`** is allowed. 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.

9.5.1.1 opening and closing files during multiple analyses

Verilog AMS HDL supports multiple analyses during the same simulation process (see Clause 8).

If a file is opened in a write mode in the first analysis and reopened in that write mode in following analysis, then content written from the following analyses shall be appended to the content written during the previous analyses.

9.5.1.2 Sharing of file descriptors between the analog and digital contexts

The file I/O system functions and tasks in both the analog and digital contexts can use file descriptors opened in either context, if the file descriptors are opened for writing or appending.

9.5.2 File output system tasks

The syntax for **`$fdisplay`**, **`$fwrite`**, **`$fmonitor`**, **`$fstrobe`** and **`$fdebug`** system tasks is shown in *Syntax 9-3*.

---

file_open_function ::=
    file_output_task_name **`(`** fd [**`,`** list_of_arguments] **`)`** **`;`**

file_output_task_name ::=
    `$fdisplay` | **`$fwrite`** | **`$fstrobe`** | **`$fmonitor`** | **`$fdebug`**

Syntax 9-3—Syntax for file output system tasks

Each of the 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`**, **`$fmonitor`**, **`$fstrobe`**, and **`$fdebug`** — accept the same type of arguments as the tasks upon which they are based, with one exception: The first argument shall be either a multichannel descriptor or a file descriptor, which indicates where to direct the file output. Multichannel descriptors are described in detail in 9.5.1. 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 file descriptor.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

9.5.3 Formatting data to a string

The syntax for the **$swrite** family of tasks and for **$sformat** system task is shown in *Syntax 9-4*.

---

string_output_task ::=
    $swrite **(** string_variable **,** list_of_arguments **)** **;**
variable-format_string_output_task ::=
    $sformat **(** string_variable **,** format_string **,** list_of_arguments **)** **;**

Syntax 9-4—Syntax for formatting data tasks

The **$swrite** family of tasks is based on the **$fwrite** family of tasks and accepts the same type of arguments as the tasks upon which it is based, with one exception: The first argument to **$swrite** shall be a string variable to which the resulting string shall be written, instead of a variable specifying the file to which to write the resulting string.

The system task **$sformat** is similar to the system task **$swrite**, with one major difference.

Unlike the display and write family of output system tasks, **$sformat** always interprets its second argument, and only its second argument, as a format string. This format argument can be a static string, such as `"data is %d"` or can be a string variable whose content is interpreted as the format string. No other arguments are interpreted as format strings. **$sformat** supports all the format specifiers supported by **$display**, as documented in Table 9-22.

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.

If the *format_string* is a string variable, it might not be possible to determine its value at compile time.

9.5.4 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 9.5.2 for more information about opening files.

9.5.4.1 Reading a line at a time

For example:

```
integer code ;
code = $fgets ( str, fd );
```

reads characters from the file specified by *fd* into the string variable, *str* until a newline character is read and transferred to *str*, or an EOF condition is encountered.

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 9.5.7).

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

9.5.4.2 Reading formatted data

For example:

```
integer code ;
code = $fscanf ( fd, format, args );
code = $sscanf ( str, format, args );
```

$fscanf reads from the files specified by the file descriptor *fd*.

$sscanf reads from the string, *str*. The string *str*, shall be a string variable, string parameter or a string literal.

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 AMS HDL in the analog context can be used. However, if the destination is a **real**, then the value **inf**(or **-inf**) is transferred. The format is a string expression. The string contains conversion specifications, which direct the conversion of input into the arguments. The control string can contain the following:

a) White space characters (spaces, tabs, newlines, or formfeeds) that, except in one case described below, cause input to be read up to the next nonwhite space character. For **$sscanf**, null characters shall also be considered white space.

b) An ordinary character (not %) that must match the next character of the input stream.

c) Conversion specifications consisting of the character %, an optional assignment suppression character *, a decimal digit string that specifies an optional numerical maximum field width, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable specified in the corresponding argument unless assignment suppression was indicated by the character *. In this case, no argument shall be supplied.

The suppression of assignment provides a way of describing an input field that is to be skipped. An *input field* is defined as a string of nonspace characters; it extends to the next inappropriate character or until the maximum field width, if one is specified, is exhausted. For all descriptors except the character c, white space leading an input field is ignored.

%	A single % is expected in the input at this point; no assignment is done.
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 _.
f, e, or g	Matches a floating point number. The format of a floating point number is an optional sign (either + or -), followed by a string of digits from the set 0,1,2,3,4,5,6,7,8,9 optionally containing a decimal point character (.), followed by an optional exponent part including e or E, followed by an

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

	optional sign, followed by a string of digits from the set 0,1,2,3,4,5,6,7,8,9.
r	Matches a ‘real’ number in engineering notation, using the scale factors defined in 2.6.2
s	Matches a string, which is a sequence of nonwhite space 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 `EOF` is encountered during input, conversion is terminated. If `EOF` occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure. Otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including newline characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable.

The number of successfully matched and assigned input items is returned in code; this number can be 0 in the event of an early matching failure between an input character and the control string. If the input ends before the first matching failure or conversion, EOF is returned. Applications can call **`$ferror`** to determine the cause of the most recent error (see 9.5.7).

9.5.5 File positioning

Example 1

```
integer pos ;
pos = $ftell ( fd );
```

returns in *pos* the offset from the beginning of the file of the current byte of the file *fd*, which shall be read or written by a subsequent operation on that file descriptor.

This value can be used by subsequent **`$fseek`** calls to reposition the file to this point. Any repositioning shall cancel any **`$ungetc`** operations. If an error occurs, `EOF` is returned. Applications can call **`$ferror`** to determine the cause of the most recent error (see 17.2.7 of IEEE Std 1364 Verilog).

Example 2

```
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 sets position equal to offset bytes
- 1 sets position to current location plus offset
- 2 sets position to `EOF` plus offset

`$rewind` is equivalent to `$fseek (fd,0,0);`

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Repositioning the current file position with **$fseek** or **$rewind** shall cancel any **$ungetc** operations.

$fseek() allows the file position indicator to be set beyond the end of the existing data in the file. If data are later written at this point, subsequent reads of data in the gap shall return zero until data are actually written into the gap. **$fseek**, by itself, does not extend the size of the file.

When a file is opened for append (that is, when type is "a" or "a+"), it is impossible to overwrite information already in the file. $fseek can be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output.

If an error occurs repositioning the file, then *code* is set to -1. Otherwise, code is set to 0.

Applications can call **$ferror** to determine the cause of the most recent error (see 9.5.7).

9.5.6 Flushing output

For example:

```
$fflush ( mcd );
$fflush ( fd );
$fflush ( );
```

writes any buffered output to the file(s) specified by mcd, to the file specified by fd, or if **$fflush** is invoked with no arguments, to all open files.

9.5.7 I/O error status

Should any error be detected by one of the file I/O routines, an error code is returned. Often this is sufficient for normal operation (i.e., if the opening of an optional configuration file fails, the application typically would simply continue using default values). However, sometimes it is useful to obtain more information about the error for correct application operation. In this case, the **$ferror** function can be used:

```
integer errno ;
errno = $ferror ( fd, str );
```

A string description of type of error encountered by the most recent file I/O operation is written into str, which should be at least 640 bits wide. The integral value of the error code is returned in errno. If the most recent operation did not result in an error, then the value returned shall be zero, and the string variable str shall be empty.

9.5.8 Detecting EOF

For example:

```
integer code;
code = $feof ( fd );
```

returns a nonzero value when EOF has previously been detected reading the input file fd. It returns zero otherwise.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

9.5.9 Behavior of the file I/O tasks in the analog block during iterative solving

If a file is being read from during an iterative solve and if that iteration is rejected, then the file pointer is reset to the file position that it pointed to before the iterative solve started.

If a file is being written to during an iterative solve, then the file write operations shall not be performed unless the iteration is accepted. The exception to this is the $fdebug. If $fdebug is evaluated during an iteration, the write operation shall occur even if the evaluation occurred during an iteration that was rejected.

The features of the underlying implementation of file I/O on the host system may prevent the file position being reset after an iteration is rejected. In this case, a fatal error will be reported.

9.6 Timescale system tasks

Verilog AMS HDL does not extend the timescale tasks defined in IEEE Std 1364 Verilog.

9.7 Simulation control system tasks

Verilog AMS HDL extends the two simulation control tasks, **`$finish`** and **`$stop`** so that they can be run in the analog context.

This section describes their behavior if used in the analog context.

Verilog AMS HDL also supports three new simulation control tasks in the analog context only; **`$fatal`**, **`$error`**, **`$warning`**.

9.7.1 $finish

The syntax for this task is shown in Syntax 9-5.

```
finish_task ::=
    $finish [ ( n ) ] ;
```

Syntax 9-5—Syntax for the finish_task

If **`$finish`** is called during an accepted iteration, then the simulator shall exit after the current solution is complete. **`$finish`** called during a rejected iteration shall have no effect. As a result of the simulation terminating due to a **`$finish`** task, it is expected that all appropriate **`final_step`** blocks are also triggered. If **`$finish`** is called from an **`analog initial`** block, the simulator shall exit without performing the simulation.

If an expression is supplied to this task, its value determines which diagnostic messages are printed after the **`$finish`** call is executed, as shown in Table 9-25. One (1) is the default if no argument is supplied.

Table 9-25—Diagnostic messages

Parameter	Message
0	Prints nothing

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 9-25—Diagnostic messages *(continued)*

Parameter	Message
1	Prints simulation time and location
2	Prints simulation time, location, and statistics about the memory and CPU time used in simulation

If **$finish** is called from within an **analog initial** block, the simulator shall report that the call was made during initialization in place of the simulation time. If **$finish** is called from the analog context during a dc sweep (but outside of an **analog initial** block), the simulator shall report the current value of the swept variable in place of the simulation time.

9.7.2 $stop

The syntax for this task is shown in Syntax 9-6.

```
stop_task ::=
    $stop [ ( n ) ] ;
```

Syntax 9-6—Syntax for the stop_task

A call to **$stop** during an accepted iteration causes simulation to be suspended at a converged time point. This task takes an optional expression argument (0, 1, or 2), which determines what type of diagnostic message is printed. The amount of diagnostic messages output increases with the value of *n*, as shown in Table 9-25. The **$stop** task shall not be used within an **analog initial** block.

The mechanism for resuming simulation is left to the implementation.

9.7.3 $fatal, $error, $warning, and $info

The syntax form for the severity system task is as follows:

```
assert_severity_task ::=
        fatal_message_task
      | nonfatal_message_task
fatal_message_task ::= $fatal [ ( finish_number [ , message_argument { , message_argument } ] ) ] ;
nonfatal_message_task ::= severity_task [ ( [ message_argument { , message_argument] } ] ) ] ;
severity_task ::= $error | $warning | $info
finish_number ::= 0 | 1 | 2
```

Syntax 9-7—Assertion severity tasks

The behavior of assert severity tasks is as follows:

— **$fatal** shall generate a run-time fatal assertion error, which terminates the simulation with an errorcode. The first argument passed to **$fatal** shall be consistent with the corresponding argument to the Verilog **$finish** system task, which sets the level of diagnostic information reported by the tool. Calling **$fatal** results in an implicit call to **$finish**.

— **$error** shall be a run-time error.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

— **`$warning`** shall be a run-time warning, which can be suppressed in a tool-specific manner.

— **`$info`** shall indicate that the assertion failure carries no specific severity.

Non-fatal system severity tasks (**`$error`**, **`$warning`**, **`$info`**) called during a rejected iteration shall have no effect. **`$fatal`** terminates the simulation without checking whether the iteration would be rejected.

If **`$fatal`** is executed within an **`analog initial`** block, then after outputting the message, the initialization may be aborted, and in no case shall simulation proceed past initialization. Some of the system severity task calls may not be executed either. The *finish_number* may be used in an implementation-specific manner.

If **`$error`** is executed within an **`analog initial`** block, then the message is issued and the initialization continues. However, the simulation shall not proceed past initialization.

The other two tasks, **`$warning`** and **`$info`**, only output their text message but do not affect the rest of the initialization and the simulation.

For simulation tools, these tasks shall also report the simulation run time at which the severity system task is called. If any of these tasks is called from an analog context during a dc sweep, the simulator shall report the current value of the swept variable in place of the simulation run time. If the task is called from an **`analog initial`** block, the simulator shall report that the call was made during initialization.

Each of these system tasks can also include additional user-specified information using the same format as the Verilog **`$display`**.

9.8 PLA modeling system tasks

Verilog AMS HDL does not extend the PLA modeling tasks defined in IEEE Std 1364 Verilog.

9.9 Stochastic analysis system tasks

Verilog AMS HDL does not extend the stochastic analysis tasks defined in IEEE Std 1364 Verilog.

9.10 Simulator time system functions

Verilog AMS HDL extends the simulator time functions defined in IEEE Std 1364 Verilog as follows;

— A new function is added called **`$abstime`** that can be used from the analog and digital contexts. **`$abstime`** returns the absolute time, that is a real value number representing time in seconds.

NOTE—In previous versions of the Verilog-AMS LRM, **`$realtime`** was supported in the analog context and it had an additional argument. This version of the LRM deprecates using **`$realtime`** in the analog context.

9.11 Conversion system functions

Verilog AMS HDL extends the conversion functions defined in IEEE Std 1364 Verilog so that **`$bitstoreal`** and **`$realtobits`**,**`$rtoi`** and **`$itor`** can be used in the analog context.

9.12 Command line input

Verilog AMS HDL extends the command line input functions defined in IEEE Std 1364 Verilog so that they can be used in the analog context.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

9.13 Probabilistic distribution system functions

Verilog-AMS HDL extends the probabilistic distribution functions so that they are supported in the analog context. Also, real versions of the probabilistic distribution functions are introduced. Also, real versions of the probabilistic distribution functions are introduced as well as a special analog random system task called **$arandom**.

9.13.1 $random and $arandom

This subclause describes how the **$random** and **$arandom** system functions are supported in the analog context.

The syntax for these functions is shown in Syntax 9-8.

```
random_function ::=
        $random [ ( random_seed ) ]
random seed ::=
          integer_variable_identifier
        | reg_variable_identifier
        | time_variable_identifier
analog_random_function ::=
        $arandom [ ( analog_random_seed [, type_string] ) ]
analog_random_seed ::=
          integer_variable_identifier
        | reg_variable_identifier
        | time_variable_identifier
        | integer_parameter_identifier
        | [ sign] decimal_number
type_string ::=
        "global"
      | "instance"
```

Syntax 9-8—Syntax for the random_function and analog random function

he system functions **$random** and **$arandom** provide a mechanism for generating random numbers. The random number returned is a 32-bit signed integer; it can be positive or negative. The two functions differ in the arguments they take. **$arandom** is upwardly compatible with **$random** — **$arandom** can take the same arguments as **$random** and has the same behavior.

The *random_seed* argument may take one of several forms. It may be omitted, in which case the simulator picks a seed. If the call to **$random** is within the analog context, the *random_seed* may be an analog **integer** variable. If the call to **$random** is within the digital context it may be a **reg**, **integer**, or **time** variable. If the *random_seed* argument is specified it is an **inout** argument; that is, a value is passed to the function and a different value is returned. The variable should be initialized by the user prior to calling **$random** and only updated by the system function. The function returns a new 32-bit random number each time it is called.

The system function **$random** shall always return the same stream of values given the same initial *random_seed*. This facilitates debugging by making the operation of the system repeatable.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

$arandom supports the seed argument *analog_random_seed*. The *analog_random_seed* argument can also be a parameter or a constant, in which case the system function does not update the parameter value. However an internal seed is created which is assigned the initial value of the parameter or constant and the internal seed gets updated every time the call to **$arandom** is made. This allows the **$arandom** system function to be used for parameter initialization. In order to get different random values when the *analog_random_seed* argument is a parameter, the user can override the parameter using a method in 6.3.

The *type_string* is an additional argument that **$arandom** supports beyond **$random**. The *type_string* provides support for Monte-Carlo analysis and shall only by used in calls to **$arandom** from within a paramset. If the *type_string* is "`global`" (or not specified in a call within a paramset), then one value is generated for each Monte-Carlo trial. If the *type_string* is "`instance`" then one value is generated for each instance that references this value, and a new set of values for these instances is generated for each Monte-Carlo trial.

Examples:

Where `b > 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:

```
integer rand;
rand = $random % 60;
```

9.13.2 Distribution functions

The section describes how the distribution functions are supported in the analog context.

The syntax for these functions are shown in Syntax 9-9.

---

distribution_functions ::=
  $digital_dist_functions (args **)**
| **$rdist_uniform (** seed **,** start_expression **,** end_expression [**,** type_string] **)**
| **$rdist_normal (** seed **,** mean_expression **,** standard_deviation_expression [**,** type_string] **)**
| **$rdist_exponential (** seed **,** mean_expression [**,** type_string] **)**
| **$rdist_poisson (** seed **,** mean_expression [**,** type_string] **)**
| **$rdist_chi_square (** seed **,** degree_of_freedom_expression [**,** type_string] **)**
| **$rdist_t (** seed **,** degree_of_freedom_expression [**,** type_string] **)**
| **$rdist_erlang (** seed **,** k_stage_expression **,** mean_expression [**,** type_string] **)**

seed ::=
  integer_variable_identifier
| integer_parameter_identifier
| [sign] decimal_number

type_string ::=
  "global"
| **"instance"**

Syntax 9-9—Syntax for the probabilistic distribution functions

The following rules apply to these functions.

— All arguments to the system functions are real values, except for *seed* (which is defined by **$random**). For the **$rdist_exponential**, **$rdist_poisson**, **$rdist_chi_square**,

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

$rdist_t, and **$rdist_erlang** functions, the arguments *mean*, *degree_of_freedom*, and *k_stage* shall be greater than zero (0). Otherwise an error shall be reported.

— Each of these functions returns a pseudo-random number whose characteristics are described by the function name, e.g., **$rdist_uniform** returns random numbers uniformly distributed in the interval specified by its arguments.

— For each system function, the *seed* argument shall be an integer. If it is an integer variable, then it is an `inout` argument; that is, a value is passed to the function and a different value is returned. The variable is initialized by the user and only updated by the system function. This ensures the desired distribution is achieved upon successive calls to the system function. If the *seed* argument is a parameter or constant, then the system function does not update the parameter value. However an internal seed is created which is assigned the initial value of the parameter or constant and the internal seed gets updated every time the call to the system function is made. This allows the system function to be used for parameter initialization.

— The system functions shall always return the same value given the same *seed*. This facilitates debugging by making the operation of the system repeatable. In order to get different random values when the *seed* argument is a parameter, the user can override the parameter.

— All functions return a real value.

— In **$rdist_uniform**, the *start* and *end* arguments are real inputs which bound the values returned. The *start* value shall be smaller than the *end* value.

— The *mean* argument used by **$rdist_normal**, **$rdist_exponential**, **$rdist_poisson**, and **$rdist_erlang** is an real input which causes the average value returned by the function to approach the value specified.

— The *standard_deviation* argument used by **$rdist_normal** is a real input, which helps determine the shape of the density function. Using larger numbers for *standard_deviation* spreads the returned values over a wider range. Using a *mean* of zero (0) and a *standard_deviation* of one (1), **$rdist_normal** generates Gaussian distribution.

— The *degree_of_freedom* argument used by **$rdist_chi_square** and **$rdist_t** is a real input, which helps determine the shape of the density function. Using larger numbers for *degree_of_freedom* spreads the returned values over a wider range.

— The *type_string* provides support for Monte-Carlo analysis and shall only by used in calls to a distribution function from within a paramset. If the *type_string* is "`global`" (or not specified in a call within a paramset), then one value is generated for each Monte-Carlo trial. If the *type_string* is "`instance`" then one value is generated for each instance that references this value, and a new set of values for these instances is generated for each Monte-Carlo trial. See 6.4.1 for an example.

9.13.3 Algorithm for probabilistic distribution

17.9.3 of IEEE Std 1364 Verilog contains the C-code to describe the algorithm of probabilistic system functions based on the seed value passed to them.

This code also describe the algorithm of the IEEE Std 1364 Verilog probabilistic functions extensions in Verilog-AMS HDL as indicated in Table 9-26.

Table 9-26—Verilog AMS to C function cross-listing

Verilog AMS Function	C function in IEEE Std 1364 Verilog (subclause 17.9.3)
$rdist_uniform	uniform
$rdist_normal	normal
$rdist_exponential	exponential

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 9-26—Verilog AMS to C function cross-listing

Verilog AMS Function	C function in IEEE Std 1364 Verilog (subclause 17.9.3)
$rdist_poisson	poisson
$rdist_chi_square	chi_square
$rdist_t	t
$rdist_erlang	erlang

9.14 Math system functions

Verilog-AMS HDL extends the IEEE Std 1364-2005 Verilog HDL math functions so that they can be used from the analog context.

All of these functions, except **`$clog2`**, are aliases of the analog math operators described in 4.3.1 and Table 4-14 shows which analog math operators are aliases of which math system functions.

The system function **`$clog2`** shall return the ceiling of the log base 2 of the argument (the log rounded up to an integer value). **`$clog2`** is defined more completely in IEEE Std 1364-2005 Verilog HDL.

Users are encourage to use the system function version of the math operation instead of the operator for better compatibility with IEEE Std 1364-2005 Verilog HDL.

9.15 Analog kernel parameter system functions

Verilog AMS HDL adds a set of system functions called the analog kernel parameter functions.

The syntax for these functions are shown in Syntax 9-10.

```
environment_parameter_functions ::=
      $temperature
    | $vt [ ( temperature_expression ) ]
    | $simparam ( param_name [, expression] )
    | $simparam$str ( param_name )
```

Syntax 9-10—Syntax for the environment parameter functions

These functions return information about the current environment parameters as a real value.

`$temperature` does not take any input arguments and returns the circuit's ambient temperature in Kelvin units.

`$vt` can optionally have temperature (in Kelvin units) as an input argument and returns the thermal voltage (kT/q) at the given temperature. **`$vt`** without the optional input temperature argument returns the thermal voltage using **`$temperature`**.

`$simparam()` queries the simulator for a real-valued simulation parameter named *param_name*. The argument *param_name* is a string value, either a string literal, string parameter, or a string variable. If *param_name* is known, its value is returned. If *param_name* is not known, and the optional *expression* is not supplied, then an error is generated. If the optional *expression* is supplied, its value is returned if

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

param_name is not known and no error is generated. **`$simparam()`** shall always return a real value; simulation parameters that have integer values shall be coerced to real. There is no fixed list of simulation parameters. However, simulators shall accept the strings in Table 9-27 to access commonly-known simulation parameters, if they support the parameter. Simulators can also accept other strings to access the same parameters.

Table 9-27—Simulation real and integer parameter names

String	Units	Description
gdev	1/Ohms	Additional conductance to be added to nonlinear branches for conductance homotopy convergence algorithm.
gmin	1/Ohms	Minimum conductance placed in parallel with nonlinear branches.
imax	Amps	Branch current threshold above which the constitutive relation of a nonlinear branch should be linearized.
imelt	Amps	Branch current threshold indicating device failure.
iteration		Iteration number of the analog solver.
scale		Scale factor for device instance geometry parameters.
shrink		Optical linear shrink factor.
simulatorSubversion		The simulator sub-version.
simulatorVersion		The simulator version.
sourceScaleFactor		Multiplicative factor for independent sources for source stepping homotopy convergence algorithm.
tnom	Celsius	Default value of temperature at which model parameters were extracted.
timeUnit	s	Time unit as specified in **`` `timescale ``** in seconds.
timePrecision	s	Time precision as specified in **`` `timescale ``** in seconds.

The values returned by simulatorVersion and simulatorSubversion are at the vendor's discretion, but the values shall be monotonically increasing for new versions or releases of the simulator, to facilitate checking that the simulator supports features that were added in a certain version or sub-version.

Examples:

In this first example, the variable `gmin` is set to the simulator's parameter named `gmin`, if it exists, otherwise, an error is generated.

```
gmin = $simparam("gmin");
```

In this second example, the variable `sourcescale` is set to the simulator's parameter named `sourceScaleFactor`, if it exists, otherwise, the value 1.0 is returned.

```
sourcescale = $simparam("sourceScaleFactor", 1.0);
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

$simparam$str is similar to **$simparam**. However it is used for returning string-valued simulation parameters. Table 9-28 gives a list of simulation string parameter names that shall be supported by **$simparam$str**.

Table 9-28—Simulation string parameter names

String	Description
analysis_name	The name of the current analysis e.g. tran1, mydc
analysis_type	The type of the current analysis e.g. dc, tran, ac
cwd	The current working directory in which the simulator was started
module	The name of the module from which $simparam$str is called.
instance	The hierarchical name of the instance from which $simparam$str is called.
path	The hierarchical path to the $simparam$str function.

Example:

```
module testbench;
   dut dut1;
endmodule
module dut;
   task mytask;
      $display( "%s\n%s\n%s\n", $simparam$str( "module"),
                        $simparam$str( "instance"),
                        $simparam$str( "path"));
   endtask
endmodule
```

produces

```
dut
testbench.dut1
testbench.dut1.mytask
```

9.16 Dynamic simulation probe function

Verilog-AMS HDL supports a system function that allows the probing of values within a sibling instance during simulation.

---

dynamic_monitor_function ::=
    $simprobe (inst_name , param_name [, expression])

Syntax 9-11—Syntax for the dynamic monitor function

$simprobe() queries the simulator for an output variable named *param_name* in a sibling instance called *inst_name*. The arguments *inst_name* and *param_name* are string values, either a string literal, string parameter, or a string variable. To resolve the value, the simulator will look for an instance called *inst_name* in the parent of the current instance i.e. a sibling of the instance containing the **$simprobe()** expression. Once the instance is resolved, it will then query that instance for an output variable called *param_name*. If either the *inst_name* or *param_name* cannot be resolved, and the optional *expression* is not supplied, then an error shall be generated. If the optional *expression* is supplied, its value will be returned in lieu of raising an error. The intended use of this function is to allow dynamic monitoring of instance quantities.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Example:

```
module monitor;
   parameter string inst = "default";
   parameter string quant = "default";
   parameter real threshold = 0.0;
   real probe;
   analog begin
      probe = $simprobe(inst,quant);
      if (probe > threshold) begin
         $strobe("ERROR: Time %e: %s#%s (%g) > threshold (%e)",
                  $abstime, inst,quant, probe, threshold);
         $finish;
      end
   end
endmodule
```

The module `monitor` will probe the `quant` in instance `inst`. If its value becomes larger than `threshold`, then the simulation will raise an error and stop.

```
module top(d,g,s);
   electrical d,g,s;
   inout d,g,s;
   electrical gnd; ground gnd;
   SPICE_pmos #(.w(4u),.l(0.1u),.ad(4p),.as(4p),.pd(10u),.ps(10u))
      mp(d,g,s,s);
   SPICE_nmos #(.w(2u),.l(0.1u),.ad(2p),.as(2p),.pd(6u),.ps(6u))
      mn(d,g,gnd,gnd);
   monitor #(.inst("mn"),.quant("id"),.threshold(4.0e-3))
      amonitor();
endmodule
```

Here the monitor instance `amonitor` will keep track of the dynamic quantity `id` in the mosfet instance `mn`. If the value of `id` goes above the specified threshold of `4.0e-3` amps then instance `amonitor` will generate the error message and stop the simulation.

9.17 Analog kernel control system tasks and functions

Verilog AMS HDL adds a set of tasks and functions to control the analog solver's behavior on a signals and instances called the analog kernel control tasks.

9.17.1 $discontinuity

The **`$discontinuity`** task is used to give hints to the simulator about the behavior of the module so the simulator can control its simulation algorithms to get accurate results in exceptional situations. This task does not directly specify the behavior of the module. **`$discontinuity`** shall be executed whenever the analog behavior changes discontinuously.

The general form is

```
$discontinuity [ ( constant_expression ) ];
```

where *constant_expression* indicates the degree of the discontinuity if the argument to **`$discontinuity`** is non-negative, i.e. **`$discontinuity(i)`** implies a discontinuity in the *i*'th derivative of the constitutive equation with respect to either a signal value or time where *i* must be a non-negative integer. Hence, `$`**`discontinuity(0)`** indicates a discontinuity in the equation, **`$discontinuity(1)`** indicates a discontinu-

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

ity in its slope, etc. A special form of the **$discontinuity** task, **$discontinuity(-1)**, is used with the **$limit()** function so -1 is also a valid argument of **$discontinuity**. See 9.17.3 for an explanation.

Because discontinuous behavior can cause convergence problems, discontinuity shall be avoided whenever possible.

The filter functions (**transition()**, **slew()**, **laplace()**, etc.) can be used to smooth discontinuous behavior. However, in some cases it is not possible to implement the desired functionality using these filters. In those cases, the **$discontinuity** task shall be executed when the signal behavior changes abruptly.

Discontinuity created by switch branches and filters, such as **transition()** and **slew()**, does not need to be announced.

The following example uses the discontinuity task to model a relay.

```
module relay (c1, c2, pin, nin) ;
   inout c1, c2 ;
   input pin, nin ;
   electrical c1, c2, pin, nin ;
   parameter real r=1 ;
   analog begin
      @(cross(V(pin,nin))) $discontinuity ;
      if (V(pin,nin) >= 0)
         I(c1,c2) <+ V(c1,c2)/r;
      else
         I(c1,c2) <+ 0 ;
   end
endmodule
```

In this example, **cross()** controls the time step so the time when the relay changes position is accurately resolved. It also triggers the **$discontinuity** task, which causes the simulator to react properly to the discontinuity. This would have been handled automatically if the type of the branch `(c1,c2)` had been switched between voltage and current.

Another example is a source which generates a triangular wave. In this case, neither the model nor the waveforms generated by the model are discontinuous. Rather, the waveform generated is piecewise linear with discontinuous slope. If the simulator is aware of the abrupt change in slope, it can adapt to eliminate problems resulting from the discontinuous slope (typically changing to a first order integration method).

```
module triangle(out);
   output out;
   voltage out;
   parameter real period = 10.0, amplitude = 1.0;
   integer slope;
   real offset;

   analog begin
      @(timer(0, period)) begin
         slope = +1;
         offset = $abstime ;
         $discontinuity;
      end

      @(timer(period/2, period)) begin
         slope = -1 ;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
            offset = $abstime;
            $discontinuity ;
         end

         V(out) <+ amplitude*slope*
            (4*($abstime - offset)/period - 1);
      end
   endmodule
```

9.17.2 $bound_step task

The **$bound_step()** task puts a bound on the next time step. It does not specify exactly what the next time step is, but it bounds how far the next time point can be from the present time point. The task takes the maximum time step as an argument. It does not return a value.

The general form is

```
$bound_step ( expression ) ;
```

where *expression* is a required argument and represents the maximum timestep the simulator can advance. The *expression* argument shall be non-negative. If the value is less than the simulator's minimum allowable time step, the simulator's minimum time step shall be used instead. Refer to the simulator's documentation for further information regarding limits on step size for time dependent analysis.

For a given time step, the simulator shall ensure that the next time step taken is no larger than the smallest **$bound_step()** argument currently active. The $bound_step() statement shall be ignored during a non time-domain analysis.

The example below implements a sinusoidal voltage source and uses the **$bound_step()** task to assure the simulator faithfully follows the output signal (it is forcing 20 points per cycle).

```
module vsine(out);
   output out;
   voltage out;
   parameter real freq=1.0, ampl=1.0, offset=0.0;

   analog begin
      V(out) <+ ampl*sin(2.0*`M_PI*freq*$abstime) + offset;
      $bound_step(0.05/freq);
   end
endmodule
```

9.17.3 $limit

The **$limit()** function is a special-purpose system function whose purpose, like that of the **limexp()** function of 4.5.13, is to improve convergence of the analog solver. While **limexp()** is specifically intended for the exponential function, **$limit()** may be used for the exponential as well as other nonlinear functions and provides a method to recommend a specific approach for improving the convergence. Syntax 9-12 shows the methods of using the **$limit()** function.

---

limit_call ::=
```
    $limit ( access_function_reference )
  | $limit ( access_function_reference , string, arg_list )
  | $limit ( access_function_reference , analog_function_identifier , arg_list )
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Syntax 9-12—Syntax for $limit()

As with **`limexp()`**, the **`$limit()`** function has internal state containing information about the argument on previous iterations. It returns a real value that is derived from its first argument (the access function reference, such as a branch voltage), but it internally limits the change of the output from iteration to iteration in order to improve convergence. On any iteration where the output value is not the same as the value of the access function (within appropriate tolerances), the simulator is prevented from terminating the iteration. In all cases, the simulator is responsible for determining if limiting should be applied and what the return value is on a given iteration. In particular, the simulator may simply choose to have **`$limit()`** return the value of its first argument, if it has other methods for ensuring convergence.

When more than one argument is supplied to the **`$limit()`** function, the second argument recommends a function to use to compute the return value. When the second argument is a string, it refers to a built-in function of the simulator. The two most common such functions are *pnjlim* and *fetlim*, which are found in SPICE and many SPICE-like simulators. Simulators may support other built-in functions and need not support *pnjlim* or *fetlim*. If the string refers to an unknown or unsupported function, the simulator is responsible for determining the appropriate limiting algorithm, just as if no string had been supplied.

pnjlim is intended for limiting arguments to exponentials, and the **`limexp()`** function of 4.5.13 may be implemented through a function derived from *pnjlim*. Two additional arguments to the **`$limit()`** function are required when the second argument to the limit function is the string "pnjlim": the third argument to **`$limit()`** indicates a step size `vte` and the fourth argument is a critical voltage `vcrit`. The step size `vte` is usually the product of the thermal voltage **`$vt`** and the emission coefficient of the junction. The critical voltage is generally obtained from the formula $V_{crit} = vte \cdot \ln(vte/(\sqrt{2} \cdot I_s))$, where I_s is the saturation current of the junction.

fetlim is intended for limiting the potential across the oxide of a MOS transistor. One additional argument to the **`$limit()`** function is required when the second argument to the limit function is the string `"fetlim"`: the third argument to **`$limit()`** is generally the threshold voltage of the MOS transistor.

In the case that none of the built-in functions of the simulator is appropriate for limiting the potential (or flow) used in a nonlinear equation, the second argument of the **`$limit()`** function may be an identifier referring to a user-defined analog function. User-defined functions are described in 4.7. In this case, if the simulator determines that limiting is needed to improve convergence, it will pass the following quantities as arguments to the user-defined function:

- The first argument of the user-defined function shall be the value of the access function reference for the current iteration.
- The second argument shall be the appropriate internal state; generally, this is the value that was returned by the **`$limit()`** function on the previous iteration.
- If more than two arguments are given to the **`$limit()`** function, then the third and subsequent arguments are passed as the third and subsequent arguments of the user-defined function.

The arguments of the user-defined function shall all be declared **`input`**.

In order to prevent convergence when the output of the **`$limit()`** function is not sufficiently close to the value of the access function reference, the user-defined function shall call **`$discontinuity`**`(-1)` (see 9.17) when its return value is not sufficiently close to the value of its first argument.

The module below defines a diode and includes an analog function that mimics the behavior of *pnjlim* in SPICE. Though **`limexp()`** could have been used for the exponential in the current, using **`$limit()`** allows the same voltage to be used in the charge calculation.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
module diode(a,c);
   inout a, c;
   electrical a, c;
   parameter real IS = 1.0e-14;
   parameter real CJO = 0.0;
   analog function real spicepnjlim;
      input vnew, vold, vt, vcrit;
      real vnew, vold, vt, vcrit, vlimit, arg;
      begin
         vlimit=vnew;
         if ((vnew > vcrit) && (abs(vnew-vold) > (vt+vt))) begin
            if (vold > 0) begin
               arg = 1 + (vnew-vold) / vt;
               if (arg > 0)
                  vlimit = vold + vt * ln(arg);
               else
                  vlimit = vcrit;
            end else
               vlimit = vt * ln(vnew/vt);
            $discontinuity(-1);
         end
         spicepnjlim = vlimit;
      end
   endfunction
   real vdio, idio, qdio, vcrit;
   analog begin
      vcrit=0.7;
      vdio = $limit(V(a,c), spicepnjlim, $vt, vcrit);
      idio = IS * (exp(vdio/$vt) - 1);
      I(a,c) <+ idio;
      if (vdio < 0.5) begin
         qdio = 0.5 * CJO * (1-sqrt(1-V(a,c)));
      end else begin
         qdio = CJO* (2.0*(1.0-sqrt(0.5))
                  + sqrt(2.0)/2.0*(vdio*vdio+vdio-3.0/4.0));
      end
      I(a,c) <+ ddt(qdio);
   end
endmodule
```

9.18 Hierarchical parameter system functions

Verilog AMS HDL adds system functions that can return hierarchically inherited values in a particular instance.

The syntax for these functions are shown in Syntax 9-13.

---

hierarchical_parameter_system_functions ::=

```
  $mfactor
| $xposition
| $yposition
| $angle
| $hflip
| $vflip
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Syntax 9-13—Syntax for the hierarchical parameter system functions

These functions return hierarchical information about the instance of a module or paramset. Subclause 6.3.6 discusses how these parameters are specified for an instance, as well as the automatic rules applied to instances with a non-unity value of **$mfactor**. The remaining hierarchical system parameters do not have any automatic effect on the simulation.

$mfactor is the shunt multiplicity factor of the instance, that is, the number of identical devices that should be combined in parallel and modeled.

$xposition and **$yposition** are the offsets, in meters, of the location of the center of the instance.

$hflip and **$vflip** are used to indicate that the instance has been mirrored about its center, and **$angle** indicates that the instance has been rotated some number of degrees in the counter-clockwise directions.

Hierarchical parameter system functions can also be used as targets in parameter alias declarations (see 3.4.7)

The value returned for each of these functions is computed by combining values from the top of the hierarchy down to the instance making the function call. The rules for combining the values are given in Table 9-29. The top-level value is the starting value at the top of the hierarchy. If a module is instantiated without specifying a value of one of these system parameters (using any of the methods in 6.3), then the value of that system parameter will be unchanged from the instantiating module. If a value is specified, then it is combined with the value from the instantiating module according to the appropriate rule from Table 9-29: the subscript “specified” indicates the value specified for the instance, and the subscript “hier” indicates the value obtained by traversing the hierarchy from the top to the instantiating module.

Table 9-29— Hierarchical parameter values

System parameter	Top-level value	Resolved value for instance	Allowed values
$angle	0 degrees	**$anglespecified + $anglehier**, modulo 360 degrees	0 ≤ **$angle** < 360
$hflip	+1	**$hflipspecified * $hfliphier**	**$hflip** = +1 or -1
$mfactor	1.0	**$mfactorspecified * $mfactorhier**	**$mfactor** > 0
$vflip	+1	**$vflipspecified * $vfliphier**	**$vflip** = +1 or -1
$xposition	0.0 m	**$xpositionspecified + $xpositionhier**	Any
$yposition	0.0 m	**$ypositionspecified + $ypositionhier**	Any

For example, when a module makes a call to **$mfactor**, the simulator computes the product of the multiplicity factor specified for the instance (or 1.0, if no override was specified) times the value for the parent module that instantiated the module, times the parent’s parent’s value, and so on, until the top level is reached.

Note that **$angle** is specified and returned in degrees, but the trigonometric functions of 4.3.2 operate in radians.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Example 1

```
module test_module(p,n);
   inout p,n;
   electrical p,n;
   module_a A1(p,n);
endmodule

module module_a(p,n);
   inout p,n;
   electrical p,n;
   module_b #(.$mfactor(2)) B1(p,n);  // mfactor = 3 * 2
endmodule

module module_b(p,n);
   inout p,n;
   electrical p,n;
   module_c #(.$mfactor(7)) C1(p,n); // mfactor = 3 * 2 * 7 = 42
endmodule

// linear resistor
module module_c(p,n);
   inout p,n;
   electrical p,n;
   parameter r=1.0;
   (* desc = "effective resistance" *) real reff;
   analog begin
      I(p,n) <+ V(p,n)/r;  // mfactor scaling of currents
                           // handled automatically
      reff = r / $mfactor; // the effective resistance = 1/42
   end
endmodule
```

shows how the effect mfactor of an instance, `test_module.A1.B1.C1` of a linear resistance is determined.

Example 2

```
module test_module(p,n);
   inout p,n;
   electrical p,n;
   module_a A1(p,n);
endmodule

module module_a(p,n);
   inout p,n;
   electrical p,n;
   module_b #(.$xposition(1u)) B1(p,n); // xposition=1.1u + 1u
endmodule

module module_b(p,n);
   inout p,n;
   electrical p,n;
   module_c #(.$xposition(2u)) C1(p,n); // xposition=1.1u + 1u + 2u = 4.1u
endmodule

// linear resistor
module module_c(p,n);
   inout p,n;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    electrical p,n;
    parameter r=1.0;
    analog begin
       // Expected value of xposition=4.1e-6
       if ($xposition == 4.1u)
          I(p,n) <+ V(p,n)/1.0;
       else
          I(p,n) <+ V(p,n)/2.0;
    end
endmodule
```

9.19 Explicit binding detection system functions

Verilog AMS HDL adds functions that can be used to check whether a parameter or port binding was explicitly made.

The behavioral code of a module can depend on the way in which it was instantiated. The hierarchy detection functions shown in Syntax 9-14 may be used to determine information about the instantiation.

---

genvar_system_function ::=
    $param_given (module_parameter_identifier)
  | **$port_connected** (port_scalar_expression)

Syntax 9-14—Syntax for the hierarchy detection functions

Note that the return values of these functions shall be constant during a simulation; the value is fixed during elaboration. As such, these functions can be used in genvar expressions controlling conditional or looping behavior of the analog operators of 4.5.

The **$param_given()** function can be used to determine whether a parameter value was obtained from the default value in its declaration statement or if that value was overridden. The **$param_given()** function takes a single argument, which must be a parameter identifier. The return value shall be one (1) if the parameter was overridden, either by a **defparam** statement or by a module instance parameter value assignment, and zero (0) otherwise.

The following example sets the variable `temp` to represent the device temperature. Note that **$temperature** is not a *constant_expression*, so it cannot be used as the default value of the parameter `tdevice`. It is important to be able to distinguish the case where `tdevice` has its default value (say, 27) from the declaration statement from the case where the value 27 was in fact specified as an override, if the simulation is performed at a different temperature.

```
if ($param_given(tdevice))
   temp = tdevice + `P_CELSIUS0;
else
   temp = $temperature;
```

Module ports need not be connected when the module is instantiated. The **$port_connected()** function can be used to determine whether a connection was specified for a port. The **$port_connected()** function takes one argument, which must be a port identifier. The return value shall be one (1) if the port was connected to a net (by order or by name) when the module was instantiated, and zero (0) otherwise. Note that the port may be connected to a net that has no other connections, but **$port_connected()** shall still return one.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

In the following example, **$port_connected()** is used to skip the **transition** filter for unconnected nodes. In module twoclk, the instances of myclk only have connections for their vout_q ports, and thus the filter for vout_qbar is not implemented for either instance. In module top, the vout_q2 port is not connected, so that the vout_q port of topclk1.clk2 is not ultimately used in the circuit; however, the filter for vout_q of clk2 is implemented, because it vout_q is connected on clk2's instantiation line.

```
module myclk(vout_q, vout_qbar);
   output vout_q, vout_qbar;
   electrical vout_q, vout_qbar;
   parameter real tdel = 3u from [0:inf);
   parameter real trise = 1u from (0:inf);
   parameter real tfall = 1u from (0:inf);
   parameter real period = 20u from (0:inf);
   integer q;
   analog begin
      @ (timer(0, period))
         q = 0;
      @ (timer(period/2, period))
         q = 1;
      if ($port_connected(vout_q))
         V(vout_q) <+ transition( q, tdel, trise, tfall);
      else
         V(vout_q) <+ 0.0;
      if ($port_connected(vout_qbar))
         V(vout_qbar) <+ transition( !q, tdel, trise, tfall);
      else
         V(vout_qbar) <+ 0.0;
   end
endmodule

module twoclk(vout_q1, vout_q2);
   output vout_q1, vout_q2;
   electrical vout_q1, vout_q1b;
   myclk clk1(.vout_q(vout_q1));
   myclk clk2(.vout_q(vout_q2));
endmodule

module top(clk_out);
   output clk_out;
   electrical clk_out;
   twoclk topclk1(.vout_q1(clk_out));
endmodule
```

9.20 Analog node alias system functions

Verilog-AMS HDL adds system functions that allows a local node to be aliased to a hierarchical node via a string reference. The node alias system functions are shown in Syntax 9-15.

---

analog_node_alias_system_function ::=
    $analog_node_alias (analog_net_reference , hierarchical_reference_string **)**
  | **$analog_port_alias (** analog_net_reference , hierarchical_reference_string **)**

Syntax 9-15—Syntax for the analog node alias system functions

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Both system functions take two arguments. The *analog_net_reference* shall be either a scalar or vector continuous node declared in the module containing the system function call. If the *analog_net_reference* is a vector node, it shall reference the full vector node, it shall be an error for it to be a bit select or part select of a vector node. It shall be an error for the *analog_net_reference* to be a port or to be involved in port connections.

The *hierarchical_reference_string* shall be a constant string value (string literal or string parameter) containing a hierarchical reference to a continuous node. It shall be an error for the *hierarchical_reference_string* to reference a node that is used as an *analog_net_reference* in another **`$analog_node_alias()`** or **`$analog_port_alias()`** system function call. The *hierarchical_reference_string* shall follow the resolution rules for hierarchical references as described in 6.7.

It shall be an error for the **`$analog_node_alias()`** and **`$analog_port_alias()`** system functions to be used outside the **`analog initial`** block. Along with their enclosing **`analog initial`** block scopes, both system functions shall be re-evaluated each sweep point of a dc sweep as needed (see 5.2.1 and 8.2 for details).

The **`$analog_node_alias()`** and **`$analog_port_alias()`** system functions shall not be used inside conditional (**`if`**, **`case`**, or **`?:`**) statements unless the conditional expression controlling the statement consists of terms which can not change during the course of a simulation.

The return value for both system functions shall be one (1) if the *hierarchical_reference_string* points to a valid continuous node and zero (0) otherwise. If the *hierarchical_reference_string* references a valid continuous node, then the *analog_net_reference* will be aliased to that hierarchical node and shall refer to the same circuit matrix position.

If the return value is zero (0), then the node referenced by the *analog_net_reference* shall be treated as a normal continuous node declared in the module containing the system function call. As such, the user is encouraged to check the return value from the system function call and to take necessary steps to avoid runtime topology issues like a singular matrix.

In addition, a node that is aliased to a valid hierarchical port reference via the **`$analog_port_alias()`** function shall be allowed as an input to the port access function (see 5.4.3) and shall measure the flow through the port of the instance referred to by the hierarchical reference.

If a particular node is involved in multiple calls to either system function, then the last evaluated call shall take precedence.

The following rules shall be applied to determine whether the node or port referenced by the hierarchical_reference_string is considered valid. If any of these rules are violated then the system function shall return a value of zero (0).

- The *hierarchical_reference_string* shall refer to a scalar continuous node or a scalar element of a continuous vector node
- The discipline of the *analog_net_reference* and the resolved hierarchical node reference shall be compatible (see 3.11)
- For the **`$analog_port_alias()`** system function, the resolved hierarchical node reference shall be a port

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

An example showing the use of these two system functions is as follows:

```
module top;
   electrical a, b, c, gnd;
   ground gnd;
   resistor #(.r(1k)) r1(.p(a), .n(gnd));
   checker #(.n1_str("top.a")) ch1();
   analog V(a,gnd) <+ 5;
endmodule

module checker;
   electrical n1, n2;
   parameter string n1_str = "(not_given)";
   integer status;
   real iprobe, vprobe;
   analog initial begin
      // node n1 will be aliased to top.gnd
      status = $analog_node_alias(n1, "top.gnd");

      // Invalid alias as top.a is not a port. Node n2 at this stage
      // is still just a local node in ch1(). The integer status will
      // be assigned a value of 0.
      status = $analog_port_alias(n2, "$root.top.a");

      // even though n1 was assigned a valid alias to top.gnd, this
      // one to top.a will take precedence.
      status = $analog_node_alias(n1, n1_str);

      // Here n2 is now successfully aliased to the port p in instance r1.
      status = $analog_port_alias(n2, "top.r1.p");
   end
   analog begin
      // since n2 is aliased to the port p of instance top.r1
      // we are allowed to probe the port current. In this case,
      // the probe will return a value of 5mA.
      iprobe = I(<n2>);

      // since n1 is aliased to the node top.a, we will be
      // probing the potential of that node. In this case,
      // the probe will return a value of 5V.
      vprobe = V(n1);
   end
endmodule
```

9.21 Table based interpolation and lookup system function

Verilog-AMS HDL provides a multidimensional interpolation and lookup function called **`$table_model`**. The function is designed to operate specifically on multidimensional data in a form that is commonly generated via parametric sweeping schemes available in most analog simulators. This type of data is generated when simulating a system while varying (sweeping) a parameter across some range. Data dimensionality increases when parameter sweeps are nested. While the samples are those of a multidimensional function, sample generation via parametric sweeping leads to a simple recursive interpolation and extrapolation process defined by the **`$table_model`** function.

A typical example will help to explain the process. A user may wish to create a data based model of some function *f(x,y)* over some range of *x* and *y* and use that data as the basis of a behavioral model described in Verilog-AMS.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

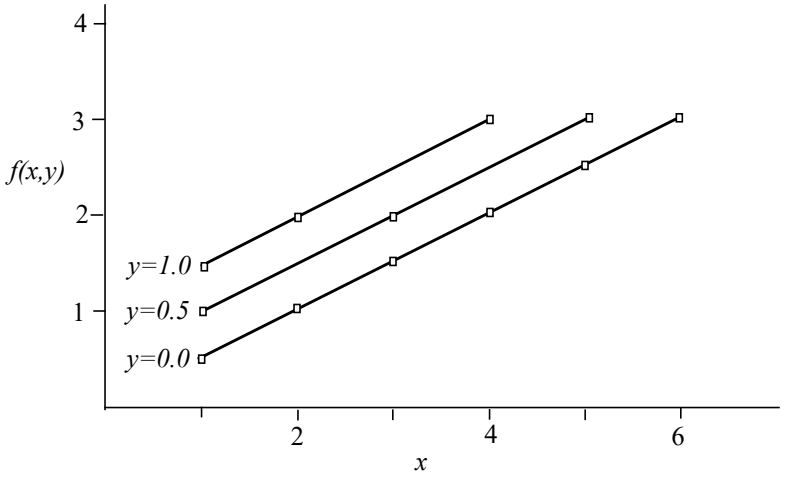


Figure 9-1: Samples on isolines

We can say that *f(x,y)* is sampled on a set of *isolines*. An isoline for each value of *y* is generated when *y* is held constant and *x* is varied across a desired range. Each isoline may exist over a different range of *x* values and the number and spacing of samples may be different on each isoline.

When describing the sampled set, *x* and *y* are called independent variables and *f(x,y)* is called the dependent variable. The sampling scheme also introduces the concept of an innermost and outermost dimension. In this example, *x* is the fastest changing or innermost dimension associated with the sampled function *f(x,y)* and *y* is the slowest changing or outermost dimension.

Understanding that the underlying multidimensional function is sampled on a set of isolines, we can now describe a simple recursive process to interpolate, extrapolate, or perform lookup on this sampled function.

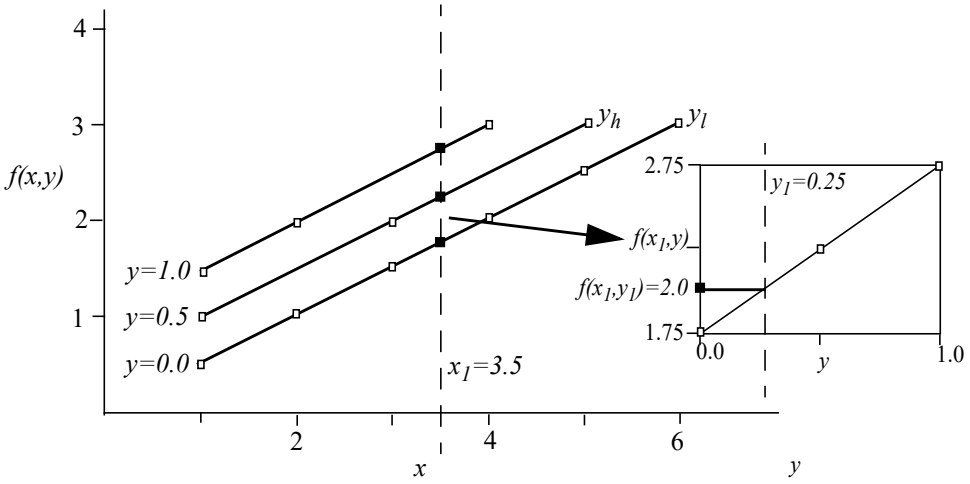


Figure 9-2: Interpolation on isolines

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Using the above example let us assume the user requests a value for the lookup pair (x_1,y_1). We first look through the set of isolines in y and find the pair that bracket y_1. Now for each isoline in y we find the two points that bracket x_1 and interpolate each isoline to find $f(x_1,y_h)$ and $f(x_1,y_l)$. Having thus generated an isoline in y for the point x_1 in x, we may interpolate this isoline to find the value $f(x1,y1)$. If the lookup point falls off the end of any given isoline then we extrapolate its value on that isoline.

As a consequence of this algorithm, the interpolation and extrapolation schemes always operate in a single dimension analogous to how the data was originally generated, so the interpolation and extrapolation schemes used may be specified on a per dimension basis.

The minimum data requirement is to have the product of at least two points per dimension (2N for N dimensions). In addition, the result of the bracketing to produce intermediate points (as per Figure 9-2 and description above) must also produce at least two points per subsequent lower dimension. Within the data set, each point shall be distinct in terms of its independent variable values. If there are two or more data points with the same independent and dependent values, then the duplicates shall be ignored and the tool may generate a warning. If there are two or more data points with the same independent values but different dependent values then an error is generated.

The **`$table_model`** function defines a format to represent the isolines of multidimensional data and a set of interpolation schemes that we need only define for single dimensional data. The data may be stored in a file or as a sequence of one-dimension arrays or a single two-dimensional array.

The interpolation schemes are closest point (discrete) lookup, linear, quadratic splines, and cubic splines. Extrapolation may be specified as being constant, linear, or error (meaning if extrapolation occurs the system should error out).

The lookup variables, (x_1, y_1) in the example above (`table_inputs` in Syntax 9-16) may be any legal expression that can be assigned to an analog signal.

The syntax for the **`$table_model`** function is shown in Syntax 9-16.

```
table_model_function ::=
        $table_model ( table_inputs , table_data_source [, table_control_string] )
table_inputs ::=
        expression [, 2nd_dim_expression [, nth_dim_expression]]
table_data_source ::=
        file_name | table_model_array
file_name ::=
        string_literal | string_parameter
table_model_array ::=
        1st_dim_array_identifier [, 2nd_dim_array_identifier [, nth_dim_array_identifier]],
        output_array_identifier
table_control_string::=
        "[interp_control[;dependent_selector]]"
interp_control::=
        1st_dim_table_ctrl_substr_or_null [, 2nd_dim_table_ctrl_substr_or_null [, nth_dim_table_ctrl_-
      substr_or_null]]
dependent_selector::=
        integer
table_ctrl_substr ::=
        [table_interp_char][table_extrap_char [higher_table_extrap_char]]
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
table_interp_char ::=
      I | D | 1 | 2 | 3
table_extrap_char ::=
      C | L | E
```

Syntax 9-16—Syntax for table model function

9.21.1 Table data source

table_data_source specifies the data source, samples of a multidimensional function arranged on isolines. The data is specified as columns in a text file or as a set of arrays (hence the name *table* model). In either case the layout is conceptually the same.

A table of M dependent variables of dimension N are laid out in N+M columns in the file, with the independent variables appearing in the first N columns followed by the dependent variables in the remaining M columns. The independent variables are ordered from the outermost (slowest changing) variable to the innermost (fastest changing) variable. Though an isoline ordinate does not change for a given isoline, in this scheme the ordinate is repeated for each point of that isoline (thus keeping the input data as a set of data rows all with the same number of points). The result is a sequential listing of each isoline with the total number of points in the listing being equal to the total number of samples on all isolines.

Again, the above example described via samples will help illustrate the layout. The function being described is

$$f(x,y)=0.5x + y$$

f(x,y) is the only dependent variable we consider in this case, and there are three isolines for values of *y* 0.0, 0.5 and 1.0; *x* is sampled at various points on each of the three isolines.

```
# 2-D table model sample example
#
#  y     x     f(x,y)
#y=0 isoline
   0.0   1.0   0.5
   0.0   2.0   1.0
   0.0   3.0   1.5
   0.0   4.0   2.0
   0.0   5.0   2.5
   0.0   6.0   3.0
#y=0.5 isoline
   0.5   1.0   1.0
   0.5   3.0   2.0
   0.5   5.0   3.0
#y=1.0 isoline
   1.0   1.0   1.5
   1.0   2.0   2.0
   1.0   4.0   3.0
```

As can be seen here, the slowly changing outer independent variable appears to the left while the rapidly changing inner independent variable appears to the right; isoline ordinates are repeated for each sample on a given isoline.

Each sample point is separated by a newline and each column is separated by one or more spaces or tabs. Comments begin with '#' and continue to the end of that line. They may appear anywhere in the file. Blank lines are ignored. The numbers shall be real or integer.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

When the data source is a sequence of 1-D arrays the isolines are laid out in conceptually the same way with each array being just as a column in the file format described above. Arrays may be specified directly via the concatenation operator or via array variable names.

The state of the data source is captured on the first call to the table model function. Any change after this point is ignored.

While it is suggested that the user arrange the sampled isolines in sorted order (one isoline following another in all dimensions); if the user provides the data in random order the system will sort the data into isolines in each dimension. Whether the data is sorted or not, the system determines the isoline ordinate by reading its exact value from the file or array. Any noise on the isoline ordinate may cause the system to incorrectly generate multiple isolines where the user intended a single isoline.

The input example above illustrated the isoline format for a single two-dimensional function (or dependent variable). The file may contain multiple dependent variables, all sharing the same set of isoline samples. A column in the data source may also be marked as *ignore*. These and all interpolation control settings are provided via the interpolation control string.

9.21.2 Control string

The control string is used to specify how the **`$table_model`** function should interpolate or lookup the data in each dimension and how it should extrapolate at the boundaries of each dimension. It also provides for some control on how to treat columns of the input data source. The string consists of a set of comma separated sub-strings followed by a semicolon and the dependent selector. The first group of sub-strings provide control over each independent variable with the first sub-string applying to the outermost dimension and so on. The dependent variable selector is a column number allowing us to specify which dependent variable in the data source we wish to interpolate. This number runs 1 though M with M being the total number of dependent variables specified in the data source.

Each sub-string associated with interpolation control has at most 3 characters. The first character controls interpolation and obeys Table 9-30.

Table 9-30—Interpolation control character

Control character	Description
`I`	Ignore this input column
`D`	Closest point (discrete) lookup
`1`	Linear interpolation (default)
`2`	Quadratic spline interpolation
`3`	Cubic spline interpolation

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The remaining character(s) in the sub-string specify the extrapolation behavior.

Table 9-31—Extrapolation control character

Control character	Description
C	Constant extrapolation
L	Linear extrapolation (default)
E	Error on an extrapolation request

The constant extrapolation method returns the table endpoint value. Linear extrapolation extends linearly to the requested point from the endpoint using a slope consistent with the selected interpolation method. The user may also disable extrapolation by choosing the error extrapolation method. With this method, an extrapolation error is reported if the **`$table_model`** function is requested to evaluate a point beyond the interpolation region.

For each dimension, users may use up to 2 extrapolation method characters to specify the extrapolation method used for each end. When no extrapolation method character is given, the linear extrapolation method will be used for both ends as default. Error extrapolation results in a fatal error being raised. When one extrapolation method character is given, the specified extrapolation method will be used for both ends. When two extrapolation method characters are given, the first character specifies the extrapolation method used for the end with the lower coordinate value, and the second character is used for the end with the higher coordinate value.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

9.21.3 Example control strings

Table 9-32—Example control strings

Control string	Description
`"" or control string omitted`	Null string, default linear interpolation and extrapolation. Dimensionality of the data is assumed to be N. Column N+1 is taken as the dependent.
`"1L,1L"`	Data is 2-D, linear interpolation and extrapolation in both dimensions.
`"1LL,1LL"`	Same as above, extrapolation method specified for both ends in each dimension.
`"1LL,1LL;1"`	Same as above, dependent variable 1 is specified. This is the default behavior when there are multiple dependent variables in the file and there is no dependent variable selector specified in the control string
`"D,1,3"`	Closest point lookup in the outer dimension, linear interpolation on dimension two and cubic spline interpolation on the inner dimension.
`"I,1CC,1CC;3"`	Ignore column 1, linear interpolation and constant extrapolation in all dimensions, interpolation applies to dependent variable 3. There are at least 6 column in the data file.
`"3,D,I,1;3"`	Cubic spline interpolation in dimension 3, (column 1), closest lookup in dimension 2 (column 2), ignore column 3, and use linear interpolation on the innermost dimension (dimension 1, column 4). Interpolate dependent variable 3 (column 7). This file has at least 7 columns.
`"C,,3"`	Data is 3D, equivalent to “1CC, 1LL, 3LL”.

9.21.4 Interpolation algorithms

The closest point lookup algorithm returns the closest point in the specified dimension. When the lookup ordinate is equidistant from two bracketing samples the function snaps away from zero.

The linear interpolation algorithm provides a simple linear interpolation between the closest sample points on a given isoline. Cubic spline interpolation[1] generates a spline for each isoline being interpolated. The extrapolation option specified is taken into account when generating the spline coefficients so as to avoid end point discontinuities in the first order derivative of the interpolated function.

When formulating the cubic spline equations the desired derivative of the interpolation function at both end points must be specified in order to provide the complete set of constraints for the cubic spline equations. It is convenient then that the table model function specifies end point extrapolation behavior. If the user selects linear extrapolation this leads to a *natural* spline. If constant extrapolation is specified the end point derivative is set to zero thus avoiding a discontinuity in the first order derivative at that end point.

Quadratic splines are similar to cubic splines, offering more efficient evaluation with generally less favorable interpolation results. Again one should attempt to avoid end point discontinuities, though it is not always possible in this case.

As a general rule cubic splines are best applied to smoothly varying samples (such as the DC I-V characteristic of a diode) while linear interpolation is a better option for data with abrupt transitions (such as a transient pulsed waveform).

[1]Numerical Methods in Scientific Computing, Germund Dahlquist and Åke Björck.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

9.21.5 Example

A simple call to the **$table_model** function is illustrated using the two dimensional sample set given in the above plots and tables. Let's first assume this data is stored in a file called *sample.dat*. The data is two dimensional with a single dependent variable. The independent variables were named *x* and *y* above. *y* is the outermost independent variable in the sample set while *x* is the innermost independent variable.

```
module example_tb(a, b);
   electrical a, b;
   inout a, b;
   analog begin
      I(a, b) <+ $table_model(0.0, V(a,b),"sample.dat");
   end
endmodule
```

This instance specifies zero for *y* and uses a module potential to interpolate *x*. No control string is specified and so the function defaults to performing linear interpolation and linear extrapolation in both dimensions.

It is possible to specify how to perform the interpolation via the control string:

```
      I(a, b) <+ $table_model(0.0, V(a,b),"sample.dat", "1LL,3LL");
```

Linear interpolation and extrapolation are specified in *y* and cubic interpolation with linear extrapolation in *x*.

The data source may also be specified as an array.

```
module example_tb(a, b);
   electrical a, b;
   inout a, b;
   real y[0:11], x[0:11],f_xy[0:11];
   analog begin
      @(initial_step) begin
         // y=0.0 isoline
         y[0] =0.0; x[0] =1.0; f_xy[0] =0.5;
         y[1] =0.0; x[1] =2.0; f_xy[1] =1.0;
         y[2] =0.0; x[2] =3.0; f_xy[2] =1.5;
         y[3] =0.0; x[3] =4.0; f_xy[3] =2.0;
         y[4] =0.0; x[4] =5.0; f_xy[4] =2.5;
         y[5] =0.0; x[5] =6.0; f_xy[5] =3.0;
         // y=0.5 isoline
         y[6] =0.5; x[6] =1.0; f_xy[6] =1.0;
         y[7] =0.5; x[7] =3.0; f_xy[7] =2.0;
         y[8] =0.5; x[8] =5.0; f_xy[8] =3.0;
         // y=1.0 isoline
         y[9] =1.0; x[9] =1.0; f_xy[9] =1.5;
         y[10]=1.0; x[10]=2.0; f_xy[10]=2.0;
         y[11]=1.0; x[11]=4.0; f_xy[11]=3.0;
      end
      I(a, b) <+ $table_model(0, V(a,b), y, x, f_xy);
   end
endmodule
```

Here the array is specified via array variables. The variables are initialized inside an **initial_step** block ensuring that they do not change after the first call to **$table_model**. Arrays may also be specified directly via an array assignment pattern.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

9.22 Connectmodule driver access system functions and operator

Verilog-AMS HDL extends IEEE Std 1364-2005 Verilog HDL with a set of driver access functions. Verilog-AMS HDL also adds a new operator, **`driver_update`** that is used in combination with the driver access functions which is described here for that reason.

Access to individual drivers and net resolution is necessary for accurate implementation of connect modules (see 7.5). A *driver* of a signal is a process which assigns a value to the signal, or a connection of the signal to an output port of a module instance or simulation primitive.

The driver access functions described here only access drivers found in ordinary modules and not to those found in connect modules. Driver access functions can only be called from connect modules.

A signal can have any number of drivers; for each driver the current status, value, and strength can be accessed.

9.22.1 $driver_count

`$driver_count` returns an integer representing the number of drivers associated with the signal in question. The syntax is shown in Syntax 9-17.

---

driver_count_function ::=
    `$driver_count` **`(`** signal_name **`)`**

Syntax 9-17—Syntax for $driver_count

The drivers are arbitrarily numbered from `0` to `N-1`, where `N` is the total number of ordinary drivers contributing to the signal value. For example, if this function returns a value `5` then the signal has five drivers numbered from `0` to `4`.

9.22.2 $receiver_count

`$receiver_count` returns an integer representing the number of receivers associated with the signal in question. The syntax is shown in Syntax 9-18.

---

receiver_count_function ::=
    `$receiver_count` **`(`** signal_name **`)`**

Syntax 9-18—Syntax for $receiver_count

The receivers are arbitrarily numbered from `0` to `N-1`, where `N` is the total number of ordinary receivers being contributed by the signal value. For example, if this function returns a value `5` then the signal has five receivers numbered from `0` to `4`.

9.22.3 $driver_state

`driver_state` returns the current value contribution of a specific driver to the state of the signal. The syntax is shown in Syntax 9-19.

---

driver_state_function ::=

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
$driver_state ( signal_name , driver_index )
```

Syntax 9-19—Syntax for $driver_state

driver_index is an integer value between `0` and `N-1`, where `N` is the total number of drivers contributing to the signal value. The state value is returned as `0`, `1`, `x`, or `z`.

9.22.4 $driver_strength

`driver_strength` returns the current strength contribution of a specific driver to the strength of the signal. The syntax is shown in Syntax 9-20.

```
driver_strength_function ::=
    $driver_strength ( signal_name , driver_index )
```

Syntax 9-20—Syntax for $driver_strength

driver_index is an integer value between `0` and `N-1`, where `N` is the total number of drivers contributing to the signal value. The strength value is returned as two strengths, `Bits 5-3` for `strength0` and `Bits 2-0` for `strength1` (see IEEE Std 1364-2005 Verilog HDL, subclauses 7.10 and 7.11).

If the value returned is `0` or `1`, `strength0` returns the high-end of the strength range and `strength1` returns the low-end of the strength range. Otherwise, the strengths of both `strength0` and `strength1` is defined as shown in Figure 9-3 below.

strength0									**strength1**								
Bits	7 Su0	6 St0	5 Pu0	4 La0	3 We 0	2 Me0	1 Sm0	0 HiZ0	0 HiZ1	1 Sm1	2 Me1	3 We 1	4 La1	5 Pu1	6 St1	7 Su1	Bits
B5	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	B2
B4	1	1	0	0	1	1	0	0	0	0	1	1	0	0	1	1	B1
B3	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	B0

Figure 9-3: Strength value mapping

9.22.5 driver_update

The status of drivers for a given signal can be monitored with the event detection keyword **`driver_update`**. It can be used in conjunction with the event detection operator `@` to detect updates to any of the drivers of the signal.

Example:

```
always @(driver_update clock)
   statement;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

causes the `statement` to execute any time a driver of the signal `clock` is updated. Here, an update is defined as the addition of a new pending value to the driver. This is true whether or not there is a change in the resolved value of the signal.

9.22.6 Receiver net resolution

As a result of driver receiver segregation, the drivers and receivers are separated so that any analog connected to a mixed net has the opportunity to influence the value driving the digital receivers. Since a single digital port is used in the connect module, the user must specify the value that the receivers will see. By not specifying the receiver value directly in the connect module driver, receiver segregation will be ignored, which is the default case. This assignment of the receiver value is done via the **`assign`** statement in which the digital port will be used to read the driver values as well as to set the receiver value.

1) The default is equivalent of `assign d_receivers = d_drivers ;`

   Where the value passed to the receivers through driver receiver segregation is the value being driven without delay or any impact from analog connections to the net. This is essentially bypassing driver receiver segregation.

2) Anything else is done explicitly, such as:

```
reg out; // value of out determined in CM, see example in 9.22.7
assign d = out;
```

   In this case, the digital port of the connect module will drive the receivers with a value determined in the connect module. This value may potentially be different from the value of the drivers of the connect module digital port.

9.22.7 Connect module example using driver access functions

Using the example shown in Figure 9-4, a connect module can be created using driver access functions to accurately model the effects of multiple drivers on an interface.

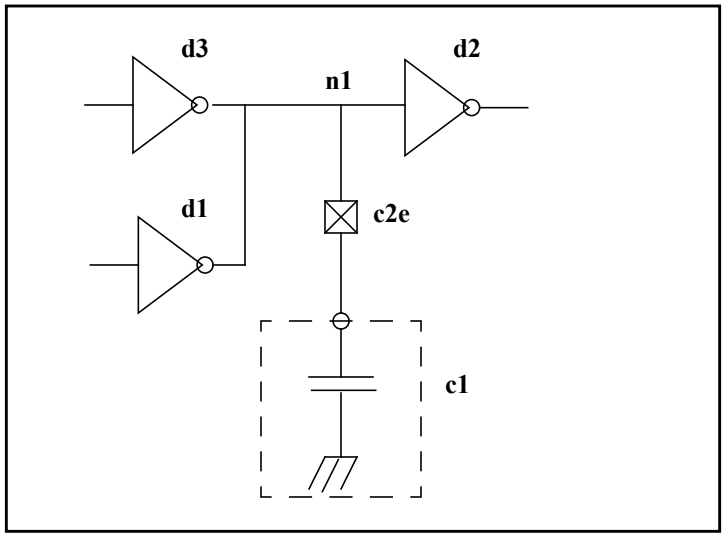


Figure 9-4: Driver-receiver segregation connect module example

The connect module below takes advantage of much of the mixed-signal language including driver access functions. This module effectively adds another parallel resistor from output to ground whenever a digital output connected to the net goes high, and another parallel resistor from output to rail (*supply*) whenever a digital output connected to the net goes low. If this is used as the connect module in Figure 9-4, not only is

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

the delay from digital outputs to the digital input a function of the value of the capacitor, for a given capacitance it takes approximately half the time (since two gates are driving the signal rather than one).

```
connectmodule c2e(d,a);
   input d;
   output a;

   ddiscrete d;
   electrical a, rail, gnd;

   reg out;
   ground gnd;
   branch (rail,a)pull_up;
   branch (a,gnd)pull_down;
   branch (rail,gnd)power;
   parameter real impedance0 = 120.0;
   parameter real impedance1 = 100.0;
   parameter real impedanceOff = 1e6;
   parameter real vt_hi = 3.5;
   parameter real vt_lo = 1.5;
   parameter real supply = 5.0;
   integer i, num_ones, num_zeros;

   assign d=out;
   initial begin
      num_ones = 0;
      num_zeros = 0;
   end

   always @(driver_update(d)) begin
      num_ones = 0;
      num_zeros = 0;
      for ( i = 0; i < $driver_count(d); i=i+1)
         if ( $driver_state(d,i) == 1 )
            num_ones = num_ones + 1;
         else
            num_zeros = num_zeros + 1;
   end

   always @(cross(V(a) - vt_hi, -1) or cross(V(a) - vt_lo, +1))
      out = 1'bx;
   always @(cross(V(a) - vt_hi, +1))
      out = 1'b1;
   always @(cross(V(a) - vt_lo, -1))
      out = 1'b0;

   analog begin
      // Approximately one impedance1 resistor to rail per high output
      // connected to the digital net
      V(pull_up) <+ 1/((1/impedance1)*num_ones+(1/impedanceOff)) *
         I(pull_up);
      // Approximately one impedance0 resistor to ground per low output
      // connected to the digital net
      V(pull_down) <+ 1/((1/impedance0)*num_zeros+(1/impedanceOff)) *
         I(pull_down);
      V(power) <+ supply;
   end
endmodule
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

9.23 Supplementary connectmodule driver access system functions

Verilog-AMS HDL extends IEEE Std 1364-2005 Verilog HDL so that a set of supplementary driver access functions are supported in the digital context of connectmodules.

These driver access functions are provided for access to digital events which have been scheduled onto a driver but might not have matured by the current simulation time.

These functions can be used to create analog waveforms which cross a specified threshold at the same time the digital event matures, thus providing accurate registration of analog and digital representations of a signal. This assumes there is at least as long a delay in the maturation of the digital signal as the required rise/fall times of the analog waveform.

NOTE—Because the scheduled digital events can be scheduled with an insufficient delay or canceled before they mature, be careful when using these functions.

9.23.1 $driver_delay

`$driver_delay` returns the delay, from current simulation time, after which the pending state or strength becomes active. If there is no pending value on a signal, it returns the value minus one (-1.0). The syntax is shown in Syntax 9-21.

---

driver_delay_function ::=
    `$driver_delay` **`(`** signal_name **`,`** driver_index **`)`**

Syntax 9-21—Syntax for $driver_delay

driver_index is an integer value between `0` and `N-1`, where `N` is the total number of drivers contributing to the signal value. The returned delay value is a real number, which is defined by the **`` `timescale``** for that module where the call has been made. The fractional part arises from the possibility of a driver being updated by an A2D event off the digital timeticks.

9.23.2 $driver_next_state

`$driver_next_state` returns the pending state of the driver, if there is one. If there is no pending state, it returns the current state. The syntax is shown in Syntax 9-22.

---

driver_next_state_function ::=
    `$driver_next_state` **`(`** signal_name **`,`** driver_index **`)`**

Syntax 9-22—Syntax for $driver_next_state

driver_index is an integer value between `0` and `N-1`, where `N` is the total number of drivers contributing to the signal value. The pending state value is returned as `1'b0`, `1'b1`, `1'bx`, or `1'bz`.

9.23.3 $driver_next_strength

`$driver_next_strength` returns the strength associated with the pending state of the driver, if there is one. If there is no pending state, it returns the current strength. The syntax is shown in Syntax 9-23.

---

driver_next_strength_function ::=

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
$driver_next_strength ( signal_name , driver_index )
```

Syntax 9-23—Syntax for$ driver_next_strength

driver_index is an integer value between `0` and `N-1`, where `N` is the total number of drivers contributing to the signal value. The pending strength value is returned as an integer between `0` and `7`.

9.23.4 $driver_type

`$driver_type` returns an integer value with its bits set according to the system header file “driver_access.vams” (refer to Annex D for the header file) for the driver specified by the signal_name and the driver_index. Connect modules for digital to analog conversion can use the returned information to help minimize the difference between the digital event time and the analog crossover when the user swaps between coding styles and performs backannotation[1]. A simulator that cannot provide proper information for a given driver type should return 0 (‘DRIVER_UNKNOWN). All drivers on *wor* and *wand* nets will have a bit set indicating such, and any extra drivers added by the kernel for pull-up or pull-down will be marked as belonging to the kernel. The syntax is shown in Syntax 9-24.

```
driver_type_function ::=
    $driver_type ( signal_name , driver_index )
```

Syntax 9-24—Syntax for$ driver_type

Digital primitives (like nand and nor gates) should always provide data about their scheduled output changes; i.e., a gate with a 5ns delay should provide 5ns of look-ahead. Behavioral code with blocking assigns cannot provide look-ahead, but non-blocking assigns with delays can. However, since the capability is implementation- and configuration-dependent, this function is provided so that the connect module can adapt for a particular instance.

[1]SDF backannotation will not change which D2A is inserted.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

10. Compiler directives

10.1 Overview

All Verilog-AMS HDL compiler directives are preceded by the (`) character. This character is called accent grave (ASCII 0x60). It is different from the character ('), which is the apostrophe character (ASCII 0x27). 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.

The following compiler directives are supported:

`__FILE__	[10.7]
`__LINE__	[10.7]
`begin_keywords	[10.6]
`celldefine	[IEEE Std 1364 Verilog]
`default_discipline	[10.2]
`default_nettype	[IEEE Std 1364 Verilog]
`default_transition	[10.3]
`define	[10.4]
`else	[IEEE Std 1364 Verilog]
`elsif	[IEEE Std 1364 Verilog]
`end_keywords	[10.6]
`endcelldefine	[IEEE Std 1364 Verilog]
`endif	[IEEE Std 1364 Verilog]
`ifdef	[IEEE Std 1364 Verilog]
`ifndef	[IEEE Std 1364 Verilog]
`include	[IEEE Std 1364 Verilog]
`line	[IEEE Std 1364 Verilog]
`nounconnected_drive	[IEEE Std 1364 Verilog]
`pragma	[IEEE Std 1364 Verilog]
`resetall	[IEEE Std 1364 Verilog]
`timescale	[IEEE Std 1364 Verilog]
`unconnected_drive	[IEEE Std 1364 Verilog]
`undef	[10.4]

10.2 `default_discipline

The default discipline is applied by discipline resolution (see 7.4 and Annex F)to all discrete signals without a discipline declaration that appear in the text stream following the use of the **`default_discipline** directive, until either the end of the text stream or another **`default_discipline** directive with the qualifier (if applicable) is found in the subsequent text, even across source file boundaries. Therefore, more than one **`default_discipline** directive can be in force simultaneously, provided each differs in qualifier.

In addition to **`resetall**, if this directive is used without a discipline name, discipline resolution will not use a default discipline for nets declared after this directive is encountered in the text stream. Syntax 10-1 shows the syntax for this directive.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
default_discipline_directive ::=
      `default_discipline [discipline_identifier [ qualifier ] ]
qualifier ::=
      integer | real | reg | wreal | wire | tri | wand | triand
    | wor | trior | trireg | tri0 | tri1 | supply0 | supply1
```

Syntax 10-1—Syntax for the default discipline compiler directive

Example:

```
`default_discipline ddiscrete
module behavnand(in1, in2, out);
   input in1, in2;
   output out;
   reg out;
   always begin
      out = ~(in1 && in2);
   end
endmodule
```

This example illustrates the usage of the **`default_discipline** directive. The nets `in1`, `in2`, and `out` all have discipline `ddiscrete` by default.

There is a precedence of compiler directives; the more specific directives have higher precedence over general directives.

10.3 `default_transition

The scope of this directive is similar to the scope of the **`define** compiler directive although it can be used only outside of module definitions. This directive specifies the default value for rise and fall time for the transition filter (see 4.5.8). There are no scope restrictions for this directive. The syntax for this directive is shown in Syntax 10-2.

```
default_transition_compiler_directive ::=
      `default_transition transition_time
transition_time ::=
      constant_expression
```

Syntax 10-2—Syntax for default transition compiler directive

transition_time is a real value.

For all transition filters which follow this directive and do not have rise time and fall time arguments specified, *transition_time* is used as the default rise and fall time values. If another **`default_transition** directive is encountered in the subsequent source description, the transition filters following the newly encountered directive derive their default rise and fall times from the transition time value of the newly encountered directive. In other words, the default rise and fall times for a transition filter are derived from the *transition_time* value of the directive which immediately precedes the transition filter.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

If a **`default_transition** directive is not used in the description, *transition_time* is controlled by the simulator.

10.4 `define and `undef

The **`define** and **`undef** compiler directives are described in IEEE Std 1364 Verilog.

To avoid conflicts with predefined Verilog-AMS macros (10.5), the **`define** compiler directive's macro text shall not begin with __VAMS_. The **`undef** compiler directive shall have no effect on predefined Verilog-AMS macros; the simulator may issue a warning for an attempt to undefine one of these macros.

The syntax for text macro definitions is given in Syntax 10-3

---

text_macro_definition ::=
    `define text_macro_name macro_text
text_macro_name ::=
    text_macro_identifier [**(** list_of_formal_arguments **)**]
list_of_formal_arguments ::=
    formal_argument_identifier { **,** formal_argument_identifier }
formal_argument_identifier ::=
    simple_identifier
text_macro_identifier ::=
    identifier

Syntax 10-3—Syntax for text macro definition (not in Annex A)

10.5 Predefined macros

Verilog-AMS HDL supports a predefined macro to allow modules to be written that work with both IEEE Std 1364 Verilog and Verilog-AMS HDL.The predefined macro is called **__VAMS_ENABLE__**.

This macro shall always be defined during the parsing of Verilog-AMS source text. Its purpose is to support the creation of modules which are both legal Verilog and Verilog-AMS. The Verilog-AMS features of such modules are made visible only when the **__VAMS_ENABLE__** macro has previously been defined.

Example:

```
module not_gate(in, out);
   input in;
   output out;
   reg out;
   `ifdef __VAMS_ENABLE__
      parameter integer del = 1 from [1:100];
   `else
      parameter del = 1;
   `endif
   always @ in
      out = #del !in;
endmodule
```

Verilog-AMS HDL version 2.2 introduced a number of extensions to support compact modeling. A predefined macro allows modules to add functionality if these extensions are supported, or to generate warnings

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

or errors if they are not. This predefined macro is called **__VAMS_COMPACT_MODELING__** and shall be defined during the parsing of Verilog-AMS source text if and only if all the compact modeling extensions are supported by the simulator.

The following example computes noise of a nonlinear resistor only if the extensions, specifically `ddx`, are supported.

```
module nonlin_res(a, b);
   input a, b;
   electrical a, b;
   parameter real rnom = 1;
   parameter real vc1 = 0;
   real reff, iab;
   analog begin
      iab = V(a,b) / (rnom * (1.0 + vc1 * V(a,b)));
      I(a,b) <+ iab;
   `ifdef __VAMS_COMPACT_MODELING__
      reff = ddx(iab, V(a));
      I(a,b) <+ white_noise(4.0*`P_K*$temperature*reff, "thermal");
   `else
      if (analysis("noise"))
         $strobe("Noise not computed.");
   `endif
   end
endmodule
```

Verilog-AMS simulators shall also provide a predefined macro so that the module can conditionally include (or exclude) portions of the source text specific to a particular simulator. This macro shall be documented in the Verilog-AMS section of the simulator manual.

10.6 `begin_keywords and `end_keywords

Verilog-AMS HDL extends the **`begin_keywords** and **`end_keywords** compiler directives from IEEE Std 1364 Verilog as well existing extensions made in previous versions of this standard by adding a "VAMS-2023" version specifier.

The version_specifier specifies the valid set of reserved keywords in effect when a design unit is parsed by an implementation. The **`begin_keywords** and **`end_keywords** directives can only be specified outside of a design element (module, primitive, configuration, paramset, connectrules or connectmodule). The **`begin_keywords** directive affects all source code that follows the directive, even across source code file boundaries, until the matching **`end_keywords** directive is encountered.

The version_specifier, "VAMS-2023" specifies that only the identifiers listed as reserved keywords in the Verilog-AMS HDL are considered to be reserved words. These identifiers are listed in Annex B.

The **`begin_keywords** and **`end_keywords** directives only specify the set of identifiers that are reserved as keywords. The directives do not affect the semantics, tokens, and other aspects of the Verilog-AMS language.

The version specifiers "1364-1995", "1364-2001", "1364-2005" and "VAMS-2.3" must also be supported. "1364-1995" represents the keywords for IEEE Std 1364-1995. "1364-2001" represents the keywords for IEEE Std 1364-2001. "1364-2005" represents the keywords for IEEE Std 1364-2005. "VAMS-2.3" represents the keywords for Verilog-AMS 2.3 HDL.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

In the example below, it is assumed that the definition of module `m1` does not have a **`begin_keywords** directive specified prior to the module definition. Without this directive, the set of reserved keywords in effect for this module shall be the implementation's default set of reserved keywords.

```
module m1; // module definition with no `begin_keywords directive
   ...
endmodule
```

The following example specifies a **`begin_keywords** `"1364-2005"` directive. The source code within the module uses the identifier `sin` as a port name. The **`begin_keywords** directive would be necessary in this example if an implementation uses Verilog-AMS as its default set of keywords because **sin** is a reserved keyword in Verilog-AMS but not in 1364-2005. Specifying the `"1364-1995"` or `"1364-2001"` Verilog keyword lists would also work with this example.

```
`begin_keywords "1364-2005" // use IEEE Std 1364-2005 Verilog keywords
module m2 (sin ...);
   input sin; // OK since sin is not a keyword in 1364-2005
   ...
endmodule
`end_keywords
```

The next example is the same code as the previous example, except that it explicitly specifies that the Verilog-AMS keywords should be used. This example shall result in an error because **sin** is reserved as a keyword in this standard.

```
`begin_keywords "VAMS-2023" // use Verilog-AMS LRM 2023 keywords
module m2 (sin, ... ); // ERROR: "sin" is a keyword in Verilog-AMS
   input sin;
   ...
endmodule
`end_keywords
```

The following example uses several Verilog-AMS constructs, and designates that the Verilog-AMS version 2023 set of keywords should be used. Note that the word "logic" is not a keyword in Verilog-AMS 2023, whereas it is a keyword in the IEEE Std 1800 SystemVerilog.

```
`begin_keywords "VAMS-2023"
discipline \logic;
   domain discrete;
enddiscipline
module a2d(dnet, anet);
   input dnet;
   wire dnet;
   logic dnet;
   output anet;
   electrical anet;
   real avar;
   analog begin
      if (dnet === 1'b1)
         avar = 5;
      else if (dnet === 1'bx)
         avar = avar;            // hold value
      else if (dnet === 1'b0)
         avar = 0;
      else if (dnet === 1'bz)
         avar = 2.5;             // high impedance - float value
      V(anet) <+ avar;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    end
  endmodule

`end_keywords
```

10.7 `__FILE__ and `__LINE__

`__FILE__ expands to the name of the current input file, in the form of a string literal. This is the path by which a tool opened the file, not the short name specified in `include or as a tool's input file name argument. The format of this path name may be implementation dependent.

`__LINE__ expands to the current input line number, in the form of a simple decimal number.

`__FILE__ and `__LINE__ are useful in generating an error message to report a problem; the message can state the source line at which the problem was detected.

For example:

```
$display("Internal error: null handle at %s, line %d.",
`__FILE__, `__LINE__);
```

An `include directive changes the expansions of `__FILE__ and `__LINE__ to correspond to the included file. At the end of that file, when processing resumes on the input file that contained the `include directive, the expansions of `__FILE__ and `__LINE__ revert to the values they had before the `include (but `__LINE__ is then incremented by one as processing moves to the line after the `include).

A `line directive (as defined in IEEE Std 1364 Verilog) changes `__LINE__ and may change `__FILE__ as well.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11. Using VPI routines

11.1 Overview

Clause 11 and Clause 12 specify the Verilog Procedural Interface (VPI) for the Verilog-AMS HDL. This clause describes how the VPI routines are used and Clause 12 defines each of the routines in alphabetical order.

11.2 The VPI interface

The VPI interface provides routines which allow Verilog-AMS product users to access information contained in a Verilog-AMS design and allow facilities to interact dynamically with a software product. Applications of the VPI interface can include delay calculators and annotators, connecting a Verilog-AMS 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-AMS HDL objects and simulation specific objects

11.2.1 VPI callbacks

Dynamic software product interaction shall be accomplished with a registered callback mechanism. VPI callbacks shall allow a user to request a Verilog-AMS HDL software product, such as a logic simulator, call a user-defined application when a specific activity occurs. For example, the user can request the user application `my_monitor()` be called when a particular net changes value or `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. to be used.

The reasons for providing callbacks 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()`**, **`vpi_register_systf()`** and **`vpi_register_analog_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-AMS HDL product is first invoked. A primary use of this facility shall be for the registration of user-defined system tasks and functions.

11.2.2 VPI access to Verilog-AMS HDL objects and simulation objects

Accessible Verilog-AMS HDL objects and simulation objects and their relationships and properties are described using data model diagrams. These diagrams are presented in 11.6. The data diagrams indicate the

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

routines and constants which are required to access and manipulate objects within an application environment. An associated set of routines to access these objects is defined in Clause 12.

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 Clause 12.

VPI routines provide access to objects in an *instantiated* Verilog-AMS 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-AMS 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.

11.2.3 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.

11.3 VPI object classifications

VPI objects are classified with data model diagrams. These diagrams provide a graphical representation of those objects within a Verilog-AMS design to which the VPI routines shall provide access. The diagrams shall show the relationships between objects and the properties of each object. Objects with sufficient commonality are placed in groups. Group relationships and properties apply to all the objects in the group.

As an example, the simplified diagram in Figure 11-1 shows there is a *one-to-many relationships* from objects of type `module` to objects of type `net` and a *one-to-one relationship* from objects of type `net` to objects of type `module`. Objects of type `net` have properties **`vpiName`**, **`vpiVector`**, and **`vpiSize`**, with the C data types string, Boolean, and integer respectively.

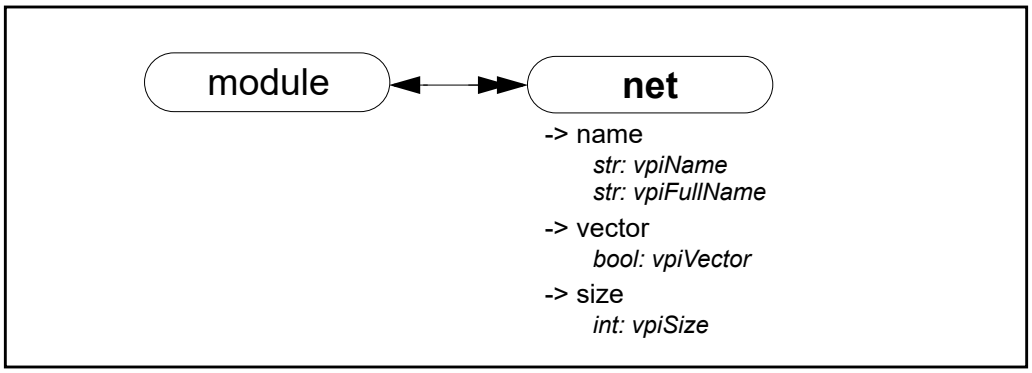


Figure 11-1: Object relationships

The VPI object data diagrams are presented in 11.6.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.3.1 Accessing object relationships and properties

The VPI interface defines the C data type of **vpiHandle**. All objects are manipulated via a **vpiHandle** variable. Object handles can be accessed from a relationship with another object, or from a hierarchical name, as the following example demonstrates.

Examples:

```
vpiHandle net;
net = vpi_handle_by_name("top.m1.w1", NULL);
```

This example call retrieves a handle to wire `top.m1.w1` and assigns it to the **vpiHandle** variable *net*. The `NULL` second argument directs the routine to search for the name from the top level of the design.

The VPI interface provides generic functions for tasks, such as traversing relationships and determining property values. One-to-one relationships are traversed with routine **vpi_handle()**.

In the following example, the module containing *net* is derived from a handle to that net:

```
vpiHandle net, mod;
net = vpi_handle_by_name("top.m1.w1", NULL);
mod = vpi_handle(vpiModule, net);
```

The call to **vpi_handle()** in the above example shall return a handle to module `top.m1`.

Properties of objects shall be derived with routines in the **vpi_get** family. The routine **vpi_get()** returns integer and Boolean properties. The routine **vpi_get_str()** accesses string properties.

To retrieve a pointer to the full hierarchical name of the object referenced by handle *mod*, the following call would be made:

```
char *name = vpi_get_str(vpiFullName, mod);
```

In the above example, character pointer `name` shall now point to the string `top.m1`.

One-to-many relationships are traversed with an iteration mechanism. The routine **vpi_iterate()** creates an object of type **vpiIterator**, which is then passed to the routine **vpi_scan()** to traverse the desired objects.

In the following example, each net in module `top.m1` is displayed:

```
vpiHandle itr;
itr = vpi_iterate(vpiNet,mod);
while (net = vpi_scan(itr) )
   vpi_printf("\t%s\n", vpi_get_str(vpiFullName, net) );
```

As the above examples illustrate, the routine naming convention is a *vpi* prefix with '_' word delimiters (with the exception of callback-related defined values, which use the *cb* prefix). Macro-defined types and properties have the *vpi* prefix and they use capitalization for word delimiters.

The routines for traversing Verilog-AMS HDL structures and accessing objects are described in IEEE Std 1364 Verilog, section 22.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.3.2 Delays and values

Properties are of type integer, boolean, real or string. Delay and logic value properties, however, are more complex and require specialized routines and associated structures. The routines **`vpi_get_delays()`** and **`vpi_put_delays()`** use structure pointers, where the structure contains the pertinent information about delays. Similarly, simulation values are also handled with the routines **`vpi_get_value()`** and **`vpi_put_value()`**, along with an associated set of structures. For analog tasks and functions, **`vpi_handle_multi()`** and **`vpi_put_value()`** support declaration and assignment of derivatives for the task arguments and function return values.

The routines and C structures for handling delays, derivatives, and logic values are presented in IEEE Std 1364 Verilog, section 22.

11.4 List of VPI routines by functional category

The VPI routines can be divided into groups based on primary functionality.

- VPI routines for simulation-related callbacks
- VPI routines for system task/function callbacks
- VPI routines for traversing Verilog-AMS HDL hierarchy
- VPI routines for accessing properties of objects
- VPI routines for accessing objects from properties
- VPI routines for delay processing
- VPI routines for logic and strength value processing
- VPI routines for task parameters derivatives processing
- VPI routines for analysis and simulation time processing
- VPI routines for miscellaneous utilities

Table 11-1 through Table 11-9 list the VPI routines by major category. IEEE Std 1364-2005 Verilog HDL, Section 22 defines each of the VPI routines, listed in alphabetical order.

Table 11-1—VPI routines for simulation related callbacks

To	Use
Register a simulation-related callback	**vpi_register_cb()**
Remove a simulation-related callback	**vpi_remove_cb()**
Get information about a simulation-related callback	**vpi_get_cb_info()**

Table 11-2—VPI routines for system task/function callbacks

To	Use
Register a system task/function callback	**vpi_register_systf()**
Get information about a system task/function callback	**vpi_get_systf_info()**

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 11-3—VPI routines for analog system task/function callbacks

To	Use
Register an analog system task/function callback	**vpi_register_analog_systf()**
Get information about an analog system task/function callback	**vpi_get_analog_systf_info()**

Table 11-4—VPI routines for traversing Verilog-AMS HDL hierarchy

To	Use
Obtain a handle for an object with a one-to-one relationship	**vpi_handle()**
Obtain handles for objects in a one-to-many relationship	**vpi_iterate()** **vpi_scan()**
Obtain a handles for an object in a many-to-one relationship	**vpi_handle_multi()**

Table 11-5—VPI routines for accessing properties of objects

To	Use
Get the value of objects with types of `int` or `bool`	**vpi_get()**
Get the value of objects with types of string	**vpi_get_str()**
Get the value of objects with types of real	**vpi_get_real()**

Table 11-6—VPI routines for accessing objects from properties

To	Use
Obtain a handle for a named object	**vpi_handle_by_name()**
Obtain a handle for an indexed object	**vpi_handle_by_index()**

Table 11-7—VPI routines for delay processing

To	Use
Retrieve delays or timing limits of an object	**vpi_get_delays()**
Write delays or timing limits to an object	**vpi_put_delays()**

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 11-8—VPI routines for logic, real, strength and analog value processing

To	Use
Retrieve logic value or strength value of an object	**vpi_get_value()**
Write logic value or strength value to an object	**vpi_put_value()**
Retrieve values of an analog object	**vpi_get_analog_value()**

Table 11-9—VPI routines for analysis and simulation time processing

To	Use
Find the current simulation time or the scheduled time of future events	**vpi_get_time()**
Find the current simulation time value in the continuous domain.	**vpi_get_analog_time()**
Find the current simulation time delta value in continuous domain.	**vpi_get_analog_delta()**
Find the current simulation frequency in the small-signal domain.	**vpi_get_analog_freq()**

Table 11-10—VPI routines for miscellaneous utilities

To	Use
Write to `stdout` and the current log file	**vpi_printf()**
Open a file for writing	**vpi_mcd_open()**
Close one or more files	**vpi_mcd_close()**
Write to one or more files	**vpi_mcd_printf()**
Retrieve the name of an open file	**vpi_mcd_name()**
Retrieve data about product invocation options	**vpi_get_vlog_info()**
See if two handles refer to the same object	**vpi_compare_objects()**
Obtain error status and error information about the previous call to a VPI routine	**vpi_chk_error()**
Free memory allocated by VPI routines	**vpi_free_object()**

11.5 Key to object model diagrams

This clause contains the keys to the symbols used in the object model diagrams. Keys are provided for objects and classes, traversing relationships, and accessing properties.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.5.1 Diagram key for objects and classes

obj defn	Object Definition**:** **Bold** letters in a solid enclosure indicate an object definition. The properties of the object are defined in this location.
object	Object Reference: Normal letters in a solid enclosure indicate an object reference.
class defn *class* **obj defn** object	Class Definition: ***Bold italic*** letters in a dotted enclosure indicate a class definition, where the class groups other objects and classes. Properties of the class are defined in this location. The class definition can contain an object definition.
class	Class Reference: *Italic* letters in a dotted enclosure indicate a class reference.
obj1 obj2	Unnamed Class: A dotted enclosure with no name is an unnamed class. It is sometimes convenient to group objects although they shall not be referenced as a group elsewhere, so a name is not indicated.

11.5.2 Diagram key for accessing properties

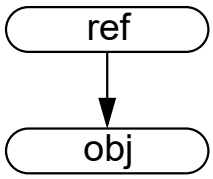
obj
-> vector
bool: vpiVector
-> size
int: vpiSize
	Integer and Boolean properties are accessed with the routine **vpi_get()**.
	
	Example: Given a **vpiHandle** `obj_h` to an object of type **vpiObj**, get the size of the object.
	
	`bool vect_flag = vpi_get(vpivector, obj_h);`
	`int size = vpi_get_size(vpiSize, obj_h);`

obj
-> name
str: vpiName
str: vpiFullName
	String properties are accessed with routine **vpi_get_str()**.
	
	Example:
	
	`char name[nameSize];`
	`vpi_get_str(vpiName, obj_h);`

object -> complex *func1()* *func2()*	Complex properties for time and logic value are accessed with the indicated routines. See the descriptions of the routines for usage.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

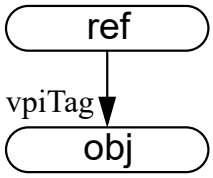
11.5.3 Diagram key for traversing relationships



A single arrow indicates a *one-to-one* relationship accessed with the routine **vpi_handle()**.

Example: Given **vpiHandle** variable `ref_h` of type `ref`, access `obj_h` of type **vpiObj**:

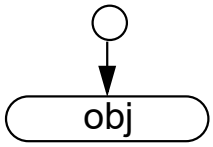
```
obj_h = vpi_handle(vpiObj, ref_h);
```



A tagged *one-to-one* relationship is traversed similarly, using **vpiTag** instead of **vpiObj**:

Example:

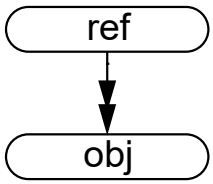
```
obj_h = vpi_handle(vpiTag, ref_h);
```



A top-level *one-to-one* relationship is traversed similarly, using `NULL` instead of `ref_h`:

Example:

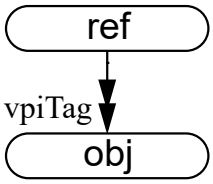
```
obj_h = vpi_handle(vpiObj, NULL);
```



A double arrow indicates a *one-to-many* relationship accessed with the routine **vpi_scan()**.

Example: Given **vpiHandle** variable `ref_h` of type `ref`, scan objects of type **vpiObj**:

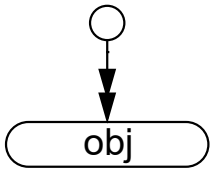
```
itr = vpi_iterate(vpiObj, ref_h);
while (obj_h = vpi_scan(itr) )
   /* process 'obj_h' */
```



A tagged *one-to-many* relationship is traversed similarly, using **vpiTag** instead of **vpiObj**:

Example:

```
itr = vpi_iterate(vpiTag, ref_h);
while (obj_h = vpi_scan(itr) )
   /* process 'obj_h' */
```



A top-level *one-to-many* relationship is traversed similarly, using `NULL` instead of `ref_h`:

Example:

```
itr = vpi_iterate(vpiObj, NULL);
while (obj_h = vpi_scan(itr) )
   /* process 'obj_h' */
```

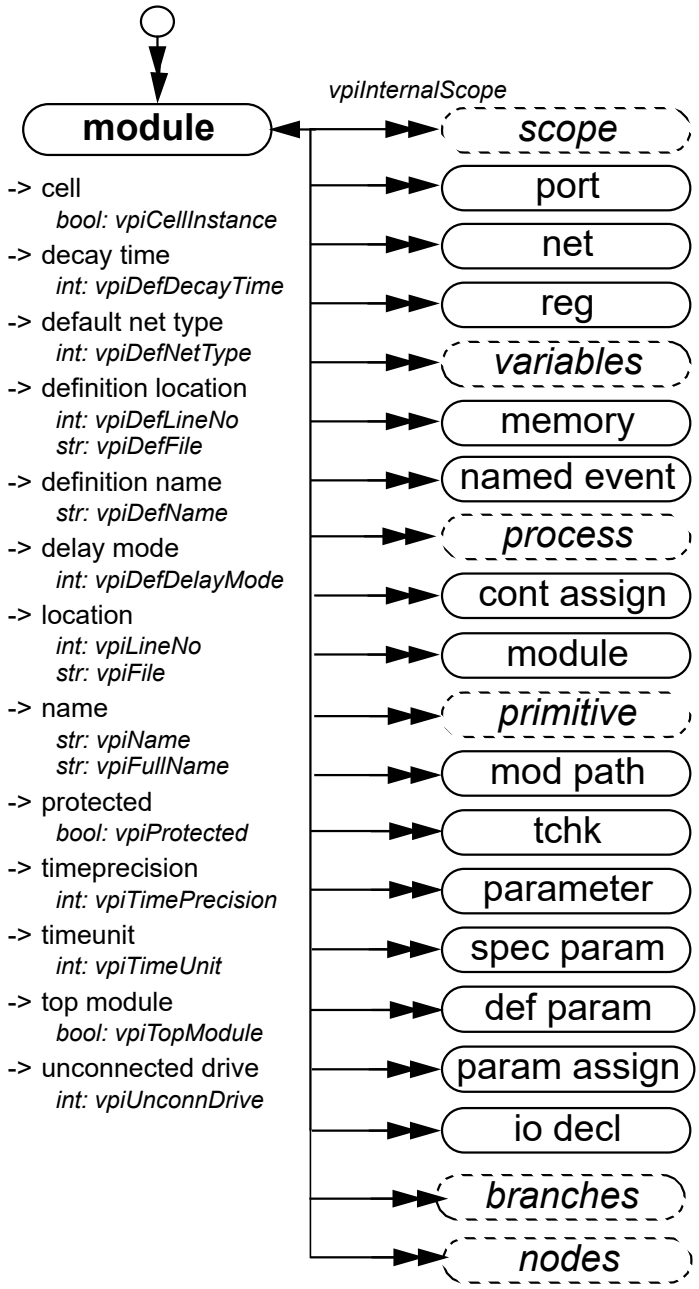

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6 Object data model diagrams

Subclauses in 11.6.1 through 11.6.25 contain the data model diagrams that define the accessible objects and groups of objects, along with their relationships and properties.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.1 Module



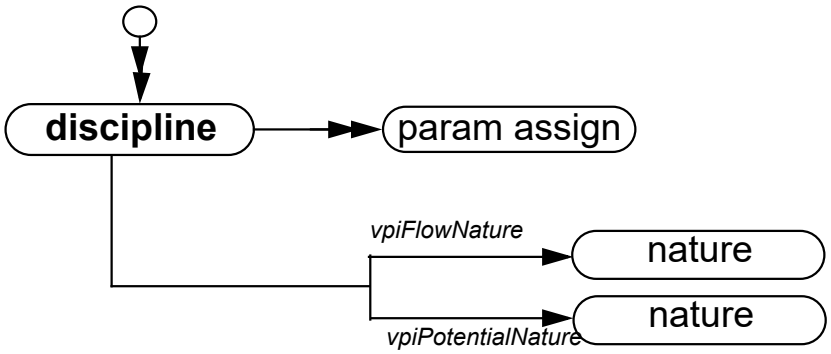
NOTES

1—Top-level modules shall be accessed using **vpi_iterate()** with a `NULL` reference object.

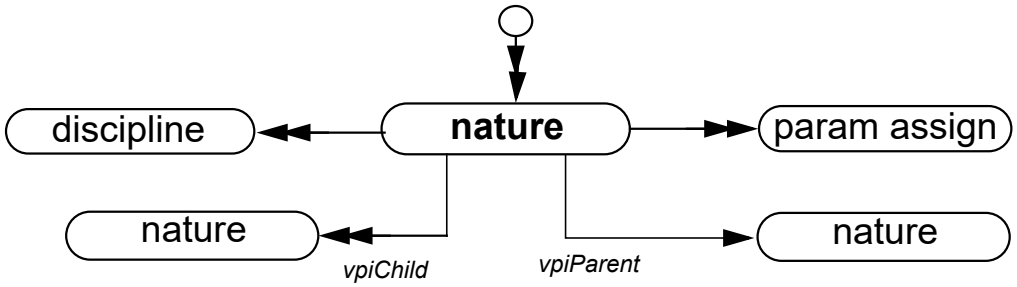
2—Passing a `NULL` handle to **vpi_get()** with types **vpiTimePrecision** or **vpiTimeUnit** shall return the smallest time precision of all modules in the instantiated design.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.2 Nature, discipline



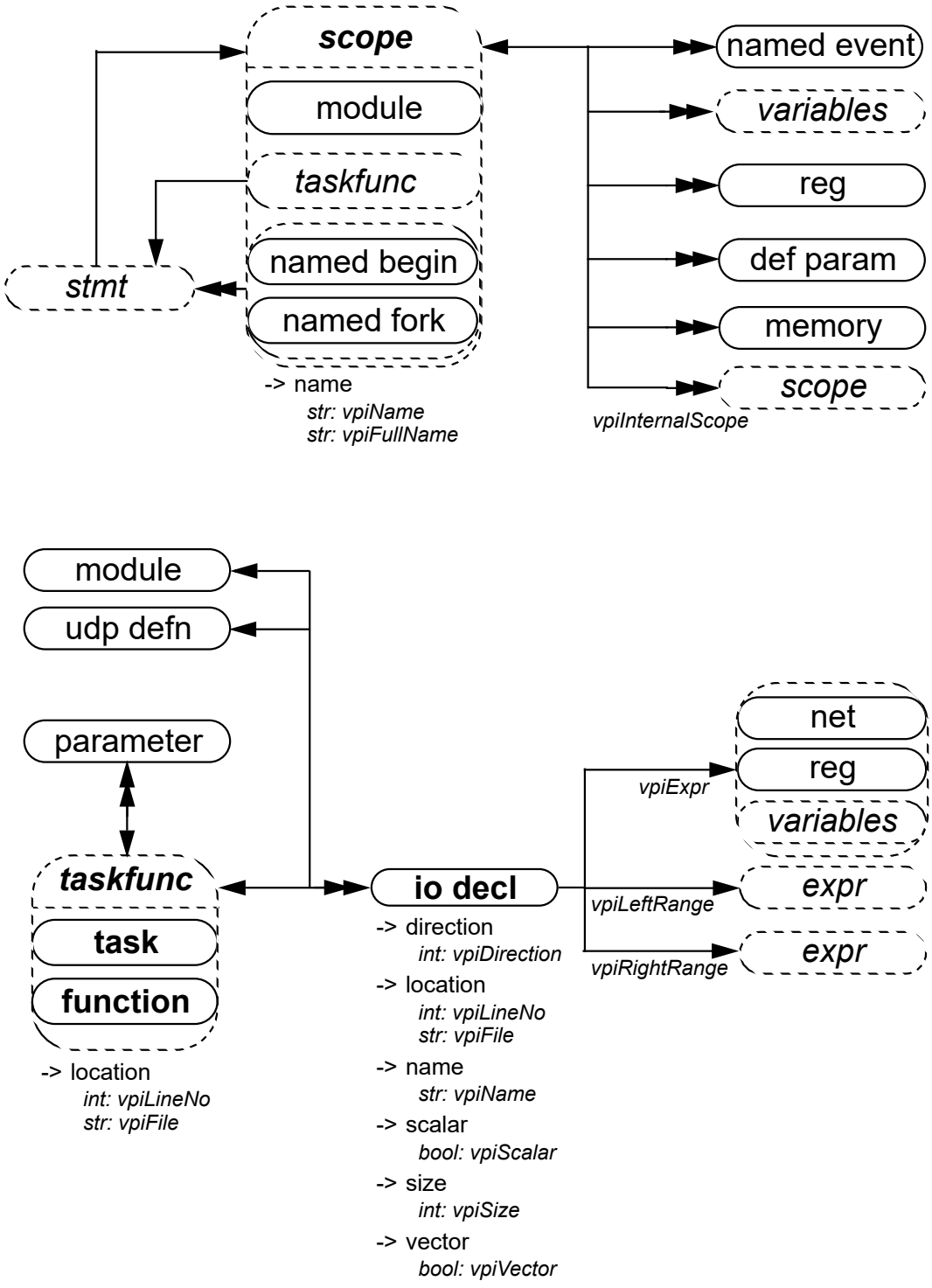
-> name
str: vpiName
str: vpiFullName



-> name
str: vpiName
str: vpiFullName

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

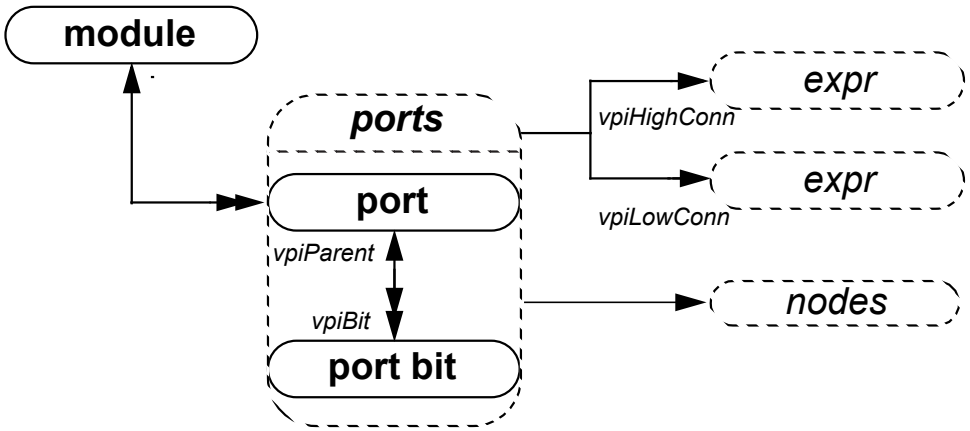
11.6.3 Scope, task, function, IO declaration



NOTE—A Verilog-AMS HDL function shall contain an object with the same name, size, and type as the function.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.4 Ports



-> connected by name
bool: vpiConnByName
-> delay (mipd)
vpi_get_delays()
vpi_put_delays()
-> direction
int: vpiDirection
-> explicitly named
bool: vpiExplicitName
-> index
int: vpiPortIndex
-> location
int: vpiLineNo
str: vpiFile
-> name
str: vpiName
str: vpiFullName
-> scalar
bool: vpiScalar
-> size
int: vpiSize
-> vector
bool: vpiVector

NOTES

1—**vpiHighConn** shall indicate the hierarchically higher (closer to the top module) port connection.

2—**vpiLowConn** shall indicate the lower (further from the top module) port connection.

3—Properties *scalar* and *vector* shall indicate if the port is 1 bit or more than 1 bit. They shall not indicate anything about what is connected to the port.

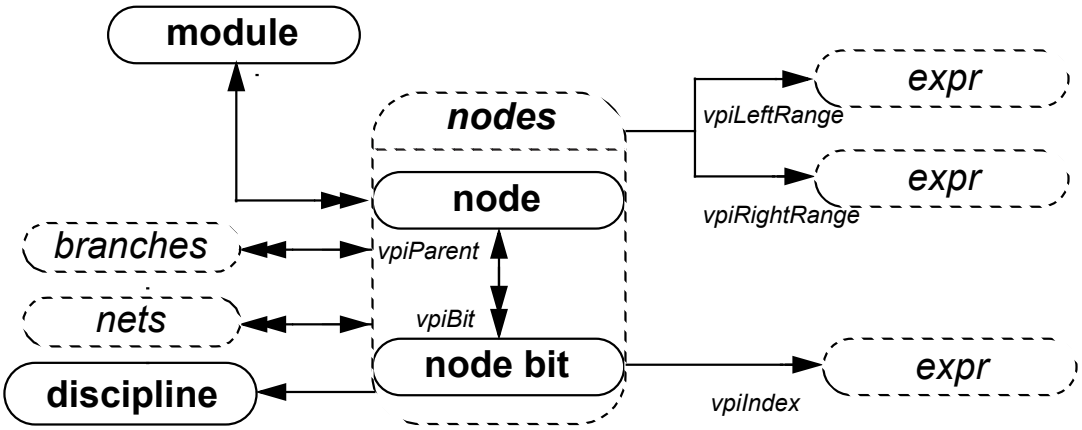
4—Properties *index* and *name* shall not apply for port bits.

5—If a port is explicitly named, then the explicit name shall be returned. If not, and a name exists, that name shall be returned. Otherwise, `NULL` shall be returned.

6—**vpiPortIndex** can be used to determine the port order.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.5 Nodes



-> implicitly declared
bool: vpiImplicitDecl

-> location
int: vpiLineNo
str: vpiFile

-> name
str: vpiName
str: vpiFullName

-> scalar
bool: vpiScalar

-> size
int: vpiSize

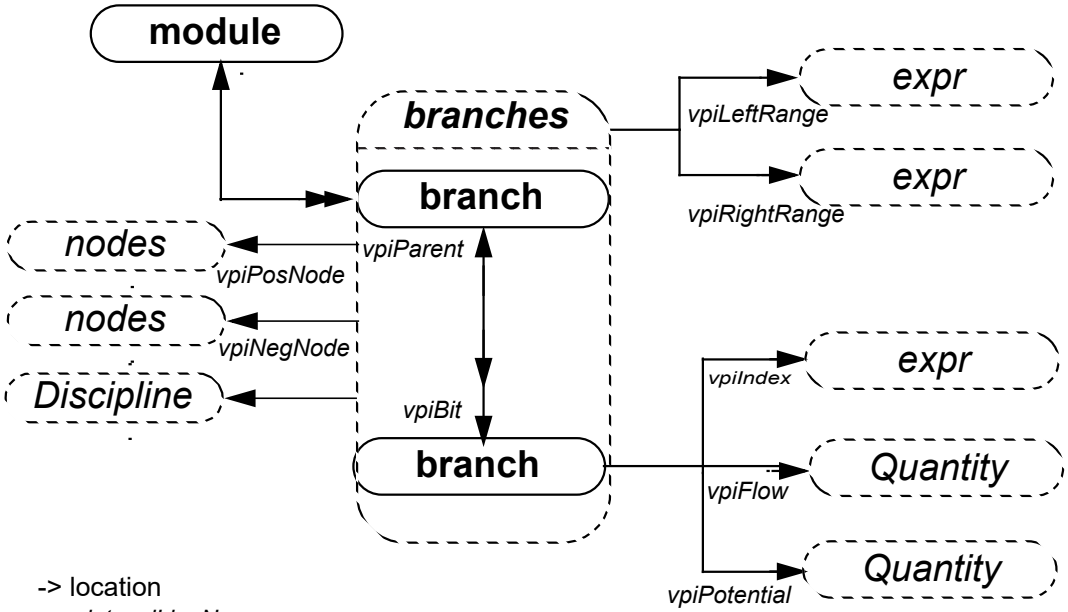
-> vector
bool: vpiVector

NOTES

1—Properties *scalar* and *vector* shall indicate if the node is 1 bit or more than 1 bit.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.6 Branches

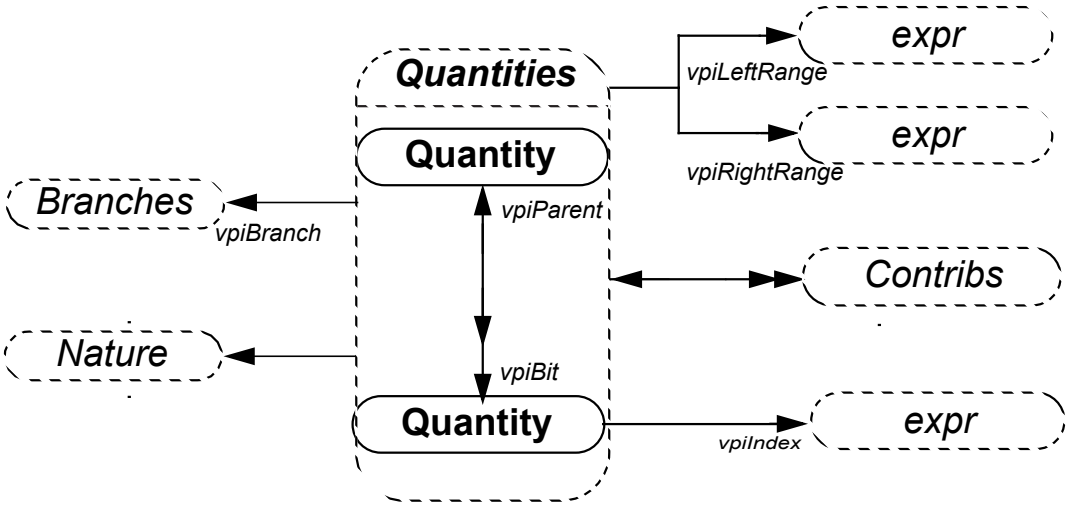


-> location
int: vpiLineNo
str: vpiFile
-> name
str: vpiName
str: vpiFullName
-> scalar
bool: vpiScalar
-> size
int: vpiSize
-> vector
bool: vpiVector

NOTE—Properties *scalar* and *vector* shall indicate if the branch is 1 bit or more than 1 bit.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

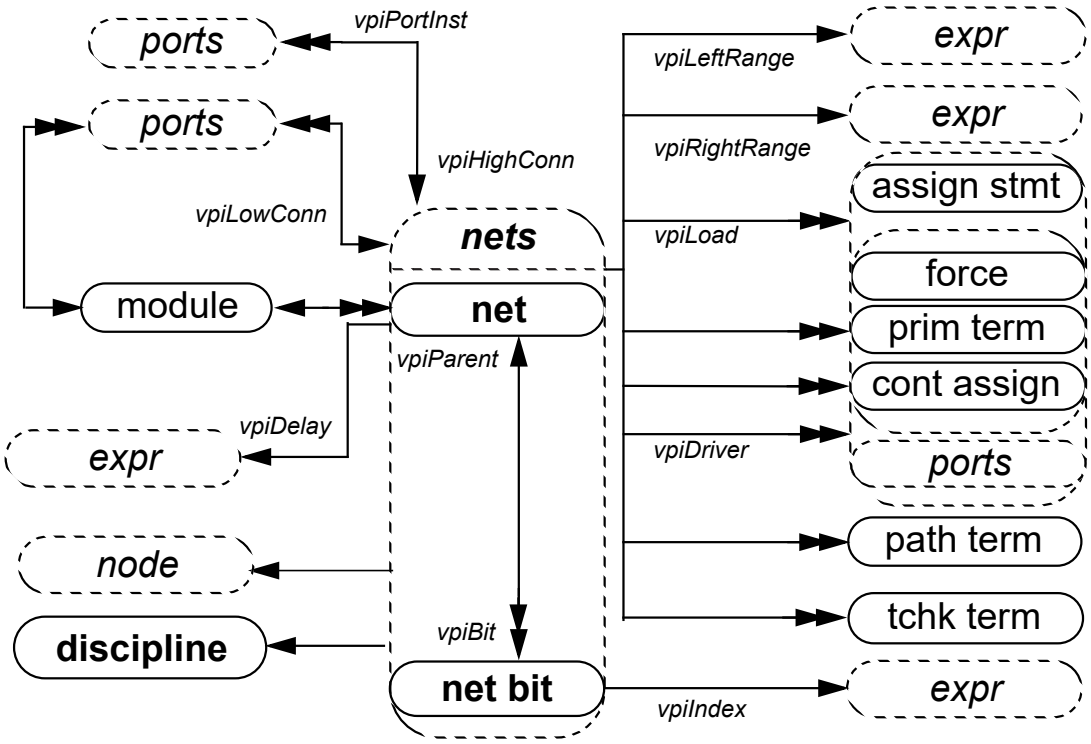
11.6.7 Quantities



-> implicitly declared
bool: vpiImplicitDecl
-> real value
vpi_get_analog_value()
-> imaginary value
vpi_get_analog_value()
-> scalar
bool: vpiScalar
-> size
int: vpiSize
-> vector
bool: vpiVector
-> source
bool: vpiSource
-> equation target
bool: vpiEquationTarget

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.8 Nets



-> delay
vpi_get_delays()
-> expanded
bool: vpiExpanded
-> implicitly declared
bool: vpiImplicitDecl
-> location
int: vpiLineNo
str: vpiFile
-> name
str: vpiName
str: vpiFullName

-> net decl assign
bool: vpiNetDeclAssign
-> net type
int: vpiNetType
-> scalar
bool: vpiScalar
-> scalared declaration
bool: vpiExplicitScalared
-> size
int: vpiSize
-> domain
int vpiDomain

-> strength
int: vpiStrength0
int: vpiStrength1
int: vpiChargeStrength
-> value
vpi_get_value()
vpi_put_value()
-> vector
bool: vpiVector
-> vectored declaration
bool: vpiExplicitVectored

NOTES

1—For vectors, net bits shall be available regardless of vector expansion.

2—Continuous assignments and primitive terminals shall be accessed regardless of hierarchical boundaries.

3—Continuous assignments and primitive terminals shall only be accessed from scalar nets or bit selects.

4—For **vpiPortInst** and **vpiPort**, if the reference handle is a bit or the entire vector, the relationships shall return a handle to either a port bit or the entire port, respectively.

5—For implicit nets, **vpiLineNo** shall return **0**, and **vpiFile** shall return the filename where the implicit net is first referenced.

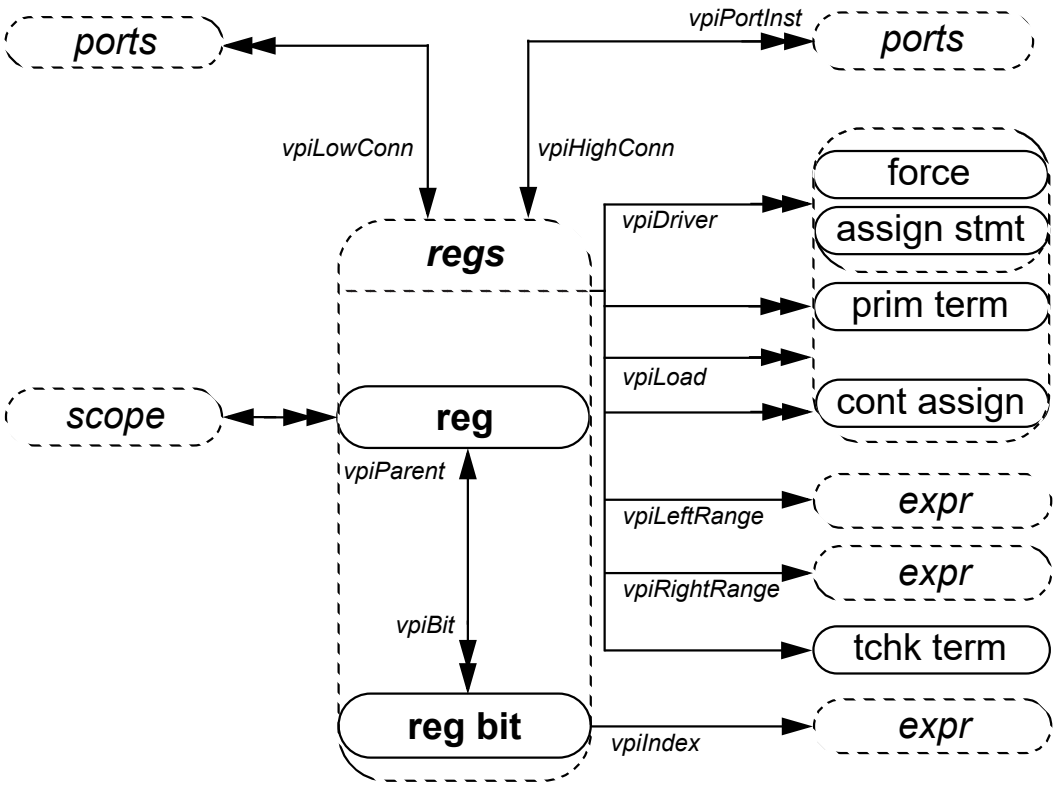
6—Only active forces and assign statements shall be returned for **vpiLoad**.

7—Only active forces shall be returned for **vpiDriver**.

8—**vpiDriver** shall also return ports which are driven by objects other than nets and net bits.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.9 Regs



-> location
int: vpiLineNo
str: vpiFile

-> name
str: vpiName
str: vpiFullName

-> scalar
bool: vpiScalar

-> size
int: vpiSize

-> value
vpi_get_value()
vpi_put_value()

-> vector
bool: vpiVector

NOTES

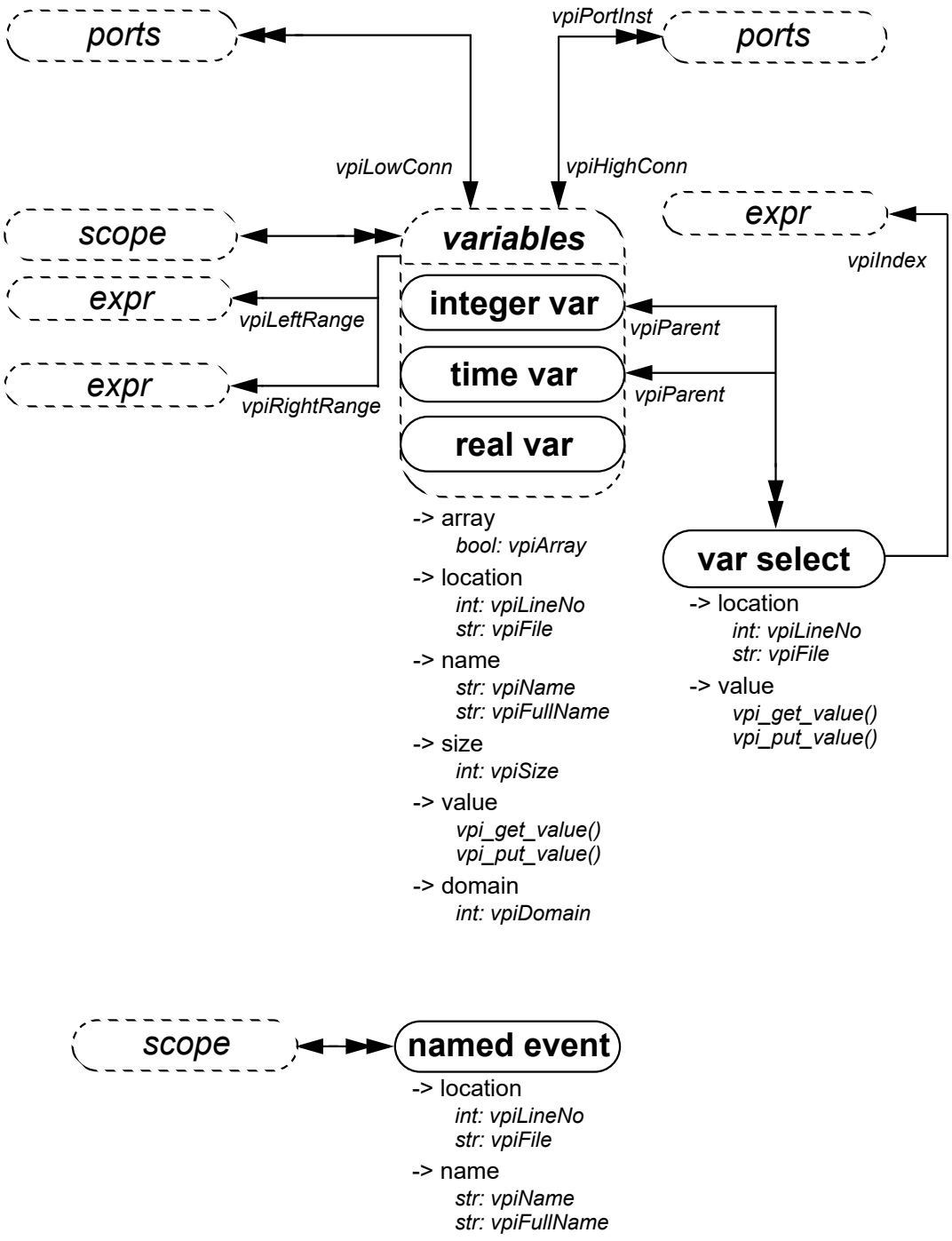
1—Continuous assignments and primitive terminals shall be accessed regardless of hierarchical boundaries.

2—Continuous assignments and primitive terminals shall only be accessed from scalar regs and bit selects.

3—Only active forces and assign statements shall be returned for **vpiLoad** and **vpiDriver**.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

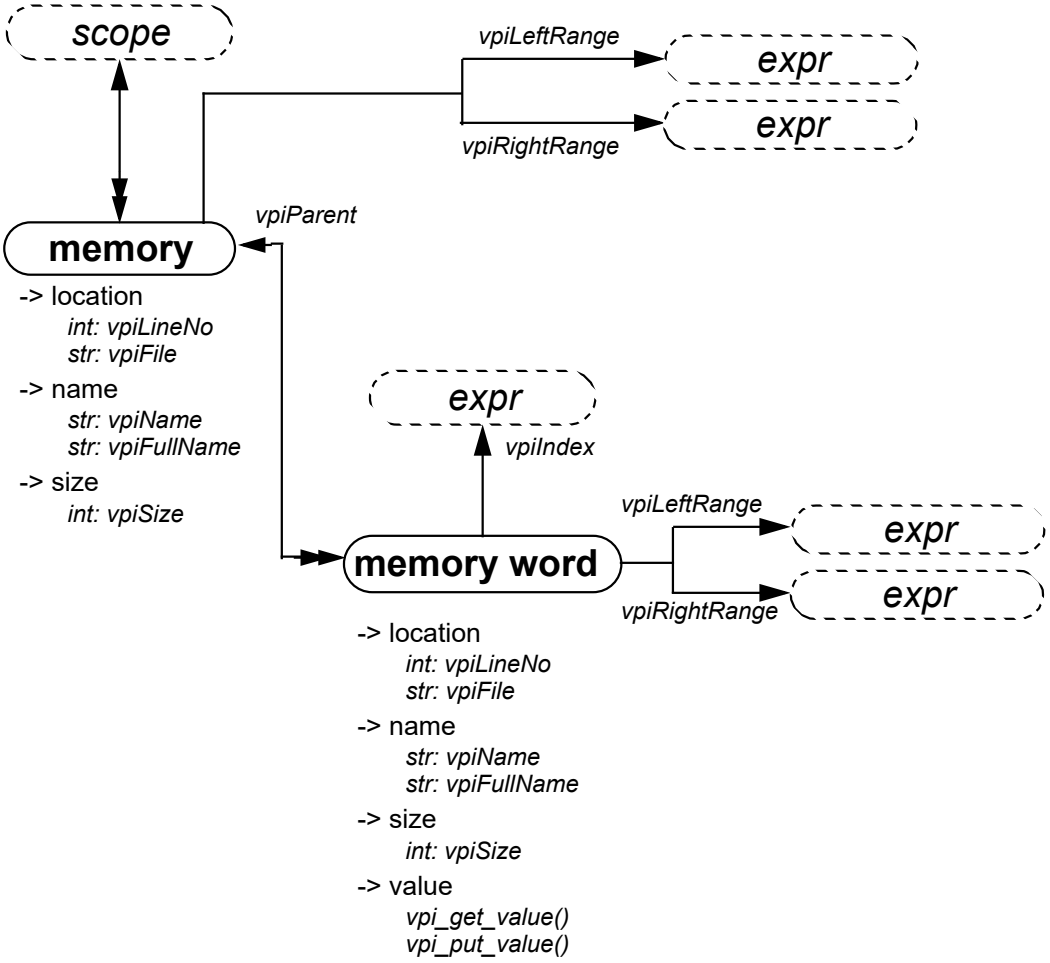
11.6.10 Variables, named event



NOTE—**vpiLeftRange** and **vpiRightRange** shall be invalid for reals, since there can not be arrays of reals.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.11 Memory



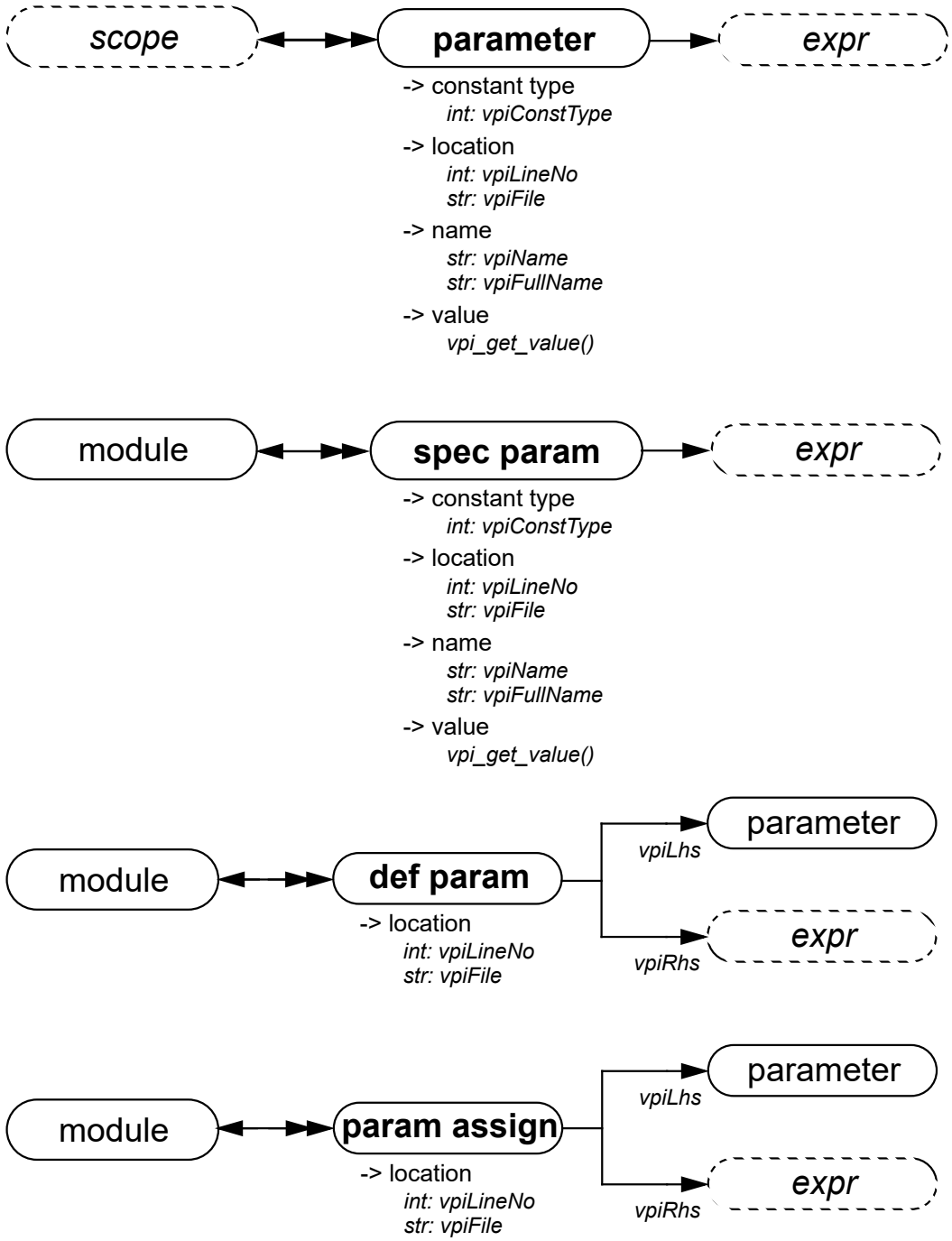
NOTES

1—**vpiSize** for a memory shall return the number of words in the memory.

2—**vpiSize** for a memory word shall return the number of bits in the word.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.12 Parameter, specparam



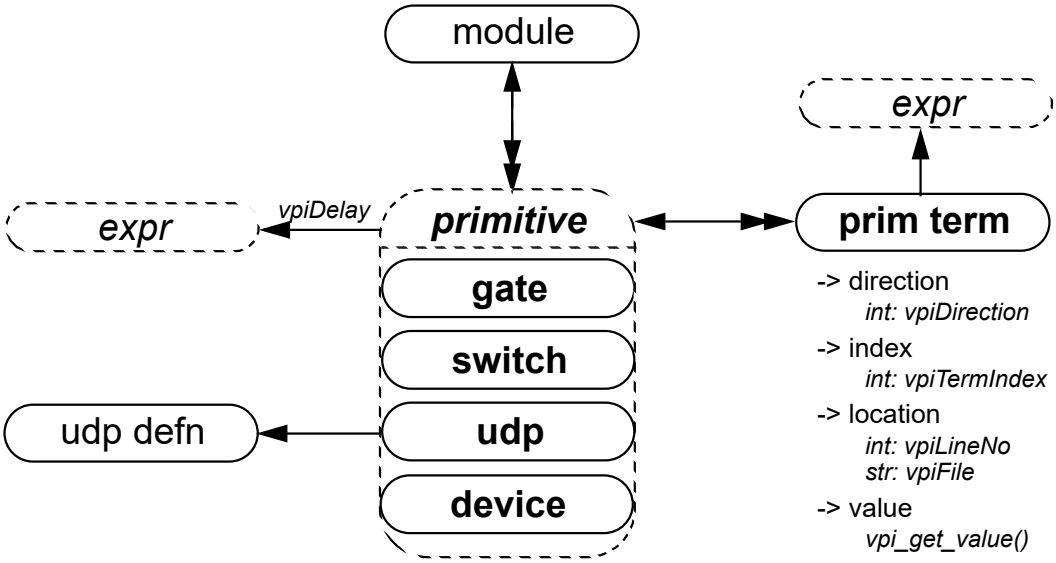
NOTES

1—Obtaining the value from the object **parameter** shall return the final value of the parameter after all module instantiation overrides and defparams have been resolved.

2—**vpiLhs** from a param assign object shall return a handle to the overridden parameter.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.13 Primitive, prim term



-> definition name
str: vpiDefName

-> delay
vpi_get_delays()
vpi_put_delays()

-> location
int: vpiLineNo
str: vpiFile

-> name
str: vpiName
str: vpiFullName

-> primitive type
int: vpiPrimType

-> number of inputs
int: vpiSize

->strength
int: vpiStrength0
int: vpiStrength1

-> value
vpi_get_value()
vpi_put_value()

-> domain
int: vpiDomain

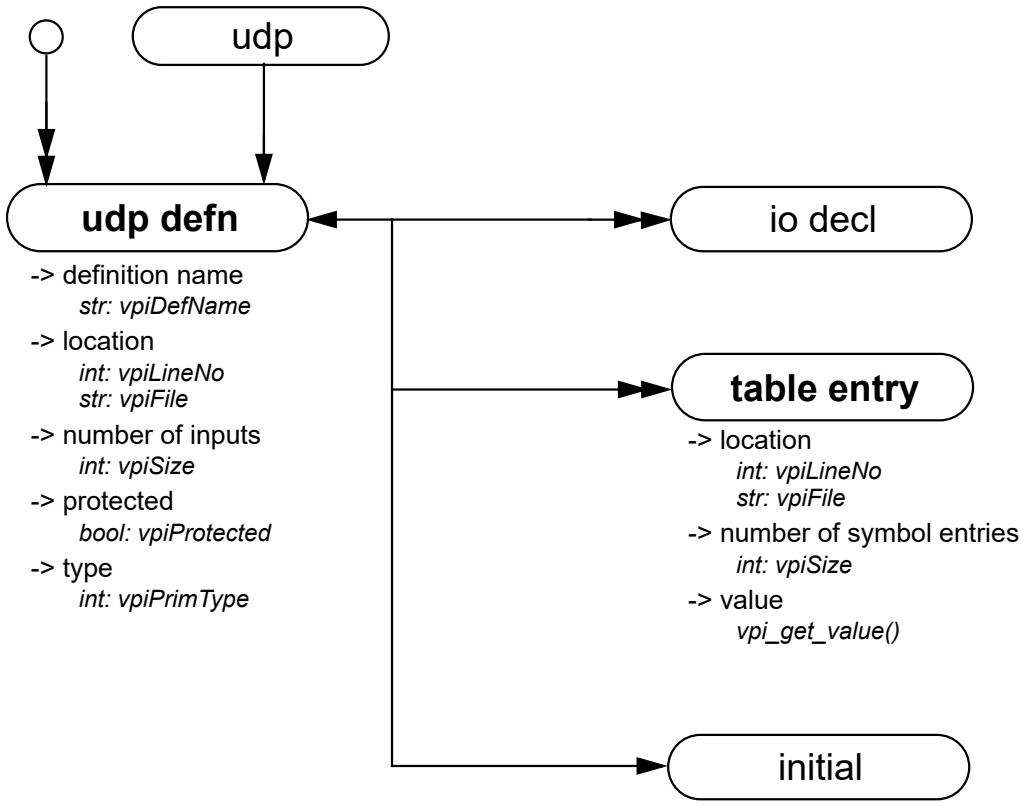
NOTES

1—**vpiSize** shall return the number of inputs.

2—For primitives, **vpi_put_value()** shall only be used with sequential UDP primitives.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

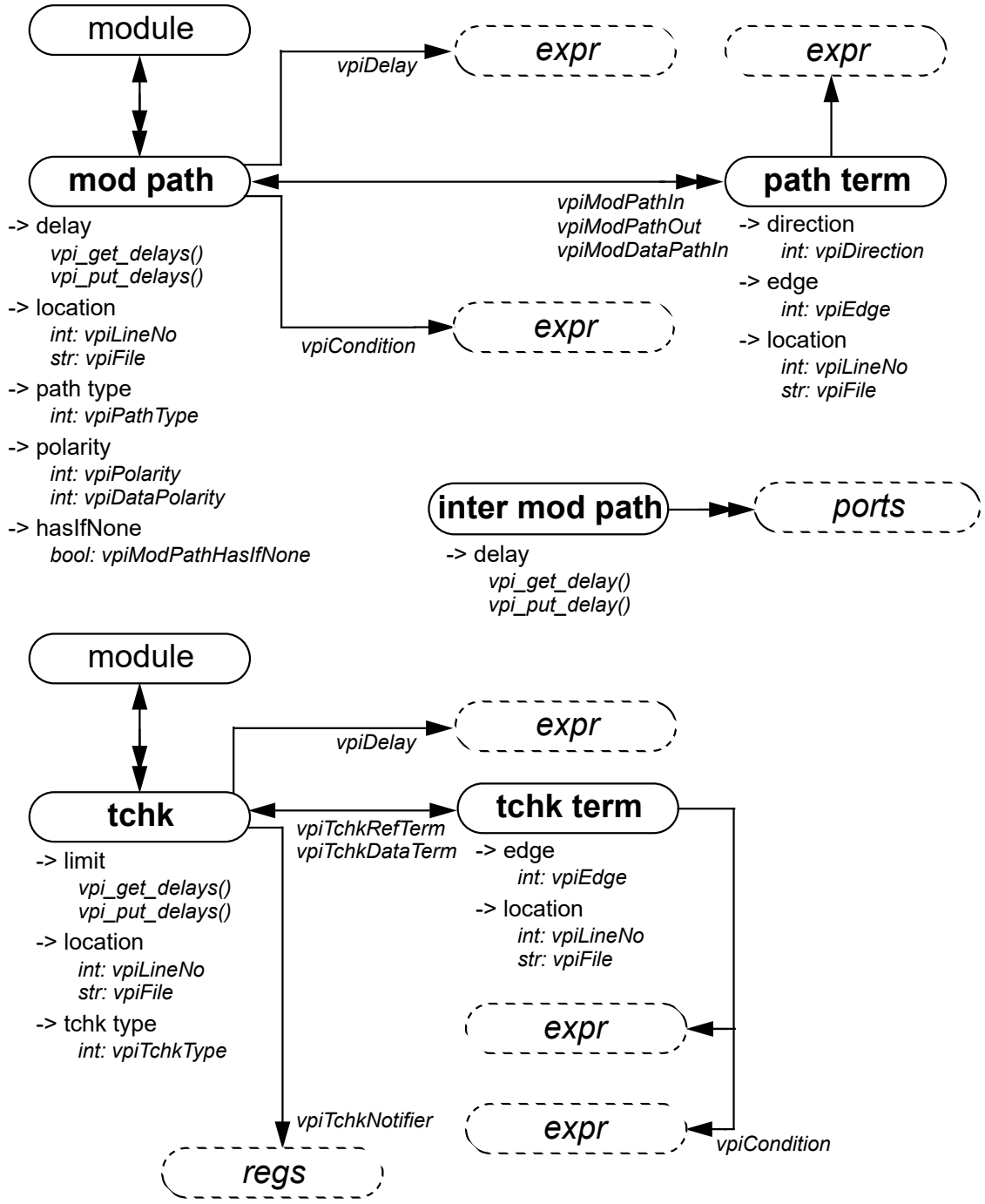
11.6.14 UDP



NOTE—Only string (decompilation) and vector (ASCII values) shall be obtained for table entry objects using **vpi_get_value()**. Refer to the definition of **vpi_get_value()** for additional details.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.15 Module path, timing check, intermodule path



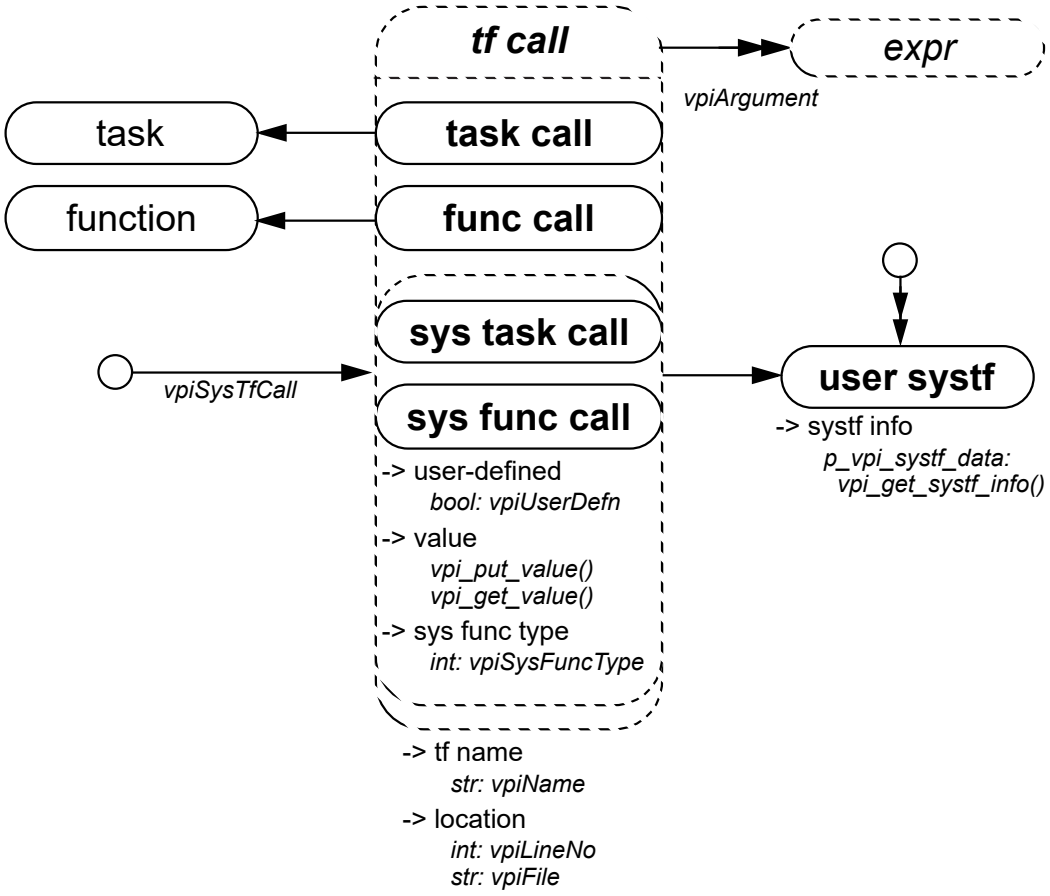
NOTES

1—The **vpiTchkRefTerm** is the first terminal for all tchks except **$setup**, where **vpiTchkDataTerm** is the first terminal and **vpiTchkRefTerm** is the second terminal.

2—To get to an intermodule path, **vpi_handle_multi(vpiInterModPath, port1, port2)** can be used.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.16 Task and function call



NOTES

1—The system task or function which invoked an application shall be accessed with **vpi_handle(vpiSysTfCall, NULL)**

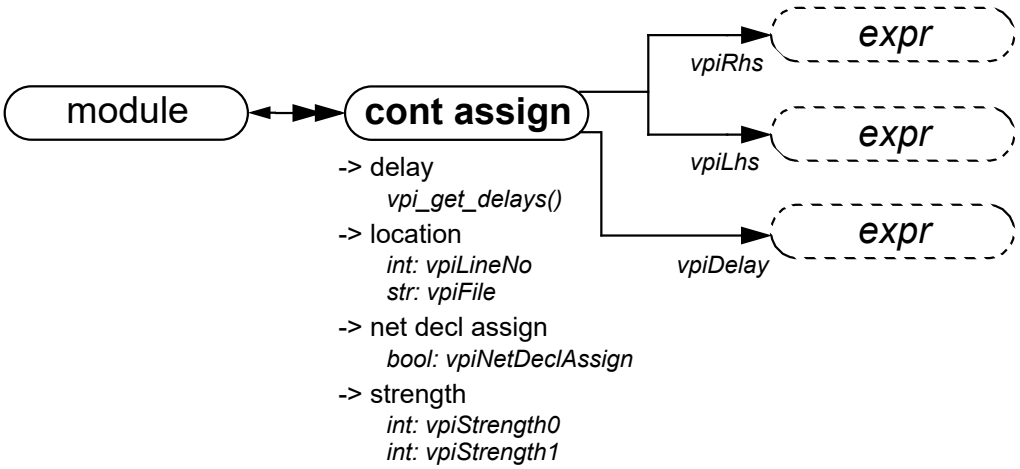
2—**vpi_get_value()** shall return the current value of the system function.

3—If the **vpiUserDefn** property of a system task or function call is true, then the properties of the corresponding systf object shall be obtained via **vpi_get_systf_info()**.

4—All user-defined system tasks or functions shall be retrieved using **vpi_iterate()**, with **vpiUserSystf** as the type argument, and a `NULL` reference argument.

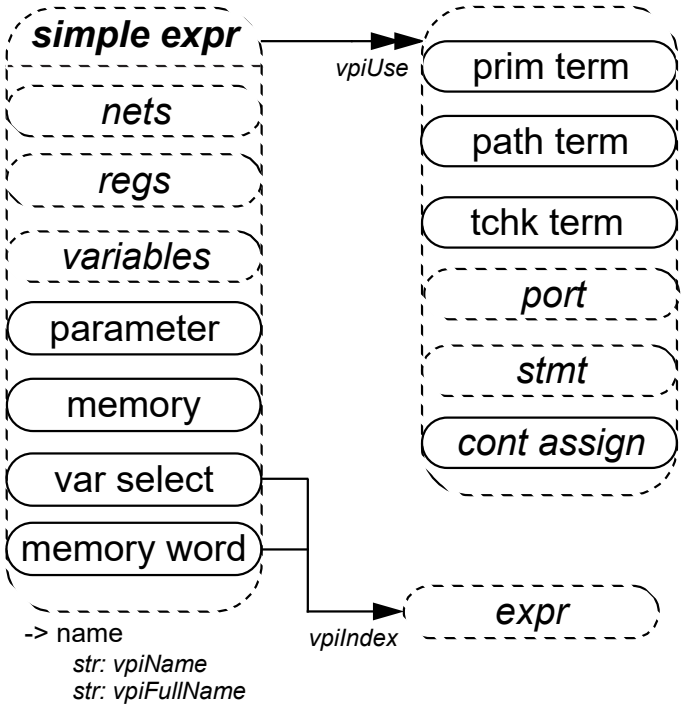
Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.17 Continuous assignment



Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.18 Simple expressions



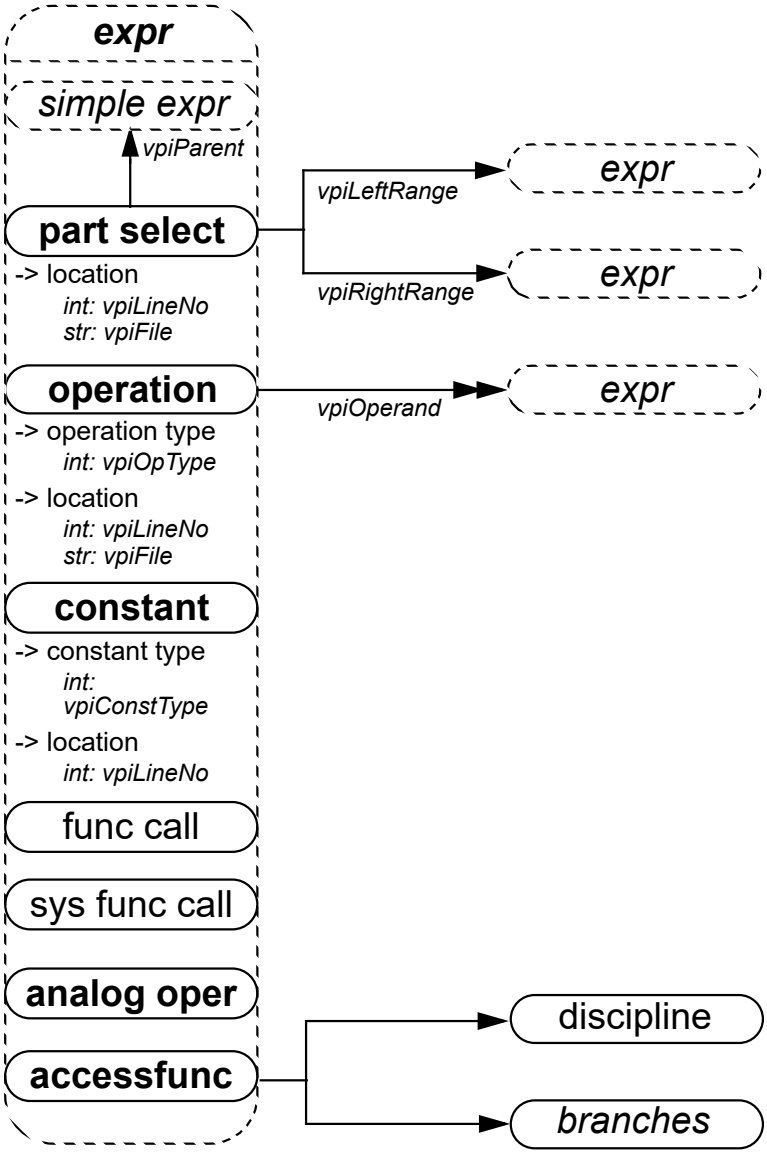
NOTES

1—For vectors, the **vpiUse** relationship shall access any use of the vector or part-selects or bit-selects thereof.

2—For bit-selects, the **vpiUse** relationship shall access any specific use of that bit, any use of the parent vector, and any part-select which contains that bit.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

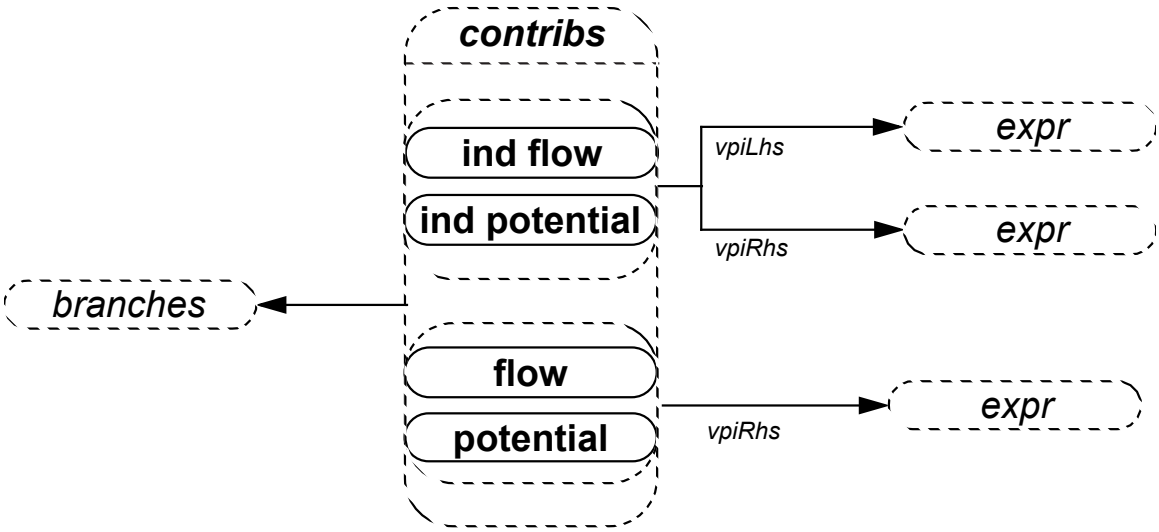
11.6.19 Expressions



NOTE—For an operator whose type is **vpiMultiConcat**, the first operand shall be the multiplier expression.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

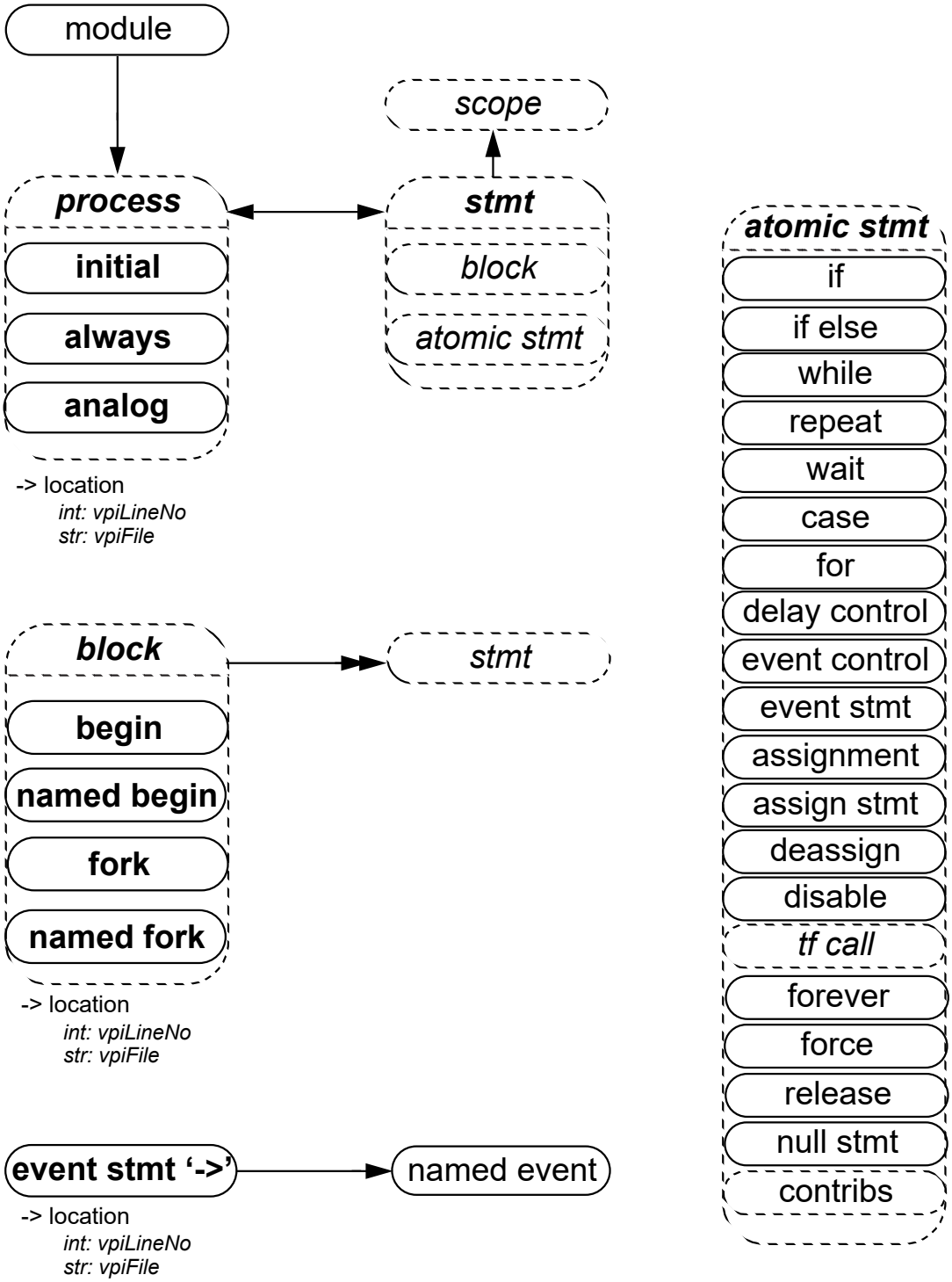
11.6.20 Contribs



-> value
vpi_get_value()
-> direct
bool:vpiDirect
-> flow
bool: vpiFlow

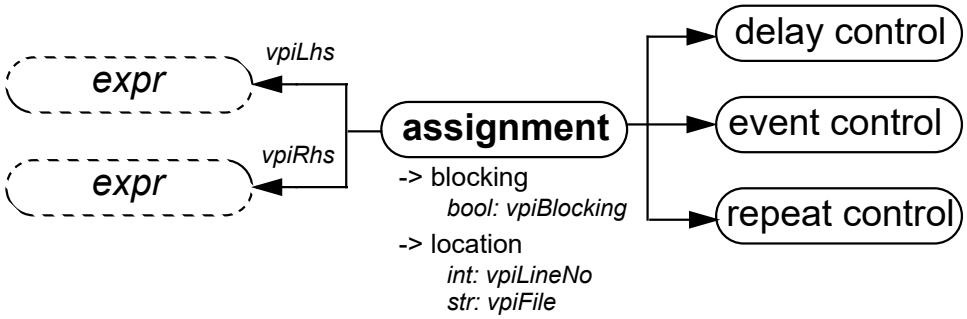
Copyright © 2024 Accellera Systems Initiative. All rights reserved.

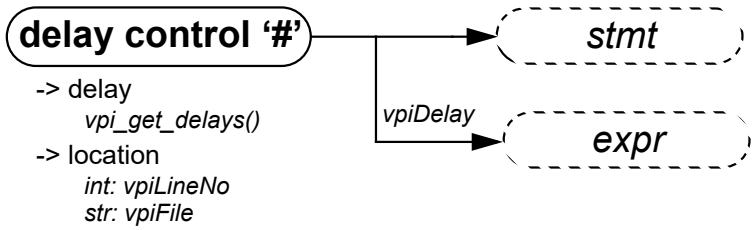
11.6.21 Process, block, statement, event statement

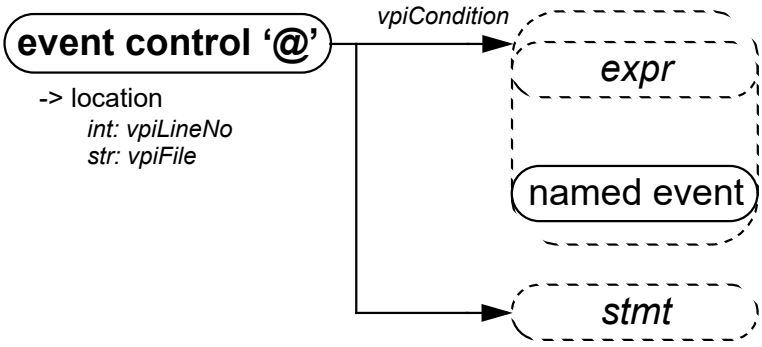


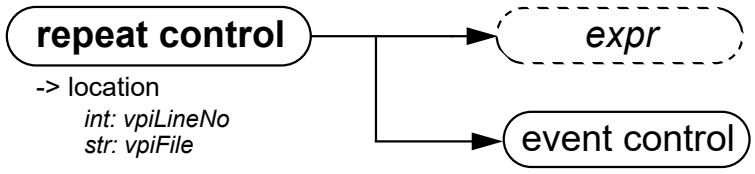
Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.22 Assignment, delay control, event control, repeat control





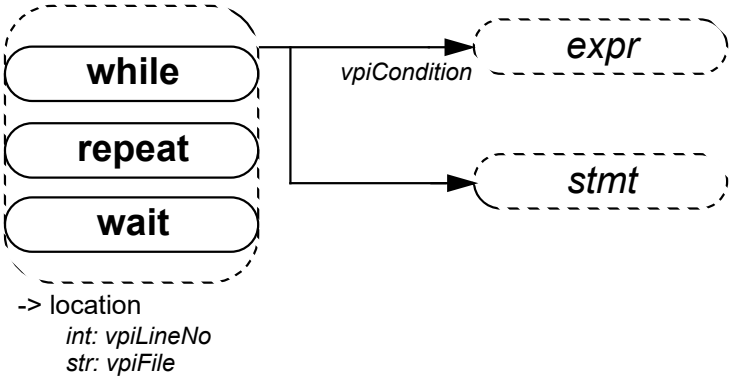


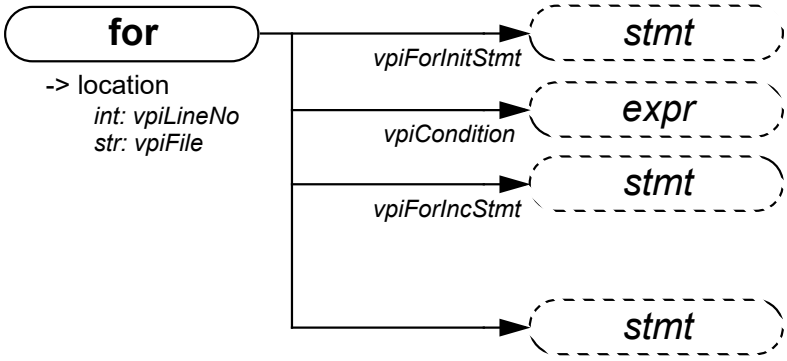


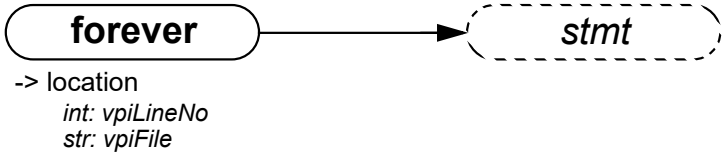
NOTE—For delay control and event control associated with assignment, the statement shall always be `NULL`.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

While, repeat, wait, for, forever

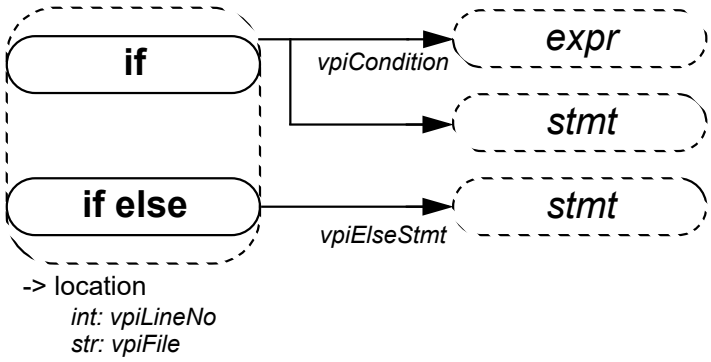


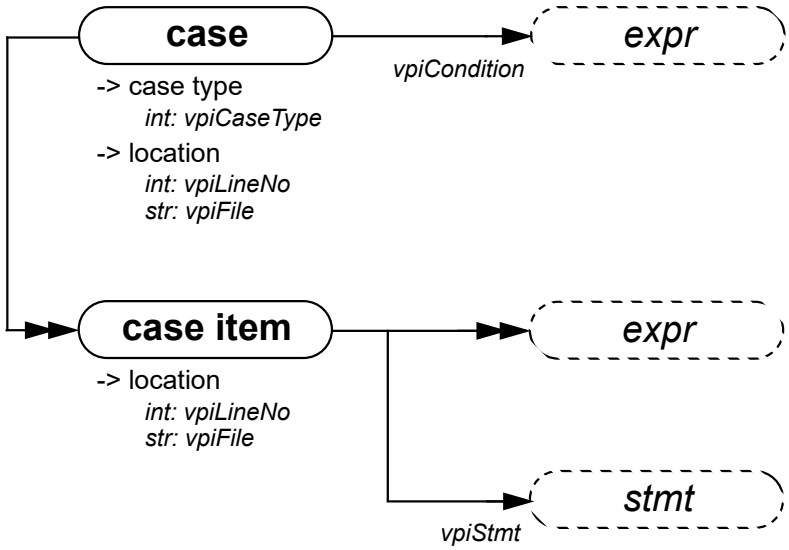




Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.23 If, if-else, case





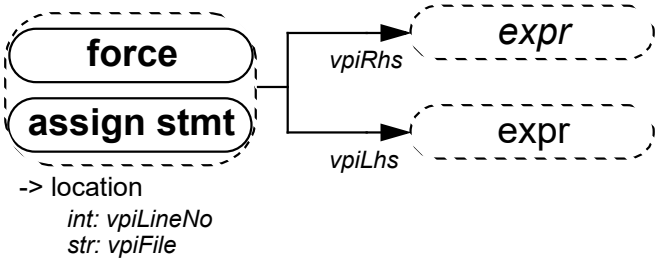
NOTES

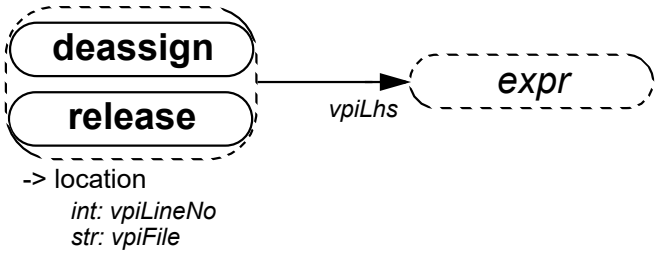
1—The *case item* shall group all case conditions which branch to the same statement.

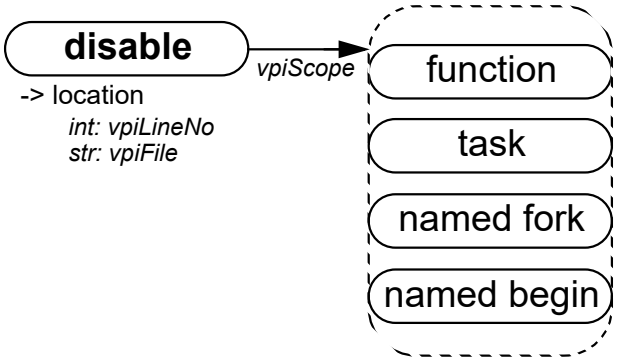
2—**vpi_iterate()** shall return `NULL` for the default case item since there is no expression with the default case.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.24 Assign statement, deassign, force, release, disable

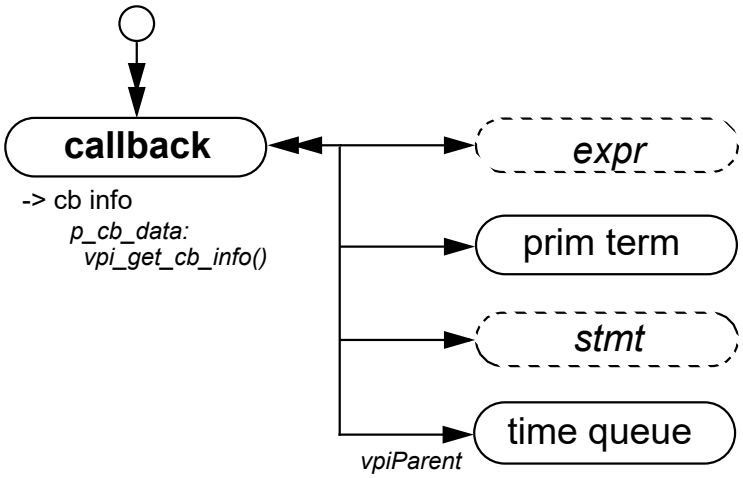


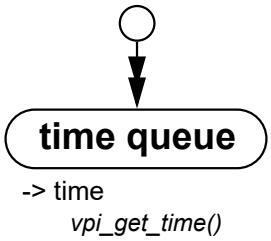




Copyright © 2024 Accellera Systems Initiative. All rights reserved.

11.6.25 Callback, time queue





NOTES

1—To get information about the callback object, the routine **vpi_get_cb_info()** can be used.

2—To get callback objects not related to the above objects, the second argument to **vpi_iterate()** shall be `NULL`.

3—The time queue objects shall be returned in increasing order of simulation time.

4—**vpi_iterate()** shall return `NULL` if there is nothing left in the simulation queue.

5—If any events after read only sync remain in the current queue, then it shall not be returned as part of the iteration.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12. VPI routine definitions

12.1 Overview

This clause describes the Verilog Procedural Interface (VPI) routines, explaining their function, syntax, and usage. The routines are listed in alphabetical order. The following conventions are used in the definitions of the VPI routines.

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 which 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 which 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 might 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 can have default values within the PLI, but they can 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 which 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 the related routines have to be used.

12.2 vpi_chk_error()

vpi_chk_error()	
Synopsis:	Retrieve information about VPI routine errors.
Syntax:	vpi_chk_error(error_info_p)
Type	Description

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

vpi_chk_error()			
Returns:	int	returns the error severity level if the previous VPI routine call resulted in an error and FALSE if no error occurred	
	Type	**Name**	**Description**
Arguments:	p_vpi_error_info	error_info_p	Pointer to a structure containing error information

The VPI routine **`vpi_chk_error()`** shall return `an integer constant representing an error severity level` if the previous call to a VPI routine resulted in an error. The error constants are shown in Table 12-1. If the previous call to a VPI routine did not result in an error, then **`vpi_chk_error()`** shall return `FALSE`. The error status shall be reset by any VPI routine call except **`vpi_chk_error()`**. Calling **`vpi_chk_error()`** shall have no effect on the error status.

Table 12-1—Return error constants for vpi_chk_error()

Error constant	Severity level
vpiNotice	lowest severity
vpiWarning	
vpiError	
vpiSystem	
vpiInternal	highest severity

If an error occurred, the `s_vpi_error_info` structure shall contain information about the error. If the error information is not needed, a `NULL` can be passed to the routine. The `s_vpi_error_info` structure used by **`vpi_chk_error()`** is defined in `vpi_user.h` and is listed in Figure 12-1.

```
typedef struct t_vpi_error_info {
   int state;  /* vpi[Compile,PLI,Run] */
   int level;  /* vpi[Notice, Warning, Error, System, Internal] */
   char *message;
   char *product;
   char *code;
   char *file;
   int line;
} s_vpi_error_info, *p_vpi_error_info;
```

Figure 12-1: The s_vpi_error_info structure definition

12.3 vpi_compare_objects()

vpi_compare_objects()	
Synopsis:	Compare two handles to determine if they reference the same object.
Syntax:	vpi_compare_objects(obj1, obj2)
Type	Description

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

vpi_compare_objects()			
Returns:	bool	`true` if the two handles refer to the same object. Otherwise, `false`	
	Type	**Name**	**Description**
Arguments:	vpiHandle	obj1	Handle to an object
	vpiHandle	obj2	Handle to an object

The VPI routine **`vpi_compare_objects()`** shall return `TRUE` if the two handles refer to the same object. Otherwise, `FALSE` shall be returned. Handle equivalence can not be determined with a C '==' comparison.

12.4 vpi_free_object()

vpi_free_object()			
Synopsis:	Free memory allocated by VPI routines.		
Syntax:	vpi_free_object(obj)		
	Type	Description	
Returns:	bool	`true` on success and `false` on failure	
	Type	**Name**	**Description**
Arguments:	vpiHandle	obj	Handle of an object

The VPI routine **`vpi_free_object()`** shall free memory allocated for objects. It shall generally be used to free memory created for iterator objects. The iterator object shall automatically be freed when **`vpi_scan()`** returns `NULL` either because it has completed an object traversal or encountered an error condition. If neither of these conditions occur (which can happen if the code breaks out of an iteration loop before it has scanned every object), **`vpi_free_object()`** needs to be called to free any memory allocated for the iterator. This routine can also optionally be used for implementations which have to allocate memory for objects. The routine shall return `TRUE` on success and `FALSE` on failure.

12.5 vpi_get()

vpi_get()			
Synopsis:	Get the value of an integer or Boolean property of an object.		
Syntax:	vpi_get(prop, obj)		
	Type	Description	
Returns:	int	Value of an integer or Boolean property	
	Type	**Name**	**Description**
Arguments:	int	prop	An integer constant representing the property of an object for which to obtain a value

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

vpi_get()			
	vpiHandle	obj	Handle to an object
Related routines:	Use vpi_get_str() to get string properties		

The VPI routine **`vpi_get()`** shall return the value of object properties, for properties of type *int* and *bool* (*bool* shall be defined to *int*). Object properties of type *bool* shall return `1` for `TRUE` and `0` for `FALSE`. For object properties of type *int* such as **`vpiSize`**, any integer shall be returned. For object properties of type *int* which return a defined value, refer to Annex C of the IEEE Std 1364 Verilog specification for the value that shall be returned. Note for object property **`vpiTimeUnit`** or **`vpiTimePrecision`**, if the object is `NULL`, then the simulation time unit shall be returned. Should an error occur, **`vpi_get()`** shall return **`vpi-Undefined`**.

12.6 vpi_get_cb_info()

vpi_get_cb_info()			
Synopsis:	Retrieve information about a simulation-related callback.		
Syntax:	vpi_get_cb_info(obj, cb_data_p)		
	Type	Description	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	vpiHandle	obj	Handle to a simulation-related callback
	p_cb_data	cb_data_p	Pointer to a structure containing callback information
Related routines:	Use vpi_get_systf_info() to retrieve information about a system task/function callback		

The VPI routine **`vpi_get_cb_info()`** shall return information about a simulation-related callback in an `s_cb_data` structure. The memory for this structure shall be allocated by the user.

The `s_cb_data` structure used by **`vpi_get_cb_info()`** is defined in `vpi_user.h` and is listed in Figure 12-2.

```
typedef struct t_cb_data {
   int reason;
   int (*cb_rtn)();
   vpiHandle obj;
   p_vpi_time time;  /* structure with simulation time info */
   p_vpi_value value;/* structure with simulation value info */
   char *user_data;  /* user data to be passed to callback function */
} s_cb_data, *p_cb_data;
```

Figure 12-2: The s_cb_data structure definition

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.7 vpi_get_analog_delta()

vpi_get_analog_delta()			
Synopsis:	Get the time elapsed since the previous solution.		
Syntax:	vpi_get_analog_delta()		
	Type	true on success and false on failureDescription	
Returns:	double	time elapsed between the solution being calculated and the last converged solution	
	Type	**Name**	**Description**
Arguments:	NONE		this function accepts no arguments

The VPI routine **`vpi_get_analog_delta()`** shall be used determine the size of the analog time step being attempted. It returns the elapsed time between the latest converged and accepted solution and the solution being calculated. The function shall return zero (`0`) during DC or the time zero transient solution.

12.8 vpi_get_analog_freq()

vpi_get_analog_freq()			
Synopsis:	Get the frequency for the current small-signal analysis.		
Syntax:	vpi_get_analog_freq()		
	Type	true on success and false on failureDescription	
Returns:	double	time elapsed between the solution being calculated and the last converged solution	
	Type	**Name**	**Description**
Arguments:	NONE		this function accepts no arguments

The VPI routine **`vpi_get_analog_freq()`** shall be used determine the current frequency used in the small-signal analysis. The function shall return zero (`0`) during DC or transient analysis.

12.9 vpi_get_analog_time()

vpi_get_analog_time()			
Synopsis:	Get the time of the current solution.		
Syntax:	vpi_get_analog_time()		
	Type	true on success and false on failureDescription	
Returns:	double	time associated with the current solution	
	Type	**Name**	**Description**
Arguments:	NONE		this function accepts no arguments

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The VPI routine **`vpi_get_analog_time()`** shall be used determine the time of the solution attempted or of the latest converged and accepted solution otherwise. The function shall return zero (`0`) during DC or the time zero transient solution.

12.10 vpi_get_analog_value()

vpi_get_analog_value()			
Synopsis:	Retrieve the simulation value of an analog quantity object.		
Syntax:	vpi_get_analog_value(obj, value_p)		
Returns:	**Type**	Description	
	void		
Arguments:	**Type**	**Name**	**Description**
	vpiHandle	obj	Handle to an analog quantity object
	p_vpi_value	value_p	Pointer to a structure containing value information
Related routines:	Use vpi_get_value() to get simulation values of digital objects. Use vpi_put_value() to set the value of an object		

The VPI routine **`vpi_get_analog_value()`** shall retrieve the simulation value of VPI analog **`vpiFlow`** or **`vpiPotential`** (node or branch) quantity objects. The value shall be placed in an `s_vpi_analog_value` structure, which has been allocated by the user. The format of the value shall be set by the *format* field of the structure.

The buffer this routine uses for string values shall be different from the buffer which **`vpi_get_str()`** shall use. The string buffer used by **`vpi_get_analog_value()`** is overwritten with each call. If the value is needed, it needs to be saved by the application.

The `s_vpi_analog_value` structure used by **`vpi_get_analog_value()`** is defined in `vpi_user.h` and listed in Figure 12-3.

```
typedef struct t_vpi_analog_value {
   int format; /* vpiRealVal,vpiExpStrVal,vpiDecStrVal,vpiStringVal
*/
   union {
      char *str;
      double real;
      char *misc;
   } real;
   union {
      char *str;
      double real;
      char *misc;
   } imaginary;
} s_vpi_analog_value, *p_vpi_analog_value;
```

Figure 12-3: The s_vpi_analog_value structure definition

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The memory for the union members *str* and *misc* of the value for real and imaginary unions in the `s_vpi_analog_value` structure shall be provided by the routine **`vpi_get_analog_value()`**. This memory shall only be valid until the next call to **`vpi_get_analog_value()`**.

Table 12-2—Return value field of the s_vpi_analog_value structure union

Format	Union members	Return description
vpiDecStrVal	str	Real and imaginary values of object are returned as strings of decimal char(s) [0–9]
vpExpStrVal	str	Real and imaginary values of object are returned as strings formatted like printf %e.
vpiRealVal	real	Real and imaginary values of the object are returned as doubles.
vpiStringVal	str	Real and imaginary parts are returned as strings formatted like printf %g. The call shall reset the format field to **vpiExpStrVal** or **vpiDecStrVal** to the selected format.

NOTE—The user shall provide the memory for these members when calling **`vpi_put_value()`**.

12.11 vpi_get_delays()

vpi_get_delays()			
Synopsis:	Retrieve the delays or pulse limits of an object.		
Syntax:	vpi_get_delays(obj, delay_p)		
Returns:	**Type**	Description	
	void		
Arguments:	**Type**	**Name**	**Description**
	vpiHandle	obj	Handle to an object
	p_vpi_delay	delay_p	Pointer to a structure containing delay information
Related routines:	Use vpi_put_delays() to set the delays or timing limits of an object		

The VPI routine **`vpi_get_delays()`** shall retrieve the delays or pulse limits of an object and place them in an `s_vpi_delay` structure which has been allocated by the user. The format of the delay information shall be controlled by the *time_type* flag in the `s_vpi_delay` structure. This routine shall ignore the value of the *type* flag in the `s_vpi_time` structure.

The `s_vpi_delay` and `s_vpi_time` structures used by both **`vpi_get_delays()`** and **`vpi_put_delays()`** are defined in `vpi_user.h` and are listed in Figure 12-4 and Figure 12-5.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
typedef struct t_vpi_delay {
   struct t_vpi_time *da;  /* ptr to user allocated array of delay
                           values */
   int no_of_delays;       /* number of delays */
   int time_type;          /* [vpiScaledRealTime, vpiSimTime] */
   bool mtm_flag;          /* true for mtm */
   bool append_flag;       /* true for append, false for replace */
   bool pulsere_flag;      /* true for pulsere values */
} s_vpi_delay, *p_vpi_delay;
```

Figure 12-4: The s_vpi_delay structure definition

```
typedef struct t_vpi_time
{
   int type;               /* [vpiScaledRealTime, vpiSimTime] */
   unsigned int high, low; /* for vpiSimTime */
   double real;            /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 12-5: The s_vpi_time structure definition

The *da* field of the `s_vpi_delay` structure shall be a user-allocated array of `s_vpi_time` structures. This array shall store delay values returned by **`vpi_get_delays()`**. The number of elements in this array shall be determined by

- The number of delays to be retrieved
- The **`mtm_flag`** setting
- The **`pulsere_flag`** setting

The number of delays to be retrieved shall be set in the *no_of_delays* field of the `s_vpi_delay` structure. Legal values for the number of delays shall be determined by the type of object.

- For primitive objects, the *no_of_delays* value shall be `2` or `3`.
- For path delay objects, the *no_of_delays* value shall be `1`, `2`, `3`, `6`, or `12`.
- For timing check objects, the *no_of_delays* value shall match the number of limits existing in the timing check.
- For inter-module path objects, the *no_of_delays* value shall be `2` or `3`.

The user-allocated `s_vpi_delay` array shall contain delays in the same order in which they occur in the Verilog-AMS HDL description. The number of elements for each delay shall be determined by the flags **`mtm_flag`** and **`pulsere_flag`**, as shown in Table 12-3.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 12-3—Size of the s_vpi_delay->da array

Flag values	Number of `s_vpi_time` array elements required for *s_vpi_delay->da*	Order in which delay elements shall be filled
mtm_flag = `false` **pulsere_flag** = `false`	*no_of_delays*	1st delay: da[0] -> 1st delay 2nd delay: da[1] -> 2nd delay ...
mtm_flag = `true` **pulsere_flag** = `false`	3 * *no_of_delays*	1st delay: da[0] -> min delay da[1] -> typ delay da[2] -> max delay 2nd delay: ...
mtm_flag = `false` **pulsere_flag** = `true`	3 * *no_of_delays*	1st delay: da[0] -> delay da[1] -> reject limit da[2] -> error limit 2nd delay element: ...
mtm_flag = `true` **pulsere_flag** = `true`	9 * *no_of_delays*	1st delay: da[0] -> min delay da[1] -> typ delay da[2] -> max delay da[3] -> min reject da[4] -> typ reject da[5] -> max reject da[6] -> min error da[7] -> typ error da[8] -> max error 2nd delay: ...

The delay structure has to be allocated before passing a pointer to **`vpi_get_delays()`**.

In the following example, a static structure, **`prim_da`**, is allocated for use by each call to the **`vpi_get_delays()`** function.

```
display_prim_delays(prim)
vpiHandle prim;t2

{
   static s_vpi_time prim_da[3];
   static s_vpi_delay delay_s = {NULL, 3, vpiScaledRealTime};
   static p_vpi_delay delay_p = &delay_s;

   delay_s.da = &prim_da;
   vpi_get_delays(prim, delay_p);
   vpi_printf("Delays for primitive %s: %6.2f %6.2f %6.2f\n",
   vpi_get_str(vpiFullName, prim)
   delay_p->da[0].real, delay_p->da[1].real, delay_p->da[2].real);
}
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.12 vpi_get_str()

vpi_get_str()			
Synopsis:	Get the value of a string property of an object.		
Syntax:	vpi_get_str(prop, obj)		
Returns:	**Type**	Description	
	char *	Pointer to a character string containing the property value	
Arguments:	**Type**	**Name**	**Description**
	int	prop	An integer constant representing the property of an object for which to obtain a value
	vpiHandle	obj	Handle to an object
Related routines:	Use vpi_get() to get integer and Boolean properties		

The VPI routine **`vpi_get_str()`** shall return string property values. The string shall be placed in a temporary buffer which shall be used by every call to this routine. If the string is to be used after a subsequent call, the string needs to be copied to another location. A different string buffer shall be used for string values returned through the `s_vpi_value` structure.

The following example illustrates the usage of **`vpi_get_str()`**.

```
char *str;
vpiHandle mod = vpi_handle_by_name("top.mod1",NULL);
vpi_printf ("Module top.mod1 is an instance of %s\n",
         vpi_get_str(vpiDefName, mod));
```

12.13 vpi_get_analog_systf_info()

vpi_get_analog_systf_info()			
Synopsis:	Retrieve information about a user-defined analog system task/function-related callback.		
Syntax:	vpi_get_analog_systf_info(obj, systf_data_p)		
Returns:	**Type**	Description	
	void		
Arguments:	**Type**	**Name**	**Description**
	vpiHandle	obj	Handle to a system task/function-related callback
	p_vpi_analog_systf_data	systf_data_p	Pointer to a structure containing callback information
Related routines:	Use vpi_get_cb_info() to retrieve information about a simulation-related callback		

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The VPI routine **vpi_get_analog_systf_info()** shall return information about a user-defined analog system task or function callback in an s_vpi_analog_systf_data structure. The memory for this structure shall be allocated by the user.

The s_vpi_systf_data structure used by **vpi_get_analog_systf_info()** is defined in vpi_user.h and is listed in Figure 12-6.

```
typedef struct t_vpi_analog_systf_data {
   int type;         /* vpiSys[Task,Function] */
   int sysfunctype;  /* vpi[IntFunc,RealFunc,TimeFunc,SizedFunc] */
   char *tfname;     /* first character shall be "$" */
   int (*calltf)();
   int (*compiletf)();
   int (*sizetf)();  /* for vpiSizedFunc system functions only */
   p_vpi_stf_partials (*derivtf)(); /* for partial derivatives */
   char *user_data;
} s_vpi_analog_systf_data, *p_vpi_analog_systf_data;
```

Figure 12-6: The s_vpi_systf_data structure definition

12.14 vpi_get_systf_info()

vpi_get_systf_info()			
Synopsis:	Retrieve information about a user-defined system task/function-related callback.		
Syntax:	vpi_get_systf_info(obj, systf_data_p)		
Returns:	**Type**	Description	
	void		
Arguments:	**Type**	**Name**	**Description**
	vpiHandle	obj	Handle to a system task/function-related callback
	p_vpi_systf_data	systf_data_p	Pointer to a structure containing callback information
Related routines:	Use vpi_get_cb_info() to retrieve information about a simulation-related callback		

The VPI routine **vpi_get_systf_info()** shall return information about a user-defined system task or function callback in an s_vpi_systf_data structure. The memory for this structure shall be allocated by the user.

The s_vpi_systf_data structure used by **vpi_get_systf_info()** is defined in vpi_user.h and is listed in Figure 12-7.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
typedef struct t_vpi_systf_data {
   int type;          /* vpiSys[Task,Function] */
   int sysfunctype;   /* vpi[IntFunc,RealFunc,TimeFunc,SizedFunc] */
   char *tfname;      /* first character shall be "$" */
   int (*calltf)();
   int (*compiletf)();
   int (*sizetf)();   /* for vpiSizedFunc system functions only */
   char *user_data;
} s_vpi_systf_data, *p_vpi_systf_data;
```

Figure 12-7: The s_vpi_systf_data structure definition

12.15 vpi_get_time()

vpi_get_time()			
Synopsis:	Retrieve the current simulation.		
Syntax:	vpi_get_time(obj, time_p)		
Returns:	**Type**	Description	
	void		
Arguments:	**Type**	**Name**	**Description**
	vpiHandle	obj	Handle to an object
	p_vpi_time	time_p	Pointer to a structure containing time information
Related routines:			

The VPI routine **`vpi_get_time()`** shall retrieve the current simulation time, using the time scale of the object. If *obj* is `NULL`, the simulation time is retrieved using the simulation time unit. The *time_p->type* field shall be set to indicate if scaled real, analog, or simulation time is desired. The memory for the *time_p* structure shall be allocated by the user.

The `s_vpi_time` structure used by **`vpi_get_time()`** is defined in `vpi_user.h` and is listed in Figure 12-8 (this is the same time structure as used by **`vpi_put_value()`**).

```
typedef struct t_vpi_time {
   int type;          /* for vpiScaledRealTime, vpiSimTime,
vpiAnalogTime */
   unsigned int high, low;   /* for vpiSimTime */
   double real;       /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 12-8: The s_vpi_time structure definition

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.16 vpi_get_value()

vpi_get_value()			
Synopsis:	Retrieve the simulation value of an object.		
Syntax:	vpi_get_value(obj, value_p)		
Returns:	**Type**	Description	
	void		
Arguments:	**Type**	**Name**	**Description**
	vpiHandle	obj	Handle to an expression
	p_vpi_value	value_p	Pointer to a structure containing value information
Related routines:	Use vpi_get_analog_value() for simulation value of quantity objects. Use vpi_put_value() to set the value of an object		

The VPI routine **`vpi_get_value()`** shall retrieve the simulation value of VPI objects (use **`vpi_get_analog_value()`** for the simulation value of VPI analog quantity objects). The value shall be placed in an `s_vpi_value` structure, which has been allocated by the user. The format of the value shall be set by the *format* field of the structure.

When the *format* field is **`vpiObjTypeVal`**, the routine shall fill in the value and change the *format* field based on the object type, as follows:

- For an integer, **`vpiIntVal`**
- For a real, **`vpiRealVal`**
- For a scalar, either **`vpiScalar`** or **`vpiStrength`**
- For a time variable, **`vpiTimeVal`** with **`vpiSimTime`**
- For a vector, **`vpiVectorVal`**

The buffer this routine uses for string values shall be different from the buffer which **`vpi_get_str()`** shall use. The string buffer used by `vpi_get_value()` is overwritten with each call. If the value is needed, it needs to be saved by the application.

The `s_vpi_value`, `s_vpi_vecval` and `s_vpi_strengthval` structures used by **`vpi_get_value()`** are defined in `vpi_user.h` and are listed in Figure 12-9, Figure 12-10, and Figure 12-11.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
typedef struct t_vpi_value {
   int format; /* vpi[[Bin,Oct,Dec,Hex]Str,Scalar,Int,Real,String,
                  Time,Vector,Strength,ObjType]Val*/
   union {
      char *str;
      int scalar; /* vpi[0,1,X,Z] */
      int integer;
      double real;
      struct t_vpi_time *time;
      struct t_vpi_vecval *vector;
      struct t_vpi_strengthval *strength;
      char *misc;
   } value;
} s_vpi_value, *p_vpi_value;
```

Figure 12-9: The s_vpi_value structure definition

```
typedef struct t_vpi_vecval {
   int aval, bval; /* bit encoding: ab: 00=0, 10=1, 11=X, 01=Z */
} s_vpi_vecval, *p_vpi_vecval;
```

Figure 12-10: The s_vpi_vecval structure definition

```
typedef struct t_vpi_strengthval {
   int logic;     /* vpi[0,1,X,Z] */
   int s0, s1;    /* refer to strength coding in the LRM */
} s_vpi_strengthval, *p_vpi_strengthval;
```

Figure 12-11: The s_vpi_strengthval structure definition

For vectors, the *p_vpi_vecval* field shall point to an array of `s_vpi_vecval` structures. The size of this array shall be determined by the size of the vector, where *array_size = ((vector_size-1)/32 + 1)*. The lsb of the vector shall be represented by the lsb of the `0`-indexed element of `s_vpi_vecval` array. The 33rd bit of the vector shall be represented by the lsb of the `1`-indexed element of the array, and so on. The memory for the union members *str*, *time*, *vector*, *strength*, and *misc* of the value union in the `s_vpi_value` structure shall be provided by the routine **`vpi_get_value()`**. This memory shall only be valid until the next call to **`vpi_get_value()`**. (The user shall provide the memory for these members when calling **`vpi_put_value()`**). When a value change callback occurs for a value type of **`vpiVectorVal`**, the system shall create the associated memory (an array of `s_vpi_vecval` structures) and free the memory upon the return of the callback.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table 12-4—Return value field of the s_vpi_value structure union

Format	Union member	Return description
vpiBinStrVal	str	String of binary char(s) [**1, 0, x, z**]
vpiOctStrVal	str	String of octal char(s) [**0–7, x, X, z, Z**] **x**When all the bits are **x** **X**When some of the bits are **x** **z**When all the bits are **z** **Z**When some of the bits are **z**
vpiDecStrVal	str	String of decimal char(s) [0–9]
vpiHexStrVal	str	String of hex char(s) [**0–f, x, X, z, Z**] **x**When all the bits are **x** **X**When some of the bits are **x** **z**When all the bits are **z** **Z**When some of the bits are **z**
vpiScalarVal	scalar	vpi1, vpi0, vpiX, vpiZ, vpiH, vpiL
vpiIntVal	integer	Integer value of the handle. Any bits x or z in the value of the object are mapped to a 0
vpiRealVal	real	Value of the handle as a double
vpiStringVal	str	A string where each 8-bit group of the value of the object is assumed to represent an ASCII character
vpiTimeVal	time	Integer value of the handle using two integers
vpiVectorVal	vector	*aval/bval* representation of the value of the object
vpiStrengthVal	strength	Value plus strength information of a scalar object only
vpiObjectVal	—	Return a value in the closest format of the object

NOTE—If the object has a real value, it shall be converted to an integer using the rounding defined by the Verilog-AMS HDL before being returned in a format other than **vpiRealVal**.

To get the ASCII values of UDP table entries (as explained in 8.1.6, Table 8-1 of IEEE Std 1364 Verilog), the *p_vpi_vecval* field shall point to an array of `s_vpi_vecval` structures. The size of this array shall be determined by the size of the table entry (no. of symbols per table entry), where *array_size = ((table_entry_size-1)/4 + 1)*. Each symbol shall require a byte; the ordering of the symbols within `s_vpi_vecval` shall be the most significant byte of *abit* first, then the least significant byte of *abit*, then the most significant byte of *bbit*, and then the least significant byte of *bbit*. Each symbol can be either one or two characters; when it is a single character, the second half of the byte shall be an ASCII "\0".

The *misc* field in the `s_vpi_value` structure shall provide for alternative value types, which can be implementation specific. If this field is utilized, one or more corresponding format types shall also be provided.

In the following example, the binary value of each net which is contained in a particular module and whose name begins with a particular string is displayed. (This function makes use of the **`strcmp()`** facility normally declared in a `string.h` C library.)

```
void display_certain_net_values(mod, target)
vpiHandle mod;
char *target;
{
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   static s_vpi_value value_s = {vpiBinStrVal};
   static p_vpi_value value_p = &value_s;
   vpiHandle net, itr;

   itr = vpi_iterate(vpiNet, mod);
   while (net = vpi_scan(itr))
   {
      char *net_name = vpi_get_str(vpiName, net);
      if (strcmp(target, net_name) == 0)
      {
         vpi_get_value(net, value_p);
         vpi_printf("Value of net %s: %s\n",
            vpi_get_str(vpiFullName, net),value_p->value.str);
      }
   }
}
```

The following example illustrates the use of **`vpi_get_value()`** to access UDP table entries. Two sample outputs from this example are provided after the example.

```
/*
 * hUDP shall be a handle to a UDP definition
 */
static void dumpUDPTableEntries(vpiHandle hUDP)

{
   vpiHandle hEntry, hEntryIter;
    s_vpi_value value;
    int numb;
    int udpType;
    int item;
    int entryVal;
    int *abItem;
    int cnt, cnt2;
    numb = vpi_get(vpiSize, hUDP);
    udpType = vpi_get(vpiPrimType, hUDP);
    if (udpType == vpiSeqPrim)
        numb++; /* There is one more table entry for state */
    numb++;   /* There is a table entry for the output */
    hEntryIter = vpi_iterate(vpiTableEntry, hUDP);
    if (!hEntryIter)
      return;
    value.format = vpiVectorVal;
    while(hEntry = vpi_scan(hEntryIter))
   {
      vpi_printf("\n");
      /* Show the entry as a string */
      value.format = vpiStringVal;
      vpi_get_value(hEntry, &value);
      vpi_printf("%s\n", value.value.str);
      /* Decode the vector value format */
      value.format = vpiVectorVal;
      vpi_get_value(hEntry, &value);
      abItem = (int *)value.value.vector;
      for(cnt=((numb-1)/2+1);cnt>0;cnt--)
      {
         entryVal = *abItem;
         abItem++;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
            /* Rip out 4 characters */
            for (cnt2=0;cnt2<4;cnt2++)
            {
               item = entryVal&0xff;
               if (item)
                  vpi_printf("%c", item);
               else
                  vpi_printf("_");
               entryVal = entryVal>>8;
            }
         }
      }
      vpi_printf("\n");
   }
```

For a UDP table of

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

The output from the preceding example is

```
10:1
_0_1___1
01:0
_1_0___0
00:1
_0_0___1
```

For a UDP table entry of

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

The output from the preceding example is

```
10:?:1
_0_1_1_?
0(01):?:-
10_0_-_?
(10)0:0:1
001_1_0
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.17 vpi_get_vlog_info()

vpi_get_vlog_info()			
Synopsis:	Retrieve information about Verilog-AMS simulation execution.		
Syntax:	vpi_get_vlog_info(vlog_info_p)		
	Type	Description	
Returns:	bool	`true` on success and `false` on failure	
	Type	**Name**	**Description**
Arguments:	p_vpi_vlog_info	vlog_info_p	Pointer to a structure containing simulation information

The VPI routine **`vpi_get_vlog_info()`** shall obtain the following information about Verilog-AMS product execution:

— The number of invocation options (*argc*)
— Invocation option values (*argv*)
— Product and version strings

The information shall be contained in an `s_vpi_vlog_info` structure. The routine shall return `TRUE` on success and `FALSE` on failure.

The `s_vpi_vlog_info` structure used by **`vpi_get_vlog_info()`** is defined in `vpi_user.h` and is listed in Figure 12-12.

```
typedef struct t_vpi_vlog_info {
   int argc;
   char **argv;
   char *product;
   char *version;
} s_vpi_vlog_info, *p_vpi_vlog_info;
```

Figure 12-12: The s_vpi_vlog_info structure definition

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.18 vpi_get_real()

vpi_get_real()			
Synopsis:	Fetch a real property value associated with an object.		
Syntax:	vpi_get_real(prop,obj)		
	Type	Description	
Returns:	double	value of a real property	
	Type	**Name**	**Description**
Arguments:	int	prop	An integer constant representing the property of an object for which to obtain a value
	vpiHandle	obj	Handle to an object

The VPI routine **`vpi_get_real()`** shall return the value of object properties, for properties of type *real.* Note for object properties shown below, if the object is `NULL`, then the corresponding value shall be returned.

- **`vpiStartTime`** for beginning of transient analysis time
- **`vpiEndTime`** for end of transient analysis time
- **`vpiTransientMaxStep`** for maximum analog time step
- **`vpiStartFrequency`** for the start frequency of AC analysis
- **`vpiEndFrequency`** for the end frequency of AC analysis

This function is available to analog tasks and functions only. Should an error occur, **`vpi_get_real()`** shall return **`vpiUndefined`**.

12.19 vpi_handle()

vpi_handle()			
Synopsis:	Obtain a handle to an object with a one-to-one relationship.		
Syntax:	vpi_handle(type, ref)		
	Type	Description	
Returns:	vpiHandle	Handle to an object	
	Type	**Name**	**Description**
Arguments:	int	type	An integer constant representing the type of object for which to obtain a handle
	vpiHandle	ref	Handle to a reference object
Related routines:	Use vpi_iterate() and vpi_scan() to obtain handles to objects with a one-to-many relationship Use vpi_handle_multi() to obtain a handle to an object with a many-to-one relationship		

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The VPI routine **vpi_handle()** shall return the object of type *type* associated with object *ref*. The one-to-one relationships which are traversed with this routine are indicated as single arrows in the data model diagrams.

The following example application displays each primitive that an input net drives.

```
void display_driven_primitives(net)
vpiHandle net;
{
   vpiHandle load, prim, itr;
   vpi_printf("Net %s drives terminals of the primitives: \n",
      vpi_get_str(vpiFullName, net));
   itr = vpi_iterate(vpiLoad, net);
   if (!itr)
      return;
   while (load = vpi_scan(itr))
   {
      switch(vpi_get(vpiType, load))
      {
         case vpiGate:
         case vpiSwitch:
         case vpiUdp:
            prim = vpi_handle(vpiPrimitive, load);
            vpi_printf("\t%s\n", vpi_get_str(vpiFullName, prim));
      }
   }
}
```

12.20 vpi_handle_by_index()

vpi_handle_by_index()			
Synopsis:	Get a handle to an object using its index number within a parent object.		
Syntax:	vpi_handle_by_index(obj, index)		
Returns:	**Type**	Description	
	vpiHandle	Handle to an object	
Arguments:	**Type**	**Name**	**Description**
	vpiHandle	obj	Handle to an object
	int	index	Index number of the object for which to obtain a handle

The VPI routine **vpi_handle_by_index()** shall return a handle to an object based on the index number of the object within a parent object. This function can be used to access all objects which can access an expression using **vpiIndex**. Argument *obj* shall represent the parent of the indexed object. For example, to access a net-bit, *obj* is the associated net, while for a memory word, *obj* is the associated memory.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.21 vpi_handle_by_name()

vpi_handle_by_name()			
Synopsis:	Get a handle to an object with a specific name.		
Syntax:	vpi_handle_by_name(name, scope)		
Returns:	**Type**	Description	
	vpiHandle	Handle to an object	
Arguments:	**Type**	**Name**	**Description**
	char *	name	A character string or pointer to a string containing the name of an object
	vpiHandle	scope	Handle to a Verilog-AMS HDL scope

The VPI routine **`vpi_handle_by_name()`** shall return a handle to an object with a specific name. This function can be applied to all objects with a *fullname* property. The *name* can be hierarchical or simple. If *scope* is `NULL`, then *name* shall be searched for from the top level of hierarchy. Otherwise, *name* shall be searched for from *scope* using the scope search rules defined by the Verilog-AMS HDL.

12.22 vpi_handle_multi()

vpi_handle_multi()			
Synopsis:	Obtain a handle to inter-module paths with a many-to-one relationship.		
Syntax:	vpi_handle_multi(type, ref1, ref2, ...)		
Returns:	**Type**	Description	
	vpiHandle	Handle to an object	
Arguments:	**Type**	**Name**	**Description**
	int	type	An integer constant representing the type of object for which to obtain a handle
	vpiHandle	ref1, ref2, ...	Handles to two or more reference objects
Related routines:	Use vpi_iterate() and vpi_scan() to obtain handles to objects with a one-to-many relationship Use vpi_handle() to obtain handles to objects with a one-to-one relationship		

The VPI routine **`vpi_handle_multi()`** shall return a handle to objects of type **`vpiInterModPath`** associated with a list of *output port* and *input port* reference objects. The ports shall be of the same size and can be at different levels of the hierarchy. This routine performs a *many-to-one* operation instead of the usual one-to-one or one-to-many.

12.22.1 Derivatives for analog system task/functions

The VPI routine **`vpi_handle_multi()`** is used to access the derivative handles associated with analog system task/functions (see also: **`vpi_register_analog_systf()`**). The first argument is the type **`vpiDerivative`**. The second is the handle for the task/function argument for which a partial derivative is

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

to be declared. The third argument indicates the value with respect to which the derivative being declared shall be calculated. For example, assuming `argHandle2` and `argHandle3` are handles to the second and third arguments of an analog system task, then `vpi_handle_multi(vpiDerivative, argHandle2, argHandle3)` indicates the partial derivative of the returned value with respect to the third argument. For **`vpiDerivative`**, the **`vpi_handle_multi()`** function can only be called for those derivatives allocated during the *derivtf* phase of execution.

12.22.2 Examples

The following example illustrates the declaration and use of derivative handles in an analog task `$resistor()`, which implements a conductance relationship. The task can be used as follows:

```
module resistor(p, n);
   electrical p, n;
   parameter real r = 1k;
   real curr;
   analog begin
      $resistor(curr, V(p, n), r);
      I(p, n) <+ curr;
   end
endmodule
```

The implementation of the analog task can be performed by the **`resistor_compile_tf()`** and **`resistor_call_tf()`** routines shown below:

```
#include "vpiutils.h"

/* compiletf() */
static int resistor_compiletf(p_cb_data cb_data) {
   vpiHandle funcHandle, i_handle, v_handle, r_handle, didv_handle;
   int type;
   s_vpi_value value;
   double g;
   p_resistor_data res;

   /* Retrieve handle to current function */
   funcHandle = vpi_handle(vpiSysTfCall, NULL);

   /* Get the handle on the first function argument*/
   i_handle = vpi_handle_by_index(funcHandle, 1);

   /* Check that argument exists */
   if (!i_handle) {
      vpi_error("Not enough arguments for $resistor function.");
   }

   /* Check that argument #1 is a real variable */
   type = vpi_get(vpiType, v_handle);
   if (type != vpiRealVar) {
      vpi_error("Arg #1 of $resistor should be a real variable");
      return 1;
   }

   /* Get the handle on the second function argument*/
   v_handle = vpi_handle_by_index(funcHandle,  2);

   /* Check that argument exists */
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   if (!v_handle) {
      vpi_error("Not enough arguments for $resistor function.");
      return 1;
   }

   /* Check that argument #1 is a real valued */
   type = vpi_get(vpiType, v_handle);
   if (type != vpiRealVar && type != vpiRealVal) {
      vpi_error("Arg #2 of $resistor should be a real variable");
      return 1;
   }
   /* Get the handle on the third function argument*/
   r_handle = vpi_handle_by_index(funcHandle, 3);

   /* Check that argument exists */
   if (!v_handle) {
      vpi_error("Not enough arguments for $resistor function.");
      return 1;
   }

   /* Check that argument #3 is real valued */
   type = vpi_get(vpiType, r_handle);
   if (type != vpiRealVar && type != vpiRealVal) {
      vpi_error("Arg #3 of $resistor should be a real variable");
      return 1;
   }

   return 0;
}

/* derivtf() */
static p_vpi_stf_partials resistor_derivtf(p_cb_data cb_data) {
   static t_vpi_stf_partials  derivs;
   static int deriv_of[] = { 1 };
   static int deriv_to[] = { 2 };

   derivs.count = 1;
   derivs.derivative_of = deriv_of;
   derivs.derivative_to = deriv_to;

   return &derivs;
}

/* load() */
static int resistor_calltf(int data, int reason) {
   vpiHandle funcHandle, i_handle, v_handle, didv_handle;
   double g;
   s_vpi_value value;

   /* Retrieve handle to current function */
   funcHandle = vpi_handle(vpiSysTfCall, NULL);
   i_handle = vpi_handle_by_index(funcHandle, 1);
   v_handle = vpi_handle_by_index(funcHandle,  2);
   didv_handle = vpi_handle_multi(vpiDerivative, i_handle, v_handle);

   /* Get resistance value, compute conductance and store it as */
   /* derivative */
   value.format = vpiRealVal;
   vpi_get_value(r_handle, &value);
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   g = 1.0 / value.value.real;

   value.value.real = g;
   vpi_put_value(didv_handle, &value, NULL, vpiNoDelay);

   /* Get voltage value, compute current and store it into "I"*/
   vpi_get_value(v_handle, &value);
   value.value.real *= g;
   vpi_put_value(i_handle, &value, NULL, vpiNoDelay);
   return 0;
}

/*
 * Public structure declaring the task
 */
static s_vpi_analog_systf_data resistor_systf = {
   vpiSysAnalogTask,    /* type: function / task   */
   0,                   /* returned type           */
   "$resistor",         /* name                    */
   resistor_calltf,     /* calltf callback         */
   resistor_compiletf,  /* compiletf callback      */
   0,                   /* unused: sizetf callback */
   resistor_derivtf,    /* derivtf callback        */
   0                    /* user_data: nothing      */
};
```

12.23 vpi_iterate()

vpi_iterate()			
Synopsis:	Obtain an iterator handle to objects with a one-to-many relationship.		
Syntax:	vpi_iterate(type, ref)		
Returns:	**Type**	Description	
	vpiHandle	Handle to an iterator for an object	
Arguments:	**Type**	**Name**	**Description**
	int	type	An integer constant representing the type of object for which to obtain iterator handles
	vpiHandle	ref	Handle to a reference object
Related routines:	Use vpi_scan() to traverse the HDL hierarchy using the iterator handle returned from vpi_iterate() Use vpi_handle() to obtain handles to object with a one-to-one relationship Use vpi_handle_multi() to obtain a handle to an object with a many-to-one relationship		

The VPI routine **`vpi_iterate()`** shall be used to traverse one-to-many relationships, which are indicated as double arrows in the data model diagrams. The **`vpi_iterate()`** routine shall return a handle to an iterator, whose type shall be **`vpiIterator`**, which can used by **`vpi_scan()`** to traverse all objects of type *type* associated with object *ref*. To get the reference object from the iterator object use `vpi_handle(vpiUse, iterator_handle)`. If there are no objects of type *type* associated with the reference handle *ref*, then the **`vpi_iterate()`** routine shall return `NULL`.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The following example application uses **vpi_iterate()** and **vpi_scan()** to display each net (including the size for vectors) declared in the module. The example assumes it shall be passed a valid module handle.

```
void display_nets(mod)
vpiHandle mod;
{
   vpiHandle net;
   vpiHandle itr;

   vpi_printf("Nets declared in module %s\n",
      vpi_get_str(vpiFullName, mod));

   itr = vpi_iterate(vpiNet, mod);
   while (net = vpi_scan(itr))
   {
      vpi_printf("\t%s", vpi_get_str(vpiName, net));
      if (vpi_get(vpiVector, net))
      {
         vpi_printf(" of size %d\n", vpi_get(vpiSize, net));
      }
      else vpi_printf("\n");
   }
}
```

12.24 vpi_mcd_close()

vpi_mcd_close()			
Synopsis:	Close one or more files opened by vpi_mcd_open().		
Syntax:	vpi_mcd_close(mcd)		
	Type	Description	
Returns:	unsigned int	0 if successful, the mcd of unclosed channels if unsuccessful	
	Type	**Name**	**Description**
Arguments:	unsigned int	mcd	A multichannel descriptor representing the files to close
Related routines:	Use vpi_mcd_open() to open a file Use vpi_mcd_printf() to write to an opened file Use vpi_mcd_name() to get the name of a file represented by a channel descriptor		

The VPI routine **vpi_mcd_close()** shall close the file(s) specified by a multichannel descriptor, *mcd*. Several channels can be closed simultaneously, since channels are represented by discrete bits in the integer *mcd*. On success this routine returns a zero (0); on error it returns the *mcd* value of the unclosed channels.

The following descriptors are predefined and can not be closed using **vpi_mcd_close()**:

— descriptor 1 is *stdout*
— descriptor 2 is *stderr*
— descriptor 3 is the current log file

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.25 vpi_mcd_name()

vpi_mcd_name()			
Synopsis:	Get the name of a file represented by a channel descriptor.		
Syntax:	vpi_mcd_name(cd)		
	Type	Description	
Returns:	char *	Pointer to a character string containing the name of a file	
	Type	**Name**	**Description**
Arguments:	unsigned int	cd	A single-channel descriptor representing a file
Related routines:	Use vpi_mcd_open() to open a file Use vpi_mcd_close() to close files Use vpi_mcd_printf() to write to an opened file		

The VPI routine **`vpi_mcd_name()`** shall return the name of a file represented by a single-channel descriptor, *cd*. On error, the routine shall return `NULL`. This routine shall overwrite the returned value on subsequent calls. If the application needs to retain the string, it shall copy it.

12.26 vpi_mcd_open()

vpi_mcd_open()			
Synopsis:	Open a file for writing.		
Syntax:	vpi_mcd_open(file)		
	Type	Description	
Returns:	unsigned int	A multichannel descriptor representing the file which was opened	
	Type	**Name**	**Description**
Arguments:	char *	file	A character string or pointer to a string containing the file name to be opened
Related routines:	Use vpi_mcd_close() to close a file Use vpi_mcd_printf() to write to an opened file Use vpi_mcd_name() to get the name of a file represented by a channel descriptor		

The VPI routine **`vpi_mcd_open()`** shall open a file for writing and return a corresponding multichannel descriptor number (*mcd*). The following channel descriptors are predefined and shall be automatically opened by the system:

- Descriptor 1 is *stdout*
- Descriptor 2 is *stderr*
- Descriptor 3 is the current log file

The **`vpi_mcd_open()`** routine shall return a zero (`0`) on error. If the file is already opened, **`vpi_mcd_open()`** shall return the descriptor number.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.27 vpi_mcd_printf()

vpi_mcd_printf()			
Synopsis:	Write to one or more files opened with vpi_mcd_open().		
Syntax:	vpi_mcd_printf(mcd, format, ...)		
	Type	Description	
Returns:	int	The number of characters written	
	Type	**Name**	**Description**
Arguments:	unsigned int	mcd	A multichannel descriptor representing the files to which to write
	char *	format	A format string using the C fprintf() format
Related routines:	Use vpi_mcd_open() to open a file Use vpi_mcd_close() to close a file Use vpi_mcd_name() to get the name of a file represented by a channel descriptor		

The VPI routine **`vpi_mcd_printf()`** shall write to one or more channels (up to 32) determined by the *mcd*. An *mcd* of `1` (bit `0` set) corresponds to Channel `1`, a *mcd* of `2` (bit `1` set) corresponds to Channel 2, a *mcd* of `4` (bit `2` set) corresponds to Channel `3`, and so on. Channel `1` is *stdout*, channel `2` is *stderr*, and channel `3` is the current log file. Several channels can be written to simultaneously, since channels are represented by discrete bits in the integer *mcd*. The format strings shall use the same format as the C **`fprintf()`** routine.The routine shall return the number of characters printed or `EOF` if an error occurred.

12.28 vpi_printf()

vpi_printf()			
Synopsis:	Write to stdout and the current product log file.		
Syntax:	vpi_printf(format, ...)		
	Type	Description	
Returns:	int	The number of characters written	
	Type	**Name**	**Description**
Arguments:	char *	format	A format string using the C printf() format
Related routines:	Use vpi_mcd_printf() to write to an opened file		

The VPI routine **`vpi_printf()`** shall write to both *stdout* and the current product log file. The format string shall use the same format as the C **`printf()`** routine. The routine shall return the number of characters printed or `EOF` if an error occurred.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.29 vpi_put_delays()

vpi_put_delays()			
Synopsis:	Set the delays or timing limits of an object.		
Syntax:	vpi_put_delays(obj, delay_p)		
Returns:	**Type**	Description	
	void		
Arguments:	**Type**	**Name**	**Description**
	vpiHandle	obj	Handle to an object
	p_vpi_delay	delay_p	Pointer to a structure containing delay information
Related routines:	Use vpi_get_delays() to retrieve delays or timing limits of an object		

The VPI routine **`vpi_put_delays()`** shall set the delays or timing limits of an object as indicated in the *delay_p* structure. The same ordering of delays shall be used as described in the **`vpi_get_delays()`** function. If only the delay changes, and not the pulse limits, the pulse limits shall retain the values they had before the delays where altered.

The `s_vpi_delay` and `s_vpi_time` structures used by both **`vpi_get_delays()`** and **`vpi_put_delays()`** are defined in `vpi_user.h` and are listed in Figure 12-13 and Figure 12-14.

```
typedef struct t_vpi_delay {
   struct t_vpi_time *da;  /* ptr to user allocated array of delay
                           values */
   int no_of_delays;       /* number of delays */
   int time_type;          /* [vpiScaledRealTime, vpiSimTime] */
   bool mtm_flag;          /* true for mtm */
   bool append_flag;       /* true for append, false for replace */
   bool pulsere_flag;      /* true for pulsere values */
} s_vpi_delay, *p_vpi_delay;
```

Figure 12-13: The s_vpi_delay structure definition

```
typedef struct t_vpi_time
{
   int type;               /* [vpiScaledRealTime, vpiSimTime] */
   unsigned int high, low; /* for vpiSimTime */
   double real;            /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 12-14: The s_vpi_time structure definition

The *da* field of the `s_vpi_delay` structure shall be a user-allocated array of `s_vpi_time` structures. This array shall store the delay values to be written by **`vpi_put_delays()`**. The number of elements in this array shall be determined by:

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

— The number of delays to be retrieved
— The **`mtm_flag`** setting
— The **`pulsere_flag`** setting

The number of delays to be retrieved shall be set in the *no_of_delays* field of the `s_vpi_delay` structure. Legal values for the number of delays shall be determined by the type of object.

— For primitive objects, the *no_of_delays* value shall be `2` or `3`.
— For path delay objects, the *no_of_delays* value shall be `1`, `2`, `3`, `6`, or `12`.
— For timing check objects, the *no_of_delays* value shall match the number of limits existing in the timing check.
— For inter-module path objects, the *no_of_delays* value shall be `2` or `3`.

The user-allocated `s_vpi_delay` array shall contain delays in the same order in which they occur in the Verilog-AMS HDL description. The number of elements for each delay shall be determined by the flags **`mtm_flag`** and **`pulsere_flag`**, as shown in Table 12-5.

Table 12-5—Size of the s_vpi_delay->da array

Flag values	**Number of `s_vpi_time` array elements required for *s_vpi_delay->da***	**Order in which delay elements shall be filled**
mtm_flag = `false` **pulsere_flag** = `false`	*no_of_delays*	1st delay: da[0] -> 1st delay 2nd delay: da[1] -> 2nd delay ...
mtm_flag = `true` **pulsere_flag** = `false`	3 * *no_of_delays*	1st delay: da[0] -> min delay da[1] -> typ delay da[2] -> max delay 2nd delay: ...
mtm_flag = `false` **pulsere_flag** = `true`	3 * *no_of_delays*	1st delay: da[0] -> delay da[1] -> reject limit da[2] -> error limit 2nd delay element: ...
mtm_flag = `true` **pulsere_flag** = `true`	9 * *no_of_delays*	1st delay: da[0] -> min delay da[1] -> typ delay da[2] -> max delay da[3] -> min reject da[4] -> typ reject da[5] -> max reject da[6] -> min error da[7] -> typ error da[8] -> max error 2nd delay: ...

The following example application accepts a module path handle, rise and fall delays, and replaces the delays of the indicated path.

```
void set_path_rise_fall_delays(path, rise, fall)
vpiHandle path;
double rise, fall;
{
   static s_vpi_time path_da[2];
   static s_vpi_delay delay_s = {NULL, 2, vpiScaledRealTime};
   static p_vpi_delay delay_p = &delay_s;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    delay_s.da = &path_da;
    path_da[0].real = rise;
    path_da[1].real = fall;

    vpi_put_delays(path, delay_p);
}
```

12.30 vpi_put_value()

vpi_put_value()			
Synopsis:	Set a value on an object.		
Syntax:	vpi_put_value(obj, value_p, time_p, flags)		
Returns:	**Type**	Description	
	vpiHandle	Handle to the scheduled event caused by vpi_put_value()	
Arguments:	**Type**	**Name**	**Description**
	vpiHandle	obj	Handle to an object
	p_vpi_value	value_p	Pointer to a structure with value information
	p_vpi_time	time_p	Pointer to a structure with delay information
	int	flags	Integer constants which set the delay mode
Related routines:	Use vpi_get_value() to retrieve the value of an expression		

The VPI routine **`vpi_put_value()`** shall set simulation logic values on an object. The value to be set shall be stored in an `s_vpi_value` structure which has been allocated. The delay time before the value is set shall be stored in an `s_vpi_time` structure which has been allocated. The routine can be applied to nets, regs, variables, memory words, system function calls, sequential UDPs, and schedule events. The *flags* argument shall be used to direct the routine to use one of the following delay modes:

`vpiInertialDelay` All scheduled events on the object shall be removed before this event is scheduled.

`vpiTransportDelay` All events on the object scheduled for times later than this event shall be removed (modified transport delay).

`vpiPureTransportDelay` No events on the object shall be removed (transport delay).

`vpiNoDelay` The object shall be set to the passed value with no delay. Argument *time_p* shall be ignored and can be set to `NULL`.

`vpiForceFlag` The object shall be forced to the passed value with no delay (same as the Verilog-AMS HDL procedural **`force`**). Argument *time_p* shall be ignored and can be set to `NULL`.

`vpiReleaseFlag` The object shall be released from a forced value (same as the Verilog-AMS HDL procedural **`release`**). Argument *time_p* shall be ignored and can be set to `NULL`. The *value_p* shall contain the current value of the object.

`vpiCancelEvent` A previously scheduled event shall be canceled. The object passed to **`vpi_put_value()`** shall be a handle to an object of type **`vpiSchedEvent`**.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

If the *flags* argument also has the bit mask **vpiReturnEvent**, **vpi_put_value()** shall return a handle of type **vpiSchedEvent** to the newly scheduled event, provided there is some form of a delay and an event is scheduled. If the bit mask is not used, or if no delay is used, or if an event is not scheduled, the return value shall be NULL.

The handle to the event can be canceled by calling **vpi_put_value()** with the flag set to **vpiCancelEvent**. It shall not be an error to cancel an event which has already occurred. The scheduled event can be tested by calling **vpi_get()** with the flag **vpiScheduled**. If an event is canceled, it shall simply be removed from the event queue. Any effects which were caused by scheduling the event shall remain in effect (e.g., events which were canceled due to inertial delay).

Calling **vpi_free_object()** on the handle shall free the handle but shall not effect the event.

Sequential UDPs shall be set to the indicated value with no delay regardless of any delay on the primitive instance.

NOTE—**vpi_put_value()** shall only return a function value in a **calltf** application, when the call to the function is active. The action of **vpi_put_value()** to a function shall be ignored when the function is not active.

The s_vpi_value and s_vpi_time structures used by **vpi_put_value()** are defined in vpi_user.h and are listed in Figure 12-15 and Figure 12-16.

```
typedef struct t_vpi_value {
   int format; /* vpi[[Bin,Oct,Dec,Hex]Str,Scalar,Int,Real,String,
                  Time,Vector,Strength,ObjType]Val*/
   union {
      char *str;
      int scalar; /* vpi[0,1,X,Z] */
      int integer;
      double real;
      struct t_vpi_time *time;
      struct t_vpi_vecval *vector;
      struct t_vpi_strengthval *strength;
      char *misc;
   } value;
} s_vpi_value, *p_vpi_value;
```

Figure 12-15: The s_vpi_value structure definition

```
typedef struct t_vpi_time {
   int type;               /* for vpiScaledRealTime, vpiSimTime */
   unsigned int high, low; /* for vpiSimTime */
   double real;            /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 12-16: The s_vpi_time structure definition

For **vpiScaledRealTime**, the indicated time shall be in the timescale associated with the object.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.31 vpi_register_cb()

vpi_register_cb()			
Synopsis:	Register simulation-related callbacks.		
Syntax:	vpi_register_cb(cb_data_p)		
Returns:	**Type**	Description	
	vpiHandle	Handle to the callback object	
Arguments:	**Type**	**Name**	**Description**
	p_cb_data	cb_data_p	Pointer to a structure with data about when callbacks should occur and the data to be passed
Related routines:	Use vpi_register_systf() to register callbacks for user-defined system tasks and functions Use vpi_remove_cb() to remove callbacks registered with vpi_register_cb()		

The VPI routine **`vpi_register_cb()`** is used for registration of simulation-related callbacks to a user-provided application for a variety of reasons during a simulation. The reasons for which a callback can occur are divided into three categories:

- Simulation event
- Simulation time
- Simulation action or feature

How callbacks are registered for each of these categories is explained in the following paragraphs.

The *cb_data_p* argument shall point to a `s_cb_data` structure, which is defined in `vpi_user.h` and given in Figure 12-17.

```
typedef struct t_cb_data {
   int reason;
   int (*cb_rtn)();
   vpiHandle obj;
   p_vpi_time time;     /* structure defined in vpi_user.h */
   p_vpi_value value;   /* structure defined in vpi_user.h */
   int index;  /* index of memory word or var select which changed */
   char *user_data;     /* user data to be passed to callback function */
} s_cb_data, *p_cb_data;
```

Figure 12-17: The s_cb_data structure definition

For all callbacks, the *reason* field of the `s_cb_data` structure shall be set to a predefined constant, such as **`cbValueChange`**, **`cbAtStartOfSimTime`**, **`cbEndOfCompile`**, etc. The reason constant shall determine when the user application shall be called back. Refer to the `vpi_user.h` file listing in *Annex G* of the IEEE Std 1364 Verilog specification for a list of all callback reason constants.

The *cb_rtn* field of the `s_cb_data` structure shall be set to the application routine name, which shall be invoked when the simulator executes the callback. The use of the remaining fields are detailed in the following sub clauses.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.31.1 Simulation-event-related callbacks

The **vpi_register_cb()** callback mechanism can be registered for callbacks to occur for simulation events, such as value changes on an expression or terminal, or the execution of a behavioral statement. When the *cb_data_p->reason* field is set to one of the following, the callback shall occur as described below:

cbValueChange	After value change on an expression or terminal
cbStmt	Before execution of a behavioral statement
cbForce/cbRelease	After a force or release has occurred
cbAssign/cbDeassign	After a procedural assign or deassign statement has been executed
cbDisable	After a named block or task containing a system task or function has been disabled

The following fields shall need to be initialized before passing the `s_cb_data` structure to **vpi_register_cb()**:

cb_data_p->obj	This field shall be assigned a handle to an expression, terminal, or statement for which the callback shall occur. For force and release callbacks, if this is set to `NULL`, every force and release shall generate a callback.
cb_data_p->time->type	This field shall be set to either **vpiScaledRealTime** or **vpiSimTime**, depending on what time information the user application requires during the callback. If simulation time information is not needed during the callback, this field can be set to **vpiSuppressTime**.
cb_data_p->value->format	This field shall be set to one of the value formats indicated in Table 12-6. If value information is not needed during the callback, this field can be set to **vpiSuppressVal**. For **cbStmt** callbacks, value information is not passed to the callback routine, so this field shall be ignored.

Table 12-6—Value format field of cb_data_p->value->format

Format	Registers a callback to return
vpiBinStrVal	String of binary char(s) [**1, 0, x, z**]
vpiOctStrVal	String of octal char(s) [**0–7, x, X, z, Z**]
vpiDecStrVal	String of decimal char(s) [0–9]
vpiHexStrVal	String of hex char(s) [**0–f, x, X, z, Z**]
vpiScalarVal	**vpi1, vpi0, vpiX, vpiZ, vpiH, vpiL**
vpiIntVal	Integer value of the handle
vpiRealVal	Value of the handle as a double
vpiStringVal	An ASCII string
vpiTimeVal	Integer value of the handle using two integers
vpiVectorVal	*aval/bval* representation of the value of the object
vpiStrengthVal	Value plus strength information of a scalar object only
vpiObjectVal	Return a value in the closest format of the object

When a simulation event callback occurs, the user application shall be passed a single argument, which is a pointer to an `s_cb_data` structure (this is not a pointer to the same structure which was passed to

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

vpi_register_cb()). The *time* and *value* information shall be set as directed by the time *type* and *value* format fields in the call to **vpi_register_cb()**. The *user_data* field shall be equivalent to the *user_data* field passed to **vpi_register_cb()**. The user application can use the information in the passed structure and information retrieved from other VPI interface routines to perform the desired callback processing.

For a **cbValueChange** callback, if the *obj* is a memory word or a variable array, the *value* in the `s_cb_data` structure shall be the value of the memory word or variable select which changed value. The *index* field shall contain the index of the memory word or variable select which changed value.

For **cbForce**, **cbRelease**, **cbAssign**, and **cbDeassign** callbacks, the object returned in the *obj* field shall be a handle to the force, release, assign or deassign statement. The *value* field shall contain the resultant value of the LHS expression. In the case of a release, the *value* field shall contain the value after the release has occurred.

The following example shows an implementation of a simple monitor functionality for scalar nets, using a simulation-event-related callback.

```
setup_monitor(net)
vpiHandle net;
{
   static s_vpi_time time_s = {vpiScaledRealTime};
   static s_vpi_value value_s = {vpiBinStrVal};
   static s_cb_data cb_data_s =
      {cbValueChange, my_monitor, NULL, &time_s, &value_s};
   char *net_name = vpi_get_str(vpiFullName, net);
   cb_data_s.obj = net;
   cb_data_s.user_data = malloc(strlen(net_name)+1);
   strcpy(cb_data_s.user_data, net_name);
   vpi_register_cb(&cb_data_s);
}

my_monitor(cb_data_p)
p_cb_data cb_data_p; {
   vpi_printf("%d %d: %s = %s\n",
      cb_data_p->time->high, cb_data_p->time->low,
      cb_data_p->user_data,
      cb_data_p->value->value.str);
}
```

12.31.2 Simulation-time-related callbacks

The **vpi_register_cb()** can register callbacks to occur for simulation time reasons, include callbacks at the beginning or end of the execution of a particular time queue. The following time-related callback reasons are defined:

cbAtStartOfSimTime	Callback shall occur before execution of events in a specified time queue. A callback can be set for any time, even if no event is present.
cbReadWriteSynch	Callback shall occur after execution of events for a specified time.
cbReadOnlySynch	Same as **cbReadWriteSynch**, except writing values or scheduling events before the next scheduled event is not allowed.
cbNextSimTime	Callback shall occur before execution of events in the next event queue.
cbAfterDelay	Callback shall occur after a specified amount of time, before execution of events in a specified time queue. A callback can be set for anytime, even if no event is present.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The following fields shall need to be set before passing the `s_cb_data` structure to **`vpi_register_cb()`**:

cb_data_p->time->type This field shall be set to either **`vpiScaledRealTime`** or **`vpiSimTime`**, depending on what time information the user application requires during the callback.

cb_data_p->[time->low,time->high,time->real] These fields shall contain the requested time of the callback or the delay before the callback.

The *value* fields are ignored for all reasons with simulation-time-related callbacks.

When the *cb_data_p->time->type* is set to **`vpiScaledRealTime`**, the *cb_data_p->obj* field shall be used as the object for determining the time scaling.

For reason **`cbNextSimTime`**, the time structure is ignored.

When a simulation-time-related callback occurs, the user callback application shall be passed a single argument, which is a pointer to an `s_cb_data` structure (this is not a pointer to the same structure which was passed to **`vpi_register_cb()`**). The *time* structure shall contain the current simulation time. The *user_data* field shall be equivalent to the *user_data* field passed to **`vpi_register_cb()`**.

The callback application can use the information in the passed structure and information retrieved from other interface routines to perform the desired callback processing.

12.31.3 Simulator analog and related callbacks

The **`vpi_register_cb()`** callback mechanism can be registered for callbacks to occur for analog simulation events, such as acceptance of the initial or final analog solution. When the *cb_data_p->reason* field is set to one of the following, the callback shall occur as described below:

`acbInitialStep` Upon acceptance of the first analog solution

`acbFinalStep` Upon acceptance of the last analog solution

`acbAbsTime` Upon acceptance of the analog solution for the given time (this callback shall force a solution at that time)

`acbElapsedTime` Upon acceptance of the solution advanced from the current solution by the given interval (this callback shall force a solution at that time)

`acbConvergenceTest` Prior acceptance of the analog solution for the given time (this callback allows rejection of the analog solution at that time and backup to an earlier time)

`acbAcceptedPoint` Upon acceptance of the solution at the given time

12.31.4 Simulator action and feature related callbacks

The **`vpi_register_cb()`** can register callbacks to occur for simulator action reasons or simulator feature reasons. *Simulator action reasons* are callbacks such as the end of compilation or end of simulation. *Simulator feature reasons* are software-product-specific features, such as restarting from a saved simulation state or entering an interactive mode. Actions are differentiated from features in that actions shall occur in all VPI-compliant products, whereas features might not exist in all VPI-compliant products.

The following action-related callbacks shall be defined:

`cbEndOfCompile` End of simulation data structure compilation or build

`cbStartOfSimulation` Start of simulation (beginning of time `0` simulation cycle)

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

cbEndOfSimulation	End of simulation (e.g., `$finish` system task executed)
cbError	Simulation run-time error occurred
cbPLIError	Simulation run-time error occurred in a PLI function call
cbTchkViolation	Timing check error occurred

Examples of possible feature related callbacks are

cbStartOfSave	Simulation save state command invoked
cbEndOfSave	Simulation save state command completed
cbStartOfRestart	Simulation restart from saved state command invoked
cbEndOfRestart	Simulation restart command completed
cbEnterInteractive	Simulation entering interactive debug mode (e.g., `$stop` system task executed)
cbExitInteractive	Simulation exiting interactive mode
cbInteractiveScopeChange	Simulation command to change interactive scope executed
cbUnresolvedSystf	Unknown user-defined system task or function encountered

The only fields in the `s_cb_data` structure which need to be setup for simulation action/feature callbacks are the *reason*, *cb_rtn*, and *user_data* (if desired) fields.

When a simulation action/feature callback occurs, the user routine shall be passed a pointer to an `s_cb_data` structure. The *reason* field shall contain the reason for the callback. For **cbTchkViolation** callbacks, the *obj* field shall be a handle to the timing check. For **cbInteractiveScopeChange**, *obj* shall be a handle to the new scope. For **cbUnresolvedSystf**, *user_data* shall point to the name of the unresolved task or function. On a **cbError** callback, the routine **vpi_chk_error()** can be called to retrieve error information.

The following example shows a callback application which reports cpu usage at the end of a simulation. If the user routine `setup_report_cpu()` is placed in the `vlog_startup_routines` list, it shall be called just after the simulator is invoked.

```
static int initial_cputime_g;

void report_cpu()
{
   int total = get_current_cputime() - initial_cputime_g;
   vpi_printf("Simulation complete. CPU time used: %d\n", total);
}

void setup_report_cpu()
{
   static s_cb_data cb_data_s = {cbEndOfSimulation, report_cpu};
   initial_cputime_g = get_current_cputime();
   vpi_register_cb(&cb_data_s);
}
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.32 vpi_register_analog_systf()

vpi_register_analog_systf()			
Synopsis:	Register user-defined system task/function-related callbacks.		
Syntax:	vpi_register_analog_systf(systf_data_p)		
Returns:	**Type**	Description	
	vpiHandle	Handle to the callback object	
Arguments:	**Type**	**Name**	**Description**
	p_vpi_analog_systf_data	systf_analog_data_p	Pointer to a structure with data about when callbacks should occur and the data to be passed
Related routines:	Use vpi_register_systf() to register digital domain system tasks/functions. Use vpi_register_cb() to register callbacks for simulation-related events		

The VPI routine **`vpi_register_analog_systf()`** shall register callbacks for user-defined analog system tasks or functions. Callbacks can be registered to occur when a user-defined system task or function is encountered during compilation or execution of analog Verilog-AMS HDL source code. Tasks or functions can be registered with either the analog or digital domain. The registration function (**`vpi_register_analog_systf()`** or **`vpi_register_systf()`**) with which the task or function is registered shall determine the context or contexts from which the task or function can be invoked and how and when the call backs associated with the function shall be called. The task or function name shall be unique in the domain in which it is registered. That is, the same name can be shared by two sets of callbacks, provided that one set is registered in the digital domain and the other is registered in the analog.

The *systf_analog_data_p* argument shall point to a `s_vpi_systf_analog_data` structure, which is defined in `vpi_user.h` and listed in Figure 12-18.

```
typedef struct t_vpi_systf_analog_data {
   int type;         /* vpiAnalogSysTask,vpiAnalogSysFunc */
   int sysfunctype;  /* vpi[IntFunc,RealFunc] */
   char *tfname;     /* first character shall be "$" */
   int (*calltf)();
   int (*compiletf)();
   int (*sizetf)();  /* for vpiSizedFunc system functions only */
   p_vpi_stf_partials (*derivtf)(); /* for partial derivatives */
   char *user_data;
} s_vpi_analog_systf_data, *p_vpi_analog_systf_data;
```

Figure 12-18: The s_vpi_analog_systf_data structure definition

12.32.1 System task and function callbacks

User-defined Verilog-AMS system tasks and functions which use VPI routines can be registered with **`vpi_register_systf()`** or **`vpi_register_analog_systf()`**. The *calltf*, *compiletf*, and *sizetf* system task/function-related callbacks are defined in **`vpi_register_systf()`**.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The *type* field of the `s_vpi_systf_data` structure shall register the user application to be a system task or a system function. The *type* field value shall be an integer constant of **`vpiAnalogSysTask`** or **`vpiAnalogSysFunction`**.

The *sysfunctype* field of the `s_vpi_analog_systf_data` structure shall define the type of value which a system function shall return. The *sysfunctype* field shall be an integer constant of **`vpiIntFunc`** of **`vpiRealFunc`**. This field shall only be used when the *type* field is set to **`vpiAnalogSysFunction`**.

The *compiletf*, *calltf*, *sizetf*, and *derivtf* fields of the `s_vpi_analog_systf_data` structure shall be pointers to the user-provided applications which are to be invoked by the system task/function callback mechanism. One or more of the *compiletf*, *calltf*, *sizetf*, and *derivtf* fields can be set to `NULL` if they are not needed. Callbacks to the applications pointed to by the *compiletf* and *sizetf* fields shall occur when the simulation data structure is compiled or built (or for the first invocation if the system task or function is invoked from an interactive mode). Callbacks to the applications pointed to by the *derivtf* fields shall occur when registering partial derivatives for the analog system task/function arguments or return value. Callbacks to the application pointed to by the *calltf* routine shall occur each time the system task or function is invoked during simulation execution.

The *user_data* field of the `s_vpi_analog_systf_data` structure shall specify a user-defined value, which shall be passed back to the *compiletf*, *sizetf*, *derivtf*, and *calltf* applications when a callback occurs.

The usage of the *compiletf*, *sizetf*, and *calltf* routines for the analog system task/function are identical to those of digital system task/functions registered with **`vpi_register_systf()`**. Refer to the description of **`vpi_register_systf()`** for more information.

12.32.2 Declaring derivatives for analog system task/functions

Analog system tasks and functions require partial derivatives of the outputs (arguments for system tasks and the return value for system functions). Thus it is possible (though not necessary) to have a partial derivative of the returned value with respect to any or all of the arguments and a partial derivative of any particular argument with respect to any or all of the other arguments.

The *derivtf* field of the `t_vpi_analog_systf_data` structure can be called during the build process (similar to *sizetf*) and returns a pointer to a `t_vpi_stf_partials` data structure containing the required information. The purpose of this function is declarative only, it does not assign any value to the derivative being declared. Having declared a partial derivative using this function in the *derivtf* callback, values can then be contributed to the derivative using the `vpi_put_value` function in the *calltf* call back.

The `t_vpi_stf_partials` data structure is defined:

```
typedef struct t_vpi_stf_partials {
   int count;
   int *derivative_of;  /* 0 = returned value, 1 = 1st arg, etc. */
   int *derivative_wrt; /* 1 = 1st arg, 2 = 2nd arg, etc. */
} s_vpi_stf_partials, *p_vpi_stf_partials;
```

This data structure declares the derivative objects for the associated analog task/function. During the *call_tf* phase, their handles can be retrieved via calls to **`vpi_handl_multi()`**.

12.32.3 Examples

The following example illustrates the declaration and use of callbacks in an analog function `$sampler()` which implements a sample and hold. The task is used as follows:

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
module sampnhold(out, in);
   electrical out, in;
   parameter real period = 1e-3;
   analog begin
     V(out) <+ $sampler(V(in), period);
   end
endmodule
```

The VPI implementation of the sampler is as follows:

```
typedef struct {
   vpiHandle returnHandle; /* Arg #0 (returned value)           */
   vpiHandle exprHandle;   /* Arg #1 (sampled expression)       */
   double period;          /* Arg #2 (static period expression) */
   s_cb_data cb_data;     /* callback structure                 */
   s_vpi_value value;
      /* value structure (holds the expression value)  */
} s_sampler_data, *p_sampler_data;

/* Forward declarations */
static int sampler_callback(p_cb_data data);
static void schedule_callback(p_sampler_data sampler, double currTime);

/* compiletf() */
static int sampler_compiletf(p_cb_data task_cb_data) {
   vpiHandle functionHandle, returnHandle, exprHandle, periodHandle;
   s_cb_data cb_data;
   int type;
   p_sampler_data sampler;
   s_vpi_value value;

   /* Retrieve handle to current function */
   functionHandle = vpi_handle(vpiSysTfCall, NULL);

   /* Get the handle on the expression argument*/
   exprHandle = vpi_handle_by_index(functionHandle, 1);
   /* Check that expression argument exists */
   if (!exprHandle) {
      vpi_error("Not enough arguments for $sampler function.");
   }

   /* Check that expression argument is of real value */
   type = vpi_get(vpiType, exprHandle);
   if (type != vpiRealVal && type != vpiRealVar) {
      vpi_error("Arg #1 of $sampler should be real valued.");
      return 1;
   }

   /* Get the handle on the period argument */
   periodHandle = vpi_handle_by_index(functionHandle, 2);

   /* Check that period argument exists */
   if (!periodHandle) {
      vpi_error("Not enough arguments for $sampler function.");
   }

   /* Check that period argument has a real value */
   type = vpi_get(vpiType, periodHandle);
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   if (type != vpiRealVal && type != vpiRealVar) {
      vpi_error("Arg #2 of $sampler should be real valued");
      return 1;
   }

   /* Schedule callback for time = 0  */
   sampler->cb_data.reason = cbEndOfCompile;
   sampler->cb_data.cb_rtn = sampler_postcompile_cb;
   sampler->cb_data.time.type = 0;
   sampler->cb_data.user_data = (char *) functionHandle;
   sampler->cb_data.time.real = 0.0;
   schedule_callback(sampler, 0.0);

   vpi_register_cb(&sampler->cb_data);

   return 0;
}

/* calltf */
static int sampler_calltf(int data, int reason) {
   vpiHandle funcHandle;
   p_sampler_data sampler = (p_sampler_data) data;
   s_vpi_value value;

   /* Retrieve handle to current function */
   funcHandle = vpi_handle(vpiSysTfCall, NULL);

   /* Set returned value to held value */
   vpi_set_value(sampler->returnHandle, &sampler->value, NULL,
      vpiNoDelay);
   return 0;
}

/* initialization callback after compile  */
static int sampler_postcompile_cb(p_cb_data data) {
   vpiHandle functionHandle = (vpiHandle) data;
   p_sampler_data sampler;
   s_vpi_value value;

   /* Allocate the instance data and initialize it */
   sampler = (p_sampler_data)malloc(sizeof(s_sampler_data));

   /*Get the handle to the returned value, no need to check that one */
   sampler->returnHandle = vpi_handle_by_index(functionHandle, 0);
   sampler->exprHandle = vpi_handle_by_index(functionHandle, 1);
   sampler->periodHandle = vpi_handle_by_index(functionHandle, 2);

   /* Get the period value, it is assumed to be constant */
   /* (but not necessary) */
   sampler->value.format = vpiRealVal;
   vpi_get_value(periodHandle, &value);
   sampler->period = value.value.real;

   /* Schedule callback for time = period  */
   sampler->cb_data.reason = acbElapsedTime;
   sampler->cb_data.cb_rtn = sampler_update_cb;
   sampler->cb_data.time.type = vpiScaledTme;
   sampler->cb_data.user_data = (char *) sampler;
   sampler->cb_data.time.real = sampler->period;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   schedule_callback(sampler, 0.0);

   vpi_register_cb(&sampler->cb_data);

   return 0;
}

/* timer callback */
static int sampler_update_cb(p_cb_data data) {
   p_sampler_data sampler = (p_sampler_data)data->user_data;
   s_vpi_value value;

   /* Hold expression value */
   vpi_get_value(sampler->exprHandle, &value);

   /* Schedule next callback */
   sampler->cb_data.reason = acbAbsTime;
   sampler->cb_data.cb_rtn = sampler_update_cb;
   sampler->cb_data.time.type = vpiScaledTime;
   sampler->cb_data.user_data = (char *) sampler;
   sampler->cb_data.time.real =
      vpi_get_analog_time() + sampler->period;
   register_callback(&sampler->cb_data);
   return 0;
}

/*
 * Public structure declaring the function
*/
static s_vpi_systf_data sampler_systf = {
   vpiSysFunc,          /* type: function / function */
   vpiRealFunc,         /* returned type             */
   "$sampler",          /* name                      */
   sampler_calltf,      /* calltf callback           */
   sampler_compiletf,   /* compiletf callback        */
   0,                   /* unused: sizetf callback   */
   0,                   /* unused: derivtf callback  */
   0                    /* user_data: nothing        */
};
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.33 vpi_register_systf()

vpi_register_systf()			
Synopsis:	Register user-defined system task/function-related callbacks.		
Syntax:	vpi_register_systf(systf_data_p)		
Returns:	**Type**	Description	
	vpiHandle	Handle to the callback object	
Arguments:	**Type**	**Name**	**Description**
	p_vpi_systf_data	systf_data_p	Pointer to a structure with data about when callbacks should occur and the data to be passed
Related routines:	Use vpi_register_analog_systf() to register analog system task/functions. Use vpi_register_cb() to register callbacks for simulation-related events		

The VPI routine **`vpi_register_systf()`** shall register callbacks for user-defined system tasks or functions. Callbacks can be registered to occur when a user-defined system task or function is encountered during compilation or execution of Verilog-AMS HDL source code.

The *systf_data_p* argument shall point to a `s_vpi_systf_data` structure, which is defined in `vpi_user.h` and listed in Figure 12-19.

```
typedef struct t_vpi_systf_data {
   int type;         /* vpiSys[Task,TaskA,Function,FunctionA] */
   int sysfunctype;  /* vpi[IntFunc,RealFunc,TimeFunc,SizedFunc] */
   char *tfname;     /* first character shall be "$" */
   int (*calltf)();
   int (*compiletf)();
   int (*sizetf)();  /* for vpiSizedFunc system functions only */
   char *user_data;
} s_vpi_systf_data, *p_vpi_systf_data;
```

Figure 12-19: The s_vpi_systf_data structure definition

12.33.1 System task and function callbacks

User-defined Verilog-AMS system tasks and functions which use VPI routines can be registered with **`vpi_register_systf()`**. The following system task/function-related callbacks are defined.

The *type* field of the `s_vpi_systf_data` structure shall register the user application to be a system task or a system function. The *type* field value shall be an integer constant of **`vpiSysTask`** or **`vpiSysFunction`**. **`vpiSysTask`** shall register a task with the digital domain. **`vpiSysFunction`** shall register a function with the digital domain.

The *sysfunctype* field of the `s_vpi_systf_data` structure shall define the type of value the system function returns. The *sysfunctype* field shall be an integer constant of **`vpiIntFunc`**, **`vpiRealFunc`**, **`vpiTimeFunc`**, or **`vpiSizedFunc`**. This field shall only be used when the *type* field is set to **`vpiSysFunction`**.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The *compiletf*, *calltf*, and *sizetf* fields of the `s_vpi_systf_data` structure shall be pointers to the user-provided applications which are to be invoked by the system task/function callback mechanism. One or more of the *compiletf*, *calltf*, and *sizetf* fields can be set to `NULL` if they are not needed. Callbacks to the applications pointed to by the *compiletf* and *sizetf* fields shall occur when the simulation data structure is compiled or built (or for the first invocation if the system task or function is invoked from an interactive mode). Callbacks to the application pointed to by the *calltf* routine shall occur each time the system task or function is invoked during simulation execution.

The *sizetf* application shall only called if the PLI application type is **`vpiSysFunction`** and the *sysfunctype* is **`vpiSizedFunc`**. If no *sizetf* is provided, a user-defined system function of **`vpiSizedFunc`** shall return 32-bits.

The *user_data* field of the `s_vpi_systf_data` structure shall specify a user-defined value, which shall be passed back to the *compiletf*, *sizetf*, and *calltf* applications when a callback occurs.

The following example application demonstrates dynamic linking of a VPI system task. The example uses an imaginary routine, `dlink()`, which accepts a file name and a function name and then links that function dynamically. This routine derives the target file and function names from the target *systf* name.

```
link_systf(target)
char *target;
{
   char task_name[strSize];
   char file_name[strSize];
   char compiletf_name[strSize];
   char calltf_name[strSize];
   static s_vpi_systf_data task_data_s = {vpiSysTask};
   static p_vpi_systf_data task_data_p = &task_data_s;

   sprintf(task_name, "$%s", target);
   sprintf(file_name, "%s.o", target);
   sprintf(compiletf_name, "%s_compiletf", target);
   sprintf(calltf_name, "%s_calltf", target);

   task_data_p->tfname = task_name;
   task_data_p->compiletf = (int (*)()) dlink(file_name, compiletf_name);
   task_data_p->calltf = (int (*)()) dlink(file_name, calltf_name);
   vpi_register_systf(task_data_p);
}
```

12.33.2 Initializing VPI system task/function callbacks

A means of initializing system task/function callbacks and performing any other desired task just after the simulator is invoked shall be provided by placing routines in a `NULL`-terminated static array, **`vlog_startup_routines`**. A C function using the array definition shall be provided as follows:

```
void (*vlog_startup_routines[]) ();
```

This C function shall be provided with a VPI-compliant product. Entries in the array shall be added by the user. The location of **`vlog_startup_routines`** and the procedure for linking **`vlog_startup_routines`** with a software product shall be defined by the product vendor. (Callbacks can also be registered or removed at any time during an application routine, not just at startup time).

This array of C functions shall be for registering system tasks and functions. User tasks and functions which appear in a compiled description shall generally be registered by a routine in this array.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

The following example uses **vlog_startup_routines** to register system tasks and functions and to run a user initialization routine.

```
/*In a vendor product file which contains vlog_startup_routines ...*/
extern void register_my_systfs();
extern void my_init();
void (*vlog_startup_routines[])() =
{
   setup_report_cpu,/* user routine example in 23.24.3 */
   register_my_systfs,  /* user routine listed below */
   0                    /* shall be last entry in list */
}

/* In a user provided file... */
void register_my_systfs()
{
   static s_vpi_systf_data systf_data_list[] = {
      {vpiSysTask, 0 "$my_task", my_task_calltf, my_task_compiletf},
      {vpiSysFunc, vpiIntFunc,"$my_func", my_func_calltf,
         my_func_compiletf},
      {vpiSysFunc, vpiRealFunc, "$my_real_func", my_rfunc_calltf,
         my_rfunc_compiletf},
      {0}
   };

   p_vpi_systf_data systf_data_p = &amp;(systf_data_list[0]);
   while (systf_data_p-&gt;type)
      vpi_register_systf(systf_data_p++);
}
```

12.34 vpi_remove_cb()

vpi_remove_cb()			
Synopsis:	Remove a simulation callback registered with vpi_register_cb().		
Syntax:	vpi_remove_cb(cb_obj)		
	Type	Description	
Returns:	bool	**1** (true) if successful; **0** (false) on a failure	
	Type	**Name**	**Description**
Arguments:	vpiHandle	cb_obj	Handle to the callback object
Related routines:	Use vpi_register_cb() to register callbacks for simulation-related events		

The VPI routine **vpi_remove_cb()** shall remove callbacks which were registered with **vpi_register_cb()**. The argument to this routine shall be a handle to the callback object. The routine shall return a `1` (`TRUE`) if successful, and a `0` (`FALSE`) on a failure. After **vpi_remove_cb()** is called with a handle to the callback, the handle is no longer valid.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.35 vpi_scan()

vpi_scan()			
Synopsis:	Scan the Verilog-AMS HDL hierarchy for objects with a one-to-many relationship.		
Syntax:	vpi_scan(itr)		
Returns:	**Type**	Description	
	vpiHandle	Handle to an object	
Arguments:	**Type**	**Name**	**Description**
	vpiHandle	itr	Handle to an iterator object returned from vpi_iterate()
Related routines:	Use vpi_iterate() to obtain an iterator handle Use vpi_handle() to obtain handles to an object with a one-to-one relationship Use vpi_handle_multi() to obtain a handle to an object with a many-to-one relationship		

The VPI routine **`vpi_scan()`** shall traverse the instantiated Verilog-AMS HDL hierarchy and return handles to objects as directed by the iterator *itr*. The iterator handle shall be obtained by calling **`vpi_iterate()`** for a specific object type. Once **`vpi_scan()`** returns `NULL`, the iterator handle is no longer valid and can not be used again.

The following example application uses **`vpi_iterate()`** and **`vpi_scan()`** to display each net (including the size for vectors) declared in the module. The example assumes it shall be passed a valid module handle.

```
void display_nets(mod)
vpiHandle mod;
{
   vpiHandle net;
   vpiHandle itr;

   vpi_printf("Nets declared in module %s\n",
   vpi_get_str(vpiFullName, mod));

   itr = vpi_iterate(vpiNet, mod);
   while (net = vpi_scan(itr))
   {
      vpi_printf("\t%s", vpi_get_str(vpiName, net));
      if (vpi_get(vpiVector, net))
      {
         vpi_printf(" of size %d\n", vpi_get(vpiSize, net));
      }
      else vpi_printf("\n");
   }
}
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

12.36 vpi_sim_control()

vpi_sim_control()			
Synopsis:	Provide software-specific simulation control.		
Syntax:	vpi_sim_control(flag, ...)		
	Type	Description	
Returns:	bool	**1** (true) if successful; **0** (false) on a failure	
	Type	**Name**	**Description**
Arguments:	int	flag	Descriptor of the simulation control request
	var args	...	Variable number and type of arguments depending on flag
Related routines:	NONE		

The VPI routine *vpi_sim_control* shall be used to pass information from user code to Verilog simulator. All standard compliant simulators must support the following three operations:

`vpiStop` — cause **`$stop`** built-in Verilog system task to be executed upon return of user function. This operation shall be passed one additional diagnostic message level integer argument that is the same as the argument passed to **`$stop`** (see 9.7.2).

`vpiFinish` — cause **`$finish`** built-in Verilog system task to be executed upon return of user function. This operation shall be passed one additional diagnostic message level integer argument that is the same as the argument passed to **`$finish`** (see 9.7.1).

`vpiReset` — cause **`$reset`** informative built-in Verilog system task to be executed upon return of user VPI function. This operation shall be passed three integer value arguments: `stop_value`, `reset_value`, `diagnostic_level` that are the same values passed to the **`$reset`** system task (see F.7 of IEEE Std 1364 Verilog).

`vpiSetInteractiveScope` — cause interactive scope to be immediately changed to new scope. This operation shall be passed one argument that is a `vpiHandle` object with type `vpiScope`.

`vpiRejectTransientStep` — cause the current analog simulation time point to be rejected. This operation shall pass one argument which is the current timestep (delta).

`vpiTransientFailConverge` — cause the current analog simulation to continue iterating for a (valid) solution.

Because there may be a need for user VPI applications to pass simulator specific information from back from a user application to control simulation, additional operators and operation specific arguments may be defined.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Annex A

(normative)

Formal syntax definition

The formal syntax of Verilog-AMS HDL is described using Backus-Naur Form (BNF). The syntax of Verilog-AMS HDL source is derived from the starting symbol source_text. The syntax of a library map file is derived from the starting symbol library_text. The following grammar is designed as a presentation grammar and should not be interpreted as an unambiguous production grammar. The compiler developer will be required to implement the various semantic restrictions outlined throughout this reference manual to remove any ambiguities.

A.1 Source text

A.1.1 Library source text

library_text ::= { library_description }

library_description ::=
    library_declaration
  | include_statement
  | config_declaration

library_declaration ::=
    library library_identifier file_path_spec [{ , file_path_spec }]
    [**-incdir** file_path_spec { , file_path_spec }] ;

file_path_spec ::= file_path

include_statement ::= **include** file_path_spec ;

A.1.2 Verilog source text

source_text ::= { description }

description ::=
    module_declaration
  | udp_declaration
  | config_declaration
  | paramset_declaration
  | nature_declaration
  | discipline_declaration
  | connectrules_declaration

module_declaration ::=
    { attribute_instance } module_keyword module_identifier [module_parameter_port_list]
      list_of_ports ; { module_item }
    endmodule
  | { attribute_instance } module_keyword module_identifier [module_parameter_port_list]
      [list_of_port_declarations] ; { non_port_module_item }
    endmodule

module_keyword ::= **module** | **macromodule** | **connectmodule**

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

A.1.3 Module parameters and ports

```
module_parameter_port_list ::= # ( parameter_declaration { , parameter_declaration } )
list_of_ports ::= ( port { , port } )
list_of_port_declarations ::=
        ( port_declaration { , port_declaration } )
port ::=
        [ port_expression ]
      | . port_identifier ( [ port_expression ] )
port_expression ::=
        port_reference
      | { port_reference { , port_reference } }
port_reference ::=
        port_identifier [ [ constant_range_expression ] ]
port_declaration ::=
        {attribute_instance} inout_declaration
      | {attribute_instance} input_declaration
      | {attribute_instance} output_declaration
```

A.1.4 Module items

```
module_item ::=
        port_declaration ;
      | non_port_module_item
module_or_generate_item ::=
        { attribute_instance } module_or_generate_item_declaration
      | { attribute_instance } local_parameter_declaration ;
      | { attribute_instance } parameter_override
      | { attribute_instance } continuous_assign
      | { attribute_instance } gate_instantiation
      | { attribute_instance } udp_instantiation
      | { attribute_instance } module_instantiation
      | { attribute_instance } initial_construct
      | { attribute_instance } always_construct
      | { attribute_instance } loop_generate_construct
      | { attribute_instance } conditional_generate_construct
      | { attribute_instance } analog_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
      | branch_declaration
      | analog_function_declaration
non_port_module_item ::=
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
        module_or_generate_item
      | generate_region
      | specify_block
      | { attribute_instance } parameter_declaration ;
      | { attribute_instance } specparam_declaration
      | aliasparam_declaration
parameter_override ::= defparam list_of_defparam_assignments ;
```

A.1.5 Configuration source text

```
config_declaration ::=
        config config_identifier ;
            design_statement
            {config_rule_statement}
        endconfig
design_statement ::= design { [library_identifier.]cell_identifier } ;
config_rule_statement ::=
        default_clause liblist_clause ;
      | inst_clause liblist_clause ;
      | inst_clause use_clause ;
      | cell_clause liblist_clause ;
      | cell_clause use_clause ;
default_clause ::= default
inst_clause ::= instance inst_name
inst_name ::= topmodule_identifier{.instance_identifier}
cell_clause ::= cell [ library_identifier.]cell_identifier
liblist_clause ::= liblist { library_identifier }
use_clause ::= use [library_identifier.]cell_identifier[:config]
```

A.1.6 Nature Declaration

```
nature_declaration ::=
        nature nature_identifier [ : parent_nature ] [ ; ]
            { nature_item }
        endnature
parent_nature ::=
        nature_identifier
      | discipline_identifier . potential_or_flow
nature_item ::= nature_attribute
nature_attribute ::= nature_attribute_identifier = nature_attribute_expression ;
```

A.1.7 Discipline Declaration

```
discipline_declaration ::=
        discipline discipline_identifier [ ; ]
            { discipline_item }
        enddiscipline
discipline_item ::=
        nature_binding
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
        | discipline_domain_binding
        | nature_attribute_override
nature_binding ::= potential_or_flow nature_identifier ;
potential_or_flow ::= potential | flow
discipline_domain_binding ::= domain discrete_or_continuous ;
discrete_or_continuous ::= discrete | continuous
nature_attribute_override ::= potential_or_flow . nature_attribute
```

A.1.8 Connectrules Declaration

```
connectrules_declaration ::=
        connectrules connectrules_identifier ;
            { connectrules_item }
        endconnectrules
connectrules_item ::=
        connect_insertion
      | connect_resolution
connect_insertion ::= connect connectmodule_identifier [ connect_mode ]
            [ parameter_value_assignment ] [ connect_port_overrides ] ;
connect_mode ::= merged | split
connect_port_overrides ::=
        discipline_identifier , discipline_identifier
      | input discipline_identifier , output discipline_identifier
      | output discipline_identifier , input discipline_identifier
      | inout discipline_identifier , inout discipline_identifier
connect_resolution ::= connect discipline_identifier { , discipline_identifier } resolveto
            discipline_identifier_or_exclude ;
discipline_identifier_or_exclude ::=
        discipline_identifier
      | exclude
```

A.1.9 Paramset Declaration

```
paramset_declaration ::=
        { attribute_instance } paramset paramset_identifier module_or_paramset_identifier ;
            paramset_item_declaration { paramset_item_declaration }
            paramset_statement { paramset_statement }
        endparamset
paramset_item_declaration ::=
        { attribute_instance } parameter_declaration ;
      | { attribute_instance } local_parameter_declaration ;
      | aliasparam_declaration
      | { attribute_instance } integer_declaration
      | { attribute_instance } real_declaration
paramset_statement ::=
        .module_parameter_identifier = paramset_constant_expression ;
      | .module_output_variable_identifier = paramset_constant_expression;
      | .system_parameter_identifier = paramset_constant_expression;
      | analog_function_statement
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

paramset_constant_expression ::=
    constant_primary
  | hierarchical_parameter_identifier
  | unary_operator { attribute_instance } constant_primary
  | paramset_constant_expression binary_operator { attribute_instance } paramset_constant_expression
  | paramset_constant_expression **?** { attribute_instance } paramset_constant_expression **:** { attribute_instance } paramset_constant_expression

A.2 Declarations

A.2.1 Declaration types

A.2.1.1 Module parameter declarations

local_parameter_declaration ::=
    localparam [**signed**] [range] list_of_param_assignments
  | **localparam** parameter_type list_of_param_assignments

parameter_declaration ::=
    parameter [**signed**] [range] list_of_param_assignments
  | **parameter** parameter_type list_of_param_assignments

specparam_declaration ::= **specparam** [range] list_of_specparam_assignments **;**

parameter_type ::=
    integer | **real** | **realtime** | **time** | **string**

aliasparam_declaration ::= **aliasparam** parameter_identifier **=** parameter_identifier **;**

A.2.1.2 Port declarations

inout_declaration ::=
    inout [discipline_identifier] [net_type | **wreal**] [**signed**] [range] list_of_port_identifiers

input_declaration ::=
    input [discipline_identifier] [net_type | **wreal**] [**signed**] [range] list_of_port_identifiers

output_declaration ::=
    output [discipline_identifier] [net_type | **wreal**] [**signed**] [range] list_of_port_identifiers
  | **output** [discipline_identifier] **reg** [**signed**] [range] list_of_variable_port_identifiers
  | **output** output_variable_type list_of_variable_port_identifiers

A.2.1.3 Type declarations

branch_declaration ::=
    branch **(** branch_terminal [**,** branch_terminal] **)** list_of_branch_identifiers **;**
  | port_branch_declaration

port_branch_declaration ::=
    branch **(** **<** port_identifier **>** **)** list_of_branch_identifiers **;**
  | **branch** **(** **<** hierarchical_port_identifier **>** **)** list_of_branch_identifiers **;**

branch_terminal ::=
    net_identifier
  | net_identifier **[** constant_expression **]**
  | net_identifier **[** constant_range_expression **]**
  | hierarchical_net_identifier
  | hierarchical_net_identifier **[** constant_expression **]**
  | hierarchical_net_identifier **[** constant_range_expression **]**

event_declaration ::= **event** list_of_event_identifiers **;**

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

integer_declaration ::= **integer** list_of_variable_identifiers **;**

net_declaration ::=
net_type [discipline_identifier] [**signed**]
[delay3] list_of_net_identifiers **;**
| net_type [discipline_identifier] [drive_strength] [**signed**]
[delay3] list_of_net_decl_assignments **;**
| net_type [discipline_identifier] [**vectored** | **scalared**] [**signed**]
range [delay3] list_of_net_identifiers **;**
| net_type [discipline_identifier] [drive_strength] [**vectored** | **scalared**] [**signed**]
range [delay3] list_of_net_decl_assignments **;**
| **trireg** [discipline_identifier] [charge_strength] [**signed**]
[delay3] list_of_net_identifiers **;**
| **trireg** [discipline_identifier] [drive_strength] [**signed**]
[delay3] list_of_net_decl_assignments **;**
| **trireg** [discipline_identifier] [charge_strength] [**vectored** | **scalared**] [**signed**]
range [delay3] list_of_net_identifiers **;**
| **trireg** [discipline_identifier] [drive_strength] [**vectored** | **scalared**] [**signed**]
range [delay3] list_of_net_decl_assignments **;**
| discipline_identifier [range] list_of_net_identifiers **;**
| discipline_identifier [range] list_of_net_decl_assignments **;**
| **wreal** [discipline_identifier] [range] list_of_net_identifiers **;**
| **wreal** [discipline_identifier] [range] list_of_net_decl_assignments **;**
| **ground** [discipline_identifier] [range] list_of_net_identifiers **;**

real_declaration ::= **real** list_of_real_identifiers **;**

realtime_declaration ::= **realtime** list_of_real_identifiers **;**

reg_declaration ::= **reg** [discipline_identifier] [**signed**] [range]
list_of_variable_identifiers **;**

time_declaration ::= **time** list_of_variable_identifiers **;**

A.2.2 Declaration data types

A.2.2.1 Net and variable types

net_type ::=
supply0 | **supply1** |**tri** | **triand** | **trior** | **tri0** | **tri1** | **uwire** | **wire** | **wand** | **wor**

output_variable_type ::= **integer** | **time**

real_type ::=
real_identifier { dimension } [**=** constant_assignment_pattern]
| real_identifier **=** constant_expression

variable_type ::=
variable_identifier { dimension } [**=** constant_assignment_pattern]
| variable_identifier **=** constant_expression

A.2.2.2 Strengths

drive_strength ::=
(strength0 **,** strength1 **)**
| **(** strength1 **,** strength0 **)**
| **(** strength0 **,** **highz1** **)**
| **(** strength1 **,** **highz0** **)**
| **(** **highz0** **,** strength1 **)**
| **(** **highz1** **,** strength0 **)**

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

strength0 ::= **supply0** | **strong0** | **pull0** | **weak0**

strength1 ::= **supply1** | **strong1** | **pull1** | **weak1**

charge_strength ::= **(** **small** **)** | **(** **medium** **)** | **(** **large** **)**

A.2.2.3 Delays

delay3 ::=
  # delay_value
| **#** **(** mintypmax_expression [**,** mintypmax_expression [**,** mintypmax_expression]] **)**

delay2 ::=
  # delay_value
| **#** **(** mintypmax_expression [**,** mintypmax_expression] **)**

delay_value ::=
  unsigned_number
| real_number
| identifier

A.2.3 Declaration lists

list_of_branch_identifiers ::= branch_identifier [range] { **,** branch_identifier [range] }

list_of_defparam_assignments ::= defparam_assignment { **,** defparam_assignment }

list_of_event_identifiers ::= event_identifier { dimension } { **,** event_identifier { dimension } }

list_of_net_decl_assignments ::= net_decl_assignment { **,** net_decl_assignment }

list_of_net_identifiers ::= ams_net_identifier { **,** ams_net_identifier }

list_of_param_assignments ::= param_assignment { **,** param_assignment }

list_of_port_identifiers ::= port_identifier { **,** port_identifier }

list_of_real_identifiers ::= real_type { **,** real_type }

list_of_specparam_assignments ::= specparam_assignment { **,** specparam_assignment }

list_of_variable_identifiers ::= variable_type { **,** variable_type }

list_of_variable_port_identifiers ::= port_identifier [**=** constant_expression]
  { **,** port_identifier [**=** constant_expression] }

A.2.4 Declaration assignments

defparam_assignment ::= hierarchical_parameter_identifier **=** constant_mintypmax_expression

net_decl_assignment ::= ams_net_identifier **=** expression

param_assignment ::=
  parameter_identifier **=** constant_mintypmax_expression { value_range }
| parameter_identifier range **=** constant_assignment_pattern { value_range }

specparam_assignment ::=
  specparam_identifier **=** constant_mintypmax_expression
| pulse_control_specparam

pulse_control_specparam ::=
  PATHPULSE$ **=** **(** reject_limit_value [**,** error_limit_value] **)**
| **PATHPULSE$**specify_input_terminal_descriptor**$**specify_output_terminal_descriptor
  = **(** reject_limit_value [**,** error_limit_value] **)**

error_limit_value ::= limit_value

reject_limit_value ::= limit_value

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

limit_value ::= constant_mintypmax_expression

A.2.5 Declaration ranges

dimension ::= [dimension_constant_expression : dimension_constant_expression]

range ::= [msb_constant_expression : lsb_constant_expression]

value_range ::=
    value_range_type (value_range_expression : value_range_expression)
  | value_range_type (value_range_expression : value_range_expression]
  | value_range_type [value_range_expression : value_range_expression)
  | value_range_type [value_range_expression : value_range_expression]
  | value_range_type '{ string { , string } }
  | **exclude** constant_expression

value_range_type :: = **from** | **exclude**

value_range_expression ::= constant_expression | **-inf** | **inf**

A.2.6 Function declarations

analog_function_declaration ::=
    analog function [analog_function_type] analog_function_identifier ;
      analog_function_item_declaration { analog_function_item_declaration }
      analog_function_statement
    endfunction

analog_function_type ::= **integer** | **real**| **string**

analog_function_item_declaration ::=
    analog_block_item_declaration
  | input_declaration ;
  | output_declaration ;
  | inout_declaration ;

function_declaration ::=
    function [**automatic**] [function_range_or_type] function_identifier ;
      function_item_declaration { function_item_declaration }
      function_statement
    endfunction
  | **function** [**automatic**] [function_range_or_type] function_identifier (function_port_list) ;
      { block_item_declaration }
      function_statement
    endfunction

function_item_declaration ::=
    block_item_declaration
  | { attribute_instance } tf_input_declaration ;

function_port_list ::=
    { attribute_instance } tf_input_declaration { , { attribute_instance } tf_input_declaration }

function_range_or_type ::=
    [**signed**] [range]
  | **integer**
  | **real**
  | **realtime**
  | **time**

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

A.2.7 Task declarations

task_declaration ::=
    task [**automatic**] task_identifier **;**
        { task_item_declaration }
        statement_or_null
    endtask
  | **task** [**automatic**] task_identifier **(** [task_port_list] **)** **;**
        { block_item_declaration }
        statement_or_null
    endtask

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

task_port_list ::= task_port_item { **,** task_port_item }

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

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

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

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

task_port_type ::=
    integer | **real** | **realtime** | **time**

A.2.8 Block item declarations

analog_block_item_declaration ::=
    { attribute_instance } parameter_declaration **;**
  | { attribute_instance } integer_declaration
  | { attribute_instance } real_declaration
  | { attribute_instance } string_declaration

block_item_declaration ::=
    { attribute_instance } **reg** [discipline_identifier] [**signed**] [range]
        list_of_block_variable_identifiers **;**
  | { attribute_instance } **integer** list_of_block_variable_identifiers **;**
  | { attribute_instance } **time** list_of_block_variable_identifiers **;**
  | { attribute_instance } **real** list_of_block_real_identifiers **;**
  | { attribute_instance } **realtime** list_of_block_real_identifiers **;**
  | { attribute_instance } event_declaration
  | { attribute_instance } local_parameter_declaration **;**
  | { attribute_instance } parameter_declaration **;**

list_of_block_variable_identifiers ::= block_variable_type { **,** block_variable_type }

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

list_of_block_real_identifiers ::= block_real_type { , block_real_type }

block_variable_type ::= variable_identifier { dimension }

block_real_type ::= real_identifier { dimension }

A.3 Primitive instances

A.3.1 Primitive instantiation and instances

gate_instantiation ::=
    cmos_switchtype [delay3] cmos_switch_instance { , cmos_switch_instance } ;
  | enable_gatetype [drive_strength] [delay3] enable_gate_instance { , enable_gate_instance } ;
  | mos_switchtype [delay3] mos_switch_instance { , mos_switch_instance } ;
  | n_input_gatetype [drive_strength] [delay2] n_input_gate_instance { , n_input_gate_instance } ;
  | n_output_gatetype [drive_strength] [delay2] n_output_gate_instance { , n_output_gate_instance } ;
  | pass_en_switchtype [delay2] pass_enable_switch_instance { , pass_enable_switch_instance } ;
  | pass_switchtype pass_switch_instance { , pass_switch_instance } ;
  | **pulldown** [pulldown_strength] pull_gate_instance { , pull_gate_instance } ;
  | **pullup** [pullup_strength] pull_gate_instance { , pull_gate_instance } ;

cmos_switch_instance ::= [name_of_gate_instance] (output_terminal , input_terminal ,
    ncontrol_terminal , pcontrol_terminal)

enable_gate_instance ::= [name_of_gate_instance] (output_terminal , input_terminal , enable_terminal)

mos_switch_instance ::= [name_of_gate_instance] (output_terminal , input_terminal , enable_terminal)

n_input_gate_instance ::= [name_of_gate_instance] (output_terminal , input_terminal
    { , input_terminal })

n_output_gate_instance ::= [name_of_gate_instance] (output_terminal { , output_terminal } ,
    input_terminal)

pass_switch_instance ::= [name_of_gate_instance] (inout_terminal , inout_terminal)

pass_enable_switch_instance ::= [name_of_gate_instance] (inout_terminal , inout_terminal ,
    enable_terminal)

pull_gate_instance ::= [name_of_gate_instance] (output_terminal)

name_of_gate_instance ::= gate_instance_identifier [range]

A.3.2 Primitive strengths

pulldown_strength ::=
    (strength0 , strength1)
  | (strength1 , strength0)
  | (strength0)

pullup_strength ::=
    (strength0 , strength1)
  | (strength1 , strength0)
  | (strength1)

A.3.3 Primitive terminals

enable_terminal ::= expression

inout_terminal ::= net_lvalue

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

input_terminal ::= expression
ncontrol_terminal ::= expression
output_terminal ::= net_lvalue
pcontrol_terminal ::= expression

A.3.4 Primitive gate and switch types

cmos_switchtype ::= **cmos** | **rcmos**
enable_gatetype ::= **bufif0** | **bufif1** | **notif0** | **notif1**
mos_switchtype ::= **nmos** | **pmos** | **rnmos** | **rpmos**
n_input_gatetype ::= **and** | **nand** | **or** | **nor** | **xor** | **xnor**
n_output_gatetype ::= **buf** | **not**
pass_en_switchtype ::= **tranif0** | **tranif1** | **rtranif1** | **rtranif0**
pass_switchtype ::= **tran** | **rtran**

A.4 Module instantiation and generate construct

A.4.1 Module instantiation

module_instantiation ::=
    module_or_paramset_identifier [parameter_value_assignment]
        module_instance { , module_instance } ;
parameter_value_assignment ::= # (list_of_parameter_assignments)
list_of_parameter_assignments ::=
    ordered_parameter_assignment { , ordered_parameter_assignment }
  | named_parameter_assignment { , named_parameter_assignment }
ordered_parameter_assignment ::= expression
named_parameter_assignment ::=
    . parameter_identifier ([mintypmax_expression])
  | . system_parameter_identifier ([constant_expression])
module_instance ::= name_of_module_instance ([list_of_port_connections])
name_of_module_instance ::= module_instance_identifier [range]
list_of_port_connections ::=
    ordered_port_connection { , ordered_port_connection }
  | named_port_connection { , named_port_connection }
ordered_port_connection ::= { attribute_instance } [expression]
named_port_connection ::= { attribute_instance } . port_identifier ([expression])

A.4.2 Generate construct

generate_region ::=
    generate { module_or_generate_item } **endgenerate**
genvar_declaration ::=
    genvar list_of_genvar_identifiers ;
list_of_genvar_identifiers ::=
    genvar_identifier { , genvar_identifier }

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
analog_loop_generate_statement ::=
        for ( genvar_initialization ; genvar_expression ; genvar_iteration )
            analog_statement
loop_generate_construct ::=
        for ( genvar_initialization ; genvar_expression ; genvar_iteration )
            generate_block
genvar_initialization ::=
        genvar_identifier = constant_expression
genvar_expression ::=
        genvar_primary
      | unary_operator { attribute_instance } genvar_primary
      | genvar_expression binary_operator { attribute_instance } genvar_expression
      | genvar_expression ? { attribute_instance } genvar_expression : genvar_expression
genvar_iteration ::=
      genvar_identifier = genvar_expression
genvar_primary ::=
        constant_primary
      | genvar_identifier
conditional_generate_construct ::=
        if_generate_construct
      | case_generate_construct
if_generate_construct ::=
        if ( constant_expression ) generate_block_or_null
        [ else generate_block_or_null ]
case_generate_construct ::=
        case ( constant_expression ) case_generate_item { case_generate_item } endcase
case_generate_item ::=
        constant_expression { , constant_expression } : generate_block_or_null
      | default [ : ] generate_block_or_null
generate_block ::=
        module_or_generate_item
      | begin [ : generate_block_identifier ] { module_or_generate_item } end
generate_block_or_null ::=
        generate_block
      | ;
```

A.5 UDP declaration and instantiation

A.5.1 UDP declaration

```
udp_declaration ::=
        { attribute_instance } primitive udp_identifier ( udp_port_list ) ;
            udp_port_declaration { udp_port_declaration }
            udp_body
        endprimitive
      | { attribute_instance } primitive udp_identifier ( udp_declaration_port_list ) ;
            udp_body
        endprimitive
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

A.5.2 UDP ports

udp_port_list ::= output_port_identifier , input_port_identifier { , input_port_identifier }

udp_declaration_port_list ::=
    udp_output_declaration , udp_input_declaration { , udp_input_declaration }

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

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

udp_input_declaration ::= { attribute_instance } **input** list_of_port_identifiers

udp_reg_declaration ::= { attribute_instance } **reg** [discipline_identifier] variable_identifier

A.5.3 UDP body

udp_body ::= combinational_body | sequential_body

combinational_body ::= **table** combinational_entry { combinational_entry } **endtable**

combinational_entry ::= level_input_list : output_symbol ;

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

udp_initial_statement ::= **initial** output_port_identifier = init_val ;

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

sequential_entry ::= seq_input_list : current_state : next_state ;

seq_input_list ::= level_input_list | edge_input_list

level_input_list ::= level_symbol { level_symbol }

edge_input_list ::= { level_symbol } edge_indicator { level_symbol }

edge_indicator ::= (level_symbol level_symbol) | edge_symbol

current_state ::= level_symbol

next_state ::= output_symbol | **-**

output_symbol ::= **0** | **1** | **x** | **X**

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

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

A.5.4 UDP instantiation

udp_instantiation ::= udp_identifier [drive_strength] [delay2] udp_instance { , udp_instance } ;

udp_instance ::= [name_of_udp_instance] (output_terminal , input_terminal { , input_terminal })

name_of_udp_instance ::= udp_instance_identifier [range]

A.6 Behavioral statements

A.6.1 Continuous assignment statements

continuous_assign ::= **assign** [drive_strength] [delay3] list_of_net_assignments ;

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

list_of_net_assignments ::= net_assignment { , net_assignment }
net_assignment ::= net_lvalue = expression

A.6.2 Procedural blocks and assignments

analog_construct ::=
    analog analog_statement
  | analog initial analog_function_statement
analog_procedural_assignment ::= analog_variable_assignment ;
analog_variable_assignment ::=
    scalar_analog_variable_assignment
  | array_analog_variable_assignment
scalar_analog_variable_assignment ::= scalar_analog_variable_lvalue = analog_expression
initial_construct ::= initial statement
always_construct ::= always statement
blocking_assignment ::= variable_lvalue = [delay_or_event_control] expression
nonblocking_assignment ::= variable_lvalue <= [delay_or_event_control] expression
procedural_continuous_assignments ::=
    assign variable_assignment
  | deassign variable_lvalue
  | force variable_assignment
  | force net_assignment
  | release variable_lvalue
  | release net_lvalue
variable_assignment ::= variable_lvalue = expression

A.6.3 Parallel and sequential blocks

analog_seq_block ::= begin [: analog_block_identifier { analog_block_item_declaration }]
    { analog_statement } end
analog_event_seq_block ::=
  begin [: analog_block_identifier { analog_block_item_declaration }]
    { analog_event_statement} end
analog_function_seq_block ::= begin [: analog_block_identifier { analog_block_item_declaration }]
    { analog_function_statement } end
par_block ::= fork [: block_identifier
    { block_item_declaration }] { statement } join
seq_block ::= begin [: block_identifier
    { block_item_declaration }] { statement } end

A.6.4 Statements

analog_statement ::=
    { attribute_instance } analog_loop_generate_statement
  | { attribute_instance } analog_loop_statement
  | { attribute_instance } analog_case_statement
  | { attribute_instance } analog_conditional_statement
  | { attribute_instance } analog_procedural_assignment

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    | { attribute_instance } analog_seq_block
    | { attribute_instance } analog_system_task_enable
    | { attribute_instance } contribution_statement
    | { attribute_instance } indirect_contribution_statement
    | { attribute_instance } analog_event_control_statement
    | { attribute_instance } jump_statement

analog_statement_or_null ::=
      analog_statement
    | { attribute_instance } ;

analog_event_statement ::=
      { attribute_instance } analog_loop_statement
    | { attribute_instance } analog_case_statement
    | { attribute_instance } analog_conditional_statement
    | { attribute_instance } analog_procedural_assignment
    | { attribute_instance } analog_event_seq_block
    | { attribute_instance } analog_system_task_enable
    | { attribute_instance } disable_statement
    | { attribute_instance } event_trigger
    | { attribute_instance } ;

analog_function_statement ::=
      { attribute_instance } analog_function_case_statement
    | { attribute_instance } analog_function_conditional_statement
    | { attribute_instance } analog_function_loop_statement
    | { attribute_instance } analog_function_seq_block
    | { attribute_instance } analog_procedural_assignment
    | { attribute_instance } analog_system_task_enable
    | { attribute_instance } jump_statement

analog_function_statement_or_null ::=
      analog_function_statement
    | { attribute_instance } ;

statement ::=
      { attribute_instance } blocking_assignment ;
    | { attribute_instance } case_statement
    | { attribute_instance } conditional_statement
    | { attribute_instance } disable_statement
    | { attribute_instance } event_trigger
    | { attribute_instance } loop_statement
    | { attribute_instance } jump_statement
    | { attribute_instance } nonblocking_assignment ;
    | { attribute_instance } par_block
    | { attribute_instance } procedural_continuous_assignments ;
    | { attribute_instance } procedural_timing_control_statement
    | { attribute_instance } seq_block
    | { attribute_instance } system_task_enable
    | { attribute_instance } task_enable
    | { attribute_instance } wait_statement

statement_or_null ::=
      statement
    | { attribute_instance } ;

function_statement[1] ::= statement
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

A.6.5 Timing control statements

```
analog_event_control_statement ::= analog_event_control analog_event_statement
analog_event_control ::=
        @ hierarchical_event_identifier
      | @ ( analog_event_expression )
analog_event_expression ::=
        expression
      | posedge expression
      | negedge expression
      | hierarchical_event_identifier
      | initial_step [ ( " analysis_identifier "{ , " analysis_identifier " } ) ]
      | final_step [ ( " analysis_identifier " { , " analysis_identifier " } ) ]
      | analog_event_functions
      | analog_event_expression or analog_event_expression
      | analog_event_expression , analog_event_expression
analog_event_functions ::=
        cross ( analog_expression [ , analog_expression_or_null
            [ , analog_expression_or_null [ , analog_expression_or_null [ , analog_expression ] ] ] ] )
      | above ( analog_expression [ , analog_expression_or_null
            [ , analog_expression_or_null [ , analog_expression ] ] ] )
      | timer ( analog_expression [ , analog_expression_or_null
            [ , analog_expression_or_null [ , analog_expression ] ] ] )
      | absdelta ( analog_expression , analog_expression
            [ , analog_expression_or_null [ , analog_expression_or_null [ , analog_expression ] ] ] )
delay_control ::=
        # delay_value
      | # ( mintypmax_expression )
delay_or_event_control ::=
        delay_control
      | event_control
      | repeat ( expression ) event_control
disable_statement ::=
        disable hierarchical_task_identifier ;
      | disable hierarchical_block_identifier ;
event_control ::=
        @ hierarchical_event_identifier
      | @ ( event_expression )
      | @*
      | @ (*)
event_trigger ::=
        -> hierarchical_event_identifier { [ expression ] } ;
event_expression ::=
        expression
      | posedge expression
      | negedge expression
      | hierarchical_event_identifier
      | event_expression or event_expression
      | event_expression , event_expression
      | analog_event_functions
      | driver_update expression
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    | analog_variable_lvalue
procedural_timing_control ::=
      delay_control
    | event_control
procedural_timing_control_statement ::=
      procedural_timing_control statement_or_null
jump_statement ::=
      return [ expression ] ;
    | break ;
    | continue ;
wait_statement ::=
      wait ( expression ) statement_or_null
```

A.6.6 Conditional statements

```
analog_conditional_statement ::=
      if ( analog_expression ) analog_statement_or_null
      { else if ( analog_expression ) analog_statement_or_null }
      [ else analog_statement_or_null ]
analog_function_conditional_statement ::=
      if ( analog_expression ) analog_function_statement_or_null
      { else if ( analog_expression ) analog_function_statement_or_null }
      [ else analog_function_statement_or_null ]
conditional_statement ::=
      if ( expression )
        statement_or_null
      [ else statement_or_null ]
    | if_else_if_statement
if_else_if_statement ::=
      if ( expression ) statement_or_null
      { else if ( expression ) statement_or_null }
      [ else statement_or_null ]
```

A.6.7 Case statements

```
analog_case_statement ::=
      case ( analog_expression ) analog_case_item { analog_case_item } endcase
    | casex ( analog_expression ) analog_case_item { analog_case_item } endcase
    | casez ( analog_expression ) analog_case_item { analog_case_item } endcase
analog_case_item ::=
      analog_expression { , analog_expression } : analog_statement_or_null
    | default [ : ] analog_statement_or_null
analog_function_case_statement ::=
      case ( analog_expression ) analog_function_case_item {analog_function_case_item} endcase
analog_function_case_item ::=
      analog_expression { , analog_expression } : analog_function_statement_or_null
    | default [ : ] analog_function_statement_or_null
case_statement ::=
      case ( expression ) case_item { case_item } endcase
    | casez ( expression ) case_item { case_item } endcase
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

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

A.6.8 Looping statements

```
analog_loop_statement ::=
      repeat ( analog_expression ) analog_statement
    | while ( analog_expression ) analog_statement
    | for ( analog_variable_assignment ; analog_expression ; analog_variable_assignment )
        analog_statement
analog_function_loop_statement ::=
      repeat ( analog_expression ) analog_function_statement
    | while ( analog_expression ) analog_function_statement
    | for ( analog_variable_assignment ; analog_expression ; analog_variable_assignment )
        analog_function_statement
loop_statement ::=
      forever statement
    | repeat ( expression ) statement
    | while ( expression ) statement
    | for ( variable_assignment ; expression ; variable_assignment ) statement
```

A.6.9 Task enable statements

```
analog_system_task_enable ::=
      analog_system_task_identifier [ ( [ analog_expression ] { , [ analog_expression ] } ) ] ;
system_task_enable ::= system_task_identifier [ ( [ expression ] { , [ expression ] } ) ] ;
task_enable ::= hierarchical_task_identifier [ ( expression { , expression } ) ] ;
```

A.6.10 Contribution statements

```
contribution_statement ::= branch_lvalue <+ analog_expression ;
indirect_contribution_statement ::= branch_lvalue : indirect_expression == analog_expression ;
```

A.7 Specify section

A.7.1 Specify block declaration

```
specify_block ::= specify { specify_item } endspecify
specify_item ::=
      specparam_declaration
    | pulsestyle_declaration
    | showcancelled_declaration
    | path_declaration
    | system_timing_check
pulsestyle_declaration ::=
      pulsestyle_onevent list_of_path_outputs ;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    | pulsestyle_ondetect list_of_path_outputs ;
showcancelled_declaration ::=
      showcancelled list_of_path_outputs ;
    | noshowcancelled list_of_path_outputs ;
```

A.7.2 Specify path declarations

```
path_declaration ::=
      simple_path_declaration ;
    | edge_sensitive_path_declaration ;
    | state_dependent_path_declaration ;
simple_path_declaration ::=
      parallel_path_description = path_delay_value
    | full_path_description = path_delay_value
parallel_path_description ::=
      ( specify_input_terminal_descriptor [ polarity_operator ] => specify_output_terminal_descriptor )
full_path_description ::=
      ( list_of_path_inputs [ polarity_operator ] *> list_of_path_outputs )
list_of_path_inputs ::=
      specify_input_terminal_descriptor { , specify_input_terminal_descriptor }
list_of_path_outputs ::=
      specify_output_terminal_descriptor { , specify_output_terminal_descriptor }
```

A.7.3 Specify block terminals

```
specify_input_terminal_descriptor ::=
      input_identifier [ [ constant_range_expression ] ]
specify_output_terminal_descriptor ::=
      output_identifier [ [ constant_range_expression ] ]
input_identifier ::= input_port_identifier | inout_port_identifier
output_identifier ::= output_port_identifier | inout_port_identifier
```

A.7.4 Specify path delays

```
path_delay_value ::=
      list_of_path_delay_expressions
    | ( list_of_path_delay_expressions )
list_of_path_delay_expressions ::=
      t_path_delay_expression
    | trise_path_delay_expression , tfall_path_delay_expression
    | trise_path_delay_expression , tfall_path_delay_expression , tz_path_delay_expression
    | t01_path_delay_expression , t10_path_delay_expression , t0z_path_delay_expression ,
        tz1_path_delay_expression , t1z_path_delay_expression , tz0_path_delay_expression
    | t01_path_delay_expression , t10_path_delay_expression , t0z_path_delay_expression ,
        tz1_path_delay_expression , t1z_path_delay_expression , tz0_path_delay_expression ,
      t0x_path_delay_expression , tx1_path_delay_expression , t1x_path_delay_expression ,
      tx0_path_delay_expression , txz_path_delay_expression , tzx_path_delay_expression
t_path_delay_expression ::= path_delay_expression
trise_path_delay_expression ::= path_delay_expression
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

tfall_path_delay_expression ::= path_delay_expression

tz_path_delay_expression ::= path_delay_expression

t01_path_delay_expression ::= path_delay_expression

t10_path_delay_expression ::= path_delay_expression

t0z_path_delay_expression ::= path_delay_expression

tz1_path_delay_expression ::= path_delay_expression

t1z_path_delay_expression ::= path_delay_expression

tz0_path_delay_expression ::= path_delay_expression

t0x_path_delay_expression ::= path_delay_expression

tx1_path_delay_expression ::= path_delay_expression

t1x_path_delay_expression ::= path_delay_expression

tx0_path_delay_expression ::= path_delay_expression

txz_path_delay_expression ::= path_delay_expression

tzx_path_delay_expression ::= path_delay_expression

path_delay_expression ::= constant_mintypmax_expression

edge_sensitive_path_declaration ::=
    parallel_edge_sensitive_path_description **=** path_delay_value
  | full_edge_sensitive_path_description **=** path_delay_value

parallel_edge_sensitive_path_description ::=
    ([edge_identifier] specify_input_terminal_descriptor **=>**
      (specify_output_terminal_descriptor [polarity_operator] **:** data_source_expression **))**

full_edge_sensitive_path_description ::=
    ([edge_identifier] list_of_path_inputs ***>**
      (list_of_path_outputs [polarity_operator] **:** data_source_expression **))**

data_source_expression ::= expression

edge_identifier ::= **posedge** | **negedge**

state_dependent_path_declaration ::=
    if (module_path_expression **)** simple_path_declaration
  | **if (** module_path_expression **)** edge_sensitive_path_declaration
  | **ifnone** simple_path_declaration

polarity_operator ::= **+** | **-**

A.7.5 System timing checks

A.7.5.1 System timing check commands

system_timing_check ::=
    $setup_timing_check
  | $hold_timing_check
  | $setuphold_timing_check
  | $recovery_timing_check
  | $removal_timing_check
  | $recrem_timing_check
  | $skew_timing_check
  | $timeskew_timing_check
  | $fullskew_timing_check
  | $period_timing_check
  | $width_timing_check
  | $nochange_timing_check

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
$setup_timing_check ::=
        $setup ( data_event , reference_event , timing_check_limit [ , [ notifier ] ] ) ;
$hold_timing_check ::=
        $hold ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
$setuphold_timing_check ::=
        $setuphold ( reference_event , data_event , timing_check_limit , timing_check_limit
            [ , [ notifier ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
            [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ] ) ;
$recovery_timing_check ::=
        $recovery ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
$removal_timing_check ::=
        $removal ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
$recrem_timing_check ::=
        $recrem ( reference_event , data_event , timing_check_limit , timing_check_limit
            [ , [ notifier ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
            [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ] ) ;
$skew_timing_check ::=
        $skew ( reference_event , data_event , timing_check_limit [ , [ notifier ] ] ) ;
$timeskew_timing_check ::=
        $timeskew ( reference_event , data_event , timing_check_limit
            [ , [ notifier ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] ) ;
$fullskew_timing_check ::=
        $fullskew ( reference_event , data_event , timing_check_limit , timing_check_limit
            [ , [ notifier ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] ) ;
$period_timing_check ::=
        $period ( controlled_reference_event , timing_check_limit [ , [ notifier ] ] ) ;
$width_timing_check ::=
        $width ( controlled_reference_event , timing_check_limit [ , threshold [ , notifier ] ] ) ;
$nochange_timing_check ::=
        $nochange ( reference_event , data_event , start_edge_offset ,
            end_edge_offset [ , [ notifier ] ] ) ;
```

A.7.5.2 System timing check command arguments

```
checktime_condition ::= mintypmax_expression
controlled_reference_event ::= controlled_timing_check_event
data_event ::= timing_check_event
delayed_data ::=
        terminal_identifier
      | terminal_identifier [ constant_mintypmax_expression ]
delayed_reference ::=
        terminal_identifier
      | terminal_identifier [ constant_mintypmax_expression ]
end_edge_offset ::= mintypmax_expression
event_based_flag ::= constant_expression
notifier ::= variable_identifier
reference_event ::= timing_check_event
remain_active_flag ::= constant_expression
stamptime_condition ::= mintypmax_expression
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

start_edge_offset ::= mintypmax_expression

threshold ::= constant_expression

timing_check_limit ::= expression

A.7.5.3 System timing check event definitions

timing_check_event ::=
    [timing_check_event_control] specify_terminal_descriptor [**&&&** timing_check_condition]

controlled_timing_check_event ::=
    timing_check_event_control specify_terminal_descriptor [**&&&** timing_check_condition]

timing_check_event_control ::=
    posedge
  | **negedge**
  | edge_control_specifier

specify_terminal_descriptor ::=
    specify_input_terminal_descriptor
  | specify_output_terminal_descriptor

edge_control_specifier ::= **edge [** edge_descriptor { **,** edge_descriptor } **]**

edge_descriptor[2] ::=
    01
  | **10**
  | z_or_x zero_or_one
  | zero_or_one z_or_x

zero_or_one ::= **0** | **1**

z_or_x ::= **x** | **X** | **z** | **Z**

timing_check_condition ::=
    scalar_timing_check_condition
  | **(** scalar_timing_check_condition **)**

scalar_timing_check_condition ::=
    expression
  | **~** expression
  | expression **==** scalar_constant
  | expression **===** scalar_constant
  | expression **!=** scalar_constant
  | expression **!==** scalar_constant

scalar_constant ::=
    1'b0 | **1'b1** | **1'B0** | **1'B1** | **'b0** | **'b1** | **'B0** | **'B1** | **1** | **0**

A.8 Expressions

A.8.1 Concatenations and assignment patterns

analog_concatenation ::= **{** analog_expression { **,** analog_expression } **}**

analog_multiple_concatenation ::= **{** constant_expression analog_concatenation **}**

concatenation ::= **{** expression { **,** expression } **}**

constant_concatenation ::= **{** constant_expression { **,** constant_expression } **}**

constant_multiple_concatenation ::= **{** constant_expression constant_concatenation **}**

module_path_concatenation ::= **{** module_path_expression { **,** module_path_expression } **}**

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

module_path_multiple_concatenation ::= { constant_expression module_path_concatenation }

multiple_concatenation ::= { constant_expression concatenation }

assignment_pattern ::=
'{ expression { , expression } }
| '{ constant_expression { expression { , expression } } }

constant_assignment_pattern ::=
'{ constant_expression { , constant_expression } }
| '{ constant_expression { constant_expression { , constant_expression } } }

constant_assignment_pattern_or_null ::= [constant_assignment_pattern]

A.8.2 Function calls

analog_function_call ::=
analog_function_identifier { attribute_instance } (analog_expression { , analog_expression })

analog_system_function_call ::=
analog_system_function_identifier [([analog_expression] { , [analog_expression] })]

analog_built_in_function_call ::=
analog_built_in_function_name (analog_expression [, analog_expression])

analog_built_in_function_name ::=
ln | **ln1p** | **log** | **exp** | **expm1** | **sqrt** | **min** | **max** | **abs** | **pow** | **floor** | **ceil**
| **sin** | **cos** | **tan** | **asin** | **acos** | **atan** | **atan2**
| **hypot** | **sinh** | **cosh** | **tanh** | **asinh** | **acosh** | **atanh**

analysis_function_call ::= **analysis** (" analysis_identifier " { , " analysis_identifier " })

analog_filter_function_call ::=
ddt (analog_expression [, abstol_expression])
| **ddx** (analog_expression , branch_probe_function_call)
| **idt** (analog_expression [, analog_expression [, analog_expression [, abstol_expression]]])
| **idtmod** (analog_expression [, analog_expression [, analog_expression [, analog_expression
[, abstol_expression]]]])
| **absdelay** (analog_expression , analog_expression [, constant_expression])
| **transition** (analog_expression [, analog_expression [, analog_expression
[, analog_expression [, constant_expression]]]])
| **slew** (analog_expression [, analog_expression [, analog_expression]])
| **last_crossing** (analog_expression [, analog_expression])
| **limexp** (analog_expression)
| laplace_filter_name (analog_expression , [analog_filter_function_arg] ,
[analog_filter_function_arg] [, constant_expression])
| zi_filter_name (analog_expression , [analog_filter_function_arg] ,
[analog_filter_function_arg] , constant_expression
[, analog_expression [, constant_expression]])

analog_filter_function_arg ::=
parameter_identifier
| parameter_identifier [msb_constant_expression : lsb_constant_expression]
| constant_assignment_pattern_or_null

analog_small_signal_function_call ::=
ac_stim ([" analysis_identifier " [, analog_expression [, analog_expression]]])
| **white_noise** (analog_expression [, string])
| **flicker_noise** (analog_expression , analog_expression [, string])
| **noise_table** (noise_table_input_arg [, string])
| **noise_table_log** (noise_table_input_arg [, string])

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
noise_table_input_arg ::=
        parameter_identifier
      | parameter_identifier [ msb_constant_expression : lsb_constant_expression ]
      | string
      | constant_assignment_pattern
laplace_filter_name ::= laplace_zd | laplace_zp | laplace_np | laplace_nd
zi_filter_name ::= zi_zp | zi_zd | zi_np | zi_nd
nature_access_function ::=
        nature_attribute_identifier
      | potential
      | flow
branch_probe_function_call ::=
        nature_access_function ( branch_reference )
      | nature_access_function ( analog_net_reference [ , analog_net_reference ] )
port_probe_function_call ::= nature_attribute_identifier ( < analog_port_reference > )
constant_analog_function_call ::=
        analog_function_identifier { attribute_instance } ( constant_expression { , constant_expression } )
constant_function_call ::= function_identifier { attribute_instance }
        ( constant_expression { , constant_expression } )
constant_system_function_call ::= system_function_identifier
        ( constant_expression { , constant_expression } )
constant_analog_built_in_function_call ::=
        analog_built_in_function_name ( constant_expression [ , constant_expression ] )
function_call ::= hierarchical_function_identifier{ attribute_instance }
        ( expression { , expression } )
system_function_call ::= system_function_identifier
        [ ( expression { , expression } ) ]
```

A.8.3 Expressions

```
abstol_expression ::=
        constant_expression
      | nature_identifier
analog_conditional_expression ::=
        analog_expression ? { attribute_instance } analog_expression : analog_expression
analog_range_expression ::=
        analog_expression
      | msb_constant_expression : lsb_constant_expression
analog_expression_or_null ::= [ analog_expression ]
analog_expression ::=
        analog_primary
      | unary_operator { attribute_instance } analog_primary
      | analog_expression  binary_operator { attribute_instance } analog_expression
      | analog_conditional_expression
      | string
base_expression ::= expression
conditional_expression ::= expression1 ? { attribute_instance } expression2 : expression3
constant_base_expression ::= constant_expression
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

constant_expression ::=
    constant_primary
  | unary_operator { attribute_instance } constant_primary
  | constant_expression binary_operator { attribute_instance } constant_expression
  | constant_expression ? { attribute_instance } constant_expression : constant_expression

analysis_or_constant_expression ::=
    constant_primary
  | analysis_function_call
  | unary_operator { attribute_instance } analysis_or_constant_expression
  | analysis_or_constant_expression binary_operator { attribute_instance }
      analysis_or_constant_expression
  | analysis_or_constant_expression ? { attribute_instance } analysis_or_constant_expression :
      analysis_or_constant_expression

constant_mintypmax_expression ::=
    constant_expression
  | constant_expression : constant_expression : constant_expression

constant_range_expression ::=
    constant_expression
  | msb_constant_expression : lsb_constant_expression
  | constant_base_expression +: width_constant_expression
  | constant_base_expression -: width_constant_expression

dimension_constant_expression ::= constant_expression

expression ::=
    primary
  | unary_operator { attribute_instance } primary
  | expression binary_operator { attribute_instance } expression
  | conditional_expression

expression1 ::= expression

expression2 ::= expression

expression3 ::= expression

indirect_expression ::=
    branch_probe_function_call
  | port_probe_function_call
  | ddt (branch_probe_function_call [, abstol_expression])
  | ddt (port_probe_function_call [, abstol_expression])
  | idt (branch_probe_function_call [, analog_expression
      [, analog_expression [, abstol_expression]]])
  | idt (port_probe_function_call [, analog_expression [, analog_expression
      [, abstol_expression]]])
  | idtmod (branch_probe_function_call [, analog_expression [, analog_expression
      [, analog_expression [, abstol_expression]]]])
  | idtmod (port_probe_function_call [, analog_expression [, analog_expression
      [, analog_expression [, abstol_expression]]]])

lsb_constant_expression ::= constant_expression

mintypmax_expression ::=
    expression
  | expression : expression : expression

module_path_conditional_expression ::= module_path_expression ? { attribute_instance }
      module_path_expression : module_path_expression

module_path_expression ::=

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    module_path_primary
  | unary_module_path_operator { attribute_instance } module_path_primary
  | module_path_expression binary_module_path_operator { attribute_instance }
      module_path_expression
  | module_path_conditional_expression

module_path_mintypmax_expression ::=
    module_path_expression
  | module_path_expression : module_path_expression : module_path_expression

msb_constant_expression ::= constant_expression

nature_attribute_expression ::=
    constant_expression
  | nature_identifier
  | nature_access_identifier

range_expression ::=
    expression
  | msb_constant_expression : lsb_constant_expression
  | base_expression +: width_constant_expression
  | base_expression -:  width_constant_expression

width_constant_expression ::= constant_expression
```

A.8.4 Primaries

```
analog_primary ::=
    number
  | analog_concatenation
  | analog_multiple_concatenation
  | variable_reference
  | net_reference
  | genvar_identifier
  | parameter_reference
  | nature_attribute_reference
  | branch_probe_function_call
  | port_probe_function_call
  | analog_function_call
  | analog_system_function_call
  | analog_built_in_function_call
  | analog_filter_function_call
  | analog_small_signal_function_call
  | analysis_function_call
  | ( analog_expression )

constant_primary ::=
    number
  | parameter_identifier [ [ constant_range_expression ] ]
  | specparam_identifier [ [ constant_range_expression ] ]
  | constant_concatenation
  | constant_multiple_concatenation
  | constant_function_call
  | constant_system_function_call
  | constant_analog_built_in_function_call
  | ( constant_mintypmax_expression )
  | string_literal
  | system_parameter_identifier
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
        | nature_attribute_reference
        | constant_analog_function_call
module_path_primary ::=
          number
        | identifier
        | module_path_concatenation
        | module_path_multiple_concatenation
        | function_call
        | system_function_call
        | ( module_path_mintypmax_expression )
primary ::=
          number
        | hierarchical_identifier [ { [ expression ] } [ range_expression ] ]
        | concatenation
        | multiple_concatenation
        | function_call
        | system_function_call
        | ( mintypmax_expression )
        | string
        | branch_probe_function_call
        | port_probe_function_call
        | nature_attribute_reference
        | analog_function_call
        | analog_built_in_function_call
```

A.8.5 Expression left-side values

```
analog_variable_lvalue ::=
          variable_identifier
        | variable_identifier [ analog_expression ] { [ analog_expression ] }
array_analog_variable_assignment ::= array_analog_variable_lvalue = array_analog_variable_rvalue ;
array_analog_variable_rvalue ::=
          array_variable_identifier
        | array_ variable_identifier [ analog_expression ] { [ analog_expression ] }
        | assignment_pattern
branch_lvalue ::= branch_probe_function_call
net_lvalue ::=
          hierarchical_net_identifier [ { [ constant_expression ] } [  constant_range_expression ] ]
        | { net_lvalue { , net_lvalue } }
variable_lvalue ::=
          hierarchical_variable_identifier [ { [ expression ] } [  range_expression ] ]
        | { variable_lvalue { , variable_lvalue } }
```

A.8.6 Operators

```
unary_operator ::=
          + | - | ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~
binary_operator ::=
          + | - | * | / | % | == | != | === | !== | && | || | **
        | < | <= | > | >= | & | | | ^ | ^~ | ~^ | >> | << | >>> | <<<
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

unary_module_path_operator ::=
    ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~

binary_module_path_operator ::=
    == | != | && | || | & | | | ^ | ^~ | ~^

A.8.7 Numbers

number ::=
    decimal_number
  | octal_number
  | binary_number
  | hex_number
  | real_number

real_number[2] ::=
    unsigned_number . unsigned_number
  | unsigned_number [. unsigned_number] exp [sign] unsigned_number
  | unsigned_number [. unsigned_number] scale_factor

exp ::= e | E

scale_factor ::= T | G | M | K | k | m | u | n | p | f | a

decimal_number ::=
    unsigned_number
  | [size] decimal_base unsigned_number
  | [size] decimal_base x_digit { _ }
  | [size] decimal_base z_digit { _ }

binary_number ::= [size] binary_base binary_value

octal_number ::= [size] octal_base octal_value

hex_number ::= [size] hex_base hex_value

sign ::= + | -

size ::= non_zero_unsigned_number

non_zero_unsigned_number[2] ::= non_zero_decimal_digit { _ | decimal_digit}

unsigned_number[2] ::= decimal_digit { _ | decimal_digit }

binary_value[2] ::= binary_digit { _ | binary_digit }

octal_value[2] ::= octal_digit { _ | octal_digit }

hex_value[2] ::= hex_digit { _ | hex_digit }

decimal_base[2] ::= '[s|S]d | '[s|S]D

binary_base[2] ::= '[s|S]b | '[s|S]B

octal_base[2] ::= '[s|S]o | '[s|S]O

hex_base[2] ::= '[s|S]h | '[s|S]H

non_zero_decimal_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

decimal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

binary_digit ::= x_digit | z_digit | 0 | 1

octal_digit ::= x_digit | z_digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

hex_digit ::=
    x_digit | z_digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  | a | b | c | d | e | f | A | B | C | D | E | F

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

x_digit ::= **x** | **X**
z_digit ::= **z** | **Z** | **?**

A.8.8 Strings

string_literal ::= " { Any_ASCII_Characters } "

A.8.9 Analog references

nature_attribute_reference ::= net_identifier . potential_or_flow . nature_attribute_identifier

analog_port_reference ::=
    port_identifier
  | port_identifier [constant_expression]
  | hierarchical_port_identifier
  | hierarchical_port_identifier [constant_expression]

analog_net_reference ::=
    port_identifier
  | port_identifier [constant_expression]
  | net_identifier
  | net_identifier [constant_expression]
  | hierarchical_port_identifier
  | hierarchical_port_identifier [constant_expression]
  | hierarchical_net_identifier
  | hierarchical_net_identifier [constant_expression]

branch_reference ::=
    hierarchical_branch_identifier
  | hierarchical_branch_identifier [constant_expression]
  | hierarchical_unnamed_branch_reference

hierarchical_unnamed_branch_reference ::=
    hierarchical_inst_identifier.**branch** (branch_terminal [, branch_terminal])
  | hierarchical_inst_identifier.**branch** (< port_identifier >)
  | hierarchical_inst_identifier.**branch** (< hierarchical_port_identifier >)

parameter_reference ::=
    parameter_identifier
  | parameter_identifier [analog_expression]

variable_reference ::=
    variable_identifier
  | variable_identifier [analog_expression] { [analog_expression] }
  | real_identifier
  | real_identifier [analog_expression] { [analog_expression] }

net_reference ::=
    hierarchical_net_identifier
  | hierarchical_net_identifier [analog_range_expression]
  | hierarchical_port_identifier
  | hierarchical_port_identifier [analog_range_expression]

A.9 General

A.9.1 Attributes

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

attribute_instance ::= (* attr_spec { , attr_spec } *)
attr_spec ::= attr_name [= constant_expression]
attr_name ::= identifier

A.9.2 Comments

comment ::=
one_line_comment
| block_comment
one_line_comment ::= // comment_text \n
block_comment ::= /* comment_text */
comment_text ::= { Any_ASCII_character }

A.9.3 Identifiers

ams_net_identifier ::=
net_identifier { dimension }
| hierarchical_net_identifier
analog_block_identifier ::= block_identifier
analog_function_identifier ::= identifier
analog_system_task_identifier ::= system_task_identifier
analog_system_function_identifier ::= system_function_identifier
analysis_identifier ::= identifier
block_identifier ::= identifier
branch_identifier ::= identifier
cell_identifier ::= identifier
config_identifier ::= identifier
connectmodule_identifier ::= module_identifier
connectrules_identifier ::= identifier
discipline_identifier ::= identifier
escaped_identifier ::= \ {Any_ASCII_character_except_white_space} white_space
event_identifier ::= identifier
function_identifier ::= identifier
gate_instance_identifier ::= identifier
generate_block_identifier ::= identifier
genvar_identifier ::= identifier
hierarchical_block_identifier ::= hierarchical_identifier
hierarchical_branch_identifier ::= hierarchical_identifier
hierarchical_event_identifier ::= hierarchical_identifier
hierarchical_function_identifier ::= hierarchical_identifier
hierarchical_identifier ::= [$root .] { identifier [[constant_expression]] . } identifier
hierarchical_inst_identifier ::= hierarchical_identifier
hierarchical_net_identifier ::= hierarchical_identifier
hierarchical_parameter_identifier ::= hierarchical_identifier
hierarchical_port_identifier ::= hierarchical_identifier

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

hierarchical_variable_identifier ::= hierarchical_identifier

hierarchical_task_identifier ::= hierarchical_identifier

identifier ::=
    simple_identifier
  | escaped_identifier

inout_port_identifier ::= identifier

input_port_identifier ::= identifier

instance_identifier ::= identifier

library_identifier ::= identifier

module_identifier ::= identifier

module_instance_identifier ::= identifier

module_or_paramset_identifier ::=
    module_identifier
  | paramset_identifier

module_output_variable_identifier ::= identifier

module_parameter_identifier ::= identifier

nature_identifier ::= identifier

nature_access_identifier ::= identifier

nature_attribute_identifier ::= **abstol** | **access** | **ddt_nature** | **idt_nature** | **units** | identifier

net_identifier ::= identifier

output_port_identifier ::= identifier

parameter_identifier ::= identifier

paramset_identifier ::= identifier

port_identifier ::= identifier

real_identifier ::= identifier

simple_identifier[3] ::= [**a-zA-Z_**] { [**a-zA-Z0-9_$**] }

specparam_identifier ::= identifier

system_function_identifier[4] ::= **$** [**a-zA-Z0-9_$**] { [**a-zA-Z0-9_$**] }

system_parameter_identifier[4] ::= **$** [**a-zA-Z0-9_$**] { [**a-zA-Z0-9_$**] }

system_task_identifier[4] ::= **$** [**a-zA-Z0-9_$**] { [**a-zA-Z0-9_$**] }

task_identifier ::= identifier

terminal_identifier ::= identifier

text_macro_identifier ::=
    identifier
  | **__VAMS_ENABLE__**
  | **__VAMS_COMPACT_MODELING__**

topmodule_identifier ::= identifier

udp_identifier ::= identifier

udp_instance_identifier ::= identifier

variable_identifier ::= identifier

A.9.4 White space

white_space ::= space | tab | newline | eof[5]

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

A.10 Details

1) Function statements are limited by the rules of 4.7.1.
2) Embedded spaces are illegal.
3) A simple_identifier shall start with an alpha or underscore (_) character, shall have at least one character, and shall not have any spaces.
4) The $ character in a system_function_identifier, system_task_identifier, or system_parameter_identifier shall not be followed by white_space. A system_function_identifier or system_task_identifier shall not be escaped.
5) End of file.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Annex B

(normative)

List of keywords

Keywords are predefined nonescaped identifiers that define Verilog-AMS language constructs. An escaped identifier shall not be treated as a keyword. Verilog-AMS reserves the keywords listed in Table B.1.

Table B.1—Reserved keywords

above	default	idt_nature
abs	defparam	if
absdelay	design	ifnone
absdelta	disable	incdir
abstol	discipline	include
access	discrete	inf
acos	domain	initial
acosh	driver_update	initial_step
ac_stim	edge	inout
aliasparam	else	input
always	end	instance
analog	endcase	integer
analysis	endconfig	join
and	endconnectrules	laplace_nd
asin	enddiscipline	laplace_np
asinh	endfunction	laplace_zd
assign	endgenerate	laplace_zp
atan	endmodule	large
atan2	endnature	last_crossing
atanh	endparamset	liblist
automatic	endprimitive	library
begin	endspecify	limexp
branch	endtable	ln
break	endtask	ln1p
buf	event	localparam
bufif0	exclude	log
bufif1	exp	macromodule
case	expm1	max
casex	final_step	medium
casez	flicker_noise	merged
ceil	floor	min
cell	flow	module
cmos	for	nand
config	force	nature
connect	forever	negedgenmos
connectmodule	fork	noise_table
connectrules	from	noise_table_log
continue	function	nor
continuous	generate	noshowcancelled
cos	genvar	not
cosh	ground	notif0
cross	highz0	notif1
ddt	highz1	or
ddt_nature	hypot	output
ddx	idt	parameter
deassign	idtmod	paramset

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table B.1—Reserved keywords *(continued)*

pmos	showcancelled	trior
posedge	signed	trireg
potential	slew	units
pow	small	unsigned
primitive	specify	use
pull0	specparam	uwire
pull1	split	vectored
pulldown	sqrt	wait
pullup	string	wand
pulsestyle_ondetect	strong0	weak0
pulsestyle_onevent	strong1	weak1
rcmos	supply0	while
real	supply1	white_noise
realtime	table	wire
reg	tan	wor
release	tanh	wreal
repeat	task	xnor
resolveto	time	xor
return	timer	zi_nd
rnmos	tran	zi_np
rpmos	tranif0	zi_zd
rtran	tranif1	zi_zp
rtranif0	transition	
rtranif1	tri	
scalared	tri0	
sin	tri1	
sinh	triand	

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Annex C

(normative)

Analog language subset

This annex defines a working subset of Verilog-AMS HDL for analog-only products.

C.1 Verilog-A overview

This Verilog-A subset defines a behavioral language for analog only systems. Verilog-A is derived from the IEEE Std 1364 Verilog specification using a minimum number of constructs for analog and mixed-signal behavioral descriptions. This Annex is intended to cover the definition and semantics of Verilog-A.

The intent of Verilog-A is to let designers of analog systems and integrated circuits create and use modules which encapsulate high-level behavioral descriptions of systems and components. The behavior of each module can be described mathematically in terms of its terminals and external parameters applied to the module. These behavioral descriptions can be used in many disciplines such as electrical, mechanical, fluid dynamics, and thermodynamics.

Verilog-A has been defined to be applicable to both electrical and non-electrical systems description. It supports conservative and signal-flow descriptions by using the terminology for these descriptions using the concepts of nodes, branches, and terminals. The solution of analog behaviors which obey the laws of conservation fall within the generalized form of Kirchhoff's Potential and Flow Laws (KPL and KFL). Both of these are defined in terms of the quantities associated with the analog behaviors.

C.2 Verilog-A language features

The Verilog-A subset provides access to a salient set of features of the full modeling language that allow analog designers the ability to model analog systems:

- Verilog-A modules are compatible with Verilog-AMS HDL.
- Analog behavioral modeling descriptions are contained in separate **`analog`** blocks.
- Branches can be named for easy selection and access.
- Parameters can be specified with valid range limits.
- Systems can be modeled by using expressions consisting of operators, variables, and signals:
  - a full set of operators including trigonometric functions, integrals, and derivatives;
  - a set of waveform filters to modify the waveform results for faster and more accurate simulation like transition, slew, Laplace, and Z-domain;
  - a set of events to control when certain code is simulated;
  - selection of the simulation time step for simulation control;
  - support for accessing SPICE primitives from within the language.

C.3 Lexical conventions

With the exception of certain keywords required for Verilog-AMS HDL, Clause 2 shall be applicable to both Verilog-A and Verilog-AMS HDL. All Verilog-AMS HDL keywords shall be supported by Verilog-A

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

as reserved words, but IEEE Std 1364 Verilog and Verilog-AMS HDL specific keywords are not used in Verilog-A. The following Verilog-AMS HDL keywords are not required to be supported for a fully compliant Verilog-A subset:

From 2.6, Numbers: support for `x` and `z` values is limited in the **`analog`** block to the mixed signal context, as defined in 7.3.2. In the same paragraph, the use of the question mark character as an alternative for `z` is also limited to the mixed signal context.

From 2.8.2, Keywords: certain keywords are not applicable in Verilog-A, as defined in Annex C.16.

C.4 Data types

The data types of Clause 3 are applicable to both Verilog-AMS HDL and Verilog-A with the following exceptions:

- From 3.6.2.2, Domain binding: the domain binding type `discrete` shall be an error in Verilog-A.
- From 3.7, Real net declarations: the **`wreal`** data type is not supported in Verilog-A.
- From 3.8, Default discipline: the **`` `default_discipline``** compiler directive is not supported in Verilog-A. All Verilog-A modules shall have a discipline defined for each module.

This feature allows the use of digital modules in Verilog-AMS HDL without editing them to add a discipline.

C.5 Expressions

The expressions defined in Clause 4 are applicable to both Verilog-AMS HDL and Verilog-A with the following exception:

The case equality operators (**`===`**, **`!==`**) are not supported in Verilog-A.

C.6 Analog signals

The signals defined in 5.4 are applicable to both Verilog-AMS HDL and Verilog-A.

C.7 Analog behavior

The analog behavior defined in Clause 5 are applicable to both Verilog-AMS HDL and Verilog-A with the following exceptions:

- No digital behavior or events are supported in Verilog-A.
- **`casex`** and **`casez`** are not supported in Verilog-A.

C.8 Hierarchical structures

The hierarchical structure defined in Clause 6 is applicable to both Verilog-AMS HDL and Verilog-A, except support for *real value ports* is only applicable to Verilog-AMS HDL and IEEE Std 1364 Verilog (see 6.5.3).

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

C.9 Mixed signal

Clause 7only applies to Verilog-AMS HDL.

C.10 Scheduling semantics

The analog simulation cycle is applicable to both Verilog-AMS HDL and Verilog-A. The mixed-signal simulation cycle from 8.2 is only applicable to Verilog-AMS HDL.

C.11 System tasks and functions

All system tasks and functions in Clause 9 that are applicable in the analog context are applicable to Verilog-A.

C.12 Compiler directives

The compiler directives of Clause 10 are applicable to both Verilog-AMS HDL and Verilog-A.

C.13 Using VPI routines

The analog behavior defined in Clause 11 is applicable to both Verilog-AMS HDL and Verilog-A.

C.14 VPI routine definitions

The analog behavior defined in Clause 12 is applicable to both Verilog-AMS HDL and Verilog-A.

C.15 Analog language subset

This annex (Annex C) defines the differences between Verilog-AMS HDL and Verilog-A. Annex A defines the BNF for Verilog-AMS HDL.

C.16 List of keywords

The keywords in Annex B are the complete set of Verilog-AMS HDL keywords, including those from IEEE Std 1364 Verilog. The following keywords as defined in this LRM are not used by Verilog-A:

```
connect
connectmodule
connectrules
driver_update
endconnectrules
merged
resolveto
split
wreal
```

NOTE—All keywords of Verilog-AMS HDL are reserved words for Verilog-A.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

C.17 Standard definitions

The definitions of Annex D are applicable to both Verilog-AMS HDL and Verilog-A, with the exception of those disciplines with a domain of **`discrete`**. A Verilog-A implementation shall silently ignore any definition of a discipline with a domain of **`discrete`**.

C.18 SPICE compatibility

Annex E defines the SPICE compatibility for both Verilog-A and Verilog-AMS HDL.

C.19 Changes from previous Verilog-A LRM versions

Annex G describes the changes from previous LRM versions for both Verilog-A and Verilog-AMS HDL.

C.20 Obsolete functionality

Annex G also describes the statements that are no longer supported in the current version of Verilog-AMS HDL as well as the analog language subset.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Annex D

(normative)

Standard definitions

This annex contains the standard definition packages (`disciplines.vams`, `constants.vams` and `driver_access.vams`) for Verilog-AMS HDL. The copyright for these three files differs from the rest of the Verilog-AMS HDL language reference manual to reflect that verbatim copies of these three standard definition files that make up this Annex may be used and distributed without restrictions.

D.1 The disciplines.vams file

```
// Copyright(c) 2009-2023 Accellera Systems Initiative Inc.
// 8698 Elk Grove Blvd. Suite 1, #114, Elk Grove, CA, 95624, USA.
//
// The material in disciplines.vams is an essential part of the Accellera Systems
// Initiative ("Accellera") Verilog-AMS Language Standard. Verbatim copies of
// the material in this Annex may be used and distributed without restriction.
// All other uses require permission from Accellera IP Committee
// (ipr-chair@lists.accellera.org).
// All other rights reserved.
//
// VAMS-2023

`ifdef DISCIPLINES_VAMS
`else
`define DISCIPLINES_VAMS 1

//
// Natures and Disciplines
//

discipline \logic ;
   domain discrete;
enddiscipline

discipline ddiscrete;
   domain discrete;
enddiscipline

/*
 *   Default absolute tolerances may be overridden by setting the
 *   appropriate _ABSTOL prior to including this file
 */

// Electrical

// Current in amperes
nature Current;
   units        = "A";
   access       = I;
   idt_nature   = Charge;
`ifdef CURRENT_ABSTOL
   abstol       = `CURRENT_ABSTOL;
`else
   abstol       = 1e-12;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
`endif
endnature

// Charge in coulombs
nature Charge;
   units      = "coul";
   access     = Q;
   ddt_nature = Current;
`ifdef CHARGE_ABSTOL
   abstol     = `CHARGE_ABSTOL;
`else
   abstol     = 1e-14;
`endif
endnature

// Potential in volts
nature Voltage;
   units      = "V";
   access     = V;
   idt_nature = Flux;
`ifdef VOLTAGE_ABSTOL
   abstol     = `VOLTAGE_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature

// Flux in Webers
nature Flux;
   units      = "Wb";
   access     = Phi;
   ddt_nature = Voltage;
`ifdef FLUX_ABSTOL
   abstol     = `FLUX_ABSTOL;
`else
   abstol     = 1e-9;
`endif
endnature

// Conservative discipline
discipline electrical;
   potential    Voltage;
   flow         Current;
enddiscipline

// Signal flow disciplines
discipline voltage;
   potential    Voltage;
enddiscipline

discipline current;
   flow    Current;
enddiscipline

// Magnetic

// Magnetomotive force in Ampere-Turns.
nature Magneto_Motive_Force;
   units      = "A*turn";
   access     = MMF;
`ifdef MAGNETO_MOTIVE_FORCE_ABSTOL
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   abstol     = `MAGNETO_MOTIVE_FORCE_ABSTOL;
`else
   abstol     = 1e-12;
`endif
endnature

// Conservative discipline
discipline magnetic;
   potential    Magneto_Motive_Force;
   flow         Flux;
enddiscipline

// Thermal

// Temperature in Kelvin
nature Temperature;
   units      = "K";
   access     = Temp;
`ifdef TEMPERATURE_ABSTOL
   abstol     = `TEMPERATURE_ABSTOL;
`else
   abstol     = 1e-4;
`endif
endnature

// Power in Watts
nature Power;
   units      = "W";
   access     = Pwr;
`ifdef POWER_ABSTOL
   abstol     = `POWER_ABSTOL;
`else
   abstol     = 1e-9;
`endif
endnature

// Conservative discipline
discipline thermal;
   potential    Temperature;
   flow         Power;
enddiscipline

// Kinematic

// Position in meters
nature Position;
   units      = "m";
   access     = Pos;
   ddt_nature = Velocity;
`ifdef POSITION_ABSTOL
   abstol     = `POSITION_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature

// Velocity in meters per second
nature Velocity;
   units      = "m/s";
   access     = Vel;
   ddt_nature = Acceleration;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   idt_nature = Position;
`ifdef VELOCITY_ABSTOL
   abstol     = `VELOCITY_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature

// Acceleration in meters per second squared
nature Acceleration;
   units      = "m/s^2";
   access     = Acc;
   ddt_nature = Impulse;
   idt_nature = Velocity;
`ifdef ACCELERATION_ABSTOL
   abstol     = `ACCELERATION_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature

// Impulse in meters per second cubed
nature Impulse;
   units      = "m/s^3";
   access     = Imp;
   idt_nature = Acceleration;
`ifdef IMPULSE_ABSTOL
   abstol     = `IMPULSE_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature

// Force in Newtons
nature Force;
   units      = "N";
   access     = F;
`ifdef FORCE_ABSTOL
   abstol     = `FORCE_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature

// Conservative disciplines
discipline kinematic;
   potential    Position;
   flow         Force;
enddiscipline

discipline kinematic_v;
   potential    Velocity;
   flow         Force;
enddiscipline

// Rotational

// Angle in radians
nature Angle;
   units      = "rads";
   access     = Theta;
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
   ddt_nature = Angular_Velocity;
`ifdef ANGLE_ABSTOL
   abstol     = `ANGLE_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature

// Angular Velocity in radians per second
nature Angular_Velocity;
   units      = "rads/s";
   access     = Omega;
   ddt_nature = Angular_Acceleration;
   idt_nature = Angle;
`ifdef ANGULAR_VELOCITY_ABSTOL
   abstol     = `ANGULAR_VELOCITY_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature

// Angular acceleration in radians per second squared
nature Angular_Acceleration;
   units      = "rads/s^2";
   access     = Alpha;
   idt_nature = Angular_Velocity;
`ifdef ANGULAR_ACCELERATION_ABSTOL
   abstol     = `ANGULAR_ACCELERATION_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature

// Torque in Newtons
nature Angular_Force;
   units      = "N*m";
   access     = Tau;
`ifdef ANGULAR_FORCE_ABSTOL
   abstol     = `ANGULAR_FORCE_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature
// Conservative disciplines
discipline rotational;
   potential    Angle;
   flow         Angular_Force;
enddiscipline
discipline rotational_omega;
   potential    Angular_Velocity;
   flow         Angular_Force;
enddiscipline
`endif
```

D.2 The constants.vams file

```
// Copyright(c) 2009-2023 Accellera Systems Initiative Inc.
// 8698 Elk Grove Blvd. Suite 1, #114, Elk Grove, CA, 95624, USA.
//
// The material in constants.vams is an essential part of the Accellera Systems
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
// Initiative ("Accellera") Verilog-AMS Language Standard. Verbatim copies of
// the material in this Annex may be used and distributed without restriction.
// All other uses require permission from Accellera IP Committee
// (ipr-chair@lists.accellera.org).
// All other rights reserved
//
// VAMS-2023
//
// Mathematical and physical constants
`ifdef CONSTANTS_VAMS
`else
`define CONSTANTS_VAMS 1
// M_ is a mathematical constant
`define  M_E               2.7182818284590452354
`define  M_LOG2E           1.4426950408889634074
`define  M_LOG10E          0.43429448190325182765
`define  M_LN2             0.69314718055994530942
`define  M_LN10            2.30258509299404568402
`define  M_PI              3.14159265358979323846
`define  M_TWO_PI          6.28318530717958647693
`define  M_PI_2            1.57079632679489661923
`define  M_PI_4            0.78539816339744830962
`define  M_1_PI            0.31830988618379067154
`define  M_2_PI            0.63661977236758134308
`define  M_2_SQRTPI        1.12837916709551257390
`define  M_SQRT2           1.41421356237309504880
`define  M_SQRT1_2         0.70710678118654752440

// The following constants have been taken from:
// https://physics.nist.gov/cuu/Constants
// P_ is a physical constant
// charge of electron in Coulombs
`define  P_Q_SPICE         1.60219e-19
`define  P_Q_OLD           1.6021918e-19
`define  P_Q_NIST1998      1.602176462e-19
`define  P_Q_NIST2010      1.602176565e-19
`define  P_Q_NIST2018      1.602176634e-19
// speed of light in vacuum in meters/second
`define  P_C               2.99792458e8
// Boltzmann's constant in Joules/Kelvin
`define  P_K_SPICE         1.38062e-23
`define  P_K_OLD           1.3806226e-23
`define  P_K_NIST1998      1.3806503e-23
`define  P_K_NIST2010      1.3806488e-23
`define  P_K_NIST2018      1.380649e-23
// Planck's constant in Joules*second
`define  P_H_SPICE         6.62620e-34
`define  P_H_OLD           6.6260755e-34
`define  P_H_NIST1998      6.62606876e-34
`define  P_H_NIST2010      6.62606957e-34
`define  P_H_NIST2018      6.62607015e-34
// permittivity of vacuum in Farads/meter
`define  P_EPS0_SPICE      8.854214871e-12
`define  P_EPS0_OLD        8.8541879239442001396 8e-12
`define  P_EPS0_NIST1998   8.854187817e-12
`define  P_EPS0_NIST2010   8.854187817e-12
`define  P_EPS0_NIST2018   8.8541878128e-12
// permeability of vacuum in Henrys/meter
`define  P_U0_OLD          (4.0e-7 * `M_PI)
`define  P_U0_NIST2018     1.25663706212e-6
// zero Celsius in Kelvin
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
`define   P_CELSIUS0         273.15

`ifdef PHYSICAL_CONSTANTS_NIST2018
`define P_Q `P_Q_NIST2018
`define P_K `P_K_NIST2018
`define P_H `P_H_NIST2018
`define P_EPS0 `P_EPS0_NIST2018
`define P_U0 `P_U0_NIST2018
`else
`define P_U0 `P_U0_OLD
`ifdef PHYSICAL_CONSTANTS_SPICE
// from UC Berkeley SPICE 3F5
`define P_Q `P_Q_SPICE
`define P_K `P_K_SPICE
`define P_H `P_H_SPICE
`define P_EPS0 `P_EPS0_SPICE

`else
`ifdef PHYSICAL_CONSTANTS_OLD
// from Verilog-A LRM 1.0 and Verilog-AMS LRM 2.0
`define P_Q `P_Q_OLD
`define P_K `P_K_OLD
`define P_H `P_H_OLD
`define P_EPS0 `P_EPS0_OLD

`else
`ifdef PHYSICAL_CONSTANTS_NIST2010
`define P_Q `P_Q_NIST2010
`define P_K `P_K_NIST2010
`define P_H `P_H_NIST2010
`define P_EPS0 `P_EPS0_NIST2010

`else
// use NIST1998 values as in LRM 2.2 - 2.3 for backwards-compatibility
`define P_Q `P_Q_NIST1998
`define P_K `P_K_NIST1998
`define P_H `P_H_NIST1998
`define P_EPS0 `P_EPS0_NIST1998
`endif
`endif
`endif
`endif
`endif
```

D.3 The driver_access.vams file

```
// Copyright(c) 2009-2014 Accellera Systems Initiative Inc.
// 1370 Trancas Street #163, Napa, CA 94558, USA.
//
// The material in driver_access.vams is an essential part of the Accellera Systems
// Initiative ("Accellera") Verilog-AMS Language Standard. Verbatim copies of
// the material in this Annex may be used and distributed without restriction.
// All other uses require permission from Accellera IP Committee
// (ipr-chair@lists.accellera.org).
// All other rights reserved.
//
// VAMS-2023

`ifdef DRIVER_ACCESS_VAMS
`else
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
`define DRIVER_ACCESS_VAMS  1
`define DRIVER_UNKNOWN      32'b00000000000    // No information
`define DRIVER_DELAYED      32'b00000000001    // driver has fixed delay
`define DRIVER_GATE         32'b00000000010    // driver is a primitive
`define DRIVER_UDP          32'b00000000100    // driver is a user defined primitive
`define DRIVER_ASSIGN       32'b00000001000    // driver is a continuous assignment
`define DRIVER_BEHAVIORAL   32'b00000010000    // driver is a reg
`define DRIVER_SDF          32'b00000100000    // driver is from backannotated code
`define DRIVER_NODELETE     32'b00001000000    // events won't be deleted
`define DRIVER_NOPREEMPT    32'b00010000000    // events won't be preempted
`define DRIVER_KERNEL       32'b00100000000    // added by kernel (wor/wand)
`define DRIVER_WOR          32'b01000000000    // driver is on a wor net
`define DRIVER_WAND         32'b10000000000    // driver is on a wand net
`endif
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Annex E

(normative)

SPICE compatibility

E.1 Introduction

Analog simulation has long been performed with SPICE and SPICE-like simulators. As such, there is a huge legacy of SPICE netlists. In addition, SPICE provides a rich set of predefined models and it is considered neither practical nor desirable to convert these models into a Verilog-AMS HDL behavioral description. In order for Verilog-AMS HDL to be embraced by the analog design community, it is important Verilog-AMS HDL provide an appropriate degree of SPICE compatibility. This annex describes the degree of compatibility which Verilog-AMS HDL provides and the approach taken to provide that compatibility.

E.1.1 Scope of compatibility

SPICE is not a single language, but rather is a family of related languages. The first widely used version of SPICE was SPICE2g6 from the University of California at Berkeley. However, SPICE has been enhanced and distributed by many different companies, each of which has added their own extensions to the language and models. As a result, there is a great deal of incompatibility even among the SPICE languages themselves.

Verilog-AMS HDL makes no judgment as to which of the various SPICE languages should be supported. Instead, it states if a simulator which supports Verilog-AMS HDL is also able to read SPICE netlists of a particular flavor, then certain objects defined in that flavor of SPICE netlist can be referenced from within a Verilog-AMS HDL structural description. In particular, SPICE models and subcircuits can be instantiated within a Verilog-AMS HDL module. This is also true for any SPICE primitives which are built into the simulator. In general, anything that can be instantiated in the particular flavor of SPICE can also be instantiated within a Verilog-AMS HDL module.

E.1.2 Degree of incompatibility

There are four primary areas of incompatibility between versions of SPICE simulators.

1) The version of the SPICE language accepted by various simulators is different and to some degree proprietary. This issue is not addressed by Verilog-AMS HDL. So whether a particular Verilog-AMS simulator is SPICE compatible, and with which particular variant of SPICE it is compatible, is solely determined by the authors of the simulator.
2) Not all SPICE simulators support the same set of component primitives. Thus, a particular SPICE netlist can reference a primitive which is unsupported. Verilog-AMS HDL offers no alternative in this case other than the possibility that if the model equations are known, the primitive can be rewritten as a module.
3) The names of the built-in SPICE primitives, their parameters, or their ports can differ from simulator to simulator. This is particularly true because many primitives, parameters, and ports are unnamed in SPICE. When instantiating SPICE primitives in Verilog-AMS HDL, the primitives shall, and parameters and ports can, be named. Since there are no established standard names, there is a high likelihood of incompatibility cropping up in these names. To reduce this, a list of what names shall be used for the more common components is shown in Table E.1. However, it is not possible to anticipate all SPICE primitives and parameters which could be supported; so different implementations can

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

end up using different names. This level of incompatibility can be overcome by using wrapper modules to map names.

4) The mathematical description of the built-in primitives can differ. As with the netlist syntax, incompatible enhancements of the models have crept in through the years. Again, Verilog-AMS HDL offers no solution in this case other than the possibility that if the model equations are known, the primitive can be rewritten as a module.

E.2 Accessing SPICE objects from Verilog-AMS HDL

If an implementation of a Verilog-AMS tool supports SPICE compatibility, it is expected to provide the basic set of SPICE primitives (see Annex E.3) and be able to read SPICE netlists which contain models and subcircuit statements.

SPICE primitives built into the simulator shall be treated in the same manner in Verilog-AMS HDL as built-in primitives of gate- and switch-level modeling. However, while the Verilog-AMS HDL built-in primitives are standardized, the SPICE primitives are not. All aspects of SPICE primitives are implementation dependent.

In addition to SPICE primitives, it shall also be possible to access subcircuits and models defined within SPICE netlists. The subcircuits and models contained within the SPICE netlist are treated as module definitions.

E.2.1 Case sensitivity

Some SPICE netlists are case insensitive, whereas Verilog-AMS HDL descriptions are case-sensitive. From within Verilog-AMS HDL, a mixed-case name matches the same name with an identical case (if one is defined in a Verilog-AMS HDL description). However, if no exact match is found, the mixed-case name shall match the same name defined within SPICE regardless of the case.

E.2.2 Examples

This subsection shows some examples.

E.2.2.1 Accessing SPICE models

Consider the following SPICE model file being read by a Verilog-AMS HDL simulator.

```
.MODEL VERTNPN NPN BF=80 IS=1E-18 RB=100 VAF=50
+ CJE=3PF CJC=2PF CJS=2PF TF=0.3NS TR=6NS
```

This model can be instantiated in a Verilog-AMS HDL module as shown in Figure E.1.

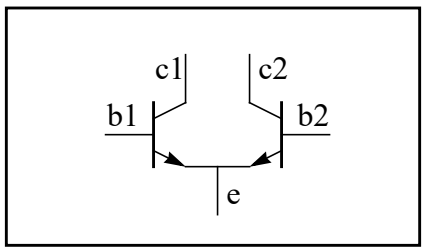


Figure E.1—Instantiated module

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
module diffPair (c1, b1, e, b2, c2);
   electrical c1, b1, e, b2, c2;

   vertNPN Q1 (c1, b1, e);
   vertNPN Q2 (.c(c2), .b(b2), .e(e));
endmodule
```

Unlike with SPICE, the first letter of the instance name, in this case `Q1` and `Q2`, is not constrained by the primitive type. For example, they can just as easily be `T1` and `T2`.

The ports and parameters of the `bjt` are determined by the `bjt` primitive itself and not by the model statement for the `bjt`. See E.3 for more details. This `bjt` primitive has 3 mandatory ports (*c*, *b*, and *e*) and one optional port (*s*). In the instantiation of `Q1`, the ports are passed by order. With `Q2`, the ports are passed by name. In both cases, the optional substrate port *s* is defaulted by simply not giving it.

E.2.2.2 Accessing SPICE subcircuits

As an example of how a SPICE subcircuit is referenced from Verilog-AMS HDL, consider the following SPICE subcircuit definition of an oscillator.

```
.SUBCKT ECPOSC (OUT GND)
   VA VCC GND 5
   IEE E GND 1MA
   Q1 VCC B1 E VCC VERTNPN
   Q2 OUT B2 E OUT VERTNPN
   L1 VCC OUT 1UH
   C1 VCC OUT 1P IC=1
   C2 OUT B1 272.7PF
   C3 B1 GND 3NF
   R1 B1 GND 10K
   C4 B2 GND 3NF
   R2 B2 GND 10K
.ENDS ECPOSC
```

This oscillator can be referenced from Verilog-AMS HDL as:

```
module osc (out, gnd);
   electrical out, gnd;
   ecpOsc Osc1 (out, gnd);
endmodule
```

NOTE—In Verilog-AMS HDL the name of the subcircuit instance is not constrained to start with `X` as it is in SPICE.

E.2.2.3 Accessing SPICE primitives

To show how various SPICE primitives can be accessed from Verilog-AMS HDL, the subcircuit in E.2.2.2 is translated to native Verilog-AMS HDL.

```
module ecpOsc (out, gnd);
   electrical out, gnd;

   vsine #(.dc(5)) Vcc (vcc, gnd);
   isine #(.dc(1m)) Iee (e, gnd);
   vertNPN Q1 (vcc, b1, e, vcc);
   vertNPN Q2 (out, b2, e, out);
   inductor #(.l(1u)) L1 (vcc, out);
   capacitor #(.c(1p), .ic(1)) C1 (vcc, out);
```


Copyright © 2024 Accellera Systems Initiative. All rights reserved.

```
    capacitor #(.c(272.7p)) C2 (out, b1);
    capacitor #(.c(3n)) C3 (b1, gnd);
    resistor #(.r(10k)) R1 (b1, gnd);
    capacitor #(.c(3n)) C4 (b2, gnd);
    resistor #(.r(10k)) R2 (b2, gnd);
endmodule
```

E.3 Preferred primitive, parameter, and port names

Table E.1 shows the required names for primitives, parameters, and ports which are otherwise unnamed in SPICE. For connection by order instead of by name, the ports and parameters shall be given in the order listed. The default discipline of the ports for these primitives shall be `electrical` and their descriptions shall be **`inout`**.

Table E.1—Names for primitives, parameters, and ports in SPICE

Primitive	Port name	Parameter name	Behavior
`resistor`	`p, n`	`r, tc1, tc2`	$V = I \cdot r \cdot (1 + tc1 \cdot T + tc2 \cdot T^2)$
`capacitor`	`p, n`	`c, ic`	$V = \frac{1}{c} \cdot \int_0^t I d\tau + ic$
`inductor`	`p, n`	`l, ic`	$I = l \cdot \int_0^t V d\tau + ic$
`iexp`	`p, n`	`dc, mag, phase, val0, val1, td0, tau0, td1, tau1`	$I = \begin{cases} val0 & t \le td0 \\ val1 - (val1 - dc) \cdot e^{\frac{td0 - t}{tau0}} & td0 < t \le td1 \\ val0 - (val0 - I_{td1}) \cdot e^{\frac{td1 - t}{tau1}} & td1 < t \end{cases}$ with I_{td1} the value of I at time $t = td1$.
`ipulse`	`p, n`	`dc, mag, phase, val0, val1, td, rise, fall, width, period`	$I = \begin{cases} val0 & t \le t0 \\ val0 + (val1 - val0) \cdot \frac{t - t0}{rise} & t0 < t \le t1 \\ val1 & t1 < t \le t2 \\ val1 + (val0 - val1) \cdot \frac{t - t2}{fall} & t2 < t \le t3 \\ val0 & t3 < t \le t4 \end{cases}$ with the following definitions (n is a non-negative integer): $t0 = td + n \cdot period$; $t1 = rise + td + n \cdot period$; $t2 = width + rise + td + n \cdot period$; $t3 = fall + width + rise + td + n \cdot period$; $t4 = td + (n + 1) \cdot period$

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table E.1—Names for primitives, parameters, and ports in SPICE *(continued)*

Primitive	Port name	Parameter name	Behavior
ipwl	p, n	dc, mag, phase, wave	I = $wave[i+1] + (wave[i+3] - wave[i+1]) \cdot \frac{t - wave[i]}{wave[i+2] - wave[i]}$ for $wave[i] \le t < wave[i+2]$ and $0 \le i < n$, $n = len(wave)$ I = $wave[n-1]$ for $wave[n-2] < t$
isine	p, n	dc, mag, phase, offset, ampl, freq, td, damp, sinephase, ammodindex, ammodfreq, ammodphase, fmmodindex, fmmodfreq	I = $offset + ampl \cdot (1 - F_{AM} \cdot \cos(2\pi \cdot f_{AM} \cdot (t - td) - \varphi_{AM})) \cdot (1 - damp \cdot (t - td)) \cdot \cos(2\pi \cdot freq \cdot (1 - F_{FM} \cdot \cos(2\pi \cdot f_{FM} \cdot (t - td))) \cdot (t - td) - \varphi_{SIN})$ with F_{AM} = ammodindex, f_{AM} = ammodfreq, φ_{AM} = ammodphase, F_{FM} = fmmodindex, f_{AM} = fmmodfreq, and φ_{SIN} =sinephase.
vexp	p, n	dc, mag, phase, val0, val1, td0, tau0, td1, tau1	V = $\begin{cases} dc & t \le td0 \\ val1 - (val1 - dc) \cdot e^{\frac{td0 - t}{tau0}} & td0 < t \le td1 \\ val0 - (val0 - V_{td1}) \cdot e^{\frac{td1 - t}{tau1}} & td1 < t \end{cases}$ with V_{td1} the value of V at time $t = td1$.
vpulse	p, n	dc, mag, phase, val0, val1, td, rise, fall, width, period	V = $\begin{cases} val0 & t \le t0 \\ val0 + (val1 - val0) \cdot \frac{t - t0}{rise} & t0 < t \le t1 \\ val1 & t1 < t \le t2 \\ val1 + (val0 - val1) \cdot \frac{t - t2}{fall} & t2 < t \le t3 \\ val0 & t3 < t \le t4 \end{cases}$ with the following definitions (n is a non-negative integer): $t0 = td + n \cdot period$ $t1 = rise + td + n \cdot period$ $t2 = width + rise + td + n \cdot period$ $t3 = fall + width + rise + td + n \cdot period$ $t4 = td + (n+1) \cdot period$

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table E.1—Names for primitives, parameters, and ports in SPICE *(continued)*

Primitive	Port name	Parameter name	Behavior
vpwl	p, n	dc, mag, phase, wave	V = $wave[i+1] + (wave[i+3] - wave[i+1]) \cdot \frac{t - wave[i]}{wave[i+2] - wave[i]}$ for $wave[i] \le t < wave[i+2]$ and $0 \le i < n$, $n = len(wave)$ I = $wave[n-1]$ for $wave[n-2] < t$.
vsine	p, n	dc, mag, phase, offset, ampl, freq, td, damp, sinephase, ammodindex, ammodfreq, ammodphase, fmmodindex, fmmodfreq	V = $offset + ampl \cdot (1 - F_{AM} \cdot \cos(2\pi \cdot f_{AM} \cdot (t - td) - \varphi_{AM})) \cdot (1 - damp \cdot (t - td)) \cdot \cos(2\pi \cdot freq \cdot (1 - F_{FM} \cdot \cos(2\pi \cdot f_{FM} \cdot (t - td)))) \cdot (t - td) - \varphi_{SIN})$ with F_{AM} = ammodindex, f_{AM} = ammodfreq, φ_{AM} = ammodphase, F_{FM} = fmmodindex, f_{AM} = fmmodfreq, and φ_{SIN} =sinephase.
tline	t1, b1, t2, b2	z0, td, f, nl	
vccs	sink, src, ps, ns	gm	I(sink, src) = $gm \cdot V(ps, ns)$
vcvs	p, n, ps, ns	gain	V(p, n) = $gain \cdot V(ps, ns)$
diode	a, c	area	
bjt	c, b, e, s	area	
mosfet	d, g, s, b	w, l, ad, as, pd, ps, nrd, nrs	
jfet	d, g, s	area	
mesfet	d, g, s	area	

Although in a SPICE context the primitives for `diode`, `bjt`, `mosfet`, `jfet`, and `mesfet` can be used only in a model definition, in Verilog-AMS they may be used directly in a paramset statement as described in 7.5.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

E.3.1 Unsupported primitives

Verilog-AMS HDL does not support the concept of passing an instance name as a parameter. As such, the following primitives are not supported: `ccvs`, `cccs`, and mutual inductors; however, these primitives can be instantiated inside a SPICE subcircuit that itself is instantiated in Verilog-AMS.

E.3.2 Discipline of primitives

To afford the ability to use analog primitive in any design, including mixed disciplines, the default discipline override is provided. The discipline of analog primitives will be resolved based on instance specific attributes, the disciplines of other instances on the same net, or default to electrical if it cannot be determined.

The precedence for the discipline of analog primitives is as follows:

1) A port_discipline attribute on the analog primitive;
2) The resolution of the discipline;
3) The default analog primitive of electrical.

E.3.2.1 Setting the discipline of analog primitives

A new optional attribute shall be provided called *port_discipline*, which shall have as a value the desired discipline for the port of the analog primitive. It shall only apply to either the analog primitive itself or the port to which it is attached. The value shall be of type string and the value must be a valid discipline of domain continuous. This attribute shall only apply to analog primitives or the ports of analog primitives; for other modules as well as the ports of all other modules it shall be ignored.

The following provides an example of this attribute applied to an analog primitive.

```
(* port_discipline="electrical" *) resistor #(.r(1k))
   r1 (node1, node2); // not needed as default
(* port_discipline="rotational" *) resistor #(.r(1k))
   r2 (node1, node2);
```

The following provides an example of this attribute applied to the ports of an analog primitive.

```
resistor #(.r(1k)) r3
   ((* port_discipline="rotational" *) node1,
    (* port_discipline="rotational" *) node2);
```

The use of these attributes can be combined to change the basic discipline of all ports for the analog primitive, but overriding the discipline for specific ports. The following provides an example of this use

```
(* port_discipline="electrical" *) vcvs #(.gain(1.45e-3))
   motor1 (n1, gnd_e,
      (* port_discipline="rotational_omega" *) shaft1,
      (* port_discipline="rotational_omega" *) gnd_rot);
```

The above model uses a voltage-controlled voltage source to model a motor as a converter from electrical potential to rotational velocity.

Attributes are described in 2.9 of this document.

E.3.2.2 Resolving the disciplines of analog primitives

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

If no attribute exists on the instance of an analog primitive, then the discipline may be determined by the disciplines of other instances connected to the same net segment. The disciplines of the vpiLoConn of all other instances on the net segment shall be evaluated to determine if they are of domain continuous and compatible with each other. If they are, then the discipline of the analog primitive shall be set to the same discipline. If they are not compatible, then an error will occur as defined in 3.11. If there are no continuous disciplines defined on the net segment, then the discipline shall default to electrical.

E.3.3 Name scoping of SPICE primitives

In the resolution hierarchy of names during elaboration a module or paramset defined in the Verilog-AMS will always be selected in favor of a SPICE primitive, model, or subcircuit using exactly the same name.

In case of a name match with differences in case, the module or paramset does not interfere with the SPICE primitive, model, or subcircuit, but the resolution method described in E.2.1 shall apply.

In case of a SPICE primitive which is always available in the Verilog-AMS simulator, a Verilog-AMS module or paramset whose name exactly matches that of the primitive will be used in module instantiations. The Verilog-AMS simulator may issue a warning stating that the Verilog-AMS module or paramset is used instead of the SPICE primitive. In case of a Verilog-AMS module or paramset whose name exactly matches that of a SPICE model or subcircuit, the Verilog-AMS simulator shall issue an warning message stating that the Verilog-AMS module or paramset is used instead of the SPICE model or subcircuit.

E.3.4 Limiting algorithms

Many SPICE simulators use limiting algorithms to improve convergence in Newton-Raphson iterations. Table E.2 lists the preferred names for three functions that may be available in a simulator, their arguments, and their intended uses. The function name, enclosed in quotation marks, can be used in the **`$limit()`** function of 9.17.3. This allows a Verilog-AMS module to use the same limiting algorithms available to built-in SPICE primitives. The arguments are described in 9.17.3.

Table E.2—SPICE limiting functions

Function name	Arguments	Meant for limiting:
fetlim	vth	gate-to-source voltage of field-effect transistors
pnjlim	vte, vcrit	voltage across diodes and pn junctions in other devices
vdslim	(none)	drain-to-source voltage of field-effect transistors

E.4 Other issues

This section highlights some other issues

E.4.1 Multiplicity factor on subcircuits

Some SPICE simulators support a multiplicity factor (M) parameter on subcircuits without the parameter being explicitly being declared. This factor is typically used to indicate the subcircuit should be modeled as if there are a specified number of copies in parallel. In previous versions of Verilog-AMS HDL, subcircuits defined as modules could not support automatic M factors.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Starting with LRM Version 2.2, the multiplicity factor is supported for subcircuits defined as modules in Verilog-AMS using the hierarchical system parameter **`$mfactor`**, as described in 6.3.6.

E.4.2 Binning and libraries

Some SPICE netlists provide mechanisms for mapping an instance to a group of models, with the final determination of which model to use being based on rules encapsulated in the SPICE netlist. Examples include model binning or corners support. From within an instance statement, it appears as if the instance is referencing a simple SPICE model; supporting these additional capabilities in Verilog-AMS HDL is supported via the instance line by default. Support of SPICE model cards is implementation specific (including those using these mechanisms).

Similar functionality for Verilog-AMS is supported through use of the paramset, as described in 6.4. Instead of referencing a specific module, and instance may refer to a paramset identifier, and there may be several paramsets with the same identifier (name). The final determination of which paramset to use is made according to rules specified in 6.4.2.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Annex F

(normative)

Discipline resolution methods

F.1 Discipline resolution

Discipline resolution is described in 7.4; it provides the semantics for two methods of resolving the discipline of undeclared interconnect. This annex provides a possible algorithm for achieving the semantics of each method. It is also possible to develop and use other algorithms to match the semantics.

F.2 Resolution of mixed signals

The following algorithms for discipline resolution of undeclared nets provide users with the ability to control the auto-insertion of connection modules. The undeclared nets are resolved at each level of the hierarchy in which *continuous* (analog) has precedence over *discrete* (digital). In both algorithms, the *continuous* domain is passed up the hierarchy from lower levels to the top level.

The algorithms traverse the hierarchy of a signal composed of nets (also known as net segments of the signal) in order to determine the discipline of all nets of undeclared discipline. See 7.2.3 for a description of how a signal consists of a set of net segments.

A net segment of a signal on the upper connection of a port shall be considered as the parent to a net segment on the lower connection of the port. The net segment on the lower connection of a port shall be considered as a child net segment of that parent net segment.

When a signal is being traversed *depth-first*, this means that the traversal shall start at the bottom (leaf) net segments of the signal – these are net segments which have no children net segments. It further means that all the children net segments of a parent net segment shall be traversed before that parent net segment is traversed. This type of depth first traversal is more precisely termed a *post-order depth-first traversal*.

When a signal is being traversed *top-down*, this means that the traversal shall start at the top net segment(s) of the signal – these are net segments which have no parent net segments. It further means that that all the parent net segments of a child net segment shall be traversed before that child net segment is traversed.

F.2.1 Default discipline resolution algorithm

This default algorithm propagates both continuous and discrete disciplines up the hierarchy to meet one another. Insertion of interface elements shall occur at each level of the hierarchy where both continuous and discrete disciplines meet. This results in connection modules being inserted higher up the design hierarchy. The algorithm is described as follows.

1) Elaborate the design

   After this step, every port in the design has both its upper (actual) connection and its lower (formal) connection defined.

2) Apply all in-context node and signal declarations

   For example, `electrical sig;` makes all instances of `sig` electrical, unless they have been overridden by an out-of-context declaration.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

3) Apply all out-of-context node and signal declarations.

   For example, `electrical top.middle.bottom.sig;` overrides any discipline which may be declared for `sig` in the module where `sig` was declared.

   More than one conflicting in-context discipline declaration or more than one conflicting out-of-context discipline declaration for the same hierarchical segment of a signal is an error. In this case, *conflicting* simply means an attempt to declare more than one discipline regardless of whether the disciplines are compatible or not.

4) Traverse each signal hierarchically (depth-first) when a net is encountered which still has not been assigned a discipline:

   a) It shall be determined whether the net is analog or digital. Any net which is used in digital behavioral code shall be considered digital. Any net whose child nets are all digital shall be considered digital (discrete domain), any others shall be considered analog (continuous domain).

   b) If the net has not yet been assigned a discipline, examine all the child nets of that net and construct a list of all disciplines of the child nets whose domains match the domain of the segment:

   — If there are no disciplines in the list apply any **`default_discipline** directives to the net, provided their domain is the same as the domain of the net. This is done according to the rules of precedence for **`default_discipline** (see 3.8).

   — If there is only a single discipline in the list, the signal is of that discipline

   — If there is more than one discipline in the list and the contents of the list match the discipline list of a resolution connect statement, the net is of the resolved discipline given by the statement.

   — Otherwise the discipline is unknown. This is legal provided the net has no mixed-port connections (i.e., it does not connect through a port to a segment of a different domain). Otherwise this is an error

At this point, connection module selection and insertion can be performed. Insert converters applying the rules and semantics of the connect statement (7.7) and auto-insertion sections (7.8).

F.2.2 Alternate expanded analog discipline resolution algorithm

This algorithm propagates continuous disciplines up and then back down to meet discrete disciplines. This may result in more connection modules being inserted lower down into discrete sections of the design hierarchy for added accuracy. The selection of this algorithm instead of the default shall be controlled by a simulator option. The algorithm is described as follows.

1) Elaborate the design

   After this step, every port in the design has both its upper (actual) connection and its lower (formal) connection defined.

2) Apply all in-context node and signal declarations

   For example, `electrical sig;` makes all instances of `sig` electrical, unless they have been overridden by an out-of-context declaration.

3) Apply all out-of-context node and signal declarations.

   For example, `electrical top.middle.bottom.sig;` overrides any discipline which may be declared for `sig` in the module where `sig` was declared.

   More than one conflicting in-context discipline declaration or more than one conflicting out-of-context discipline declaration for the same hierarchical segment of a signal is an error. In this case, *conflicting* simply means an attempt to declare more than one discipline regardless of whether the disciplines are compatible or not.

4) Traverse each signal hierarchically (depth-first) when a net is encountered which has still not been assigned a discipline:

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

a) It shall be determined whether the net is analog or digital. Any net which is used in digital behavioral code shall be considered digital. Any net whose child nets are all digital shall be considered digital. If any of the connections are analog, the net shall be considered analog. Any others shall still be considered unknown.

b) If the net has not yet been assigned a discipline, examine all the child nets of that net and construct a list of all disciplines of these child nets whose domains match the domain of the segment:

— If there are no disciplines in the list apply any **`default_discipline** directives to the net segment, provided their domain is the same as the domain of the net. This is done according to the rules of precedence for **`default_discipline** (see 3.8).

— If there is only a single discipline in the list, the signal is of that discipline

— If there is more than one discipline in the list and the contents of the list match the discipline list of a resolution connect statement, the net is of the resolved discipline given by the statement.

— Otherwise the discipline is unknown. This is legal provided the net has no mixed-port connections (i.e., it does not connect through a port to a segment of a different domain). Otherwise this is an error.

5) Traverse each signal hierarchically (top-down) when a net is encountered which still has not been assigned a discipline or which has been assigned a digital domain from step 4:

a) It shall be re-determined whether the net is analog or digital. Any net which is used in digital behavioral code shall be considered digital. Any net whose parent nets are digital shall be considered digital. Any others shall be considered analog.

b) If the net has not yet been assigned a discipline, examine all the parent nets of that net and construct a list of all disciplines of these parent nets whose domains match the domain of the segment:

— If there are no disciplines in the list apply any **`default_discipline** directives to the net, provided their domain is the same as the domain of the net. This is done according to the rules of precedence for **`default_discipline** (see 3.8).

— If there is only a single discipline in the list, the signal is of that discipline

— If there is more than one discipline in the list and the contents of the list match the discipline list of a resolution connect statement, the net is of the resolved discipline given by the statement.

— Otherwise the discipline is unknown. This is legal provided the net has no mixed-port connections (i.e., it does not connect through a port to a segment of a different domain). Otherwise this is an error.

At this point, connection module selection and insertion can be performed. Insert converters applying the rules and semantics of the connect statement (7.7) and auto-insertion sections (7.8).

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Annex G

(informative)

Change history

This annex lists the changes made to the document for each revision.

G.1 Changes from previous LRM versions

This subclause highlights some of the key differences between versions of the Verilog-AMS HDL reference manual. The syntax and semantics of this document supersede any syntax, semantics, or interpretations of previous revisions.

Table G.1—Changes from v1.0 to v2.0 syntax

Feature	OVI Verilog-A v1.0	OVI Verilog-AMS v2.0	Change type
Analog time	$realtime	$abstime	new
Ceiling operator	N/A	**ceil**(expr)	new
Floor operator	N/A	**floor**(expr)	new
Circular integrator	N/A	**idtmod**(expr)	new
Expression looping	N/A	**genvar**	new
Distribution functions	$dist_*functions*() Integer based functions	$rdist_*functions*() Real value equivalents to $dist_*functions*()	new
Empty discipline	predefined as type wire	type not defined	default definition
Implicit nodes	‘default_nodetype discipline_identifier default: **wire**	default type: empty discipline, no domain type	default definition
initial_step	default = TRAN	default = ALL	default definition
final_step	default = TRAN	default = ALL	default definition
Analog **ground**	no definition	now a declaration statement	definition
$realtime	$realtime :timescale =1 sec	$realtime :timescale= ’timescale def=1n, see $abstime	definition
Array setting	aa[0:1] = {2.1 = (1), 4.5 = (2)	aa[0:1] = {2.1,4.5}	syntax
Discontinuity function	**discontinuity**(*x*)	**$discontinuity**(x)	syntax
Limiting exponential function	$limexp(*expression*)	**limexp**(*expression*)	syntax
Port branch access	I(a,a)	I(<*a*>)	syntax

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table G.1—Changes from v1.0 to v2.0 syntax *(continued)*

Feature	OVI Verilog-A v1.0	OVI Verilog-AMS v2.0	Change type
Timestep control (maximum stepsize)	**`bound_step`**(*const_expression*)	$bound_step(*expr*)	syntax
Continuous waveform delay	**`delay()`**	**`absdelay()`**	syntax
User-defined analog functions	**`function`**	**`analog function`**	syntax
Discipline domain	N/A, assumed continuous	now continuous(default) and discrete	Extension
k scalar (10^3)	N/A, only “K” supported	now supported	Extension
Module keyword	**`module`**	**`module`** or **`macromodule`**	Extension
Modulus operator	`integers` only	now supports `integer` and `reals`	Extension
Time tolerance on timer functions	N/A	supports additional time tolerance argument for **`timer`**()	Extension
Time tolerance on transition filter	N/A	supports additional time tolerance argument for **`transition`**()	Extension
‘default_nodetype	‘**`default_nodetype`**	‘**`default_discipline`**	Obsolete
Forever statement	**`forever`**	N/A	Obsolete
Generate statement	**`generate`**	N/A	Obsolete
Null statement	;	Limited to case, conditional, and event statements (see syntax)	Obsolete

Table G.2—Changes from v2.0 to v2.1

Item	Description/Issue	Clause
1	Clarification on when range checking for parameters is done. Range check will be done only on the final value of the parameter for that instance.	3.4.2
2	Not to use “max” and use “maxval” instead since **`max`** is a keyword	3.6.1.1, 3.6.2.6
3	Support of user-defined attributes to disciplines similar to natures has been added. This would be a useful way to pass information to other tools reading the Verilog-AMS netlist	3.6.2, 3.6.1.3
4	LRM specifies TRI and WIRE as aliases. The existing AMS LRM forces nets with wiretypes other than wire or tri to become digital, but in many cases these are really interconnect also. If they are tied to behavioral code they will become digital but if they are interconnected, we should not force them until after discipline resolution. This is needed if you have configs where the blocks connected to the net can change between analog and digital. If we force these nets to be digital we force unneeded CMs when blocks are switched to analog.	3.6.2.4, 3.7

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table G.2—Changes from v2.0 to v2.1 *(continued)*

Item	Description/Issue	Clause
5	Setting an initial value on net as part of the net declaration.	3.6.3, Syntax 3-6, 3.6.3.2
6	Initial value of **`wreal`** to be set to 0.0 if the value has not been determined at t = 0.	3.7
7	Clarification on the usage of **`` `default_discipline``** and default discipline for analog and digital primitives. Analog primitives will have default discipline as electrical, whereas digital primitives shall use the **`` `default_discipline``** declaration. **`` `default_discipline``** explanation moved to the section along with other compiler directives and clarification of impact on **`` `reset_all``** on this. The usage of word 'scope' is clarified to be used as the scope of the application of the compiler directive, and not as a scope argument.	3.8, 3.9, 3.10, 10.2, 10.3
8	Reference to derived disciplines to be removed as current BNF does not support the syntax	3.11
9	Reworked discipline and nature compatibility rules for better clarity.	3.11
10	Removed the reference to neutral discipline since wire can be used in the same context.	3.11
11	**`absdelay`** instead of **`delay`**	4.5.14
12	Array declaration wrongly specified before the variable identifier. For variables, array specification is written after the name of the variable.	3.2
13	**`@(final_step)`** without arguments should not have parenthesis	5.10.2, Table 5-1
15	**`@(final_step)`** for DCOP should be 1	5.10.2, Table 5-1
16	Examples to be fixed to use assign for **`wreal`** and use **`wreal`** in instantiation, and also add a top level block for example in 7.3.3, and the testbench use **`wreal`**.	6.5.3, 3.7
17	Clarification on the port bound semantics in explaining the hierarchical structure for a port with respect to vpiLoConn and vpiHiConn and clarification on driver and receiver segregation	7.2.3
18	Figure should have NetC.c_out instead of NetC.b_out	7.2.3
19	Mixed-signal module examples to use case syntax with X & Z instead of "==" for value comparison	7.2.3
20	Clarification on accessing discrete nets and variables and X & Z bits in the analog context.	7.2.3
21	Adding Support for 'NaN & X' into Verilog-AMS. Contribution of these values to a branch would be an error; however, analog variables should be able to propagate this value. Added a section regarding NaN	7.2.3, 7.3.2.1
22	The diagram corresponding to the bidir model has been reworked, and the example module shown for bidir will match the corresponding figure.	7.6
23	Rework on *connect-resolveto* syntax section to clarify the rules	7.7.2.1
24	Use **`merged`** instead of **`merge`**	7.8.1
25	Support for digital primitive instantiation in an **`analog`** block. Port names are created for the ports of the digital primitives, and these digital port names cannot be used in child instantiations.	7.8.5.1

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table G.2—Changes from v2.0 to v2.1 *(continued)*

Item	Description/Issue	Clause
26	Net resolution function has been removed and replaced with 'Receiver Net Resolution'. Reintroduced the **`assign`** statement syntax.	7.10.5 (subclause deleted in v2.3)
27	Corrections to the connect module example using the driver access function. The errors in the example have been corrected to make it syntactically and semantically correct	7.10.6 (subclause deleted in v2.3)
28	The constraints for supplementary drivers and delays ar clearly stated.	7.11 (subclause deleted in v2.3)
29	Driver Type function: There should be a driver access function for finding type of driver. driver_type_function ::= $driver_type(signal_name, signal_index)	7.11.4 (subclause deleted in v2.3), Annex D
30	Clarification on the MS synchronization algorithm: Includes a more detailed explanation on the analog-digital synchronization mechanism.	Clause 8
31	Truncation versus Rounding mechanism for converting from analog to digital times.	8.4.3.3
32	Spelling mistake on "boltzmann" and "planck" in constants file	Annex D
33	Units for charge, angle and other definitions in disciplines.vams have been changed to adhere to SI standards.	Annex D
34	Values specified in constants file for charge, light, Boltzmann constant, and so forth have been changed to adhere to the standard definitions.	Annex D

Table G.3—Changes from v2.1 to v2.2

Item	Description/Issue	Clause
1	Attributes were added following syntax in 1364-2001.	2.9
2	Output variables were defined.	3.2.1
3	Parameters were extended to include units and descriptions, **`localparam`**, **`aliasparam`**, and **`string`** parameters.	3.4, 3.4.3, 3.4.5, 3.4.6, 3.4.7, Syntax 6-1
4	Net descriptions allowed by attributes.	3.6.3.1
5	Additional bitwise operators were added.	Table 4-2, 4.2.9
6	Modifications to the domains of functions.	Table 4-9, Table 4-10
7	Changes to the descriptions of access function examples.	Table 4-11
8	Added symbolic derivative operators **`ddx`**()	4.5.6
9	Added references to limiting algorithms, cross-reference to **`$limit()`**	4.5.13
10	Added entries for **`above()`**, **`ddx()`**, and **`$limit()`**	4.5.14
11	Clarified dc sweep behavior for **`analysis`**(), **`initial_step`**, and **`final_step`**;added section describing dc analysis.	4.2.1, 4.5.2, Table 5-1
12	Allow multiple return values for analog functions.	4.7
14	Add above event	5.10.3, 5.10.3.2

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table G.3—Changes from v2.1 to v2.2 *(continued)*

Item	Description/Issue	Clause
15	Module descriptions allowed by attributes.	6.2, Syntax 6-1
16	Allow attributes for module item declarations	Syntax 6-1
17	Add **`$param_given()`** and **`$port_connected()`**	6.3.5, 6.5.6, 9.19
18	Added hierarchical system parameters **`$mfactor`**, **`$xposition`**, **`$yposition`**, **`$angle`**, **`$hflip`**, **`$vflip`**	6.3.6, Syntax 6-2, Syntax 6-3, 9.18, Annex E.4.1
19	Add paramsets	6.4
20	Add **`$simparam()`**	9.15
21	Add support for Monte-Carlo analysis to **`$random`** and **`$rdist_`** functions; clarify descriptions of arguments.	9.13
22	Add **`$debug()`**	9.4
23	Add format specifiers %r and $R	Table 9-23
24	Add support for limiting (damped Newton-Raphson) with **`$limit()`** and **`$discontinuity(-1)`**	9.17, Annex E.3.4
25	Add interpolation function **`$table_model()`**	9.20
26	Add **`__VAMS_COMPACT_MODELING__`**	10.5
27	New keywords: **`above`**, **`aliasparam`**, **`ddx`**, **`endparamset`**, **`localparam`**, **`paramset`**, **`string`**	Annex B
28	Corrected the value of ‘M_TWO_PI, defined Planck’s constant as‘P_H (not ‘P_K, which is Boltzmann’s constant), removed parenthetical value after ‘P_U0	D.3

Table G.4—Changes from v2.2 to v2.3

Item	Description/Issue	Clause
1	Add string data type and applicable operations	3.3
2	Add apostrophe before opening { in list of values (to distinguish a list of values from the concatenation operator)	3.4.2
3	Add Verilog function style versions of standard mathematical functions **`$ln()`**, **`$log10()`**, **`$exp()`**, **`$sqrt()`**, **`$pow()`**, **`$floor()`** and **`$ceil()`**	Table 4-14
4	Add Verilog function style versions of trigonometric and hyperbolic functions **`$sin()`**, **`$cos()`**, **`$tan()`**, **`$asin()`**, **`$acos()`**, **`$atan()`**, **`$atan2()`**, **`$hypot()`**, **`$sinh()`**, **`$cosh()`**, **`$tanh()`**, **`$asinh()`**, **`$acosh()`** and **`$atanh()`**	Table 4-15
5	Specify atan2(0, 0) as equal to 0	Table 4-15
6	Disallow V(n1, n1) as legal access function usage	Table 4-16
7	Add conversion from real to integer	4.2.1.1

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table G.4—Changes from v2.2 to v2.3 *(continued)*

Item	Description/Issue	Clause
8	More strict definition of time-integral operator	4.5.4
9	The `noise_table()` function accepts a file name as argument to read table data from file.	4.6.4.3
10	Define use of locally defined parameters and module-level parameters inside user-defined analog functions	4.7
11	Define semantics of inout arguments for user-defined analog functions	4.7.2.4
12	Allow a user-defined analog function to be called from within another user-defined analog function	4.7.3
13	Support for the **`analog initial`** block	5.2.1
14	Detailed restrictions on conditional statements	5.8
15	Detailed restrictions on looping statements	5.9
16	The **`cross`**, **`above`** and **`timer`** monitored event functions have been extended with an enable argument	5.10.3, Syntax 5-13
17	Support for null arguments in the **`cross`**, **`above`** and **`timer`** monitored event functions	5.10.3, Syntax 5-13
18	Support for multiple **`analog`** blocks within a single module	6.2
19	Added extra rule on connected ports for paramset selection	6.3.3
20	Support for loop generate constructs and conditional generate constructs	6.6.2
21	Restricted use of out-of-module references (OOMRs)	6.7.1
22	Extended scope definitions to generate blocks	6.8
23	Added elaboration rules for analog and mixed-signal hierarchies	6.9
24	Support for discipline incompatibility declaration	7.7.2
25	Description of mixed-signal DC analysis process	8.4.2
26	Extended definition of **`$fopen()`**	9.5.1
27	Support for **`$fdebug()`** system task	9.2, 9.5.2
28	Support for the **`$swrite()`** and **`$sformat()`** system tasks	9.5.3
29	Support for $fatal, **`$error`**, **`$warning`**, and **`$info`** system tasks	9.7.3
30	Renamed the former **`$random`** system task to **`$arandom`**	9.13.1
31	Added the **`$simprobe()`** system task	9.16
32	Extended **`$table_model()`** system function to support isoline data, tables with multiple dependent values, and higher-order data interpolation.	9.20
33	Support for **`` `begin_keywords ``** and **`` `end_keywords ``** compiler directives; added `"VAMS-2.3"` version specifier for keywords compiler directive	10.6
34	Support for port declarations in module header	A.1.2, Syntax 6-1
35	Optional semicolon following the nature identifier in a nature declaration	A.1.6, Syntax 3-4

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table G.4—Changes from v2.2 to v2.3 *(continued)*

Item	Description/Issue	Clause
36	Optional semicolon following the discipline identifier in a discipline declaration	A.1.7, Syntax 3-5
37	Annex C of LRM v2.2 has been split and the section describing the changes from previous LRM versions has been documented in this Annex	
38	Introduced guard clauses to the driver_access.vams standard definitions	D.3
39	Corrected syntax of **`port_discipline`** attribute	E.3.2.1
40	Added name scoping of analog primitives	E.3.3
41	Annex G of version 2.2, Open Issues, removed; this information is now in the Verilog Mantis data base	
42	The keywords in Annex B.2 and Annex B.3 have been merged into the single table in Annex B.1	

Table G.5—Changes from v2.3 to v2.3.1

Mantis Item	Description/Issue	Clause
2266	The signal flow discipline for current now uses the flow nature and not potential	D.1
2391	Clarified semantics for when a branch is treated as a flow source of value zero (0)	5.4.4, 5.6.1.3
2453	Corrected summation formula for the analog filter function **`laplace_nd()`**	4.5.11.4
2458	Added **`$simparam$str`** to syntax box	Syntax 9-10
2498	Added in keywords: **`wire`**, **`wor`**, **`wreal`**, **`xnor`**, **`xor`**, **`zi_nd`**, **`zi_np`**, **`zi_zd`**, and **`zi_zp`** which were accidentally deleted in LRM v2.3	Annex B
2535	Corrected definition for multiline strings	A.8.8
2536	Corrected examples that were using invalid real numbers	3.6.2.1
2538	Removed redundant *string_parameter_declaration* and *local_string_parameter_declaration* syntax items	A.1.9
2391	Clarified definition of a switch branch	5.6.1, 5.8.1
2581	Clarified restrictions on unnamed branches	3.12
2589	Removed multiple definitions of *net_assignment*	A.2.1.3, A.2.3, A.2.4, A.8.4
2497	Added in definition of *nature_access_identifier* syntax item	A.9.3
2497	Added in definition of *text_macro* syntax item	Syntax 10-3
2497	Syntax item *analog_variable_lvalue* was missing in certain places	Syntax 5-14, Syntax 7-3
2497	Mathematical function, **`pow()`**, was missing from *analog_built_in_function_name* syntax item definition	A.8.2

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table G.5—Changes from v2.3 to v2.3.1 *(continued)*

Mantis Item	Description/Issue	Clause
2497	A new syntax item, *analog_or_constant_expression*, has been created to allow the use of the analog **`analysis()`** function as part of the constant conditional expression of an **`if-else`** statement	5.8.1, A.8.3
2537	Corrected example where the *parameter_type* was specified before the **`parameter`** keyword	5.10.3.1
2537	Missing trailing ";" in *analog_function_item_declaration* for the *input_declaration*, *output_declaration*, and *inout_declaration*	A.2.6
2538	Missing trailing ";" in the *analog_block_item_declaration* for *parameter_declaration*	A.2.8
2537	Added in a new syntax item definition, *paramset_constant_expression*, which is used as the RHS expresson in the *paramset_statement*	6.4, A.1.9

Table G.6—Changes from v2.3.1 to v2.4

Mantis Item	Description/Issue	Clause
831	Clarified ambiguity in named vector branch indexing	3.12
874	Corrected example to use spice *vsine* primitive	3.6.2.1
875	Added support for **`string`** parameters to **`$fopen()`**	9.5.1
876	Clarified ambiguity regarding vector port range specification	6.5.2.2
1638	Modified standard definition file for physical constants to allow for backward compatibility	D.2
1854	Added support for parameter aliases to the hierarchical system parameters	3.4.7, 9.18
2266	Added additional clarification for signal flownodes	1.3.4
2331	Changed rule from *hierarchical_system_parameter_functions* to *hierarchical_parameter_system_functions*	9.18
2792	Corrected table to indicate that **`$monitor`** is supported in the analog context	Table 9-1
2806	Corrected rule for **`last_crossing`** to allow the *direction* argument to be optional	4.5.10
2836	Added support for .*module_output_variable_identifier* to the **`paramset`** definition	6.4, A.1.9, A.9.3
2843	Modified **`$monitor`** description to specify that input arguments **`$abstime`** and **`$realtime`** don't cause it to fire	9.4.1
2860	Re-formatted *connect_resolution* rule to remove ambiguity	7.7.2, A.1.8
2921	Clarified $random and $arandom support for when the seed argument is a **`reg`** or **`time`** variable	9.13.1
2922	Clarified the formal argument requirements for analog user-defined functions	4.7.1

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table G.6—Changes from v2.3.1 to v2.4 *(continued)*

Mantis Item	Description/Issue	Clause
3343	Corrected table cross reference	9.5.1
3371	Corrected example with incorrect range specification	4.5.8
3435	Allow $sscanf to also accept string parameters and literals	9.5.4.2
3461	Corrected error with incorrect capitalization of the reference to the *ttl* discipline	3.6.1.1, 3.6.2.6
3462	Modified description of when short-circuit evaluation occurs	4.2.3
3464	Corrected example for **`$simprobe`** where the declared nets where separated by '.' characters and not ','	9.16
3465	Corrected default value for **`parameter`** `slewrate`	8.4.3.3
3466	Corrected range definition for **`parameter integer`** `dir`	5.10.3.1
3527	Removed ambiguities for usage of **`$table_model`**	9.20
3570	Corrected *net_decl_assignment* syntax rule	3.6.3, 6.5.2.1, A.2.4
4064	Removed redundant paragraph	6.7
4170	Corrected examples	6.6.1
4193	Clarified behavior for multiple **`$bound_step`** tasks	9.17.2
4259	Corrected syntax for the random seed argument to allow negative numbers	9.13.1, 9.13.2
4308	Added examples showing how to support multiple power regions in a mixed signal simulation	7.8.6
4320	Added explicit default for the *type_string* argument to the distribution functions	9.13.1, 9.13.2
4339	Removed several restrictions on the use of out-of-module references	3.12, 5.5.1, 5.5.4, 5.6.8.1, 6.7.1, A.2.1.3, A.8.9
4348	Specified the minimum data requirements for **`$table_model`**	9.20
4349	Added support for **`$noise_table_log`**	4.5.1, 4.6.4, 4.6.4.4, A.8.2
4350	Specified the behavior of the severity system functions when called from an **`analog initial`** block	9.7.1, 9.7.2, 9.7.3
4355	Corrected missing variable declaration in example	7.3.4
4356	Corrected missing ground declaration in example	3.6.2.1, 6.2.2
4441	Added examples for multidimensional arrays	3.2, 3.3, 3.4, 3.4.8, 4.2.14, 4.5.1, 5.7, A.6.2, A.8.1, A.8.5
4473	Corrected error in timer description with the *start_time* argument	5.10.3.3
4484	Specified behavior of **`final_step`** in conjunction with **`$finish`**	5.10.2, 9.7.1
4543	Added support for the anaog node alias system functions **`$analog_node_alias()`** and **`$analog_port_alias()`**	7.8.6, Table 9-17, 9.20

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table G.6—Changes from v2.3.1 to v2.4 *(continued)*

Mantis Item	Description/Issue	Clause
4582	Added additional standard attributes: *op* and *multiplicity*	2.9, 2.9.2
4689	Added copyright notice for the standard header files to allow distribution	Annex D
4713	Clarified probe branch semantics	1.3.1, 5.4.2.1
4754	Added support for the branch access functions: **`potential()`** and **`flow()`**	4.4, 5.5.1, 5.6.1, A.8.2
4792	Corrected problems with discipline propagation algorithm	F.2
4795	Added in a chart outlining the analog simulation initialization flow	8.2
4803	Added in support for the **`absdelta`** event function	5.10, 5.10.3.4, 8.4.6, 8.4.7, A.6.5
4815	Deprecated support for empty disciplines	3.6.2, 3.6.3, 3.6.5, 3.8, 3.10, 3.11.1, 7.4, 7.4.4, 7.4.5, 10.2
4826	Added in support for the hierarchical identifier prefix **`$root`**	6.2.1, 6.7, A.9.3
4833	Corrected description for **`cross()`** which was missing the *enable* argument	5.10.3.1
4834	Ensure that document consistently refers to “dc sweep” in all lower case	5.2.1, 6.6.2.1
4849	Modified document contributor’s table to acknowledge people who have contributed to previous versions of the standard	

Table G.7—Changes from v2.4 to VAMS-2023

Mantis Item	Description/Issue	Clause
830	Support *jump_statements* **`return`**, **`break`**, and **`continue`**	5.11, A.6.5
2594	Resolved redundancy for *list_of_port_declarations*	A.1.3
4848	Updated description for Laplace and Zi transform filters to support vector parameters	4.5.11, 4.5.12
4926	Fixed incorrect grammar production for *analog_function_case_item*	A.6.7
4935	Incorrect font for ternary operator	A.1.9, A.8.3
5027	Removed unused keyword **`net_resolution`**	B.1, C.16
5036	Fixed incorrect reference to *net* with *node*	6.7
7754	Clarified description of the behavior for the system function **`$limit()`**	9.17.3
7780	Added support for math functions expm1() and ln1p()	4-14, 9-11, A.8.2, B.1
7791	Clarified description of the behavior for the analog operator **`limexp()`**	4.5.13
7792	Clarified description of **`last_crossing()`** operator	4.5.10
7793	Support **`$receiver_count()`** function,	9.22.2

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Table G.7—Changes from v2.4 to VAMS-2023 ***(continued)***

Mantis Item	Description/Issue	Clause
7794	Clarified description on how **`analog initial`** blocks impact identification of a variable domains	7.2.2
7795	Support the alternative Verilog style for **`$min()`**, **`$max()`**, and **`$abs()`**	4.3.1, 9-11
7808	Allow string return type for analog user defined functions Support **`return`** statement for analog user defined functions	4.7.1 4.7.2.2
7809	Explicitly describe how analog named events can be triggered and detected in the analog context	5.10.4
7810	Allow tolerance arguments to **`transition()`**, **`timer()`**, **`cross()`**, **`above()`** and **`absdelta()`** to be dynamic expressions	4-20, 5.10.3.1, 5.10.3.2, 5.10.3.3, 5.10.3.4, A.6.5
7811	Rework for description of the **`transition()`** filter to clearly define behavior when interrupted	4.5.8
7812	Clarify how linear interpolation is used for the **`absdelay()`** filter	4.5.7
7817	Fixed typography error in syntax for **`idtmod()`**	4-19
7886	Fixed example for sized literal decimal number	2.6.1
7888	Fixed example to use case equality operator (**`===`**) instead of logical equality operator (**`==`**)	7.3.2
7891	**`string`** was being used as both a keyword and a non-terminal. Renamed the non-terminal to *string_literal*	A.8.8
7893		
7897	Aligned wording for when short-circuit evaluation is applied with that described in IEEE Std 1364 Verilog	4.2.3
7898	Added example of arithmetic operator ** to table	4.2.4
7900	Corrected description for domain of pow() to not use undefined reference *int(y)*, but instead explicitly state *for all integer y*	4-14
7901	Moved definition for analog_filter_function_arg non-terminal from A.8.1 to A.8.2	A.8.2
7903	In example, discrete discipline should be `\logic`	10.6
7909	In the non-terminal *analog_filter_function_arg* replaced reference to *constant_optional_arrayinit* with newly defined *constant_assignment_pattern_or_null*	A.8.1, A.8.2
7912	Add support for **`__FILE__`** and **`__LINE__`** compiler directives	10.7
7920	Add support for **`$roi()`** and **`$itor()`** in the analog context	9-8, 9.11
7921	Added *VAMS-2023* as a keywords specifier	10.6
7922	Remove error for contributing to a port declared with out **`input`** direction. This is now a warning	5.6.1

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

G.2 Obsolete functionality

The following statements are not supported in the current version of Verilog-AMS HDL; they are only noted for backward compatibility.

G.2.1 Forever

This statement is no longer supported.

G.2.2 NULL

This statement is no longer supported. Certain functions such as case, conditionals and the event statement do allow null statements as defined by the syntax.

G.2.3 Generate

The *generate statement* is a looping construct which is unrolled at elaboration time. It is the only looping statement that can contain analog operators. The syntax of generate statement is shown in Figure G-1.

```
generate_statement ::=
    generate index_identifier ( start_expr , end_expr [ , incr_expr ] )
        statement

start_expr ::=
    constant_expression

end_expr ::=
    constant_expression

incr_expr ::=
    constant_expression
```

Figure G-1—Syntax for generate statement

The index shall not be assigned or modified in any way inside the loop. In addition, it is local to the loop and is expanded when the loop is unrolled. Even if there is a local variable with the same name as the index and the variable is modified as a side effect of a function called from within the loop, the loop index is unaffected.

The start and end bounds and the increment are constant expressions. They are only evaluated at elaboration time. If the expressions used for the increment and bounds change during the simulation, it does not affect the behavior of the generate statement.

If the lower bound is less than the upper bound and the increment is negative, or if the lower bound is greater than the upper bound and the increment is positive, then the generate statement does not execute.

If the lower bound equals the upper bound, the increment is ignored and the statement execute once. If the increment is not given, it is taken to be `+1` if the lower bound is less than the upper bound, and `-1` if the lower bound is greater than the upper bound.

The statement, which can be a sequential block, is replicated with all occurrences of index in the statement replaced by a constant. In the first instance of the statement, the index is replaced with the lower bound. In

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

the second, it is replaced by the lower bound plus the increment. In the third, it is replaced by the lower bound plus two times (2x) the increment. This pattern is repeated until the lower bound plus a multiple of the increment is greater than the upper bound.

Example: This module implements a continuously running (unclocked) analog-to-digital converter.

```
module adc(in,out) ;
   parameter bits=8, fullscale=1.0, dly=0.0, ttime=10n;
   input in;
   output [0:bits-1] out;
   electrical in;
   electrical [0:bits-1] out;
   real sample, thresh;
   analog begin
      thresh = fullscale/2.0;
      generate i (bits-1,0) begin
         V(out[i]) <+ transition(sample > thresh, dly, ttime);
         if (sample > thresh) sample = sample - thresh;
         sample = 2.0*sample;
      end
   end
endmodule
```

G.2.4 `default_function_type_analog

The **`default_function_type_analog** directive is no longer supported. this compiler directive allowed user-defined functions to be treated as analog functions in Verilog-A if they did not have the key word **analog** as part of the definition.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

Annex H

(informative)

Glossary

A

AMS

See also, *Verilog-AMS.*

B

behavioral description

A mathematical mapping of inputs to outputs for a module, including intermediate variables and control flow.

behavioral model

A version of a module with a unique set of parameters designed to model a specific component.

block

A level within the behavioral description of a module, delimited by **`begin`** and **`end`**.

branch

A relationship between two nodes and their attached quantities within the behavioral description of a module. Each branch has two quantities, a value and a flow, with a reference direction for each.

C

compact model

A behavioral model or description of a semiconductor device.

component

A fundamental unit within a system which encapsulates behavior and/or structure. Modules and models can represent a single component or a subcircuit with many components.

constitutive relationships

The essential relationships (*expressions* and *statements*) between the outputs of a module and its inputs and parameters, which define the nature of the module. These relationships constitute a behavioral description.

control flow

The conditional and iterative statements controlling the behavior of a module. These statements evaluate arbitrary variables (counters, flags, and tokens) to control the operation of different sections of a behavioral description.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

child module

A module instantiated inside another, “parent” module. A complete definition of the child module needs to be defined somewhere. A child module is also known as *instantiated module*.

F

flow

One of the two fundamental quantities used to simulate the behavior of a system. In electrical systems, the flow is the current.

I

instance

Any named occurrence of an component created from a module definition. One module can occur in multiple instances.

instantiation

The process of creating an instance from a module definition or simulator primitive and defining the connectivity and parameters of that instance. (Placing the instance in the circuit or system.)

K

Kirchhoff’s Laws

The physical laws defining the interconnection relationships of nodes, branches, values, and flows. They specify a conservation of flow in and out of a node and a conservation of value around a loop of branches.

L

level

One block within a behavioral description, delimited by a pair of matching keywords such as **`begin-end`** or **`discipline-enddiscipline`**.

M

model

A named instance with a unique group of parameters specifying the behavior of one particular version of a module. Models can be used to instantiate elements with parametric specifications different from those in the original module definition.

module

A definition of the interfaces and behavior of a component.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

N

net declaration

The statement in a module definition identifying the names of the nets associated with the module ports or local to the module. A net declaration also identifies the discipline of the net, which in turn identifies the access functions.

node

A connection point in the system, with access functions for potential and/or flow through an underlying discipline.

NR method

Newton-Raphson method. A generalized method for solving systems of nonlinear algebraic equations by breaking them into a series of many small linear operations ideally suited for computer processing.

P

parameter

A constant for characterizing the behavior of an instance of a module. Parameters are defined via a parameter declaration statement in the module definition, and can be specified each time a module is called in a netlist instance statement.

parameter declaration

The statement in a module definition which defines the parameters of that module.

port

An external connection point for a module (also known as a *terminal*).

potential

One of the two fundamental quantities used to simulate the behavior of a system. In electrical systems, the potential is the voltage.

primitive

A basic component defined entirely in terms of behavior, without reference to any other primitives. A primitive is the smallest and simplest portion of a simulated circuit or system.

probe

A branch in a circuit (or system), which does not alter its behavior, but lets the simulator read out the potential or flow at that point.

R

reference direction

A convention for determining whether the value of a node, the flow through a branch, the value across a branch, or the flow in or out of a terminal, is positive or negative.

reference node

A globalanalog node (which equals zero (`0`)) against whose potentials all node values are measured. Nets declared as `ground` shall be bound to the reference node.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.

S

scope

The current level of a block statement, which includes all lines of code within one set of braces in a module definition.

structural definitions

Instantiating modules inside other modules through the use of module definitions and declarations to create a hierarchical structure in the module's behavioral description.

T

terminal

See also, *port*.

V

Verilog-A

A subset of Verilog-AMS detailing the analog version of IEEE Std 1364 Verilog (see Annex C). This is a language for the behavioral description of continuous-time systems, which uses a syntax similar to the IEEE Std 1364 Verilog specification.

Verilog-AMS

Mixed-signal version of IEEE Std 1364 Verilog. A language for the behavioral description of continuous-time and discrete-time systems based on the IEEE Std 1364 Verilog specification.

Copyright © 2024 Accellera Systems Initiative. All rights reserved.